29K Family

1990 Data Book

Advanced
Micro
Devices

Advanced
Micro
Devices

29K Family Data Book

901 Thompson Place, P.O. Box 3453, Sunnyvale, California 94088-3000
(408)732-2400 TWX: 910-339-9280 TELEX: 34-6306

INTRODUCTION

The RISC-based Am29000 Streamlined Instruction Processor from Advanced Micro Devices is the high-performance solution for your general-purpose embedded systems needs. As the heart of the 29K Family, this 32-bit CMOS microprocessor delivers outstanding performance, yet offers flexible cost-effective solutions that can quickly move your product to market.

This data book is your comprehensive guide to AMD's 29K Family of microprocessors and development tools. These products have helped current developers create applications that fully exploit the power of the Am29000 microprocessor: laser printers of all types, real-time graphics systems, networks and bridges, and a host of other peripheral and communication devices.

To provide a total system solution for you, AMD has taken the 29K Family's advantages of 17-MIPS performance, flexible memory-configuration requirements, and outstanding development tools and coupled them with our Fusion29K™ program. This program provides you with AMD and industry-standard third-party solutions, including the application-specific solutions you need for successful system integration that can substantially shorten the time-to-market factor of your design.

AMD is committed to the 29K Family, and will continue to apply substantial resources to ensure that the present levels of high performance, cost and design flexibility, and rapid design cycles are maintained and further enhanced. Qualified support is readily available for our customers—our highly trained field applications engineers are backed by experts in the factory. For further details on how the 29K Family can be the solution to your design needs, call your local AMD sales office or the authorized representative listed in the back of this publication.

[illegible]

Geoff Tate
Senior Vice President
Microprocessors & Peripherals Group

PREFACE

Advanced Micro Devices' 29K™ Family is a new generation of high-performance CMOS microprocessor components and associated software tools. The heart of the 29K Family is the RISC-based Am29000™ microprocessor. The Am29000 Streamlined Instruction Processor is a high-performance, general-purpose, 32-bit microprocessor that supports a variety of applications, by virtue of a flexible architecture and rapid execution of simple instructions which are common to a wide range of tasks. The 29K Family's microprocessors are fully described in Chapter 1.

The Am29000 Streamlined Instruction Processor efficiently performs operations common to all systems, while deferring most decisions on system policies to the system architect. It is well suited for applications in high-performance workstations, general-purpose super minicomputers, high-performance real-time controllers, laser printer controllers, network protocol converters, and many other applications where high performance, flexibility, and the ability to program using standard software tools is important.

The Am29000 microprocessor has been enhanced to support byte and half-word loads and stores. This feature is provided as an option, requiring that an external device or memory be able to write individual bytes and/or half-words of a word. The Am29000 microprocessor can perform all necessary padding, sign extension, and alignment within the word. Furthermore, this feature is defined to be compatible with existing 29K Family software.

The Am29027™ Arithmetic Accelerator is a high-computational unit intended for use with the Am29000 Streamlined Instruction Processor. It connects directly to the Am29000 microprocessor's system buses, and requires no additional interface circuitry. When added to an Am29000 microprocessor-based system, the Am29027 co-processor can improve floating-point performance by an order of magnitude or more. The Am29027 co-processor implements an extensive floating-point and integer instruction set, and can perform operations on single-, double-, or mixed-precision operands.

But the superior performance of the 29K Family of microprocessors is only part of the story: AMD also provides a comprehensive set of software and hardware development tools, as shown in Chapter 2. These tools, coupled with the growing number of development products from established third-party vendors, can drastically reduce the time-to-market factor of designs.

For software development, AMD offers the globally optimizing HighC29K™ Cross-Development Toolkit, complete with high-performance math libraries. The HighC29K compiler is packaged with the ASM29K™ Cross-Development Toolkit, which includes a relocatable macro assembler, linker/loader, librarian, and a full architectural simulator of the Am29000 microprocessor.

Several debugging tools are available, including the XRAY29K™, a source-level debugger for high-level and assembly-level debugging and the software-based MON29K™ target-resident debugger/monitor. All tools work at the Am29000 processor's clock rate to allow debugging while operating at full microprocessor speed.

The application notes in Chapter 3 make development with the 29K Family of silicon and tools a simpler task. Within these documents, AMD engineers explore solutions of common problems that stand as roadblocks in your development path. So whether you need general information on programming standalone Am29000 microprocessor-based systems or detailed specifics on how to make your product HIF compatible, these application notes can provide the answers. And with new notes constantly being written and released, this wealth of knowledge will continue to be integral to your development process.

29K FAMILY DATA BOOK
TABLE OF CONTENTS

CHAPTER 1
29K Family CMOS Devices

Preliminary

Am29000

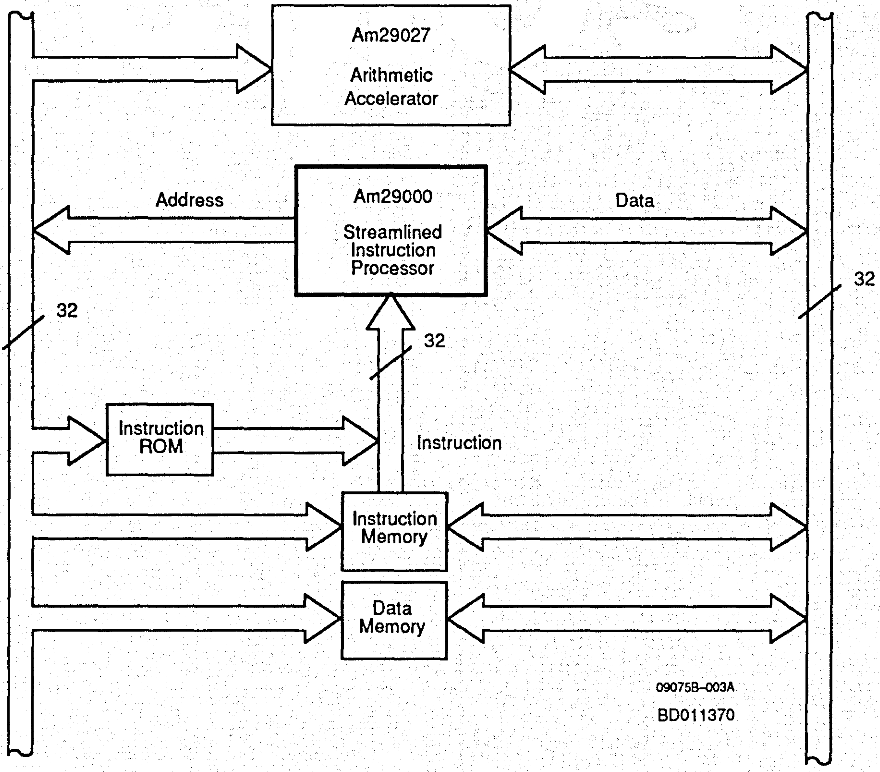
Streamlined Instruction Processor

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- Full 32-bit, three-bus architecture
- 23 million instructions per second (MIPS) sustained at 33 MHz
- 33-, 25-, 20-, and 16-MHz operating frequency
- Efficient execution of high-level language programs
- CMOS technology
- 4-gigabyte virtual address space with demand paging
- Concurrent instruction and data accesses
- Burst-mode access support
- 192 general-purpose registers
- 512-byte Branch Target Cache™
- 64-entry Memory-Management Unit
- Demultiplexed, pipelined address, instruction, and data buses
- Three-address instruction architecture
- On-chip byte-alignment support allows optional byte/half-word accesses

SIMPLIFIED BLOCK DIAGRAM



Publication # 09075 Rev. C Amendment 0
Issue Date: November 1989

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

GENERAL DESCRIPTION

The Am29000™ Streamlined Instruction Processor is a high-performance, general-purpose, 32-bit microprocessor implemented in CMOS technology. It supports a variety of applications by virtue of a flexible architecture and rapid execution of simple instructions that are common to a wide range of tasks.

The Am29000 efficiently performs operations common to all systems, while deferring most decisions on system policies to the system architect. It is well-suited for application in high-performance workstations, general-purpose super-minicomputers, high-performance real-time controllers, laser printer controllers, network protocol converters, and many other applications where high performance, flexibility, and the ability to program using standard software tools is important.

The Am29000 instruction set has been influenced by the results of high-level language, optimizing compiler research. It is appropriate for a variety of languages because it efficiently executes operations that are common to all languages. Consequently, the Am29000 is an ideal target for high-level languages such as C, FORTRAN, Pascal, Ada, and COBOL.

The processor is available in two packaging options: a 169-lead pin-grid-array (PGA) package, and a 164-lead Ceramic Quad Flat Pack (CQFP) package for the military. The PGA has 141 signal pins, 27 power and ground pins, and 1 alignment pin. The CQFP has 141 signal pins and 23 power and ground pins. A representative system diagram is shown on page 1.

29K™ Family Development Support Products

Contact your local AMD representative for information on the complete set of development support tools.

Software development products on several hosts:

- Optimizing compilers for common high-level languages
- Assembler and utility packages
- Source- and assembly-level software debuggers
- Target-resident development monitors
- Simulators

Hardware Development:

- ADAPT29K™ Advanced Development and Prototyping Tool

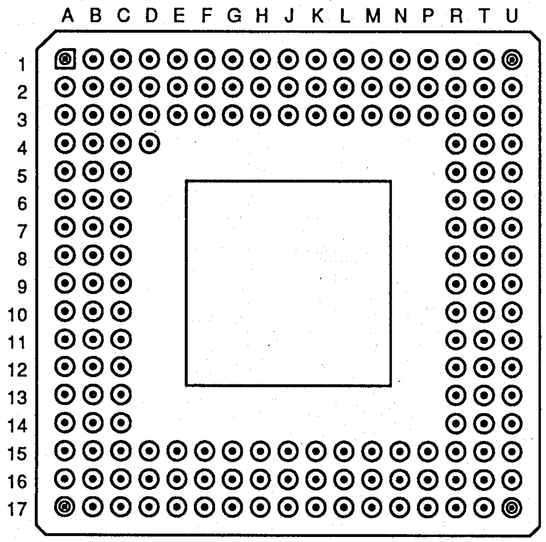
RELATED AMD PRODUCTS

Am29000 Peripheral Devices

Part No.	Description
Am29027™	Arithmetic Accelerator

CONNECTION DIAGRAM
169-Lead PGA*

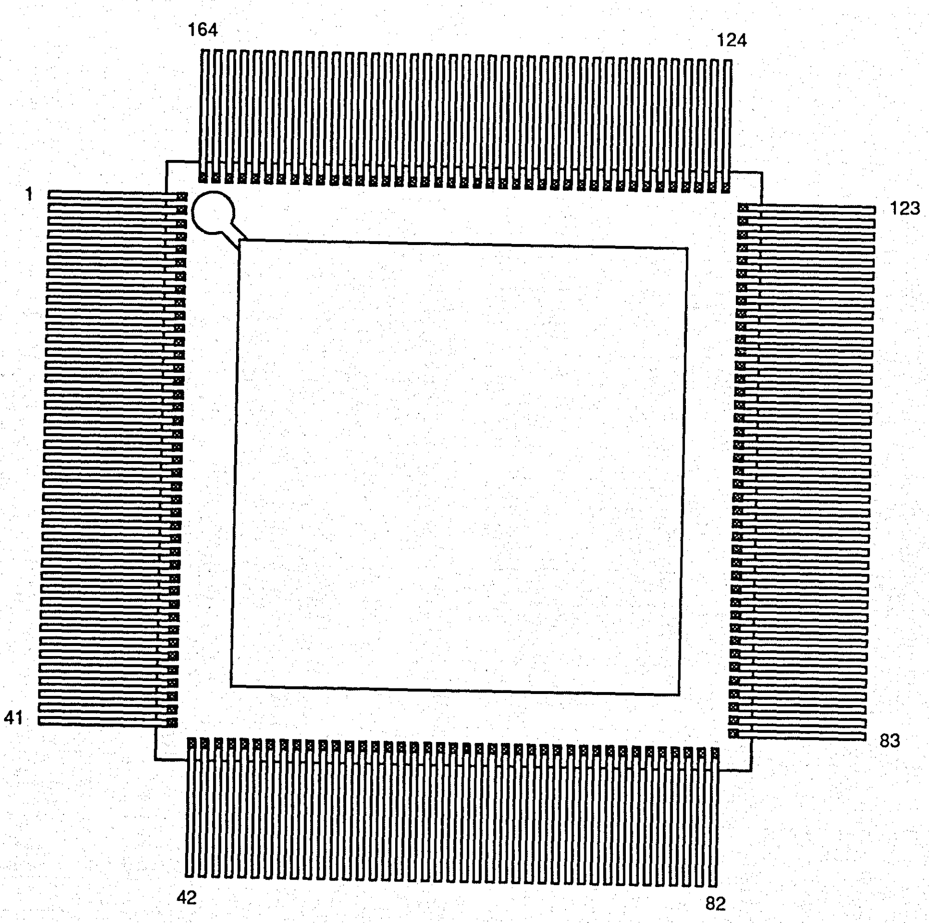
Bottom View



* Pinout observed from pin side of package.

CONNECTION DIAGRAM
164-Lead CQFP

Top View
(Lid Facing Viewer)



PGA PIN DESIGNATION
(Sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
A-1	GND	C-10	GND	J-16	A_{16}	R-12	$STAT_2$
A-2	I_1	C-11	GND	J-17	A_{14}	R-13	GND
A-3	I_0	C-12	D_{22}	K-1	I_{26}	R-14	OPT_0
A-4	D_2	C-13	D_{26}	K-2	I_{25}	R-15	A_2
A-5	D_4	C-14	V_{CC}	K-3	GND	R-16	A_6
A-6	D_6	C-15	D_{30}	K-15	V_{CC}	R-17	A_7
A-7	D_9	C-16	D_{31}	K-16	A_{12}	T-1	INCLK
A-8	D_{11}	C-17	A_{29}	K-17	A_{13}	T-2	$\overline{BREQ}$
A-9	D_{12}	D-1	I_{11}	L-1	I_{27}	T-3	$\overline{DERR}$
A-10	D_{14}	D-2	I_{10}	L-2	I_{28}	T-4	$\overline{IRDY}$
A-11	D_{16}	D-3	I_7	L-3	V_{CC}	T-5	$\overline{WARN}$
A-12	D_{18}	D-4	PIN169	L-15	V_{CC}	T-6	$\overline{INTR_2}$
A-13	D_{20}	D-15	A_{31}	L-16	A_{10}	T-7	$\overline{INTR_0}$
A-14	D_{21}	D-16	A_{28}	L-17	A_{11}	T-8	$\overline{BINV}$
A-15	D_{25}	D-17	A_{26}	M-1	I_{29}	T-9	$\overline{BGRT}$
A-16	D_{27}	E-1	I_{13}	M-2	I_{30}	T-10	$\overline{DREQ}$
A-17	GND	E-2	I_{12}	M-3	GND	T-11	$\overline{LOCK}$
B-1	I_6	E-3	V_{CC}	M-15	GND	T-12	MSERR
B-2	I_5	E-15	GND	M-16	A_0	T-13	$STAT_0$
B-3	I_3	E-16	A_{27}	M-17	A_1	T-14	$SUP/\overline{US}$
B-4	D_0	E-17	A_{23}	N-1	I_{31}	T-15	OPT_1
B-5	D_1	F-1	I_{16}	N-2	$\overline{TEST}$	T-16	A_3
B-6	D_5	F-2	I_{15}	N-3	SYSCLK	T-17	A_4
B-7	D_8	F-3	I_{14}	N-15	GND	U-1	GND
B-8	D_{10}	F-15	A_{25}	N-16	$MPGM_1$	U-2	$\overline{PEN}$
B-9	D_{13}	F-16	A_{24}	N-17	$MPGM_0$	U-3	$\overline{IERR}$
B-10	D_{15}	F-17	A_{21}	P-1	$CNTL_1$	U-4	$\overline{IBACK}$
B-11	D_{17}	G-1	I_{19}	P-2	$CNTL_0$	U-5	$\overline{INTR_3}$
B-12	D_{19}	G-2	I_{18}	P-3	PWRCLK	U-6	$\overline{INTR_1}$
B-13	D_{23}	G-3	I_{17}	P-15	A_5	U-7	$\overline{TRAP_0}$
B-14	D_{24}	G-15	A_{22}	P-16	A_8	U-8	$\overline{IBREQ}$
B-15	D_{28}	G-16	A_{20}	P-17	A_9	U-9	$\overline{IREQ}$
B-16	D_{29}	G-17	A_{19}	R-1	$\overline{RESET}$	U-10	$\overline{PIA}$
B-17	A_{30}	H-1	I_{20}	R-2	$\overline{CDA}$	U-11	$R/\overline{W}$
C-1	I_9	H-2	I_{22}	R-3	$\overline{DRDY}$	U-12	$\overline{DREQT_1}$
C-2	I_8	H-3	I_{21}	R-4	$\overline{DBACK}$	U-13	$\overline{DREQT_0}$
C-3	I_4	H-15	GND	R-5	GND	U-14	$STAT_1$
C-4	I_2	H-16	A_{18}	R-6	V_{CC}	U-15	IREQT
C-5	GND	H-17	A_{17}	R-7	$\overline{TRAP_1}$	U-16	OPT_2
C-6	D_3	J-1	I_{23}	R-8	GND	U-17	GND
C-7	D_7	J-2	I_{24}	R-9	$\overline{DBREQ}$		
C-8	V_{CC}	J-3	GND	R-10	$\overline{PDA}$		
C-9	V_{CC}	J-15	A_{15}	R-11	V_{CC}		

Note: Pin Number D-4 is the alignment pin and is electrically connected to the package lid.

PGA PIN DESIGNATIONS
(Sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
M-16	A_0	B-6	D_5	K-3	GND	T-1	INCLK
M-17	A_1	A-6	D_6	N-15	GND	T-7	$\overline{\text{INTR}_0}$
R-15	A_2	C-7	D_7	R-5	GND	U-6	$\overline{\text{INTR}_1}$
T-16	A_3	B-7	D_8	U-1	GND	T-6	$\overline{\text{INTR}_2}$
T-17	A_4	A-7	D_9	R-13	GND	U-5	$\overline{\text{INTR}_3}$
P-15	A_5	B-8	D_{10}	R-8	GND	T-4	$\overline{\text{IRDY}}$
R-16	A_6	A-8	D_{11}	M-3	GND	U-9	$\overline{\text{IREQ}}$
R-17	A_7	A-9	D_{12}	U-17	GND	U-15	IREQT
P-16	A_8	B-9	D_{13}	A-3	I_0	T-11	$\overline{\text{LOCK}}$
P-17	A_9	A-10	D_{14}	A-2	I_1	N-17	MPGM_0
L-16	A_{10}	B-10	D_{15}	C-4	I_2	N-16	MPGM_1
L-17	A_{11}	A-11	D_{16}	B-3	I_3	T-12	MSERR
K-16	A_{12}	B-11	D_{17}	C-3	I_4	R-14	OPT_0
K-17	A_{13}	A-12	D_{18}	B-2	I_5	T-15	OPT_1
J-17	A_{14}	B-12	D_{19}	B-1	I_6	U-16	OPT_2
J-15	A_{15}	A-13	D_{20}	D-3	I_7	R-10	$\overline{\text{PDA}}$
J-16	A_{16}	A-14	D_{21}	C-2	I_8	U-2	$\overline{\text{PEN}}$
H-17	A_{17}	C-12	D_{22}	C-1	I_9	U-10	$\overline{\text{PIA}}$
H-16	A_{18}	B-13	D_{23}	D-2	I_{10}	D-4	PIN169
G-17	A_{19}	B-14	D_{24}	D-1	I_{11}	P-3	PWRCLK
G-16	A_{20}	A-15	D_{25}	E-2	I_{12}	U-11	$\text{R}/\overline{\text{W}}$
F-17	A_{21}	C-13	D_{26}	E-1	I_{13}	R-1	$\overline{\text{RESET}}$
G-15	A_{22}	A-16	D_{27}	F-3	I_{14}	T-13	STAT_0
E-17	A_{23}	B-15	D_{28}	F-2	I_{15}	U-14	STAT_1
F-16	A_{24}	B-16	D_{29}	F-1	I_{16}	R-12	STAT_2
F-15	A_{25}	C-15	D_{30}	G-3	I_{17}	T-14	$\text{SUP}/\overline{\text{US}}$
D-17	A_{26}	C-16	D_{31}	G-2	I_{18}	N-3	SYSCLK
E-16	A_{27}	R-4	$\overline{\text{DBACK}}$	G-1	I_{19}	N-2	$\overline{\text{TEST}}$
D-16	A_{28}	R-9	$\overline{\text{DBREQ}}$	H-1	I_{20}	U-7	$\overline{\text{TRAP}_0}$
C-17	A_{29}	T-3	$\overline{\text{DERR}}$	H-3	I_{21}	R-7	$\overline{\text{TRAP}_1}$
B-17	A_{30}	R-3	$\overline{\text{DRDY}}$	H-2	I_{22}	C-14	Vcc
D-15	A_{31}	T-10	$\overline{\text{DREQ}}$	J-1	I_{23}	L-15	Vcc
T-9	$\overline{\text{BGRT}}$	U-13	DREQT_0	J-2	I_{24}	C-8	Vcc
T-8	$\overline{\text{BINV}}$	U-12	DREQT_1	K-2	I_{25}	C-9	Vcc
T-2	$\overline{\text{BREQ}}$	E-15	GND	K-1	I_{26}	E-3	Vcc
R-2	$\overline{\text{CDA}}$	H-15	GND	L-1	I_{27}	K-15	Vcc
P-2	CNTL_0	M-15	GND	L-2	I_{28}	L-3	Vcc
P-1	CNTL_1	C-10	GND	M-1	I_{29}	R-6	Vcc
B-4	D_0	A-1	GND	M-2	I_{30}	R-11	Vcc
B-5	D_1	A-17	GND	N-1	I_{31}	T-5	$\overline{\text{WARN}}$
A-4	D_2	C-5	GND	U-4	$\overline{\text{IBACK}}$		
C-6	D_3	C-11	GND	U-8	$\overline{\text{IBREQ}}$		
A-5	D_4	J-3	GND	U-3	$\overline{\text{IERR}}$		

Note: Pin Number D-4 is the alignment pin and is electrically connected to the package lid.

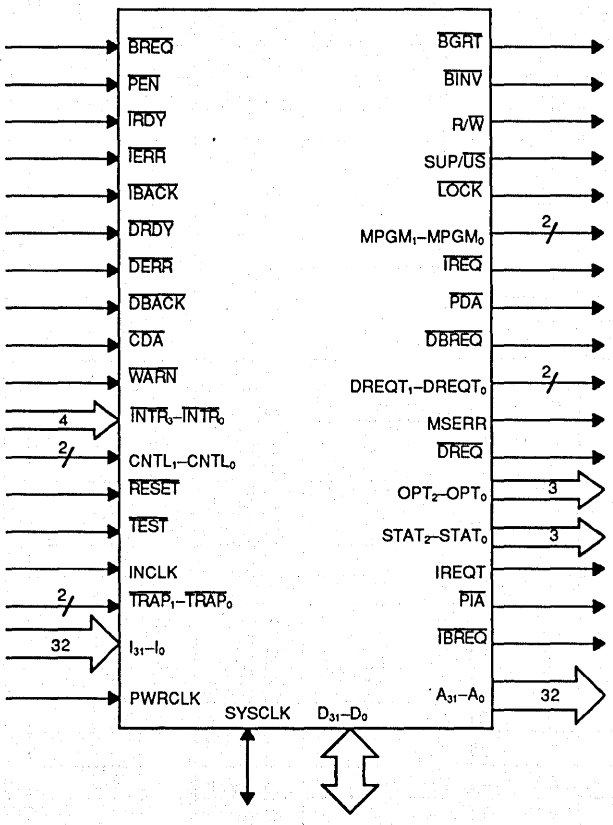
CQFP PIN DESIGNATION (Sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
1	$\overline{CDA}$	42	V_{CC}	83	V_{CC}	124	GND
2	INCLK	43	I_3	84	GND	125	OPT_0
3	PWRCLK	44	I_2	85	A_{31}	126	OPT_1
4	SYSCLK	45	I_1	86	A_{30}	127	OPT_2
5	GND	46	GND	87	A_{29}	128	$SUP/\overline{US}$
6	V_{CC}	47	I_0	88	A_{28}	129	IREQT
7	GND	48	D_0	89	A_{27}	130	$STAT_0$
8	$\overline{RESET}$	49	D_1	90	A_{26}	131	$STAT_1$
9	$CNTL_0$	50	D_2	91	A_{25}	132	$STAT_2$
10	$CNTL_1$	51	D_3	92	A_{24}	133	MSERR
11	$\overline{TEST}$	52	D_4	93	A_{23}	134	$DREQT_0$
12	I_{31}	53	D_5	94	A_{22}	135	$DREQT_1$
13	I_{30}	54	D_6	95	A_{21}	136	$\overline{LOCK}$
14	I_{29}	55	D_7	96	A_{20}	137	$R/\overline{W}$
15	I_{28}	56	D_8	97	A_{19}	138	$\overline{DREQ}$
16	I_{27}	57	D_9	98	A_{18}	139	$\overline{PDA}$
17	I_{26}	58	D_{10}	99	A_{17}	140	$\overline{PIA}$
18	I_{25}	59	D_{11}	100	A_{16}	141	$\overline{IREQ}$
19	I_{24}	60	D_{12}	101	A_{15}	142	$\overline{BGRT}$
20	GND	61	D_{13}	102	GND	143	$\overline{DBREQ}$
21	V_{CC}	62	D_{14}	103	V_{CC}	144	$\overline{IBREQ}$
22	I_{23}	63	V_{CC}	104	A_{14}	145	$\overline{BINV}$
23	I_{22}	64	GND	105	A_{13}	146	V_{CC}
24	I_{21}	65	D_{15}	106	A_{12}	147	GND
25	I_{20}	66	D_{16}	107	A_{11}	148	V_{CC}
26	I_{19}	67	D_{17}	108	A_{10}	149	GND
27	I_{18}	68	D_{18}	109	A_1	150	$\overline{TRAP_0}$
28	I_{17}	69	D_{19}	110	A_0	151	$\overline{TRAP_1}$
29	I_{16}	70	D_{20}	111	$MPGM_0$	152	$\overline{INTR_0}$
30	I_{15}	71	D_{21}	112	$MPGM_1$	153	$\overline{INTR_1}$
31	I_{14}	72	D_{22}	113	V_{CC}	154	$\overline{INTR_2}$
32	I_{13}	73	D_{23}	114	A_9	155	$\overline{INTR_3}$
33	I_{12}	74	D_{24}	115	A_8	156	$\overline{WARN}$
34	I_{11}	75	D_{25}	116	A_7	157	$\overline{IBACK}$
35	I_{10}	76	D_{26}	117	A_6	158	$\overline{IRDY}$
36	I_9	77	D_{27}	118	A_5	159	$\overline{IERR}$
37	I_8	78	D_{28}	119	A_4	160	$\overline{DERR}$
38	I_7	79	D_{29}	120	A_3	161	$\overline{DBACK}$
39	I_6	80	D_{30}	121	A_2	162	$\overline{PEN}$
40	I_5	81	D_{31}	122	GND	163	$\overline{BREQ}$
41	I_4	82	GND	123	GND	164	$\overline{DRDY}$

CQFP PIN DESIGNATIONS (Sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
110	A_0	51	D_3	82	GND	144	$\overline{IBREQ}$
109	A_1	52	D_4	84	GND	159	$\overline{IERR}$
121	A_2	53	D_5	102	GND	2	INCLK
120	A_3	54	D_6	122	GND	152	$\overline{INTR_0}$
119	A_4	55	D_7	123	GND	153	$\overline{INTR_1}$
118	A_5	56	D_8	124	GND	154	$\overline{INTR_2}$
117	A_6	57	D_9	147	GND	155	$\overline{INTR_3}$
116	A_7	58	D_{10}	149	GND	158	$\overline{IRDY}$
115	A_8	59	D_{11}	47	I_0	141	$\overline{IREQ}$
114	A_9	60	D_{12}	45	I_1	129	IREQT
108	A_{10}	61	D_{13}	44	I_2	136	$\overline{LOCK}$
107	A_{11}	62	D_{14}	43	I_3	111	$MPGM_0$
106	A_{12}	65	D_{15}	41	I_4	112	$MPGM_1$
105	A_{13}	66	D_{16}	40	I_5	133	MSERR
104	A_{14}	67	D_{17}	39	I_6	125	OPT_0
101	A_{15}	68	D_{18}	38	I_7	126	OPT_1
100	A_{16}	69	D_{19}	37	I_8	127	OPT_2
99	A_{17}	70	D_{20}	36	I_9	139	$\overline{PDA}$
98	A_{18}	71	D_{21}	35	I_{10}	162	$\overline{PEN}$
97	A_{19}	72	D_{22}	34	I_{11}	140	$\overline{PIA}$
96	A_{20}	73	D_{23}	33	I_{12}	3	PWRCLK
95	A_{21}	74	D_{24}	32	I_{13}	137	$R/\overline{W}$
94	A_{22}	75	D_{25}	31	I_{14}	8	$\overline{RESET}$
93	A_{23}	76	D_{26}	30	I_{15}	130	$STAT_0$
92	A_{24}	77	D_{27}	29	I_{16}	131	$STAT_1$
91	A_{25}	78	D_{28}	28	I_{17}	132	$STAT_2$
90	A_{26}	79	D_{29}	27	I_{18}	128	$SUP/\overline{US}$
89	A_{27}	80	D_{30}	26	I_{19}	4	SYSCLK
88	A_{28}	81	D_{31}	25	I_{20}	11	$\overline{TEST}$
87	A_{29}	161	$\overline{DBACK}$	24	I_{21}	150	$\overline{TRAP_0}$
86	A_{30}	143	$\overline{DBREQ}$	23	I_{22}	151	$\overline{TRAP_1}$
85	A_{31}	160	$\overline{DERR}$	22	I_{23}	6	V_{CC}
142	$\overline{BGRT}$	164	$\overline{DRDY}$	19	I_{24}	21	V_{CC}
145	$\overline{BINV}$	138	$\overline{DREQ}$	18	I_{25}	42	V_{CC}
163	$\overline{BREQ}$	134	$DREQT_0$	17	I_{26}	63	V_{CC}
1	$\overline{CDA}$	135	$DREQT_1$	16	I_{27}	83	V_{CC}
9	$CNTL_0$	5	GND	15	I_{28}	103	V_{CC}
10	$CNTL_1$	7	GND	14	I_{29}	113	V_{CC}
48	D_0	20	GND	13	I_{30}	146	V_{CC}
49	D_1	46	GND	12	I_{31}	148	V_{CC}
50	D_2	64	GND	157	$\overline{IBACK}$	156	$\overline{WARN}$

LOGIC SYMBOL



ORDERING INFORMATION
Standard Products

AMD standard products are available in several packages and operating ranges. The ordering number (Valid Combination) is formed by a combination of:
a. Device Number
b. Speed Option (if applicable)
c. Package Type
d. Temperature Range
e. Optional Processing

AM29000 -25 G C

e. OPTIONAL PROCESSING
Blank = Standard Processing
B = Burn-in

d. TEMPERATURE RANGE
C = Commercial (T_C = 0 to +85°C)

c. PACKAGE TYPE
G = 169-Lead Pin Grid Array without Heat Sink (CGX169)

b. SPEED OPTION
–33 = 33 MHz
–25 = 25 MHz
–20 = 20 MHz
–16 = 16 MHz

a. DEVICE NUMBER/DESCRIPTION
Am29000
Streamlined Instruction Processor

Valid Combinations	
AM29000-33	GC, GCB
AM29000-25	
AM29000-20	
AM29000-16	

Valid Combinations

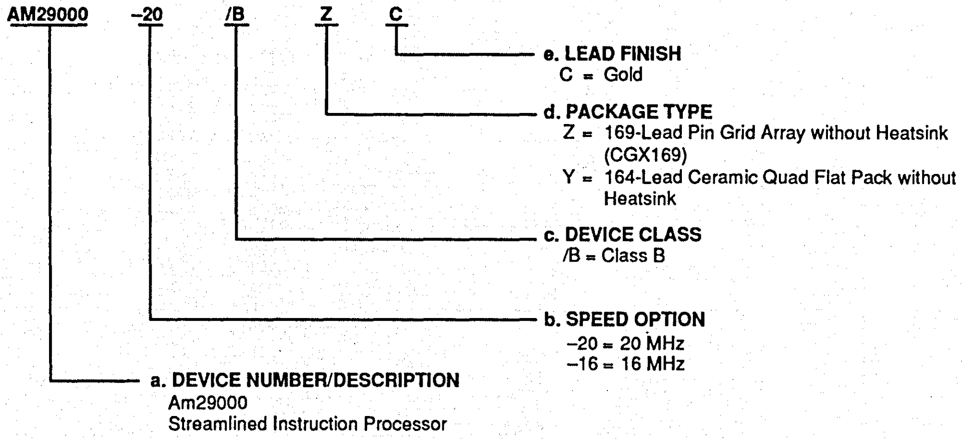
Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations, to check on newly released combinations, and to obtain additional data on AMD's standard military grade products.

ORDERING INFORMATION

APL Products

AMD products for Aerospace and Defense applications are available in several packages and operating ranges. APL (Approved Products List) products are fully compliant with MIL-STD-883C requirements. The ordering number (Valid Combination) is formed by a combination of:

a. Device Number
b. Speed Option (if applicable)
c. Device Class
d. Package Type
e. Lead Finish



AM29000 -20 /B Z C

e. LEAD FINISH
C = Gold

d. PACKAGE TYPE
Z = 169-Lead Pin Grid Array without Heatsink (CGX169)
Y = 164-Lead Ceramic Quad Flat Pack without Heatsink

c. DEVICE CLASS
/B = Class B

b. SPEED OPTION
–20 = 20 MHz
–16 = 16 MHz

a. DEVICE NUMBER/DESCRIPTION
Am29000
Streamlined Instruction Processor

Valid Combinations	
AM29000-20 AM29000-16	/BZC
AM29000-20 AM29000-16	/BYC

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations, to check on newly released combinations, and to obtain additional data on AMD's standard military grade products.

Group A Tests

Group A tests consist of Subgroups 1, 2, 3, 7, 8, 9, 10, 11.

PIN DESCRIPTION

Although certain outputs are described as being three-state or bidirectional outputs, all outputs (except MSERR) may be placed in a high-impedance state by the Test mode. The three-state and bidirectional terminology in this section is for those outputs (except SYSCLK) that are disabled when the processor grants the channel to another master.

A_{31}–A_0

Address Bus (three-state output, synchronous)

The Address Bus transfers the byte address for all accesses except burst-mode accesses. For burst-mode accesses, it transfers the address for the first access in the sequence.

$\overline{\text{BGRT}}$

Bus Grant (output, synchronous)

This output signals to an external master that the processor is relinquishing control of the channel in response to $\overline{\text{BREQ}}$.

$\overline{\text{BINV}}$

Bus Invalid (output, synchronous)

This output indicates that the address bus and related controls are invalid. It defines an idle cycle for the channel.

$\overline{\text{BREQ}}$

Bus Request (input, synchronous)

This input allows other masters to arbitrate for control of the processor channel.

$\overline{\text{CDA}}$

Coprocessor Data Accept (input, synchronous)

This signal allows the coprocessor to indicate the acceptance of operands or operation codes. For transfers to the coprocessor, the processor does not expect a $\overline{\text{DRDY}}$ response; an active level on $\overline{\text{CDA}}$ performs the function normally performed by $\overline{\text{DRDY}}$. $\overline{\text{CDA}}$ may be active whenever the coprocessor is able to accept transfers.

CNTL_1–CNTL_0

CPU Control (Input, asynchronous)

These inputs control the processor mode:

CNTL_1	CNTL_0	Mode
0	0	Load Test Instruction
0	1	Step
1	0	Halt
1	1	Normal

D_{31}–D_0

Data Bus (bidirectional, synchronous)

The Data Bus transfers data to and from the processor for load and store operations.

$\overline{\text{DBACK}}$

Data Burst Acknowledge (input, synchronous)

This input is active whenever a burst-mode data access has been established. It may be active even though no data are currently being accessed.

$\overline{\text{DBREQ}}$

Data Burst Request (three-state output, synchronous)

This signal is used to establish a burst-mode data access and to request data transfers during a burst-mode data access. $\overline{\text{DBREQ}}$ may be active even though the address bus is being used for an instruction access. This signal becomes valid late in the cycle, with respect to $\overline{\text{DREQ}}$.

$\overline{\text{DERR}}$

Data Error (input, synchronous)

This input indicates that an error occurred during the current data access. For a load, the processor ignores the content of the data bus. For a store, the access is terminated. In either case, a Data Access Exception trap occurs. The processor ignores this signal if there is no pending data access.

$\overline{\text{DRDY}}$

Data Ready (input, synchronous)

For loads, this input indicates that valid data is on the data bus. For stores, it indicates that the access is complete, and that data need no longer be driven on the data bus. The processor ignores this signal if there is no pending data access.

$\overline{\text{DREQ}}$

Data Request (three-state output, synchronous)

This signal requests a data access. When it is active, the address for the access appears on the address bus.

DREQT_1–DREQT_0

Data Request Type
(three-state output, synchronous)

These signals specify the address space of a data access, as follows (the value "x" is a "don't care"):

DREQT_1	DREQT_0	Meaning
0	0	Instruction/data memory access
0	1	Input/output access
1	x	Coprocessor transfer

An interrupt/trap vector request is indicated as a data-memory read. If required, the system can identify the vector fetch by the STAT_2–STAT_0 outputs. DREQT_1–DREQT_0 are valid only when $\overline{\text{DREQ}}$ is active.

I_{31}–I_0
Instruction Bus (Input, synchronous)

The Instruction Bus transfers instructions to the processor.

$\overline{\text{IBACK}}$
Instruction Burst Acknowledge (Input, synchronous)

This input is active whenever a burst-mode instruction access has been established. It may be active even though no instructions are currently being accessed.

$\overline{\text{IBREQ}}$
Instruction Burst Request (three-state output, synchronous)

This signal is used to establish a burst-mode instruction access and to request instruction transfers during a burst-mode instruction access. $\overline{\text{IBREQ}}$ may be active even though the address bus is being used for a data access. This signal becomes valid late in the cycle with respect to $\overline{\text{IREQ}}$.

$\overline{\text{IERR}}$
Instruction Error (Input, synchronous)

This input indicates that an error occurred during the current instruction access. The processor ignores the content of the instruction bus, and an Instruction Access Exception trap occurs if the processor attempts to execute the invalid instruction. The processor ignores this signal if there is no pending instruction access.

INCLK
Input Clock (Input)

When the processor generates the clock for the system, this is an oscillator input to the processor at twice the processor's operating frequency. In systems where the clock is not generated by the processor, this signal must be tied High or Low, except in certain master/slave configurations.

$\overline{\text{INTR}_3}$–$\overline{\text{INTR}_0}$
Interrupt Request (Input, asynchronous)

These inputs generate prioritized interrupt requests. The interrupt caused by $\overline{\text{INTR}_0}$ has the highest priority, and the interrupt caused by $\overline{\text{INTR}_3}$ has the lowest priority. The interrupt requests are masked in prioritized order by the Interrupt Mask field in the Current Processor Status Register.

$\overline{\text{IRDY}}$
Instruction Ready (Input, synchronous)

This input indicates that a valid instruction is on the instruction bus. The processor ignores this signal if there is no pending instruction access.

$\overline{\text{IREQ}}$
Instruction Request (three-state output, synchronous)

This signal requests an instruction access. When it is active, the address for the access appears on the address bus.

IREQT
Instruction Request Type (three-state output, synchronous)

This signal specifies the address space of an instruction request when $\overline{\text{IREQ}}$ is active:

IREQT	Meaning
0	Instruction/data memory access
1	Instruction read-only memory access

$\overline{\text{LOCK}}$
Lock (three-state output, synchronous)

This output allows the implementation of various channel and device interlocks. It may be active only for the duration of an access, or active for an extended period of time under control of the Lock bit in the Current Processor Status.

MPGM_1–MPGM_0
MMU Programmable (three-state output, synchronous)

These outputs reflect the value of two PGM bits in the Translation Look-Aside Buffer entry associated with the access. If no address translation is performed, these signals are both Low.

MSERR
Master/Slave Error (output, synchronous)

This output shows the result of the comparison of processor outputs with the signals provided internally to the off-chip drivers. If there is a difference for any enabled driver, this line is asserted.

OPT$_2$–OPT$_0$

Option Control
(three-state output, synchronous)

These outputs reflect the value of bits 18–16 of the load or store instruction that begins an access. Bit 18 of the instruction is reflected on OPT$_2$, bit 17 on OPT$_1$, and bit 16 on OPT$_0$.

The standard definitions of these signals (based on DREQT) are as follows (the value "x" is a "don't care"):

DREQT$_1$	DREQT$_0$	OPT$_2$	OPT$_1$	OPT$_0$	Meaning
0	x	0	0	0	Word-length access
0	x	0	0	1	Byte access
0	x	0	1	0	Half-word access
0	0	1	0	0	Instruction ROM access (as data)
0	0	1	0	1	Cache control
0	0	1	1	0	ADAPT29K accesses
	-all others-				Reserved

During an interrupt/trap vector fetch, the OPT$_2$–OPT$_0$ signals indicate a word-length access (000). Also, the system should return an entire aligned word for a read, regardless of the indicated data length.

The Am29000 does not explicitly prevent a store to the instruction ROM. OPT$_3$–OPT$_0$ are valid only when $\overline{\text{DREQ}}$ is active.

$\overline{\text{PDA}}$

Pipelined Data Access
(three-state output, synchronous)

If $\overline{\text{DREQ}}$ is not active, this output indicates that a data access is pipelined with another in-progress data access. The indicated access cannot be completed until the first access is complete. The completion of the first access is signaled by the assertion of $\overline{\text{DREQ}}$.

$\overline{\text{PEN}}$

Pipeline Enable (input, synchronous)

This signal allows devices that can support pipelined accesses (i.e., that have input latches for the address and required controls) to signal that a second access may begin while the first is being completed.

$\overline{\text{PIA}}$

Pipelined Instruction Access
(three-state output, synchronous)

If $\overline{\text{IREQ}}$ is not active, this output indicates that an instruction access is pipelined with another in-progress instruction access. The indicated access cannot be completed until the first access is complete. The completion of the first access is signaled by the assertion of $\overline{\text{IREQ}}$.

R/$\overline{\text{W}}$

Read/Write (three-state output, synchronous)

This signal indicates whether data is being transferred from the processor to the system, or from the system to the processor. R/$\overline{\text{W}}$ is valid only when the address bus is valid. R/$\overline{\text{W}}$ will be High when $\overline{\text{IREQ}}$ is active.

$\overline{\text{RESET}}$

Reset (input, asynchronous)

This input places the processor in the Reset mode.

STAT$_2$–STAT$_0$

CPU Status (output, synchronous)

These outputs indicate the state of the processor's execution stage on the previous cycle. They are encoded as follows:

STAT$_2$	STAT$_1$	STAT$_0$	Condition
0	0	0	Halt or Step Modes
0	0	1	Pipeline Hold Mode
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Wait Mode
1	0	0	Interrupt Return
1	0	1	Taking Interrupt or Trap
1	1	0	Non-sequential Instruction Fetch
1	1	1	Executing Mode

SUP/$\overline{\text{US}}$

Supervisor/User Mode
(three-state output, synchronous)

This output indicates the program mode for an access.

The processor does not relinquish the channel (in response to $\overline{\text{BREQ}}$) when $\overline{\text{LOCK}}$ is active.

SYSCLK

System Clock (bidirectional)

This is either a clock output with a frequency that is half that of INCLK, or an input from an external clock generator at the processor's operating frequency.

$\overline{\text{TEST}}$

Test Mode (input, asynchronous)

When this input is active, the processor is in Test mode. All outputs and bidirectional lines, except MSERR, are forced to the state.

$\overline{\text{TRAP}_1}$–$\overline{\text{TRAP}_0}$

Trap Request (input, asynchronous)

These inputs generate prioritized trap requests. The trap caused by $\overline{\text{TRAP}_0}$ has the highest priority. These trap requests are disabled by the DA bit of the Current Processor Status Register.

$\overline{\text{WARN}}$

Warn (Input, asynchronous, edge-sensitive)

A high-to-low transition on this input causes a non-maskable $\overline{\text{WARN}}$ trap to occur. This trap bypasses the normal trap vector fetch sequence, and is useful in situations where the vector fetch may not work (e.g., when data memory is faulty).

The following pins are not signal pins, but are named in Am29000 documentation because of their special role in the processor and system.

PWRCLK

Power Supply for SYSCLK Driver

This pin is a power supply for the SYSCLK output driver. It isolates the SYSCLK driver, and is used to determine whether or not the Am29000 generates the clock for the system. If power (+5 volts) is applied to this pin, the Am29000 generates a clock on the SYSCLK output. If this pin is grounded, the Am29000 accepts a clock generated by the system on the SYSCLK input.

PIN169

Alignment pin

In the PGA package, this pin is used to indicate proper pin-alignment of the Am29000 and is used by the ADAPT29K to communicate its presence to the system. This pin does not exist on the Am29000 in CQFP package.

FUNCTIONAL DESCRIPTION

Product Overview

The Am29000 contains a high-function execution unit, a large register file (192 locations), a Branch Target Cache (32 4-bit instruction branch targets), a memory management unit (64 entries), and a high-bandwidth, pipelined external channel with separate instruction and data buses. The flexible register file may be used as a cache for run-time variables during program execution, or as a collection of register banks allocated to separate tasks in multitasking applications.

The Am29000 provides a significant margin of performance over other processors in its class, since the majority of processor features were defined with the maximum achievable performance in mind. This section describes the features of the Am29000 from the point of view of system performance.

Cycle Time

The processor operates at a frequency of 33 MHz. The processor cycle time is a single, 30-ns clock period. The processor interface drivers can drive 80-pF loads at this frequency (for greater loads see Capacitive Output Delay table).

The Am29000 architecture and system interfaces are designed so that the processor cycle time can decrease with technology improvements.

Four-Stage Pipeline

The Am29000 utilizes a four-stage pipeline, allowing it to execute one instruction every clock cycle. The processor can complete an instruction on every cycle, even though four cycles are required from the beginning of an instruction to its completion.

At a 33-MHz operating frequency, the maximum instruction execution rate is 33 million instructions per second (MIPS). The Am29000 pipeline is designed so that the Am29000 can operate at the maximum instruction execution rate a significant portion of the time.

Pipeline interlocks are implemented by processor hardware. Except for a few special cases, it is not necessary to rearrange programs to avoid pipeline dependencies.

System Interface

The Am29000 accesses external instructions and data using three non-multiplexed buses. These buses are referred to collectively as the channel. The channel protocol minimizes the logic chains involved in a transfer, and provides a maximum transfer rate of 264 Mb/s.

Separate Address, Instruction, and Data Buses

The Am29000 incorporates two 32-bit buses for instruction and data transfers, and a third address bus that is shared between instruction and data accesses. This bus structure allows simultaneous instruction and data transfers, even though the address bus is shared. The channel achieves the performance of four separate 32-bit buses at a much-reduced pin count.

Pipelined Addresses

The Am29000 address bus is pipelined so that it can be released before an instruction or data transfer is completed. This allows a subsequent access to begin before the first has been completed, and allows the processor to have two accesses in progress simultaneously.

Support of Burst Devices and Memories

Burst-mode accesses provide high transfer rates for instructions and data at sequential addresses. For such accesses, the address of the first instruction or datum is sent, and subsequent requests for instructions or data at sequential addresses do not require additional address transfers. These instructions or data are transferred until either party involved in the transfer terminates the access.

Burst-mode accesses can occur at the rate of one access per cycle after the first address has been processed. At 33 MHz, the maximum achievable transfer bandwidth for either instructions or data is 132 Mb/s.

Burst-mode accesses may occur to input/output devices if the system design permits.

Interface to Fast Devices and Memories

The processor can be interfaced to devices and memories that complete accesses within one cycle. The channel protocol takes maximum advantage of such devices and memories by allowing data to be returned to the processor during the cycle in which the address is transmitted. This allows a full range of memory-speed tradeoffs to be made within a particular system.

Register File

An on-chip Register File containing 192 general-purpose registers allows most instruction operands to be fetched without the delay of an external access. The Register File incorporates several features that aid the retention of data required by an executing program. Because of the number of general-purpose registers, the frequency of external references for the Am29000 is significantly lower than the frequency of references in processors having only 16 or 32 registers.

Triple-port access to the Register File allows two source operands to be fetched in one cycle while a previously computed result is written. Three 32-bit internal buses prevent contention in the routing of operands. All operand fetches and result write-backs for instruction execution can be performed in a single cycle.

The registers allow efficient procedure linkage by caching a portion of a compiler's run-time stack. On the average, procedure calls and returns can be executed 5 to 10 times faster (on a cycle-by-cycle basis) than in processors that require the implementation of a run-time stack in external memory (with the attendant loading and storing of registers on procedure call and return).

The registers can contain variables, constants, addresses, and operating-system values. In multitasking applications, they can be used to hold the processor status and variables for as many as eight different tasks. A register-banking option allows the Register File to be divided into segments, which can be individually protected. In this configuration, a task switch can occur in as few as 17 cycles.

Instruction Execution

The Am29000 uses an Arithmetic/Logic Unit, a Field Shift Unit, and a Prioritizer to execute most instructions. Each of these is organized to operate on 32-bit operands and provide a 32-bit result. All operations are performed in a single cycle.

Instruction operations are overlapped with operand fetch and result write-back to the Register File. Pipeline forwarding logic detects pipeline dependencies and routes data as required, avoiding delays that might arise from these dependencies.

Branch Target Cache

In general, the Am29000 meets its instruction bandwidth requirements via instruction prefetching. However, instruction prefetching is ineffective when a branch occurs. The Am29000 therefore incorporates an on-chip Branch Target Cache to supply instructions for a branch—if this branch has been taken previously—while a new prefetch stream is established.

If branch-target instructions are in the Branch Target Cache, branches execute in a single cycle. The Branch Target Cache in the Am29000 has an average hit rate of 60%. In other words, it eliminates the branch latency for 60% of all successful branches on the average.

Branching

Branch conditions in the Am29000 are based on Boolean data contained in general-purpose registers rather than on arithmetic condition codes. Using a condition-code register for the purpose of branching—which is common in other processors—inhibits certain compiler optimizations because the condition-code register is modified by many different instructions. It is difficult for an optimizing compiler to schedule this shared use. By treating branch conditions as any other instruction operand, the Am29000 avoids this problem.

The Am29000 executes branches in a single cycle for those cases where the target of the branch is in the Branch Target Cache. The single-cycle branch is unusual for a pipelined processor, and is due to processor hardware that allows much of the branch instruction operation to be performed early in the execution of the branch. Single-cycle branching has a dramatic effect on performance, since successful branches typically represent 15% to 25% of a processor's instruction mix.

The techniques used to achieve single-cycle branching also minimize the execution time of branches in those cases where the target is not in the Branch Target Cache. To keep the pipeline operating at the maximum rate, the instruction following the branch, referred to as the delay instruction, is executed regardless of the outcome of the branch. An optimizing compiler can define a useful instruction for the delay instruction in approximately 90% of branch instructions, thereby increasing the performance of branches.

Loads and Stores

The performance degradation of load and store operations is minimized in the Am29000 by overlapping them with instruction execution, by taking advantage of pipelining, and by organizing the flow of external data onto the processor so that the impact of external accesses is minimized.

Overlapped Loads and Stores

In the Am29000, a load or store is performed concurrently with execution of instructions that do not have dependencies on the load or store operation. An optimizing compiler can schedule loads and stores in the instruction sequence so that, in most cases, data accesses are overlapped with instruction execution.

Overlapped load and store operations can achieve up to a 30% improvement in performance when data memory has a two-cycle access time. Processor hardware detects dependencies while overlapped loads and stores are being performed, so dependencies have no software implications.

The Am29000 exception restart mechanism automatically saves information required to restart any load or store until the operation is successfully completed. Thus, it allows the overlapped execution of loads and stores while properly handling address-translation exceptions.

The Am29000 data-flow organization avoids the one-cycle penalty that would result from the contention between load data and the results of overlapped instruction execution. Load data is buffered in a latch while awaiting an opportunity to be written into the register file. This opportunity is guaranteed to arise before the next load is executed. While the data is buffered in this latch, it may be used as an instruction operand in place of the destination register for the load.

Load Multiple and Store Multiple

Load Multiple and Store Multiple instructions allow the transfer of the contents of multiple registers to or from external memories or devices. This transfer can occur at a rate of one register content per cycle.

The advantage of Load Multiple and Store Multiple is best seen in task switching, register-file saving and restoring, and in block data moves. In many systems, such operations require a significant percentage of execution time.

The Load Multiple and Store Multiple sequences are interruptible so that they do not affect interrupt latency.

Forwarding of Load Data

Data that are sent to the processor at the completion of a load are forwarded directly to the appropriate execution unit if the data are required immediately by an instruction. This avoids the common one-cycle delay from bus transfer to use of data, and reduces the access latency of external data by one cycle.

Memory Management

A 64-entry Translation Look-Aside Buffer (TLB) on the Am29000 performs virtual-to-physical address translation, avoiding the cycle that would be required to transfer the virtual address to an external TLB. A number of enhancements improve the performance of address translation:

1. Pipelining—The operation of the TLB is pipelined with other processor operations.
2. Early Address Translation—Address translations for load, store, and branch instructions occur during the cycle in which these instructions are executed. This allows the physical address to be transferred externally in the next cycle.
3. Task Identifiers—Task Identifiers allow TLB entries to be matched to different processes so that TLB invalidation is not required during task switches.
4. Least-Recently Used Hardware—This hardware allows immediate selection of a TLB set to be replaced.
5. Software Reload—Software reload allows the operating system to use a page-mapping scheme that is best matched to its environment. Paged-segmented, one-level page mapping, two-level page mapping, or any other user-defined page-mapping scheme can be supported. Because Am29000 instructions execute at an average rate of nearly one instruction per cycle, software reload has a performance approaching that of hardware TLB reload.

Interrupts and Traps

When the Am29000 takes an interrupt or trap, it does not automatically save its current state information. This greatly improves the performance of temporary interruptions such as TLB reload, floating-point emulation, or other simple operating-system calls that require no saving of state information.

In cases where the processor state must be saved, the saving and restoring of state information is under the control of software. The methods and data structures used to handle interrupts—and the amount of state saved—may be tailored to the needs of a particular system.

Interrupts and traps are dispatched through a 256-entry Vector Area, which directs the processor to a routine to handle a given interrupt or trap. The Vector Area may be relocated in memory by the modification of a processor register. There may be multiple Vector Areas in the system, though only one is active at any given time.

The Vector Area is either a table of pointers to the interrupt and trap handlers, or a segment of instruction memory (possibly read-only memory) containing the handlers themselves. The choice between the two possible Vector Area definitions is determined by the cost/performance trade-offs made for a particular system.

If the Vector Area is a table of vectors in data memory, it requires only 1 kb of memory. However, this structure requires that the processor perform a vector fetch every time an interrupt or trap is taken. The vector fetch requires at least three cycles in addition to the number of cycles required for the basic memory access.

If the Vector Area is a segment of instruction memory, it requires a maximum of 64 kb of memory. The advantage of this structure is that the processor begins the execution of the interrupt or trap handler in the minimum amount of time.

Floating-Point Arithmetic Unit

The Am29027 is a double-precision, floating-point arithmetic unit for the Am29000. It can provide an order-of-magnitude performance increase over floating-point operations performed in software. It performs both single-precision and double-precision operations using IEEE and other floating-point formats. The Am29027 also supports 32- and 64-bit integer functions.

The Am29027 performs floating-point operations using combinatorial—rather than sequential—logic; therefore, operations with the Am29027 require only five Am29000 cycles. Floating-point operations may be overlapped with other processor operations. Furthermore, the Am29027 incorporates pipeline registers and eight operand registers, permitting very high throughput for certain types of operations (such as array computations).

The Am29027 attaches directly to the Am29000 using the coprocessor interface. The Am29000 can transfer two 32-bit quantities to the Am29027 in one cycle.

The Am29027 is described in detail in the Am29027 Arithmetic Accelerator Data Sheet (order# 09114) and the Am29027 Handbook (order# 11852).

ARCHITECTURE HIGHLIGHTS

This section introduces the principle architectural elements, hardware features, and system interfaces of the Am29000.

Architecture Overview

This section gives a brief description of the Am29000 from a programmer's point of view. It introduces the processor's program modes, registers, and instructions. An overview of the processor's data formats and handling is given. This section also briefly describes interrupts and traps, memory management, and the coprocessor interface. Finally, the Timer Facility and Trace Facility are introduced.

Program Modes

There are two mutually exclusive modes of program execution: the Supervisor mode and the User mode. In the Supervisor mode, executing programs have access to all processor resources. In the User mode, certain processor resources may not be accessed; any attempted access causes a trap.

Visible Registers

The Am29000 incorporates three classes of registers that are accessed and manipulated by instructions: general-purpose registers, special-purpose registers, and Translation Look-Aside Buffer (TLB) registers. (Refer to the Register Description section for greater detail of the register categories.)

General-Purpose Registers

The Am29000 has 192 general-purpose registers. With a few exceptions, general-purpose registers are not dedicated to any special use and are available for any appropriate program use.

Most processor instructions are three-address instructions. An instruction specifies any three of the 192 registers for use in instruction execution. Normally, two of these registers contain source operands for the instruction, and a third stores the result of the instruction.

The 192 registers are divided into 64 global and 128 local registers. Global registers are addressed with absolute register numbers, while local registers are addressed relative to an internal Stack Pointer.

For fast procedure calling, a portion of a compiler's runtime stack can be mapped into the local registers. Statically allocated variables, temporary values, and operating-system parameters are kept in the global registers.

The Stack Pointer for local registers is mapped to Global Register 1. The Stack Pointer is a full 32-bit virtual address for the top of the run-time stack.

The general-purpose registers may be accessed indirectly, with the register number specified by the content of a special-purpose register (see below) rather than by an instruction field. Three independent indirect register numbers are contained in three separate special-purpose registers. Indirect addressing is accomplished by specifying Global Register 0 as an instruction operand or result register. An instruction can specify an indirect register access for any or all of the source operands or result.

General-purpose registers may be partitioned into segments of 16 registers for the purpose of access protection. A register in a protected segment may be accessed only by a program executing in the Supervisor mode. An attempted access (either read or write) by a User-mode program causes a trap to occur.

Special-Purpose Registers

The Am29000 contains 27 special-purpose registers. These registers provide controls and data for certain processor functions.

Special-purpose registers are accessed by data movement only. Any special-purpose register can be written with the contents of any general-purpose register, and any general-purpose register can be written with the contents of any special-purpose register. Operations cannot be performed directly on the contents of special-purpose registers.

Some special-purpose registers are protected, and can be accessed only in the Supervisor mode. This restriction applies to both read and write accesses. An attempt by a User-mode program to access a protected register causes a trap to occur.

The protected special-purpose registers are defined as follows:

1. Vector Area Base Address—Defines the beginning of the interrupt/trap Vector Area.
2. Old Processor Status—Receives a copy of the Current Processor Status (see below) when an interrupt or trap is taken. It is later used to restore the Current Processor Status on an interrupt return.
3. Current Processor Status—Contains control information associated with the currently executing process, such as interrupt disables and the Supervisor Mode bit.
4. Configuration—Contains control information that normally varies only from system to system, and usually is set only during system initialization.
5. Channel Address—Contains the address associated with an external access, and retains the address if the access is not completed successfully. The Channel Address Register, in conjunction with the Channel Data and Channel Control registers described below, allows the restarting of unsuccessful external accesses. This

might be necessary for an access encountering a page fault in a demand-paged environment, for example.

6. Channel Data—Contains data associated with a store operation, and retains the data if the operation is not completed successfully.
7. Channel Control—Contains control information associated with a channel operation, and retains this information if the operation is not completed successfully.
8. Register Bank Protect—Restricts access of user-mode programs to specified groups of 16 registers. This facilitates register banking for multitasking applications, and protects operating system parameters kept in the global registers from corruption by user-mode programs.
9. Timer Counter—Supports real-time control and other timing-related functions.
10. Timer Reload—Maintains synchronization of the Timer Counter. It includes control bits for the Timer Facility.
11. Program Counter 0—Contains the address of the instruction being decoded when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.
12. Program Counter 1—Contains the address of the instruction being executed when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.
13. Program Counter 2—Contains the address of the instruction just completed when an interrupt or trap is taken. This address is provided for information only, and does not participate in an interrupt return.
14. MMU Configuration—Allows selection of various memory-management options, such as page size.
15. LRU Recommendation—Simplifies the reload of entries in the Translation Look-Aside Buffer (TLB) by providing information on the least recently used entry of the TLB when a TLB miss occurs.

The unprotected special-purpose registers are defined as follows:

1. Indirect Pointer C—Allows the indirect access of a general-purpose register.
2. Indirect Pointer A—Allows the indirect access of a general-purpose register.
3. Indirect Pointer B—Allows the indirect access of a general-purpose register.
4. Q—Provides additional operand bits for multiply step, divide step, and divide operations.
5. ALU Status—Contains information about the outcome of integer arithmetic and logical operations, and holds residual control for certain instruction operations.
6. Byte Pointer—Contains an index of a byte or half-word within a word. This register is also accessible via the ALU Status Register.
7. Funnel Shift Count—Provides a bit offset for the extraction of word-length fields from double-word operands. This register is also accessible via the ALU Status Register.
8. Load/Store Count Remaining—Maintains a count of the number of loads and stores remaining for Load Multiple and Store Multiple operations. The count is initialized to the total number of loads or stores to be performed before the operation is initiated. This register is also accessible via the Channel Control Register.
9. Floating-Point Environment—Controls the operation of floating-point arithmetic, such as rounding modes and exception reporting.
10. Integer Environment—Enables and disables the reporting of exceptions that occur during integer multiply and divide operations.
11. Floating-Point Status—Contains information about the outcome of floating-point operations.
12. Exception Opcode—Reports the operation code of an instruction causing a trap. This register is provided primarily for recovery from floating-point exceptions, but is also set for other instructions that cause traps.

TLB Registers

Translation Look-Aside Buffer (TLB) entries in the Am29000 Memory Management Unit are accessed via 128 TLB registers. A single TLB entry appears as two TLB registers; TLB registers are thus paired according to the corresponding TLB entry.

TLB registers are accessed by data movement only. Any TLB register can be written with the contents of any general-purpose register, and any general-purpose register can be written with the contents of any TLB register. Operations cannot be performed directly on the contents of TLB registers.

TLB registers can be accessed only in the Supervisor mode. This restriction applies to both read and write accesses. An attempt by a User-mode program to access a TLB register causes a trap to occur.

Instruction Set Overview

The three-address architecture of the Am29000 instruction set allows a compiler or assembly-language programmer to prevent the destruction of operands, and aids register allocation and operand reuse. Instruction operands may be contained in any 2 of the 192 general-purpose registers, and instruction results may be stored in any of the 192 general-purpose registers.

The compiler or assembly-language programmer has complete freedom to allocate register usage. There is no dedication of a particular register or register group to a particular class of operations. The instruction set is designed to minimize the number of side effects and implicit operations of instructions.

Most Am29000 instructions can specify an 8-bit constant as one of the source operands. Larger constants are constructed using one or two additional instructions and a general-purpose register. Relative branch instructions specify a 16-bit, signed, word offset. Absolute branches specify a 16-bit word address.

The Am29000 instruction set contains 117 instructions. These instructions are divided into nine classes:

1. Integer Arithmetic—Perform integer add, subtract, multiply, and divide operations.
2. Compare—Perform arithmetic and logical comparisons. Some instructions in this class allow the generation of a trap if the comparison condition is not met.
3. Logical—Perform a set of bit-wise Boolean operations.
4. Shift—Perform arithmetic and logical shifts, and allow the extraction of 32-bit words from 64-bit double words.
5. Data Movement—Perform movement of data fields between registers, and the movement of data to and from external devices and memories.
6. Constant—Allow the generation of large constant values in registers.
7. Floating-Point—Included for floating-point arithmetic, comparisons, and format conversions. These instructions are not currently implemented directly in processor hardware.
8. Branch—Perform program jumps and subroutine calls.
9. Miscellaneous—Perform miscellaneous control functions and operations not provided by other classes.

The Am29000 executes all instructions in a single cycle, except for interrupt returns, Load Multiple, and Store Multiple.

Figure 1 shows a complete list of Am29000 instructions, listed alphabetically by instruction mnemonic (refer to the Instruction Set section for more details).

Mnemonic	Instruction Name
ADD	Add
ADDC	Add with Carry
ADDCS	Add with Carry, Signed
ADDCU	Add with Carry, Unsigned
ADDS	Add, Signed
ADDU	Add, Unsigned
AND	AND Logical
ANDN	AND-NOT Logical
ASEQ	Assert Equal To
ASGE	Assert Greater Than or Equal To
ASGEU	Assert Greater Than or Equal To, Unsigned
ASGT	Assert Greater Than
ASGTU	Assert Greater Than, Unsigned
ASLE	Assert Less Than or Equal To
ASLEU	Assert Less Than or Equal To, Unsigned
ASLT	Assert Less Than
ASLTU	Assert Less Than, Unsigned
ASNEQ	Assert Not Equal To
CALL	Call Subroutine
CALLI	Call Subroutine, Indirect
CLASS	Classify Floating-Point Operand
CLZ	Count Leading Zeros
CONST	Constant
CONSTH	Constant, High
CONSTN	Constant, Negative
CONVERT	Convert Data Format
CPBYTE	Compare Bytes
CPEQ	Compare Equal To
CPGE	Compare Greater Than or Equal To
CPGEU	Compare Greater Than or Equal To, Unsigned
CPGT	Compare Greater Than
CPGTU	Compare Greater Than, Unsigned
CPLE	Compare Less Than or Equal To
CPLEU	Compare Less Than or Equal To, Unsigned
CPLT	Compare Less Than
CPLTU	Compare Less Than, Unsigned
CPNEQ	Compare Not Equal To
DADD	Floating-Point Add, Double-Precision
DDIV	Floating-Point Divide, Double-Precision
DEQ	Floating-Point Equal To, Double-Precision
DGE	Floating-Point Greater Than or Equal To, Double-Precision
DGT	Floating-Point Greater Than, Double-Precision
DIV	Divide Step
DIVO	Divide Initialize
DIVIDE	Integer Divide, Signed
DIVIDU	Integer Divide, Unsigned
DIVL	Divide Last Step
DIVREM	Divide Remainder
DMUL	Floating-Point Multiply, Double-Precision
DSUB	Floating-Point Subtract, Double-Precision
EMULATE	Trap to Software Emulation Routine
EXBYTE	Extract Byte
EXHW	Extract Half-Word
EXHWS	Extract Half-Word, Sign-Extended
EXTRACT	Extract Word, Bit-Aligned
FADD	Floating-Point Add, Single-Precision
FDIV	Floating-Point Divide, Single-Precision
FDMUL	Floating-Point Multiply, Single-to-Double Precision
FEQ	Floating-Point Equal To, Single-Precision
FGE	Floating-Point Greater Than or Equal To, Single-Precision

Figure 1. Am29000 Instruction Set

Mnemonic	Instruction Name
FGT	Floating-Point Greater Than, Single-Precision
FMUL	Floating-Point Multiply, Single-Precision
FSUB	Floating-Point Subtract, Single-Precision
HALT	Enter Halt Mode
INBYTE	Insert Byte
INHW	Insert Half-Word
INV	Invalidate
IRET	Interrupt Return
IRETINV	Interrupt Return and Invalidate
JMP	Jump
JMPF	Jump False
JMPFDEC	Jump False and Decrement
JMPFI	Jump False Indirect
JMPI	Jump Indirect
JMPT	Jump True
JMPTI	Jump True Indirect
LOAD	Load
LOADL	Load and Lock
LOADM	Load Multiple
LOADSET	Load and Set
MFSR	Move from Special Register
MFTLB	Move from Translation Look-Aside Buffer Register
MTSR	Move to Special Register
MTSRIM	Move to Special Register Immediate
MTTLB	Move to Translation Look-Aside Buffer Register
MUL	Multiply Step
MULL	Multiply Last Step
MULTIPLU	Integer Multiply, Unsigned
MULTIPLY	Integer Multiply, Signed
MULTM	Integer Multiply Most-Significant Bits, Signed
MULTMU	Integer Multiply Most-Significant Bits, Unsigned
MULU	Multiply Step, Unsigned
NAND	NAND Logical
NOR	NOR Logical
OR	OR Logical
SETIP	Set Indirect Pointers
SLL	Shift Left Logical
SQRT	Square Root
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
STORE	Store
STOREL	Store and Lock
STOREM	Store Multiple
SUB	Subtract
SUBC	Subtract with Carry
SUBCS	Subtract with Carry, Signed
SUBCU	Subtract with Carry, Unsigned
SUBR	Subtract Reverse
SUBRC	Subtract Reverse with Carry
SUBRCS	Subtract Reverse with Carry, Signed
SUBRCU	Subtract Reverse with Carry, Unsigned
SUBRS	Subtract Reverse, Signed
SUBRU	Subtract Reverse, Unsigned
SUBS	Subtract Signed
SUBU	Subtract Unsigned
XNOR	Exclusive-NOR Logical
XOR	Exclusive-OR Logical

Figure 1. Am29000 Instruction Set (continued)

Data Formats and Handling

This section introduces the data formats and data-manipulation mechanisms that are supported by the Am29000.

Data Types

A word is defined as 32 bits of data. A half-word consists of 16 bits, and a double word consists of 64 bits. Bytes are 8 bits in length. The Am29000 has direct support for word-integer (signed and unsigned), word-logical, word-Boolean, half-word integer (signed and unsigned), and character (signed and unsigned) data.

Other data types, such as character strings, are supported with sequences of basic instructions and/or external hardware. Single- and double-precision floating-point types are defined for the Am29000, but are not supported directly by hardware.

The format for Boolean data used by the processor is such that the Boolean values TRUE and FALSE are represented by 1 and 0, respectively, in the most-significant bit of a word.

Figure 2 illustrates the numbering conventions for data units contained in a word. Within a word, bits are numbered in increasing order from right to left, starting with the number 0 for the least-significant bit. Bytes and half-words within a word are numbered in increasing order, starting with the number 0. However, bytes and half-words may be numbered right-to-left or left-to-right, as controlled by the Configuration Register.

Note that the numbering of bits within words is strictly for notational convenience. In contrast, the numbering conventions for bytes and half-words within words affect processor operations.

External Data Accesses

External accesses move data between the processor and external devices and memories. These accesses occur only as a result of load and store instructions.

Load and store instructions move words of data to and from general-purpose registers. Each load and store instruction moves a single word. There are load and store instructions that support interlocking operations necessary for multiprocessor exclusion, synchronization, and communication.

For the movement of multiple words, Load Multiple and Store Multiple instructions move the contents of sequentially addressed external locations to or from sequentially numbered general-purpose registers. The Load Multiple and Store Multiple allow the movement of up to 192 words at a maximum rate of one word per processor cycle. The multiple load and store sequences may be interrupted, and restarted at the point of interruption.

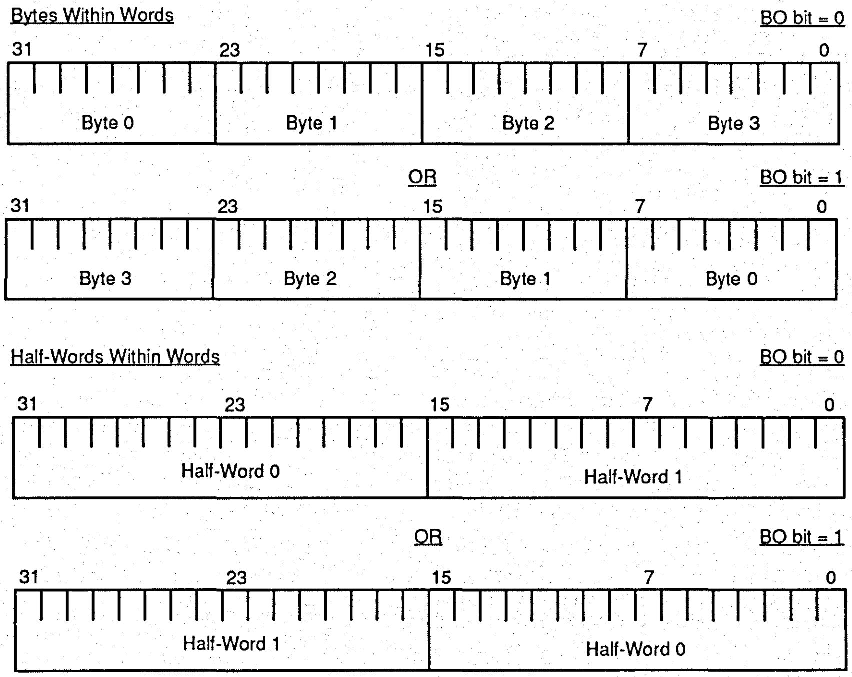


Figure 2. Data-Unit Numbering Conventions

Load and store instructions provide no mechanism for computing the address associated with the external data access. All addresses are contained in a general-purpose register at the beginning of the access, or are given by an 8-bit instruction constant. Any address computation must be performed explicitly before the load or store instruction is executed. Since address computations are expressed directly, they are exposed for compiler optimizations as any other computations are.

External data accesses are overlapped with instruction execution. Processor performance is improved if instructions that follow loads do not immediately use externally referenced data. In this manner, the time required to perform the external access is overlapped with subsequent instruction execution. Because of hardware interlocks, this concurrency has no effect on the logical behavior of an executing program.

Addressing and Alignment

External instructions and data are contained in one of four 32-bit address spaces:

1. Instruction/Data Memory
2. Input/Output
3. Coprocessor
4. Instruction Read-Only Memory (Instruction ROM)

An address in the instruction/data memory address space may be treated as virtual or physical, as determined by the Current Processor Status Register. Address translation for data accesses is enabled separately from address translation for instruction accesses. A program in the Supervisor mode may temporarily disable address translation for individual loads and stores; this permits load-real and store-real operations.

Bits contained within load and store instructions distinguish between the instruction/data memory, input/output, and coprocessor address spaces. Address translation also may determine whether an access is performed in the instruction/data memory or the input/output address space. The Current Processor Status register determines whether instruction accesses are directed to the instruction/data memory address space or to the instruction ROM address space.

The Am29000 does not support data accesses directly to the instruction ROM address space. However, this capability is possible as a system option.

All addresses are interpreted as byte addresses, although accesses are word-oriented. The number of a byte within a word is given by the two least-significant address bits. The number of a half-word within a word is given by the next-to-least-significant address bit.

Since only byte addressing is supported, it is possible that an address for the access of a word or half-word is not aligned to the desired word or half-word. For a word access, an unaligned address has a 1 in either or both of the two least-significant address bits. For a half-word access, an unaligned address has a 1 in the least-significant address bit. In many systems, address alignment can be ignored, with addresses truncated to access the word or half-word of interest. However, as a user option, the Am29000 creates a trap when a nonaligned access is attempted. The trap allows software emulation of nonaligned accesses.

In the Am29000, all instructions are 32 bits in length, and are aligned on word-address boundaries.

Byte and Half-Word Accesses

The Am29000 supports the direct external access of bytes and half-words as an option. If this option is enabled, the Am29000 selects a byte or half-word within a word on a load, and aligns it to the low-order byte or half-word of a register. On a store, the low-order byte or half-word of a register is replicated in all byte or half-word positions, so that the external memory can easily write the required byte or half-word in memory. This option requires that the external memory system be able to write individual bytes and half-words within words.

To avoid the memory-system complexity caused by writing individual bytes and half-words, the Am29000 can perform byte and half-word accesses using software alone. The Am29000 can set a byte-position indicator in the ALU Status Register as an option for load instructions, with the two least-significant bits of the address for the load. To load a byte or half-word, a word load is first performed. This load sets the byte-position indicator, and a subsequent instruction extracts the byte or half-word of interest from the accessed word. To store a byte or half-word, a load is also first performed; the byte or half-word of interest is inserted into the accessed word, and the resulting word then is stored. Even if the Am29000 is configured to perform byte and half-word accesses in hardware, this software-only technique operates correctly; this allows software to be upwardly compatible from simpler systems to more complex systems.

Interrupts and Traps

Normal program flow may be preempted by an interrupt or trap for which the processor is enabled. The effect on the processor is identical for interrupts and traps; the distinction is in the different mechanisms by which interrupts and traps are enabled. It is intended that interrupts be used for suspending current program execution and causing another program to execute, while traps are used to report errors and exceptional conditions.

The interrupt and trap mechanism supports high-speed, temporary context switching and user-defined interrupt-processing mechanisms.

Temporary Context Switching

The basic interrupt/trap mechanism of the Am29000 supports temporary context switching. During the temporary context switch, the interrupted context is held in processor registers. The interrupt or trap handler can return immediately to this context.

Temporary context switching is useful for instruction emulation, floating-point operations, TLB reload routines, and so forth. Many of its features are similar to microprogram execution; processor context does not have to be saved, interrupts are disabled for the duration of the program, and all processor resources are accessible, even if the context that was interrupted is in the User mode. The associated routine may execute from instruction/data memory or instruction ROM.

User-Defined Interrupt Processing

Since the basic interrupt/trap mechanism for the Am29000 keeps the interrupted context in the processor, dynamically nested interrupts are not supported directly. The context in the processor must be saved before another interrupt or trap can be taken.

The interrupt or trap handler executing during a temporary context switch is not required to return to the interrupted context. This routine optionally may save the interrupted context, load a new one, and return to the new context.

The implementation of the saving and restoring of contexts is completely user-defined. Thus, the context save/restore mechanism used (e.g., interrupt stack, program status word area, etc.) and the amount of context saved may be tailored to the needs of the system.

Vector Area

Interrupt and trap dispatching occur through a relocatable Vector Area, which accommodates as many as 256 interrupt and trap handling routines. Entries into the Vector Area are associated with various sources of interrupts and traps; some are predefined while others are user-defined.

The Vector Area is either a table of vectors in data memory where each vector points to the beginning of an interrupt or trap handler, or it is a segment of instruction/data memory (or instruction ROM) containing the actual routines. The latter configuration for the Vector Area yields better interrupt performance with the cost of additional memory.

Memory Management

The Am29000 incorporates a Memory Management Unit (MMU) that accepts a 32-bit virtual byte address and translates it to a 32-bit physical byte address in a single cycle. The MMU is not dedicated to any particular address-translation architecture.

Address translation in the MMU is performed by a 64-entry Translation Look-Aside Buffer (TLB), an associative table containing the most recently used address translations for the processor. If the translation for a given address cannot be performed by the TLB, a TLB miss occurs and causes a trap that allows the required translation to be placed into the TLB.

Processor hardware maintains information for each TLB line indicating which entry was least recently used; when a TLB miss occurs, this information is used to indicate the TLB entry to be replaced. Software is responsible for searching system page tables and modifying the indicated TLB entry as appropriate. This allows the page tables to be defined according to the system environment.

TLB entries are modified directly by processor instructions. A TLB entry consists of 64 bits and appears as two word-length TLB registers, which may be inspected and modified by instructions.

TLB entries are tagged with a Task Identifier field, which allows the operating system to create a unique 32-bit virtual address space for each of 256 processes. In addition, TLB entries provide support for memory protection and user-defined control information.

Coprocessor Programming

The coprocessor interface for the Am29000 allows a program to communicate with an off-chip coprocessor for performing operations not supported by processor hardware directly.

The coprocessor interface allows the program to transfer operands and operation codes to the coprocessor, and then perform other operations while the coprocessor operation is in progress. The results of the operation are read from the coprocessor by a separate transfer. The processor may transfer multiple operands to the coprocessor without retransferring operation codes or reading intermediate results. As many as 64 bits of information can be transferred to the coprocessor in a single cycle.

The Am29000 includes features that support the definition of the coprocessor as a system option. In this case, coprocessor operations are emulated by software when the coprocessor is not present in a system.

Timer Facility

The Timer Facility provides a counter for implementing a real-time clock or other software timing functions. This facility comprises two special-purpose registers: the Timer Counter Register, which decrements at a rate equal to the processor operating frequency, and the Timer Reload Register, which reinitializes the Timer Counter Register when it decrements to 0. The Timer Facility optionally may create an interrupt when the Timer Counter decrements to 0.

Trace Facility

The Trace Facility allows a debug program to emulate single-instruction stepping in a program under test. This facility allows a trap to be generated after the execution of any instruction in the program being tested.

Using the Trace Facility, the debug program can inspect and modify the state of the program at every instruction boundary. The Trace Facility is designed to work properly in the presence of normal system interrupts and traps.

FUNCTIONAL OPERATION

This section briefly describes the operation of Am29000 hardware. It introduces the processor pipeline and the three major internal functional units: the Instruction Fetch Unit, the Execution Unit, and the Memory Management Unit. Finally, the processor's operational modes are described.

Four-Stage Pipeline

The Am29000 implements a four-stage pipeline for instruction execution. The four stages are: fetch, decode, execute, and write-back. The pipeline is organized so that the effective instruction execution rate is as high as one instruction per cycle. Data forwarding and pipeline interlocks are handled by processor hardware.

Fetch Stage

During the fetch stage, the Instruction Fetch Unit determines the location of the next processor instruction and issues the instruction to the decode stage. The instruction is fetched either from the Instruction Prefetch Buffer, the Branch Target Cache, or an external instruction memory.

Decode Stage

During the decode stage, the Execution Unit decodes the instruction selected during the fetch stage and fetches and/or assembles the required operands. It also evaluates addresses for branches, loads, and stores.

Execute Stage

During the execute stage, the Execution Unit performs the operation specified by the instruction. In the case of branches, loads, and stores, the Memory Management Unit performs address translation if required.

Write-Back Stage

During the write-back stage, the results of the operation performed during the execute stage are stored. In the case of branches, loads, and stores, the physical address resulting from translation during the execute stage is transmitted to an external device or memory.

Function Organization

Figure 3 shows the Am29000 internal data-flow organization. The following sections refer to the various components on this data-flow diagram.

Instruction Fetch Unit

The Instruction Fetch Unit fetches instructions and supplies instructions to other functional units. It incorporates the Instruction Prefetch Buffer, the Branch Target Cache, and the Program Counter Unit. All components of the Instruction Fetch Unit operate during the fetch stage of the processor pipeline.

Instruction Prefetch Buffer

Most instructions executed by the Am29000 are fetched from external instruction/data memory. The processor prefetches instructions so that they are requested at least four cycles before they are required for execution.

Prefetched instructions are stored in a four-word Instruction Prefetch Buffer while awaiting execution. An instruction prefetch request occurs whenever there is a free location in this buffer (if the processor is otherwise enabled to fetch instructions). When a nonsequential instruction fetch occurs, prefetching is terminated, and then restarted for the new instruction stream.

Instruction prefetching uncouples the instruction fetch rate from the instruction access latency. For example, an instruction may be transferred to the processor two cycles after it is requested. However, as long as instructions are supplied to the processor at an average rate of one instruction per cycle, this latency has no effect on the instruction execution rate.

Branch Target Cache

The Am29000 incorporates a Branch Target Cache that contains as many as 128 instructions. The Branch Target Cache is a two-way, set-associative cache containing the first four target instructions of a number of recently taken branches. Each of the two sets in the Branch Target Cache contains 64 instructions, and the 64 instructions are further divided into 16 blocks of 4 instructions each.

The purpose of the Branch Target Cache is to provide instructions for the beginning of a nonsequential instruction-fetch sequence. This keeps the instruction pipeline full until the processor can establish a new instruction prefetch stream from the external instruction/data memory.

The processor is organized so that branch instructions can execute in a single cycle if the target instruction sequence is present in the Branch Target Cache.

Program Counter Unit

The Program Counter Unit creates and sequences addresses of instructions as they are executed by the processor.

Execution Unit

The Execution Unit executes instructions. It incorporates the Register File, the Address Unit, the Arithmetic/Logic Unit, the Field Shift Unit, and the Prioritizer. The Register File and Address Unit operate during the decode stage of the pipeline. The Arithmetic/Logic Unit, Field Shift Unit, and Prioritizer operate during the execute stage of the pipeline. The Register File operates during the write-back stage.

Register File

The general-purpose registers are implemented by a 192-location Register File. The Register File can perform two read accesses and one write access in a single cycle. Normally, two read accesses are performed during the decode-pipeline stage to fetch operands re-

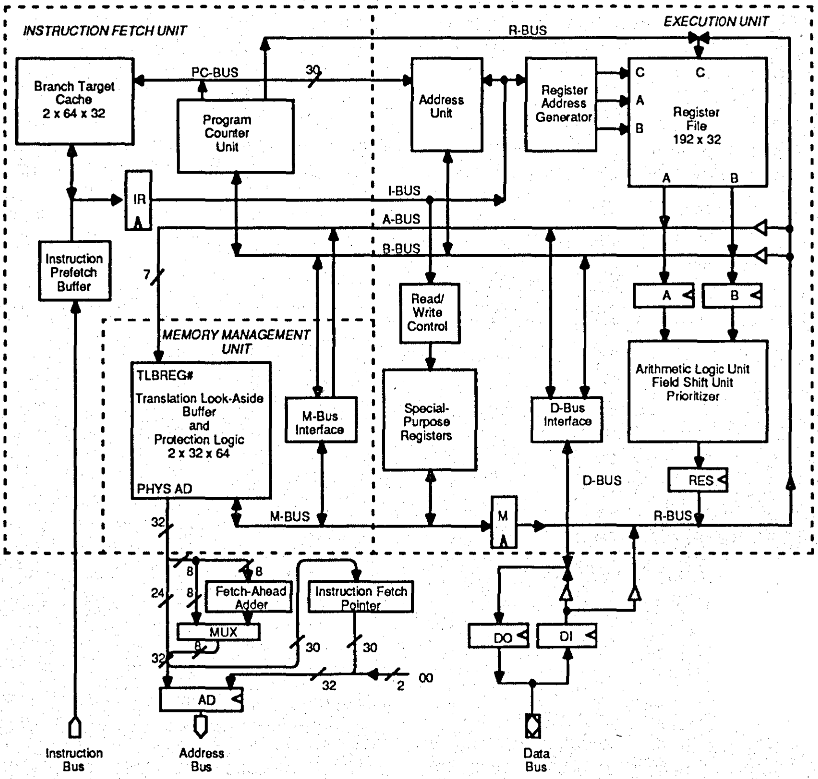


Figure 3. Am29000 Data Flow

quired by the instruction being decoded. The write access during the same cycle completes the write-back stage of a previously executed instruction.

Addressing logic associated with the Register File distinguishes between the global and local general-purpose registers, and it performs the Stack-Pointer addressing for the local registers. Register File addressing functions are performed during the decode stage.

Address Unit
The Address Unit evaluates addresses for branches, loads, and stores. It also assembles instruction-immediate data and computes addresses for Load Multiple and Store Multiple sequences.

Arithmetic/Logic Unit
The ALU performs all logical, compare, and arithmetic operations (including multiply step and divide step).

Field Shift Unit
The Field Shift Unit performs N-bit shifts. The Field Shift Unit also performs byte and half-word extract and insert operations, and it extracts words from double words.

Prioritizer
The Prioritizer provides a count of the number of leading 0 bits in a 32-bit word; this is useful for performing floating-point normalization, for example. It can also be used to implement prioritization in a multilevel interrupt handler.

Memory Management Unit

The Memory Management Unit (MMU) performs address translation and memory-protection functions for all branches, loads, and stores. The MMU operates during the execute stage of the pipeline, so the physical address that it generates is available at the beginning of the write-back stage.

All addresses for external accesses are physical addresses. MMU operation is pipelined with external accesses, so that an address translation can occur while a previous access is being completed.

Address translation is not performed for the addresses associated with instruction prefetching. Instead, these addresses are generated by an instruction prefetch pointer that is incremented by the processor. Address translation is performed only at the beginning of the prefetch sequence (as the result of a branch instruction), and when the prefetch pointer crosses a potential virtual-page boundary.

Processor Modes

The Am29000 operates in several different modes to accomplish various processor and system functions. All modes except for Pipeline Hold (see below) are under direct control of instructions and/or processor control inputs. The Pipeline Hold mode normally is determined by the relative timing between the processor and its external system for certain types of operations. The processor provides an external indication of its operational mode.

Executing

When the processor is in the Executing mode, it fetches and executes instructions as described in this manual. External accesses occur as required.

Wait

When the processor is in the Wait mode, it does not execute instructions and it performs no external accesses. The Wait mode is controlled by the Current Processor Status Register. The processor leaves this mode when an interrupt or trap for which it is enabled occurs, or when a reset occurs.

Pipeline Hold

Under certain conditions, processor pipelining might cause nonsequential instruction execution or timing-dependent results of execution. For example, the processor might attempt to execute an instruction that has not been fetched from instruction/data memory.

For such cases, pipeline-interlock hardware detects the anomalous condition and suspends processor execution until execution can proceed properly. While execution is suspended by the interlock hardware, the processor is in the Pipeline Hold mode. The processor resumes execution when the pipeline-interlock hardware determines that it is correct to do so.

Halt

The Halt mode is provided so that the processor may be placed under the control of the ADAPT29K or other hardware-development system for the purposes of hardware and software debugging. The processor enters the Halt mode as the result of instruction execution, or as the result of external controls. In the Halt mode, the processor neither fetches nor executes instructions.

Step

The Step mode allows the ADAPT29K or other hardware-development system to step through processor pipeline operation on a stage-by-stage basis. The Step mode is nearly identical to the Halt mode, except that it enables the processor to enter the Executing mode while the pipeline advances by one stage.

Load Test Instruction

The Load Test Instruction mode permits the ADAPT29K or other hardware-development system to access data contained in the processor or system. This is accomplished by allowing the ADAPT29K to supply the processor with instructions, instead of having the processor fetch instructions from instruction/data memory. The Load Test Instruction mode is defined so that, once the processor has completed the execution of instructions provided by the ADAPT29K, it may resume the execution of its normal instruction sequence.

Test

The Test mode facilitates testing of hardware associated with the processor by disabling processor outputs so that they may be driven directly by test hardware. The Test mode also allows the addition of a second processor to a system to monitor the outputs of the first and to signal detected errors.

Reset

The Reset mode provides initialization of certain processor registers and control state. This is used for power-on reset, for eliminating unrecoverable error conditions, and for supporting certain hardware debugging functions.

System Interface

This section briefly describes the features of the Am29000 that allow it to be connected to other system components.

The two major interfaces of the Am29000, introduced in this section, are the channel and the Test/Development interfaces. The other topics briefly described here are clock generation, master/slave checking, and coprocessor attachment.

Channel

The Am29000 channel consists of the following 32-bit buses and related controls:

1. An Instruction Bus, which transfers instructions into the processor
2. A Data Bus, which transfers data to and from the processor
3. An Address Bus, which provides addresses for both instruction and data accesses. The address bus also is used to transfer data to a coprocessor.

The channel performs accesses and data transfers to all external devices and memories, including instruction/data memories, instruction caches, instruction read-only memories, data caches, input/output devices, bus converters, and coprocessors.

The channel defines three different access protocols: simple, pipelined, and burst-mode. For simple accesses, the Am29000 holds the address valid throughout the entire access. This is appropriate for high-speed devices that can complete an access in one cycle, and for low-cost devices that are accessed infrequently (such as read-only memories containing initialization routines). Pipelined and burst-mode accesses provide high performance with other types of devices and memories.

For pipelined accesses, the address transfer is uncoupled from the corresponding data or instruction transfer. After transmitting an address for a request, the processor may transmit one more address before receiving the reply to the first request. This allows address transfer and decoding to be overlapped with another access.

On the other hand, burst-mode accesses eliminate the address-transfer cycle completely. Burst-mode accesses are defined so that once an address is transferred for a given access, subsequent accesses to sequentially increasing addresses may occur without retransfer of the address. The burst may be terminated at any time by either the processor or responding device.

The Am29000 determines whether an access is simple, pipelined, or burst-mode on a transfer-by-transfer (i.e., generally device-by-device) basis. However, an access that begins as a simple access may be converted to a pipelined or burst-mode access at any time during the transfer. This relaxes the timing constraints on the channel-protocol implementation, since addressed devices do not have to respond immediately to a pipelined or burst-mode request.

Except for the shared address bus, the channel maintains a strict division between instruction and data accesses. In the most common situation, the system supplies the processor with instructions using burst-mode accesses, with instruction addresses transmitted to the system only when a branch occurs. Data accesses can occur simultaneously without interfering with instruction transfer.

The Am29000 contains arbitration logic to support other masters on the channel. A single external master can arbitrate directly for the channel, while multiple masters may arbitrate using a daisy chain or other method that requires no additional arbitration logic. However, to increase arbitration performance in a multiple-master configuration, an external channel arbiter should be used. This arbiter works in conjunction with the processor's arbitration logic.

Test/Development Interface

The Am29000 supports the attachment of the ADAPT29K or other hardware-development system. This attachment is made directly to the processor in the system under development, without the removal of the processor from the system. The Test/Development Interface makes it possible for the hardware-development system to gain control over the Am29000, and inspect and modify its internal state (e.g., general-purpose register contents, TLB entries, etc.). In addition, the Am29000 may be used to access other system devices and memories on behalf of the hardware-development system.

The Test/Development Interface is made up of controls and status signals provided on the Am29000, as well as the instruction and data buses. The Halt, Step, Reset, and Load Test Instruction modes allow the hardware-development system to control the operation of the Am29000. The hardware-development system may supply the processor with instructions on the instruction bus using the load test instruction mode. The internal processor state can be inspected and modified via the data bus.

Clocks

The Am29000 generates and distributes a system clock at its operating frequency. This clock is specially designed to reduce skews between the system clock and the processor's internal clocks. The internal clock-generation circuitry requires a single-phase oscillator signal at twice the processor operating frequency.

For systems in which processor-generated clocks are not appropriate, the Am29000 also can accept a clock from an external clock generator.

The processor decides between these two clocking arrangements based on whether the power supply to the clock-output driver (PWRCLK) is tied to +5 volts or to Ground.

Master/Slave Operation

Each Am29000 output has associated logic that compares the signal on the output with the signal that the processor is providing internally to the output driver. The processor signals situations where the output of any enabled driver does not agree with its input.

For a single processor, the output comparison detects short circuits in output signals, but does not detect open circuits. It is possible to connect a second processor in parallel with the first, where the second processor has its outputs disabled due to the Test mode. The second processor detects open-circuit signals, as well as provides a check of the outputs of the first processor.

Coprocessor Attachment

A coprocessor for the Am29000 attaches directly to the processor channel. However, this attachment has features that are different from those of other channel devices. The coprocessor interface is designed to support a high operand transfer rate and to support the overlap of coprocessor operations with other processor operations, including other external accesses.

The coprocessor is assigned a special address space on the channel. This permits the transfer of operands and other information on the address bus without interfering with normal addressing functions. Since both the address bus and data bus are used for data transfer, the Am29000 can transfer 64 bits of information to the coprocessor in one cycle.

Program Modes

All system-protection features of the Am29000 are based on two mutually exclusive program modes: the Supervisor mode and the User mode. Memory protection in the Memory Management Unit is also based on the Supervisor and User modes (see Memory Management section).

Supervisor Mode

The processor is in the Supervisor mode whenever the Supervisor Mode (SM) bit of the Current Processor Status Register (see Register Description section) is 1. In this mode, executing programs have access to all processor resources.

During the address cycle of a channel request, the Supervisor mode is indicated by the SUP/$\overline{\text{US}}$ output being High.

User Mode

The processor is in the User mode whenever the SM bit in the Current Processor Status Register is 0. In this mode, any of the following actions by an executing program causes a Protection Violation trap to occur:

1. An attempted access of any TLB entry.
2. An attempted access of any general-purpose register for which a bit in the Register Bank Protect Register is 1.
3. An attempted execution of a load or store instruction for which the PA bit is 1, or for which the UA bit is 1. (The attempted execution of a translated load or store for which the AS bit is 1 also causes a Protection Violation trap. However, this trap occurs regardless of whether or not the processor is in the User mode.)
4. An attempted execution of one of the following instructions: Interrupt Return, Interrupt Return and Invalidate, Invalidate, or Halt. However, a hardware-development system such as the ADAPT29K can disable protection checking for the Halt instruction, so this instruction may be used to implement instruction breakpoints in User-mode programs.
5. An attempted access of one of the following registers: Vector Area Base Address, Old Processor Status, Current Processor Status, Configuration, Channel Address, Channel Data, Channel Control, Register Bank Protect, Timer Counter, Timer Reload, Program Counter 0, Program Counter 1, Program Counter 2, MMU Configuration, or LRU Recommendation.
6. An attempted execution of an assert or Emulate instruction that specifies a vector number between 0 and 63, inclusive.

Devices and memories on the channel also can implement protection and generate traps based on the value of the SM bit. During the address cycle of a channel request, the User mode is indicated by the SUP/$\overline{\text{US}}$ output being Low.

REGISTER DESCRIPTION

The Am29000 has three classes of registers that are accessible by instructions. These are general-purpose registers, special-purpose registers, and Translation Look-Aside Buffer (TLB) registers. Any operation available in the Am29000 can be performed on the general-purpose registers, while special-purpose registers and TLB registers are accessed only by explicit data movement to or from general-purpose registers. Various protection mechanisms prevent the access of some of these registers by User-mode programs.

General-Purpose Registers

The Am29000 incorporates 192 general-purpose registers. The organization of the general-purpose registers is diagrammed in Figure 4.

General-purpose registers hold the following types of operands for program use:

1. 32-bit data addresses
2. 32-bit signed or unsigned integers
3. 32-bit branch-target addresses
4. 32-bit logical bit strings
5. 8-bit signed or unsigned characters
6. 16-bit signed or unsigned integers
7. word-length Booleans
8. single-precision floating-point numbers
9. double-precision floating-point numbers (in two register locations)

Because a large number of general-purpose registers are provided, a large amount of frequently used data can be kept on-chip, where access time is fastest.

Am29000 instructions can specify two general-purpose registers for source operands, and one general-purpose register for storing the instruction result. These registers are specified by three 8-bit instruction fields containing register numbers. A register may be specified directly by the instruction, or indirectly by one of three special-purpose registers.

Register Addressing

The general-purpose registers are partitioned into 64 global registers and 128 local registers, differentiated by the most-significant bit of the register number. The distinction between global and local registers is the result of register-addressing considerations.

The following terminology is used to describe the addressing of general-purpose registers:

1. Register number—this is a software-level number for a general-purpose register. For example, this is the number contained in an instruction field. Register numbers range from 0 to 255.
2. Global register number—this is a software-level number for a global register. Global register numbers range from 0 to 127.
3. Local register number—this is a software-level number for a local register. Local register numbers range from 0 to 127.
4. Absolute register number—this is a hardware-level number used to select a general-purpose register in the Register File. Absolute register numbers range from 0 to 255.

Global Registers

When the most-significant bit of a register number is 0, a global register is selected. The seven least-significant bits of the register number give the global register number. For global registers, the absolute register number is equivalent to the register number.

Global Registers 2 through 63 are unimplemented. An attempt to access these registers yields unpredictable results; however, they may be protected from User-mode access by the Register Bank Protect Register.

The register numbers associated with Global Registers 0 and 1 have special meaning. The number for Global Register 0 specifies that an indirect pointer is to be used as the source of the register number; there is an indirect pointer for each of the instruction operand/result registers. Global Register 1 contains the Stack Pointer, which is used in the addressing of local registers as explained below.

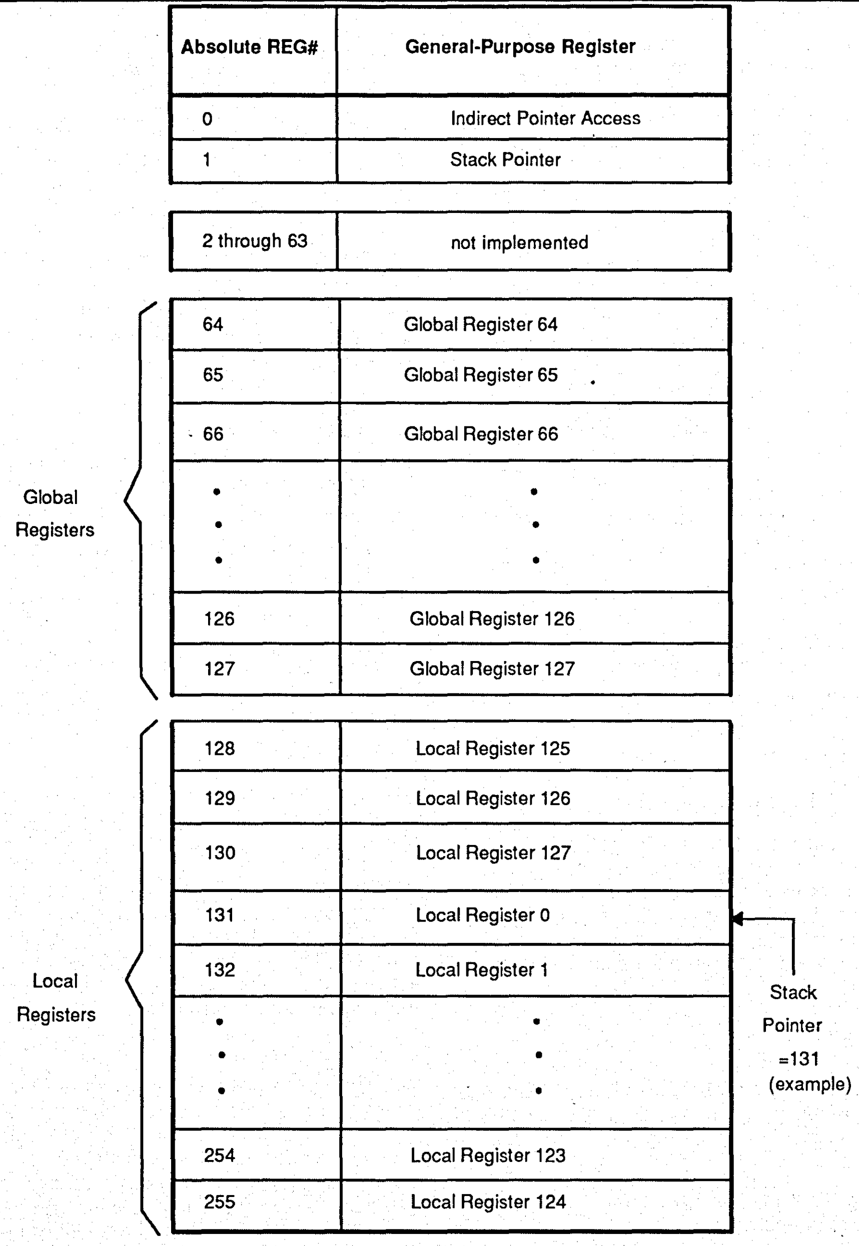
Local Register Stack Pointer

The Stack Pointer is a 32-bit register that may be an operand of an instruction as any other general-purpose register. However, a shadow copy of Global Register 1 is maintained by processor hardware to be used in local register addressing. This shadow copy is set only with the results of Arithmetic and Logical instructions. If the Stack Pointer is set with the result of any other instruction class, local registers cannot be accessed predictably until the Stack Pointer is set once again with an Arithmetic or Logical instruction.

Local Registers

When the most-significant bit of a register number is 1, a local register is selected. The seven least-significant bits of the register number give the local-register number. For local registers, the absolute register number is obtained by adding the local register number to bits 8–2 of the Stack Pointer and truncating the result to seven bits; the most-significant bit of the original register number is unchanged (i.e., it remains a 1).

The Stack Pointer addition applied to local register numbers provides a limited form of base-plus-offset addressing within the local registers. The Stack Pointer contains the 32-bit base address. This assists run-time storage management of variables for dynamically nested procedures.



Absolute REG#	General-Purpose Register
0	Indirect Pointer Access
1	Stack Pointer
2 through 63	not implemented
64	Global Register 64
65	Global Register 65
66	Global Register 66
• • •	• • •
126	Global Register 126
127	Global Register 127
128	Local Register 125
129	Local Register 126
130	Local Register 127
131	Local Register 0
132	Local Register 1
• • •	• • •
254	Local Register 123
255	Local Register 124

Figure 4. General-Purpose Register Organization

Register Banking

For the purpose of access restriction, the general-purpose registers are divided into register banks. Register banks consist of 16 registers (except for Bank 0, which contains Unimplemented Registers 2 through 15) and are partitioned according to absolute register numbers, as shown in Figure 5.

The Register Bank Protect Register contains 16 protection bits, where each bit controls User-mode accesses

Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (unimplemented)
1	16 through 31	Bank 1 (unimplemented)
2	32 through 47	Bank 2 (unimplemented)
3	48 through 63	Bank 3 (unimplemented)
4	64 through 79	Bank 4
5	80 through 95	Bank 5
6	96 through 111	Bank 6
7	112 through 127	Bank 7
8	128 through 143	Bank 8
9	144 through 159	Bank 9
10	160 through 175	Bank 10
11	176 through 191	Bank 11
12	192 through 207	Bank 12
13	208 through 223	Bank 13
14	224 through 239	Bank 14
15	240 through 255	Bank 15

Figure 5. Register Bank Organization

(read or write) to a bank of registers. Bits 0–15 of the Register Bank Protect Register protect Register Banks 0 through 15, respectively.

When a bit in the Register Bank Protect Register is 1 and a register in the corresponding bank is specified as an operand register or result register by a User-mode instruction, a Protection Violation trap occurs. Note that protection is based on absolute register numbers; in the case of local registers, Stack-Pointer addition is performed before protection checking.

When the processor is in Supervisor mode, the Register Bank Protect Register has no effect on general-purpose register accesses.

Indirect Accesses

Specification of Global Register 0 as an instruction-operand register or result register causes an indirect access to the general-purpose registers. In this case, the absolute register number is provided by an indirect pointer contained in a special-purpose register.

Each of the three possible registers for instruction execution has an associated 8-bit indirect pointer. Indirect register numbers can be selected independently for each of the three operands. Since the indirect pointers contain absolute register numbers, the number in an indirect pointer is not added to the Stack Pointer when local registers are selected.

The indirect pointers are set by the Move To Special Register, Floating-Point, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, DIVIDU, SETIP, and EMULATE instructions.

For a Move To Special Register instruction, an indirect pointer is set with bits 9–2 of the 32-bit source operand. This provides consistency between the addressing of words in general-purpose registers and the addressing of words in external devices or memories. A modification of an indirect pointer using a Move To Special Register has a delayed effect on the addressing of general-purpose registers.

For the remaining instructions, all three indirect pointers are set, simultaneously, with the absolute register numbers derived from the register numbers specified by the instruction. For any local registers selected by the instruction, the Stack-Pointer addition is applied to the register numbers before the indirect pointers are set.

Register numbers stored into the indirect pointers are checked for bank-protection violations—except when an indirect pointer is set by a Move-To-Special-Register instruction—at the time that the indirect pointers are set.

Special-Purpose Registers

The Am29000 contains 27 special-purpose registers. The organization of the special-purpose registers is shown in Figure 6.

Special-purpose registers provide controls and data for certain processor operations. Some special-purpose registers are updated dynamically by the processor, independent of software controls. Because of this, a read of a special-purpose register following a write does not necessarily get the data that was written.

Some special-purpose registers have fields that are reserved for future processor implementations. When a special-purpose register is read, a bit in a reserved field is read as a 0. An attempt to write a reserved bit with a 1 has no effect; however, this should be avoided because of upward-compatibility considerations.

The special-purpose registers are accessed by explicit data movement only. Instructions that move data to or from a special-purpose register specify the special-purpose register by an 8-bit field containing a special-purpose register number. Register numbers are specified directly by instructions.

An attempted read of an unimplemented special-purpose register yields an unpredictable value. An attempted write of an unimplemented special-purpose register has no effect; however, this should be avoided, because of upward-compatibility considerations.

The special-purpose registers are partitioned into protected and unprotected registers. Special-purpose registers numbered 0–127 and 160–255 are protected (note that not all of these are implemented). Special-purpose registers numbered 128–159 are unprotected (again, not all are implemented).

Protected special-purpose registers numbered 0–127 are accessible only by programs executing in the Supervisor mode. An attempted read or write of a protected special-purpose register by a User-mode program causes a Protection Violation trap to occur. Protected special-purpose registers numbered 160–255 are not accessible by programs in either the User mode or the Supervisor mode. These register numbers identify virtual registers in the floating-point architecture.

The Floating-Point Environment Register, Integer Environment Register, Floating-Point Status Register, and Exception Opcode Register are not implemented in processor hardware. These registers are implemented via a virtual floating-point interface provided on the Am29000.

Unprotected special-purpose registers are accessible by programs executing in both the User and Supervisor modes.

Vector Area Base Address (Register 0)

This protected special-purpose register (see Figure 7) specifies the beginning address of the interrupt/trap Vector Area. The Vector Area is either a table of 256 vectors that points to interrupt and trap handling routines, or a segment of 256 64-instruction blocks that directly contains the interrupt and trap handling routines.

The organization of the Vector Area is determined by the Vector Fetch (VF) bit of the Configuration Register. If the VF bit is 1 when an interrupt or trap is taken, the vector number for the interrupt or trap (see Interrupts and Traps section) replaces bits 9–2 of the value in the Vector Area Base Address Register to generate the physical address for a vector contained in instruction/data memory.

If the VF bit is 0, the vector number replaces bits 15–8 of the value in the Vector Area Base Address Register to generate the physical address of the first instruction of the interrupt or trap handler. The instruction fetch for this instruction is directed either to instruction memory or instruction read-only memory, as determined by the ROM Vector Area (RV) bit of the Configuration Register.

Bits 31–16: Vector Area Base (VAB)—The VAB field gives the beginning address of the Vector Area. This address is constrained to begin on a 64-kb address-boundary in instruction data memory or instruction read-only memory.

Register Number	Protected Registers	Mnemonic
0	Vector Area Base Address	VTB
1	Old Processor Status	OPS
2	Current Processor Status	CPS
3	Configuration	CFG
4	Channel Address	CHA
5	Channel Data	CHD
6	Channel Control	CHC
7	Register Bank Protect	RBP
8	Timer Counter	TMC
9	Timer Reload	TMR
10	Program Counter 0	PC0
11	Program Counter 1	PC1
12	Program Counter 2	PC2
13	MMU Configuration	MMUC
14	LRU Recommendation	LRU

	Unprotected Registers	
128	Indirect Pointer C	IPC
129	Indirect Pointer A	IPA
130	Indirect Pointer B	IPB
131	Q	Q
132	ALU Status	SR
133	Byte Pointer	BPR
134	Funnel Shift Count	FCR
135	Load/Store Count Remaining	MC
...		
160	Floating-Point Environment	FPE
161	Integer Environment	INTE
162	Floating-Point Status	FPS
...		
164	Exception Opcode	EXOP

Figure 6. Special-Purpose Registers

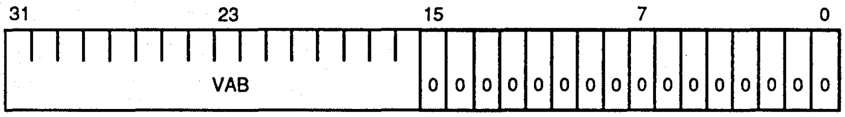


Figure 7. Vector Area Base Address Register

Bits 15–0: Zeros—These bits force the alignment of the Vector Area.

Old Processor Status (Register 1)

This protected special-purpose register has the same format as the Current Processor Status described below. The Old Processor Status stores a copy of the Current Processor Status when an interrupt or trap is taken. This is required since the Current Processor Status will be modified to reflect the status of the interrupt/trap handler.

During an interrupt return, the Old Processor Status is copied into the Current Processor Status. This allows the Current Processor Status to be set as required for the routine that is the target of the interrupt return.

Current Processor Status (Register 2)

This protected special-purpose register (see Figure 8) controls the behavior of the processor and its ability to recognize exceptional events.

Bits 31–16: reserved.

Bit 15: Coprocessor Active (CA)—The CA bit is set and reset under the control of load and store instructions that transfer information to and from a coprocessor. This bit indicates that the coprocessor is performing an operation at the time that an interrupt or trap is taken. This notifies the interrupt or trap handler that the coprocessor contains state information to be preserved. Note that this notification occurs because the CA bit of the Old Processor Status is 1 in this case, not because of the value of the CA bit of the Current Processor Status.

Bit 14: Interrupt Pending (IP)—This bit allows software to detect the presence of external interrupts while they are disabled. The IP bit is set if one or more of the external signals $\overline{INTR_3}$–$\overline{INTR_0}$ is active, but the processor is disabled from taking the resulting interrupt due to the value of the DA, DI, or IM bits. If all external interrupt signals subsequently are deasserted while still disabled, the IP bit is reset.

Bits 13–12: Trace Enable, Trace Pending (TE, TP)—The TE and TP bits implement a software-controlled, instruction single-step facility. Single stepping is not implemented directly, but rather emulated by trap sequences controlled by these bits. The value of the TE bit is copied to the TP bit whenever an instruction execution is completed. When the TP bit is 1, a Trace trap occurs.

Bit 11: Trap Unaligned Access (TU)—The TU bit enables checking of address alignment for external data-memory accesses. When this bit is 1, an Unaligned Access trap occurs if the processor either generates an address for an external word that is not aligned on a word address boundary (i.e., either of the least-significant two bits is 1), or generates an address for an external half-word that is not aligned on a half-word address boundary (i.e., the least-significant address bit is 1). When the TU bit is 0, data-memory address alignment is ignored.

Alignment is ignored for input/output accesses and coprocessor transfers. The alignment of instruction addresses is also ignored (unaligned instruction addresses can be generated only by indirect jumps). Interrupt/trap vector addresses always are aligned properly.

Bit 10: Freeze (FZ)—The FZ bit prevents certain registers from being updated during interrupt and trap processing, except by explicit data movement. The affected registers are: Channel Address, Channel Data, Channel Control, Program Counter 0, Program Counter 1, Program Counter 2, and the ALU Status Register.

When the FZ bit is 1, these registers hold their values. An affected register can be changed only by a Move To Special Register instruction. When the FZ bit is 0, there is no effect on these registers, and they are updated by

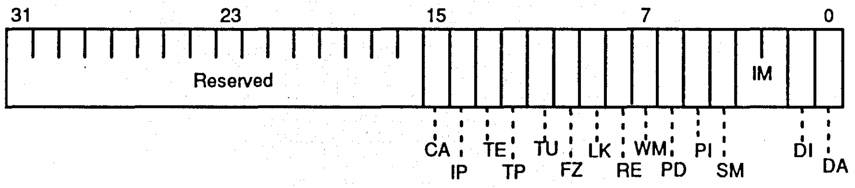


Figure 8. Current Processor Status Register

processor instruction execution as described in this manual.

The FZ bit is set whenever an interrupt or trap is taken, holding critical state in the processor so that it is not modified unintentionally by the interrupt or trap handler.

Bit 9: Lock (LK)—The LK bit controls the value of the $\overline{\text{LOCK}}$ external signal. If the LK bit is 1, the $\overline{\text{LOCK}}$ signal is active. If the LK bit is 0, the $\overline{\text{LOCK}}$ signal is controlled by the execution of the instructions Load and Set, Load and Lock, and Store and Lock. This bit is provided for the implementation of multiprocessor synchronization protocols.

Bit 8: ROM Enable (RE)—The RE bit enables instruction fetching from external instruction read-only memory (ROM). When this bit is 1, the IREQT signal directs all instruction requests to ROM. Instructions that are fetched from ROM are subject to capture and reuse by the Branch Target Cache when it is enabled; the Branch Target Cache distinguishes between instructions from ROM and those from non-ROM storage. When this bit is 0, off-chip requests for instructions are directed to instruction/data memory.

Bit 7: WAIT Mode (WM)—The WM bit places the processor in the Wait mode. When this bit is 1, the processor performs no operations. The Wait mode is reset by an interrupt or trap for which the processor is enabled, or by the Reset mode.

Bit 6: Physical Addressing/Data (PD)—The PD bit determines whether address translation is performed for load or store operations. Address translation is performed for an access only when this bit is 0, and the Physical Address (PA) bit in the load or store instruction causing the access is also 0.

Bit 5: Physical Addressing/Instructions (PI)—The PI bit determines whether address translation is performed for external instruction accesses. Address translation is performed only when this bit is 0.

Bit 4: Supervisor Mode (SM)—The SM bit protects certain processor context, such as protected special-purpose registers. When this bit is 1, the processor is in the Supervisor mode, and access to all processor context is allowed. When this bit is 0, the processor is in the User mode, and access to protected processor context is not allowed; an attempt to access (either read or write) protected processor context causes a Protection Violation trap.

For an external access, the User Access (UA) bit in the load or store instruction also controls access to protected processor context. When the UA bit is 1, the Memory Management Unit and channel perform the access as though the program causing the access was in User mode.

Bits 3–2: Interrupt Mask (IM)—The IM field is an encoding of the processor priority with respect to external interrupts. The interpretation of the interrupt mask is specified by the following table:

IM Value	Result
0 0	$\overline{\text{INTR}_0}$ enabled
0 1	$\overline{\text{INTR}_1}$–$\overline{\text{INTR}_0}$ enabled
1 0	$\overline{\text{INTR}_2}$–$\overline{\text{INTR}_0}$ enabled
1 1	$\overline{\text{INTR}_3}$–$\overline{\text{INTR}_0}$ enabled

Bit 1: Disable Interrupts (DI)—The DI bit prevents the processor from being interrupted by external interrupt requests $\overline{\text{INTR}_3}$–$\overline{\text{INTR}_0}$. When this bit is 1, the processor ignores all external interrupts. However, note that traps (both internal and external), Timer interrupts, and Trace traps will be taken. When this bit is 0, the processor will take any interrupt enabled by the IM field, unless the DA bit is 1.

Bit 0: Disable all Interrupts and Traps (DA)—The DA bit prevents the processor from taking any interrupts and most traps. When this bit is 1, the processor ignores interrupts and traps, except for the $\overline{\text{WARN}}$, Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps. When this bit is 0, all traps will be taken, and interrupts will be taken if otherwise enabled.

Configuration (Register 3)

This protected special-purpose register (see Figure 9) controls certain processor and system options. Most fields normally are modified only during system initialization. The Configuration Register definition follows.

Bits 31–24: Processor Release Level (PRL)—The PRL field is an 8-bit, read-only identification number that specifies the processor version.

Bits 23–6: reserved.

Bit 5: Data Width Enable (DW)—The DW bit enables and disables byte and half-word external accesses. If the DW bit is 0, byte and half-word accesses are not per-

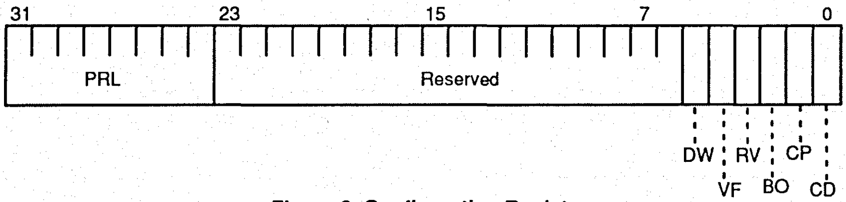


Figure 9. Configuration Register

formed in hardware, and these accesses must be emulated by software. If the DW bit is 1, byte and half-word accesses are performed by hardware: this requires that external devices and memories be able to write individual bytes and half-words within a word.

Bit 4: Vector Fetch (VF)—The VF bit determines the structure of the interrupt/trap Vector Area. If this bit is 1, the Vector Area is defined as a block of 256 vectors that specify the beginning addresses of the interrupt and trap handling routines. If the VF bit is 0, the Vector Area is a segment of 256 64-instruction blocks that contain the actual routines.

Bit 3: ROM Vector Area (RV)—If the VF bit is 0, the RV bit specifies whether the Vector Area is contained in instruction memory (RV = 0) or instruction read-only memory (RV = 1). The value of the RV bit is irrelevant if the VF bit is 1.

Bit 2: Byte Order (BO)—The BO bit determines the ordering of bytes and half-words within words. If the BO bit is 0, bytes and half-words are numbered left-to-right within a word. If the BO bit is 1, bytes and half-words are numbered right-to-left.

Bit 1: Coprocessor Present (CP)—The CP bit indicates the presence of a coprocessor that may be used by the processor. If this bit is 1, it enables the execution of load and store instructions that have a Coprocessor Enable (CE) bit of 1. If the CP bit is 0 and the processor attempts to execute a load or store instruction with a CE bit of 1, a Coprocessor Not Present trap occurs. This feature may be used to emulate coprocessor operations as well as to protect the state of a coprocessor shared between multiple processes.

Bit 0: Branch Target Cache Disable (CD)—The CD bit determines whether or not the Branch Target Cache is used for nonsequential instruction references. When this bit is 1, all instruction references are directed to external instruction memory or instruction ROM, and the Branch Target Cache is not used. When this bit is 0, the targets of nonsequential instruction fetches are stored in the Branch Target Cache and reused. The value of the CD bit does not take effect until the execution of the next branch instruction.

Channel Address (Register 4)

This protected special-purpose register (Figure 10) is used to report exceptions during external accesses or coprocessor transfers. It also is used to restart interrupted Load Multiple and Store Multiple operations, and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Address Register is updated on the execution of every load or store instruction, and on every load or store in a Load Multiple or Store Multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1.

Bits 31–0: Channel Address (CHA)—This field contains the address of the current channel transaction (if the FZ bit of the Current Processor Status Register is 0). For external data accesses, the address is virtual if address translation was enabled for the access, or physical if translation was disabled. For transfers to the coprocessor, the CHA field contains data transferred to the coprocessor.

Channel Data (Register 5)

This protected special-purpose register (Figure 11) is used to report exceptions during external accesses or coprocessor transfers. It is also used to restart the first store of an interrupted Store Multiple operation and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Data Register is updated on the execution of every load or store instruction, and on every load or store in a Load Multiple or Store Multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1. When the Channel Data Register is updated for a load operation, the resulting value is unpredictable.

Bits 31–0: Channel Data (CHD)—This field contains the data (if any) associated with the current channel

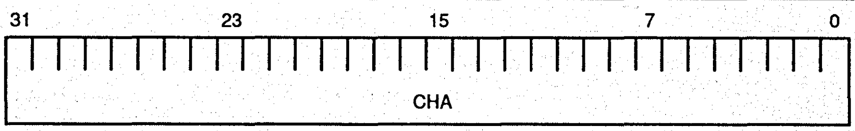


Figure 10. Channel Address Register

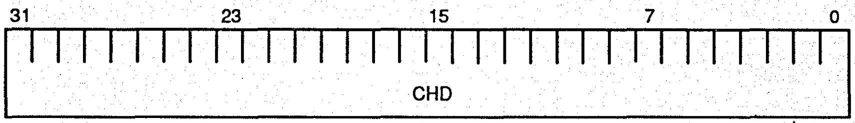


Figure 11. Channel Data Register

transaction (if the FZ bit of the Current Processor Status Register is 0). If the current channel transaction is not a store or a transfer to the coprocessor, the value of this field is irrelevant.

Channel Control (Register 6)

This protected special-purpose register (Figure 12) is used to report exceptions during external accesses or coprocessor transfers. It also is used to restart interrupted Load Multiple and Store Multiple operations, and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Control Register is updated on the execution of every load or store instruction, and on every load or store in a Load Multiple or Store Multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1.

Bits 31–24—These bits are a direct copy of bits 23–16 from the load or store instruction that started the current channel transaction.

Bits 23–16: Load/Store Count Remaining (CR)—The CR field indicates the remaining number of transfers for a Load Multiple or Store Multiple operation that encountered an exception or was interrupted before completion. This number is zero-based; for example, a value of 28 in this field indicates that 29 transfers remain to be completed. If the fault or interrupt occurs on the last transaction, the CR field contains a value of 0 and the ML bit is 1 (see below).

Bit 15: Load/Store (LS)—The LS bit is 0 if the channel transaction is a store operation, and 1 if it is a load operation.

Bit 14: Multiple Operation (ML)—The ML bit is 1 if the current channel transaction is a partially complete Load Multiple or Store Multiple operation; otherwise it is 0.

Bit 13: Set (ST)—The ST bit is 1 if the current channel transaction is for a Load and Set instruction; otherwise it is 0.

Bit 12: Lock Active (LA)—The LA bit is 1 if the current channel transaction is for a Load and Lock or Store and Lock instruction; otherwise it is 0. Note that this bit is not set as the result of the Lock (LK) bit in the Current Processor Status Register.

Bit 11: reserved.

Bit 10: Transaction Faulted (TF)—The TF bit indicates that the current channel transaction was not complete due to some exceptional circumstance. This bit is set only for exceptions reported via the $\overline{\text{DERR}}$ input, and it causes a Data Access Exception or Coprocessor Exception trap to occur (depending on the value of the CE bit) when it is 1.

The TF bit allows the proper sequencing of externally reported errors that get preempted by higher-priority traps; it is reset by software that handles the resulting trap.

Bits 9–2: Target Register (TR)—The TR field indicates the absolute register number of data operand for the current transaction (either a load target or store data source). Since the register number in this field is absolute, it reflects the Stack-Pointer addition when the indicated register is a local register.

Bit 1: Not Needed (NN)—The NN bit indicates that, even though the Channel Address, Channel Data, and Channel Control registers contain a valid representation of an uncompleted load operation, the data requested is not needed. This situation arises when a load instruction is overlapped with an instruction that writes the load target register.

Bit 0: Contents Valid (CV)—The CV bit indicates that the contents of the Channel Address, Channel Data, and Channel Control registers are valid.

Register Bank Protect (Register 7)

This protected special-purpose register (Figure 13) protects banks of general-purpose registers from User-mode program accesses.

The general-purpose registers are partitioned into 16 banks of 16 registers each (except that Bank 0 contains 14 registers). The banks are organized as shown in Figure 4.

Bits 31–16: reserved.

Bits 15–0: Bank 15 through Bank 0 Protection Bits (B15–B0)—In the Register Bank Protect Register, each bit is associated with a particular bank of registers and the bit number gives the associated bank number (e.g., B11 determines the protection for Bank 11).

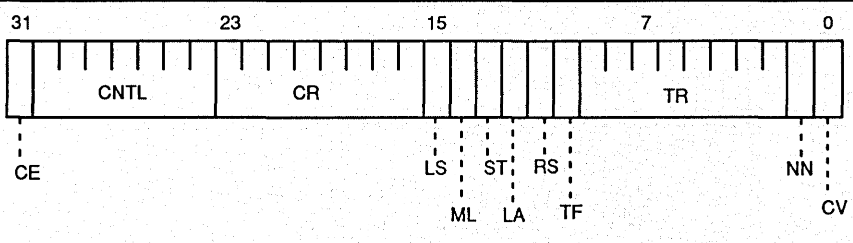


Figure 12. Channel Control Register

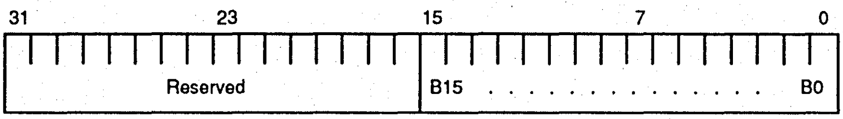


Figure 13. Register Bank Protect Register

When a protection bit is 1, the corresponding bank is protected from access by programs executing in the User mode. A Protection Violation trap occurs when a User-mode program attempts to access (either read or write) a register in a protected bank. When a bit in this register is 0, the corresponding bank is available to programs executing in the User mode.

Supervisor-mode programs are not affected by the Register Bank Protect Register.

Register protection is based on absolute register numbers. For local registers, the protection checking is performed after the Stack-Pointer addition is performed.

Timer Counter (Register 8)

This protected special-purpose register (Figure 14) contains the counter for the Timer Facility.

Bits 31–24: reserved.

Bits 23–0: Timer Count Value (TCV)—The 24-bit TCV field decrements by one on each processor clock. When the TCV field decrements to 0, it is reloaded with the content of the Timer Reload Value field in the Timer Reload Register. At this time, the Interrupt bit in the Timer Reload Register is set.

Timer Reload (Register 9)

This protected special-purpose register (Figure 15) maintains synchronization of the Timer Counter Register, enables Timer interrupts, and maintains Timer Facility status information.

Bits 31–27: reserved.

Bit 26: Overflow (OV)—The OV bit indicates that a Timer interrupt occurred before a previous Timer interrupt was serviced. It is set if the Interrupt (IN) bit is 1 (see below) when the Timer Count Value (TCV) field of the Timer Counter Register decrements to 0. In this case, a Timer interrupt caused by the IN bit has not been serviced when another interrupt is created.

Bit 25: Interrupt (IN)—The IN bit is set whenever the TCV field decrements to 0. If this bit is 1 and the IE bit is also 1, a Timer interrupt occurs. Note that the IN bit is set when the TCV field decrements to 0, regardless of the value of the IE bit. The IN bit is reset by software that handles the Timer interrupt.

The TCV field is zero-based with respect to the Timer interrupt interval; for example, a value of 28 in the TCV field causes the IN bit to be set in the 29th subsequent processor cycle. The reason for this is that the TCV field is 0 for a complete cycle before the IN bit is set.

Bit 24: Interrupt Enable (IE)—When the IE bit is 1, the Timer interrupt is enabled, and the Timer interrupt occurs whenever the IN bit is 1. When this bit is 0, the Timer interrupt is disabled. Note that Timer interrupts

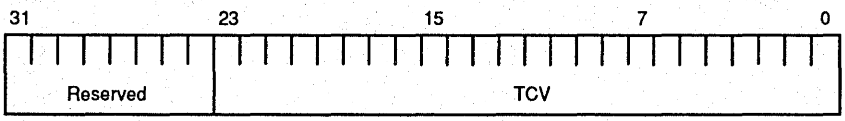


Figure 14. Timer Counter Register

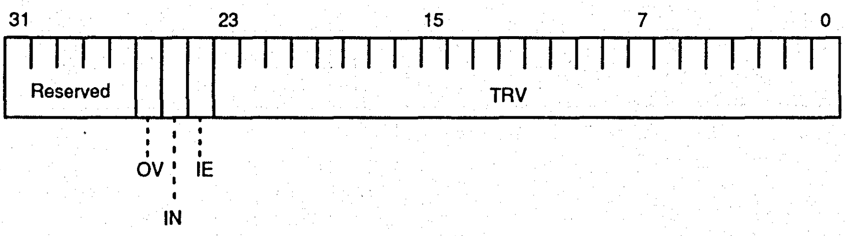


Figure 15. Timer Reload Register

may be disabled by the DA bit of the Current Processor Status Register regardless of the value of the IE bit.

Bits 23–0: Timer Reload Value (TRV)—The value of this field is written into the Timer Count Value (TCV) field of the Timer Counter Register when the TCV field decrements to 0.

Program Counter 0 (Register 10)

This protected special-purpose register (Figure 16) is used on an interrupt return to restart the instruction that was in the decode stage when the original interrupt or trap was taken.

Bits 31–2: Program Counter 0 (PC0)—This field captures the word address of an instruction as it enters the decode stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC0 holds its value.

When an interrupt or trap is taken, the PC0 field contains the word address of the instruction in the decode stage; the interrupt or trap has prevented this instruction from executing. The processor uses the PC0 field to restart this instruction on an interrupt return.

Bits 1–0: Zeros—These bits are 0 since instruction addresses are always word-aligned.

Program Counter 1 (Register 11)

This protected special-purpose register (Figure 17) is used on an interrupt return to restart the instruction that was in the execute stage when the original interrupt or trap was taken.

Bits 31–2: Program Counter 1 (PC1)—This field captures the word address of an instruction as it enters the execute stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC1 holds its value.

When an interrupt or trap is taken, the PC1 field contains the word address of the instruction in the execute stage; the interrupt or trap has prevented this instruction from completing execution. The processor uses the PC1 field to restart this instruction on an interrupt return.

Bits 1–0: Zeros—These bits are 0 since instruction addresses are always word-aligned.

Program Counter 2 (Register 12)

This protected special-purpose register (Figure 18) reports the address of certain instructions causing traps.

Bits 31–2: Program Counter 2 (PC2)—This field captures the word address of an instruction as it enters the write-back stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC2 holds its value.

When an interrupt or trap is taken, the PC2 field contains the word address of the instruction in the write-back stage. In certain cases, PC2 contains the address of the instruction causing a trap. The PC2 field is used to report the address of this instruction, and has no other use in the processor.

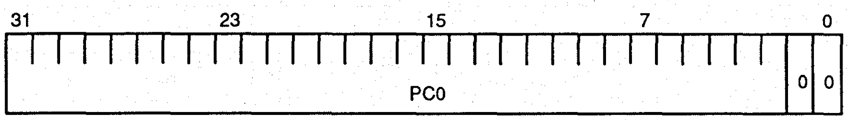


Figure 16. Program Counter 0 Register

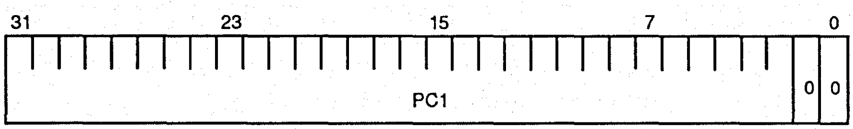


Figure 17. Program Counter 1 Register

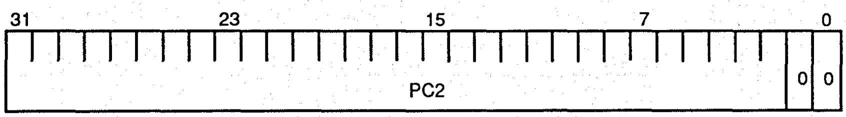


Figure 18. Program Counter 2 Register

Bits 1–0: Zeros—These bits are 0 since instruction addresses are always word-aligned.

MMU Configuration (Register 13)

This protected special-purpose register (Figure 19) specifies parameters associated with the Memory Management Unit (MMU).

Bits 31–10: reserved.

Bits 9–8: Page Size (PS)—The PS field specifies the page size for address translation. The page size affects translation as discussed in the Memory Management section. The PS field has a delayed effect on address translation. At least one cycle of delay must separate an instruction that sets the PS field and an instruction that performs address translation. The PS field is encoded as follows:

PS	Page Size
0 0	1 kb
0 1	2 kb
1 0	4 kb
1 1	8 kb

Bits 7–0: Process Identifier (PID)—For translated User-mode loads and stores, this 8-bit field is compared to Task Identifier (TID) fields in Translation Look-Aside Buffer entries when address translation is performed. For the address translation to be valid, the PID field must match the TID field in an entry. This allows a separate 32-bit virtual-address space to be allocated to each active User-mode process (within the limit of 255 such processes). Translated Supervisor-mode loads and stores use a fixed process identifier of 0, and require that the TID field be 0 for successful translation.

LRU Recommendation (Register 14)

This protected special-purpose register (Figure 20) assists Translation Look-Aside Buffer (TLB) reloading by indicating the least recently used TLB entry in the required replacement line.

Bits 31–7: reserved.

Bits 6–1: Least Recently Used Entry (LRU)—The LRU field is updated whenever a TLB miss occurs during an address translation. It gives the TLB register number of the TLB entry selected for replacement. The LRU field also is updated whenever a memory-protection violation occurs; however, it has no interpretation in this case.

Bit 0: Zero—The appended 0 serves to identify Word 0 of the TLB entry.

Indirect Pointer C (Register 128)

This unprotected special-purpose register (Figure 21) provides the RC-operand register number when an instruction RC field has the value 0 (i.e., when Global Register 0 is specified).

Bits 31–10: reserved.

Bits 9–2: Indirect Pointer C (IPC)—The 8-bit IPC field contains an absolute register number for a general-purpose register. This number directly selects a register

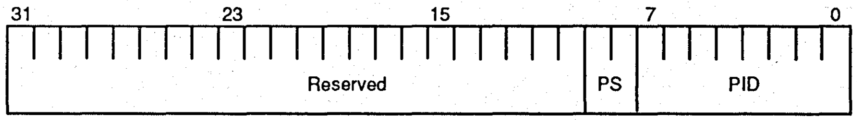


Figure 19. MMU Configuration Register

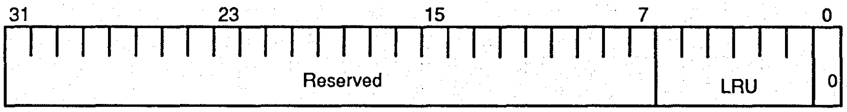


Figure 20. LRU Recommendation Register

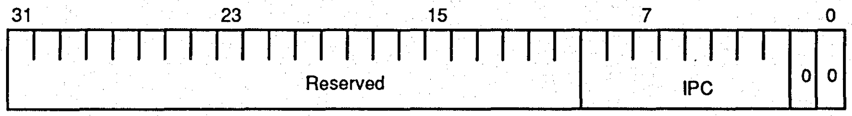


Figure 21. Indirect Pointer C Register

(Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPC field is aligned for compatibility with word addresses.

Indirect Pointer A (Register 129)

This unprotected special-purpose register (Figure 22) provides the RA-operand register number when an instruction RA field has the value 0 (i.e., when Global Register 0 is specified).

Bits 31–10: reserved.

Bits 9–2: Indirect Pointer A (IPA)—The 8-bit IPA field contains an absolute register number for either a general-purpose register or a local register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPA field is aligned for compatibility with word addresses.

Indirect Pointer B (Register 130)

This unprotected special-purpose register (Figure 23) provides the RB-operand register number when an instruction RB field has the value 0 (i.e., when Global Register 0 is specified).

Bits 31–10: reserved.

Bits 9–2: Indirect Pointer B (IPB)—The 8-bit IPB field contains an absolute register number for a general-purpose register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPB field is aligned for compatibility with word addresses.

Q (Register 131)

The Q Register is an unprotected special-purpose register (Figure 24).

Bits 31–0: Quotient/Multiplier (Q)—During a sequence of divide steps, this field holds the low-order bits of the dividend; it contains the quotient at the end of the divide. During a sequence of multiply steps, this field holds the multiplier; it contains the low-order bits of the result at the end of the multiply.

For an integer divide instruction, the Q field contains the high-order bits of the dividend at the beginning of the instruction, and contains the remainder upon completion of the instruction.

ALU Status (Register 132)

This unprotected special-purpose register (Figure 25) holds information about the outcome of Arithmetic/Logic Unit (ALU) operations as well as control for certain operations performed by the Execution Unit.

Bits 31–12: reserved.

Bit 11: Divide Flag (DF)—The DF bit is used by the instructions that implement division. This bit is set at the end of the division instructions either to 1 or to the complement of the 33rd bit of the ALU. When a Divide Step instruction is executed, the DF bit then determines whether an addition or subtraction operation is performed by the ALU.

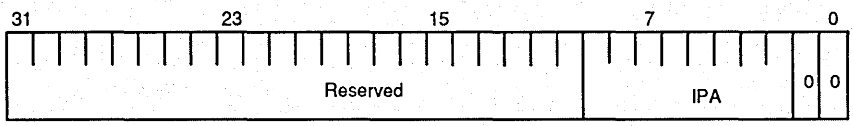


Figure 22. Indirect Pointer A Register

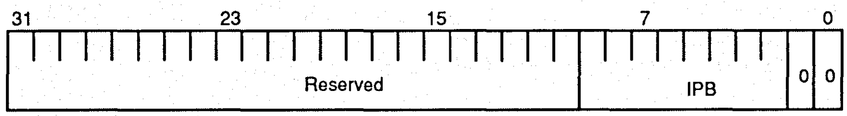


Figure 23. Indirect Pointer B Register

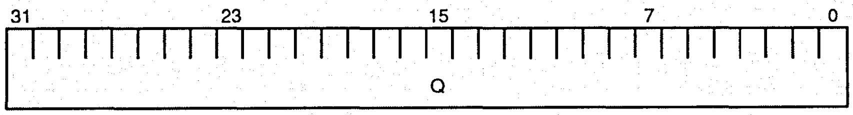


Figure 24. Q Register

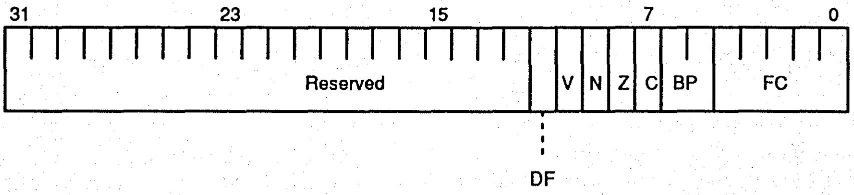


Figure 25. ALU Status Register

Bit 10: Overflow (V)—The V bit indicates that the result of a signed, twos-complement ALU operation required more than 32 bits to represent the result correctly. The value of this bit is determined by exclusive ORing the ALU carry-out with the carry-in to the most-significant bit for signed, twos-complement operations. This bit is not used for any special purpose in the processor, and is provided for information only.

Bit 9: Negative (N)—The N bit is set with the value of the most-significant bit of the result of an arithmetic or logical operation. If twos-complement overflow occurs, the N bit does not reflect the true sign of the result. This bit is used in divide operations.

Bit 8: Zero (Z)—The Z bit indicates that the result of an arithmetic or logical operation is 0. This bit is not used for any special purpose in the processor, and is provided for information only.

Bit 7: Carry (C)—The C bit stores the carry-out of the ALU for arithmetic operations. It is used by the add-with-carry and subtract-with-carry instructions to generate the carry into the Arithmetic/Logic Unit.

Bits 6–5: Byte Pointer (BP)—The BP field holds a 2-bit pointer to a byte within a word. It is used by Insert Byte and Extract Byte instructions. The exact mapping of the pointer value to the byte position depends on the value of the Byte Order (BO) bit in the Configuration Register.

The most-significant bit of the BP field is used to determine the position of a half-word within a word for the Insert Half-Word, Extract Half-Word, and Extract Half-Word, Sign-Extended instructions. The exact mapping of the most-significant bit to the half-word position depends on the value of the BO bit in the Configuration Register.

The BP field is set by a Move To Special Register instruction with either the ALU Status Register or the Byte Pointer Register as the destination. It is also set by a load or store instruction if the Set Byte Pointer (SB) bit in the instruction is 1. A load or store sets the BP field either with the two least-significant bits of the address (if the DW bit of the Configuration Register is 0) or with the complement of the Byte Order bit of the Configuration Register (if DW is 1).

Bits 4–0: Funnel Shift Count (FC)—The FC field contains a 5-bit shift count for the Funnel Shifter. The Funnel Shifter concatenates two source operands into a single 64-bit operand and extracts a 32-bit result from this 64-bit operand; the FC field specifies the number of bit positions from the most-significant bit of the 64-bit operand to the most-significant bit of the 32-bit result. The FC field is used by the Extract instruction.

The FC field is set by a Move To Special Register instruction with either the ALU Status Register or the Funnel Shift Count Register as the destination.

Byte Pointer (Register 133)

This unprotected special-purpose register (Figure 26) provides an alternate access to the BP field in the ALU Status Register.

Bits 31–2: Zeros.

Bits 1–0: Byte Pointer (BP)—This field allows a program to change the BP field without affecting other fields in the ALU Status Register.

Funnel Shift Count (Register 134)

This unprotected special-purpose register (Figure 27) provides an alternate access to the FC field in the ALU Status Register.

Bits 31–5: Zeros.

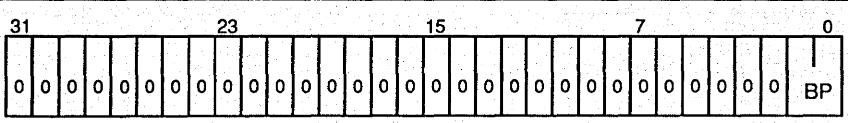


Figure 26. Byte Pointer

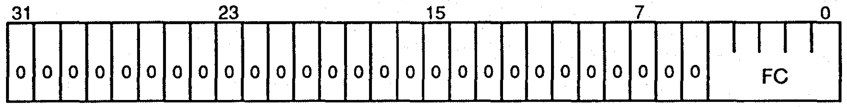


Figure 27. Funnel Shift Count

Bits 4–0: Funnel Shift Count (FC)—This field allows a program to change the FC field without affecting other fields in the ALU Status Register.

Load/Store Count Remaining (Register 135)

This unprotected special-purpose register (Figure 28) provides alternate access to the CR field in the Channel Control Register.

Bits 31–8: Zeros.

Bits 7–0: Load/Store Count Remaining (CR)—This field allows a program to change the CR field without affecting other fields in the Channel Control Register, and is used to initialize the value before a Load Multiple or Store Multiple instruction is executed.

Floating-Point Environment (Register 160)

This unprotected special-purpose register (Figure 29) contains control bits that affect the execution of floating-point operations.

Bits 31–9: reserved.

Bit 8: Fast Float Select (FF)—The FF bit being 1 enables fast floating-point operations, in which certain requirements of the IEEE floating-point specification are not met. This improves the performance of certain operations by sacrificing conformance to the IEEE specification.

Bits 7–6: Floating-Point Round Mode (FRM)—This field specifies the default mode used to round the results of floating-point operations, as follows:

FRM1–0	Round Mode
0 0	Round to nearest
0 1	Round to $-\infty$
1 0	Round to $+\infty$
1 1	Round to zero

Bit 5: Floating-Point Divide-By-Zero Mask (DM)—If the DM bit is 0, a Floating-Point Exception trap occurs when the divisor of a floating-point division operation is zero and the dividend is a non-zero, finite number. If the DM bit is 1, a Floating-Point Exception trap does not occur for divide-by-zero.

Bit 4: Floating-Point Inexact Result Mask (XM)—If the XM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is not equal to the infinitely precise result. If the XM bit is 1, a Floating-Point Exception trap does not occur for an inexact result.

Bit 3: Floating-Point Underflow Mask (UM)—If the UM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is too small to be expressed in the destination format. If the UM bit is 1, a Floating-Point Exception trap does not occur for underflow.

Bit 2: Floating-Point Overflow Mask (VM)—If the VM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is too large to be expressed in the destination format. If the VM bit is 1, a

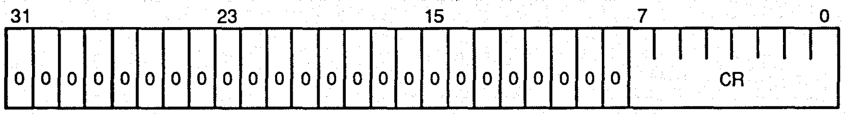


Figure 28. Load/Store Count Remaining

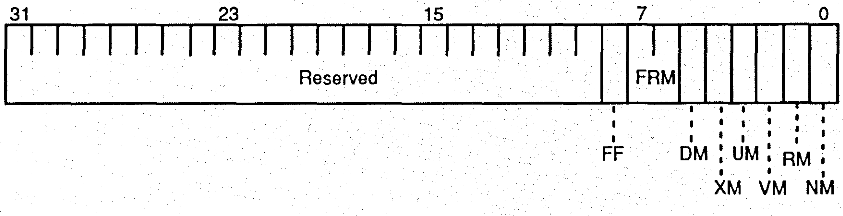


Figure 29. Floating-Point Environment

Floating-Point Exception trap does not occur for overflow.

Bit 1: Floating-Point Reserved Operand Mask (RM)—If the RM bit is 0, a Floating-Point Exception trap occurs when one or more input operands to a floating-point operation is a reserved value, or when the result of a floating-point operation is a reserved value. If the RM bit is 1, a Floating-Point Exception trap does not occur for reserved operands.

Bit 0: Floating-Point Invalid Operation Mask (NM)—If the NM bit is 0, a Floating-Point Exception trap occurs when the input operands to a floating-point operation produce an indeterminate result (e.g., ∞ times 0). If the NM bit is 1, a Floating-Point Exception trap does not occur for invalid operations.

Integer Environment (Register 161)

This unprotected special-purpose register (Figure 30) contains control bits that affect the execution of integer operations.

Bits 31–2: reserved.

Bit 1: Integer Division Overflow Mask (DO)—If the DO bit is 0, an Out of Range trap occurs when overflow of a signed or unsigned 32-bit result occurs during DIVIDE or DIVIDU instructions, respectively. If the DO bit is 1, an Out of Range trap does not occur for overflow during integer divide operations.

The DIVIDE and DIVIDU instructions always cause an Out of Range trap upon division by 0, regardless of the value of the DO bit.

Bit 0: Integer Multiplication Overflow Exception Mask (MO)—If the MO bit is 0, an Out of Range trap occurs when overflow of a signed or unsigned 32-bit result occurs during MULTIPLY or MULTIPLU instructions, respectively. If the DO bit is 1, an Out of Range trap does not occur for overflow during integer multiply operations.

Floating-Point Status (Register 162)

This unprotected special-purpose register (Figure 31) contains status bits indicating the outcome of floating-point operations. The bits of the Floating-Point Status Register are divided into two groups of status bits. The bits in each group correspond to the causes of Floating-Point Exception traps that are enabled and disabled by bits 5–0 of the Floating-Point Environment Register.

The first group of status bits (bits 13–8) are trap status bits that report the cause of a Floating-Point Exception trap. The trap status bits are set only when a Floating-Point Exception trap occurs, and indicate all conditions that apply to the trapping operation. All other operations leave the status bits unchanged. A trap status bit is set regardless of the state of the corresponding mask bit of the Floating-Point Environment Register, except that at least one of the mask bits must be 0 for the trap to occur. When a Floating-Point Exception trap occurs, all trap status bits not relevant to the trapping operation are reset.

The second group of status bits (bits 5–0) are sticky status bits that, once set, remain set until explicitly cleared by a Move to Special Register (MTSR) or Move to Special Register Immediate (MTSRIM) instruction. A sticky status bit is set only when a floating-point exception is detected and the corresponding mask bit of the Floating-Point Environment Register is 1. That is, the sticky status bit is set only if the corresponding cause of a Floating-Point Exception trap is disabled. Normally, this means that sticky status bits are not set when a Floating-Point Exception trap is taken. However, if

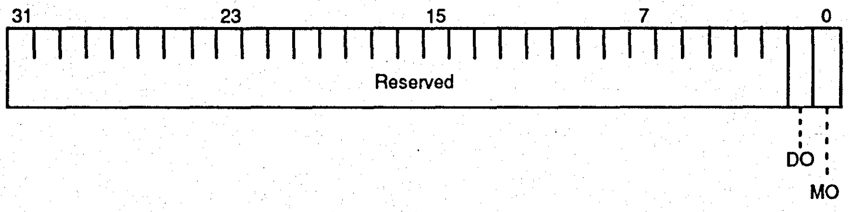


Figure 30. Integer Environment

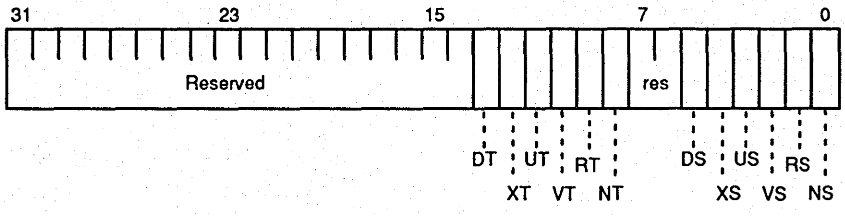


Figure 31. Floating-Point Status

multiple exceptions are detected, a sticky status bit corresponding to a masked exception may still be set if a Floating-Point Exception trap occurs for an unmasked exception.

Bits 31–14: reserved.

Bit 13: Floating-Point Divide-By-Zero Trap (DT)—The DT bit is set when a Floating-Point Exception trap occurs, and the associated floating-point operation is a divide with a zero divisor and a non-zero, finite dividend. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 12: Floating-Point Inexact Result Trap (XT)—The XT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is not equal to the infinitely precise result. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 11: Floating-Point Underflow Trap (UT)—The UT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is too small to be expressed in the destination format. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 10: Floating-Point Overflow Trap (VT)—The VT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is too large to be expressed in the destination format. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 9: Floating-Point Reserved Operand Trap (RT)—The RT bit is set when a Floating-Point Exception trap occurs, and either one or more input operands to the associated floating-point operation is a reserved value or the result of this floating-point operation is a reserved value. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 8: Floating-Point Invalid Operation Trap (NT)—The NT bit is set when a Floating-Point Exception trap occurs, and the input operands to the associated floating-point operation produce an indeterminate result. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bits 7–6: reserved.

Bit 5: Floating-Point Divide-By-Zero Sticky (DS)—The DS bit is set when the DM bit of the Floating-Point Environment Register is 1, the divisor of a floating-point division operation is a 0, and the dividend is a non-zero, finite number.

Bit 4: Floating-Point Inexact Result Sticky (XS)—The XS bit is set when the XM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is not equal to the infinitely precise result.

Bit 3: Floating-Point Underflow Sticky (US)—The US bit is set when the UM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is too small to be expressed in the destination format.

Bit 2: Floating-Point Overflow Sticky (VS)—The VS bit is set when the VM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is too large to be expressed in the destination format.

Bit 1: Floating-Point Reserved Operand Sticky (RS)—The RS bit is set when the RM bit of the Floating-Point Environment Register is 1, and either one or more input operands to a floating-point operation is a reserved value or the result of a floating-point operation is a reserved value.

Bit 0: Floating-Point Invalid Operation Sticky (NS)—The NS bit is set when the NM bit of the Floating-Point Environment Register is 1, and the input operands to a floating-point operation produce an indeterminate result.

Exception Opcode (Register 164)

This unprotected special-purpose register (Figure 32) reports the operation code (opcode) of an instruction causing a trap. It is provided primarily for recovery from floating-point exceptions, but reports the opcode of any trapping instruction.

Bits 31–8: reserved.

Bits 7–0: Instruction Opcode (IOP)—This field captures the opcode of an instruction causing a trap as a result of instruction execution; the opcode is captured as the instruction enters the write-back stage of the processor pipeline. Instructions that do not trap as a consequence of execution do not modify the IOP field.

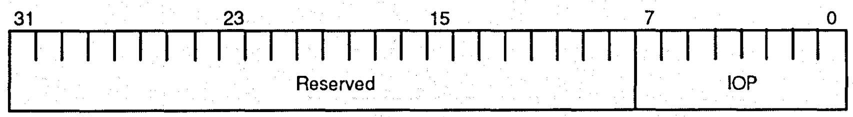


Figure 32. Exception Opcode

TLB Registers

The Am29000 contains 128 Translation Look-Aside Buffer (TLB) registers. The organization of the TLB registers is shown in Figure 33.

The TLB registers comprise the TLB entries, and are provided so that programs may inspect and alter TLB entries. This allows the loading, invalidation, saving, and restoring of TLB entries.

TLB registers have fields that are reserved for future processor implementations. When a TLB register is read, a bit in a reserved field is read as a 0. An attempt to write a reserved bit with a 1 has no effect; however, this should be avoided because of upward-compatibility considerations.

The Translation Look-Aside Buffer (TLB) registers are accessed only by explicit data movement by Supervisor-mode programs. Instructions that move data to or from a TLB register specify a general-purpose register containing a TLB register number. The TLB register number is given by the contents of bits 6–0 of the general-purpose register. TLB register numbers may only be specified indirectly by general-purpose registers.

TLB entries are accessed as registers numbered 0–127. Since two words are required to completely specify a TLB entry, two registers are required for each TLB entry. The words corresponding to an entry are paired as two sequentially numbered registers starting on an even-numbered register. The word with the even register number is called Word 0, and the word with the odd register number is called Word 1. The entries for TLB Set 0 are in registers numbered 0–63, and the entries for TLB Set 1 are in registers numbered 64–127.

TLB Entry Word 0

The TLB Entry Word 0 register is shown in Figure 34.

Bits 31–15: Virtual Tag (VTAG)—When the TLB is searched for an address translation, the VTAG field of the TLB entry must match the most significant 17, 16, 15, or 14 bits of the address being translated—for page sizes of 1, 2, 4, and 8 kb, respectively—for the search to be successful.

TLB Reg#	TLB Set 0
0	TLB Entry Line 0 Word 0
1	TLB Entry Line 0 Word 1
2	TLB Entry Line 1 Word 0
3	TLB Entry Line 1 Word 1
• • •	• • •
62	TLB Entry Line 31 Word 0
63	TLB Entry Line 31 Word 1

	TLB Set 1
64	TLB Entry Line 0 Word 0
65	TLB Entry Line 0 Word 1
• • •	• • •
126	TLB Entry Line 31 Word 0
127	TLB Entry Line 31 Word 1

Figure 33. Translation Look-Aside Buffer Registers

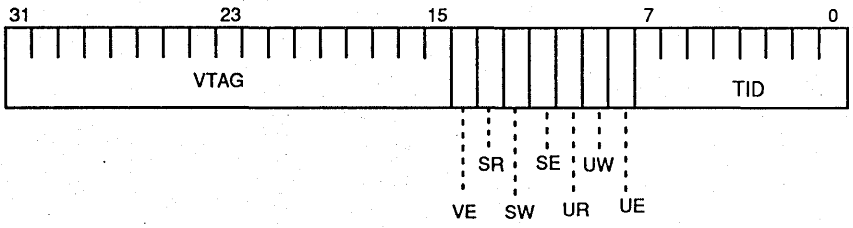


Figure 34. TLB Entry Word 0

When software loads a TLB entry with an address translation, the most significant 14 bits of the Virtual Tag are set with the most significant 14 bits of the virtual address whose translation is being loaded into the TLB. The remaining 3 bits of the Virtual Tag must be set either to the corresponding bits of the address or to 0s, depending on the page size, as follows ("A" refers to corresponding address bits):

Page Size	VTAG 2–0 (TLB Word 0 bits 17–15)
1 kb	AAA
2 kb	AA0
4 kb	A00
8 kb	000

Bit 14: Valid Entry (VE)—If this bit is 1, the associated TLB entry is valid; if it is 0, the entry is invalid.

Bit 13: Supervisor Read (SR)—If the SR bit is 1, Supervisor-mode load operations from the virtual page are allowed; if it is 0, Supervisor-mode loads are not allowed.

Bit 12: Supervisor Write (SW)—If the SW bit is 1, Supervisor-mode store operations to the virtual page are allowed; if it is 0, Supervisor-mode stores are not allowed.

Bit 11: Supervisor Execute (SE)—If the SE bit is 1, Supervisor-mode instruction accesses to the virtual page are allowed; if it is 0, Supervisor-mode instruction accesses are not allowed.

Bit 10: User Read (UR)—If the UR bit is 1, User-mode load operations from the virtual page are allowed; if it is 0, User-mode loads are not allowed.

Bit 9: User Write (UW)—If the UW bit is 1, User-mode store operations to the virtual page are allowed; if it is 0, User-mode stores are not allowed.

Bit 8: User Execute (UE)—If the UE bit is 1, User-mode instruction accesses to the virtual page are allowed; if it is 0, User-mode instruction accesses are not allowed.

Bits 7–0: Task Identifier (TID)—When the TLB is searched for an address translation, the TID must match the Process Identifier (PID) in the MMU Configuration Register for the translation to be successful. This field is allows the TLB entry to be associated with a particular process.

TLB Entry Word 1

The TLB Entry Word 1 register is shown in Figure 35.

Bits 31–10: Real Page Number (RPN)—The RPN field gives the most significant 22, 21, 20, or 19 bits of the physical address of the page for page sizes of 1, 2, 4, and 8 kb, respectively. It is concatenated to bits 9–0, 10–0, 11–0, or 12–0 of the address being translated—for 1-, 2-, 4-, and 8-kb page sizes, respectively—to form the physical address for the access.

When software loads a TLB entry with an address translation, the most significant 19 bits of the Real Page Number are set with the most significant 19 bits of the physical address associated with the translation. The remaining 3 bits of the Real Page Number must be set either to the corresponding bits of the physical address, or to 0s, depending on the page size, as follows ("A" refers to corresponding address bits):

Page Size	RPN 2–0 (TLB Word 1 bits 12–10)
1 kb	AAA
2 kb	AA0
4 kb	A00
8 kb	000

Bits 7–6: User Programmable (PGM)—These bits are placed on the $MPGM_1$–$MPGM_0$ outputs when the ad-

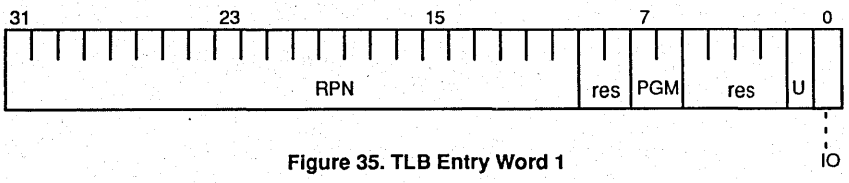


Figure 35. TLB Entry Word 1

dress is transmitted for an access. They have no predefined effect on the access; any effect is defined by logic external to the processor.

Bit 1: Usage (U)—This bit indicates which entry in a given TLB line was least recently used to perform an address translation. If this bit is a 0, then the entry in Set 0 in the line is least recently used; if it is 1, then the entry in Set 1 is least recently used. This bit has an equal value for both entries in a line. Whenever a TLB entry is used to translate an address, the Usage bit of both entries in the line used for translation are set according to the TLB set containing the translation. This bit is set whenever the translation is valid, regardless of the outcome of memory-protection checking.

Bit 0: Input/Output (IO)—The IO bit determines whether the access is directed to the instruction/data memory (IO = 0) or the input/output (IO = 1) address space.

INSTRUCTION SET

The Am29000 implements 117 instructions. All instructions execute in a single cycle except for IRET, IRETINV, LOADM, STOREM, and the trapping arithmetic instructions such as floating-point instructions.

Most instructions deal with general-purpose registers for operands and results; however, in most instructions, an 8-bit constant can be used in place of a register-based operand. Some instructions deal with special-purpose registers, TLB registers, external devices and memories, and coprocessors.

This section describes the nine instruction classes in the Am29000, and provides a brief summary of instruction operations.

If the processor attempts to execute an instruction that is not implemented, an Illegal Opcode trap occurs.

Integer Arithmetic

The Integer Arithmetic instructions perform add, subtract, multiply, and divide operations on word-length integers. Certain instructions in this class cause traps if signed or unsigned overflow occurs during the execution of the instruction. There is support for multi-precision arithmetic on operands whose lengths are multiples of words. All instructions in this class set the ALU Status Register. The integer arithmetic instructions are shown in Figure 36.

The instructions MULTIPLU, MULTMU, MULTIPLY, MULTM, DIVIDE, and DIVIDU are not implemented directly by processor hardware, but cause traps to occur in instruction-emulation routines.

Compare

The Compare instructions test for various relationships between two values. For all Compare instructions except the CPBYTE instruction, the comparisons are performed on word-length signed or unsigned integers. There are two types of Compare instructions. The first type places a Boolean value reflecting the outcome of the compare into a general-purpose register. For the second type (assert instructions), instruction execution continues only if the comparison is true; otherwise a trap occurs. The assert instructions specify a vector for the trap.

The assert instructions support run-time operand checking and operating-system calls. If the trap occurs in the User mode and a trap number between 0 and 63 is specified by the instruction, a Protection Violation trap occurs. The Compare instructions are shown in Figure 37.

Logical

The Logical instructions perform a set of bit-by-bit Boolean functions on word-length bit strings. All instructions in this class set the ALU Status Register. These instructions are shown in Figure 38.

Shift

The Shift instructions (Figure 39) perform arithmetic and logical shifts. All but the Extract instruction operate on word-length data and produce a word-length result. The Extract instruction operates on double-word data and produces a word-length result. If both parts of the double word for the Extract instruction are from the same source, the Extract operation is equivalent to a rotate operation. For each operation, the shift count is a 5-bit integer, specifying a shift amount in the range of 0 to 31 bits.

Data Movement

The Data Movement instructions (Figure 40) move bytes, half-words, and words between processor registers. In addition, they move data between general-purpose registers and external devices, memories, and the coprocessor.

Constant

The Constant instructions (Figure 41) provide the ability to place half-word and word constants into registers. Most instructions in the instruction set allow an 8-bit constant as an operand. The Constant instructions allow the construction of larger constants.

Floating-Point

The Floating-Point instructions (Figure 42) provide operations on single-precision (32-bit) or double-precision (64-bit) floating-point data. In addition, they provide conversions between single-precision, double-precision, and integer number representations. In the current processor implementation, these instructions cause traps to occur in routines that perform the floating-point operations.

Branch

The Branch instructions (Figure 43) control the execution flow of instructions. Branch target addresses may be absolute, relative to the Program Counter (with the offset given by a signed instruction constant), or contained in a general-purpose register. For conditional jumps, the outcome of the jump is based on a Boolean value in a general-purpose register. Procedure calls are unconditional and save the return address in a general-purpose register. All branches have a delayed effect; the instruction sequence following the branch is executed regardless of the outcome of the branch.

Miscellaneous

The Miscellaneous instructions (Figure 44) perform various operations that cannot be grouped into other instruction classes. In certain cases, these are control functions available only to Supervisor-mode programs.

Mnemonic	Operation Description
ADD	DEST <-SRCA + SRCB

ADDS	DEST <-SRCA + SRCB
	IF signed overflow THEN Trap (Out Of Range)

ADDU	DEST <-SRCA + SRCB
	IF unsigned overflow THEN Trap (Out Of Range)

ADDC	DEST <-SRCA + SRCB + C

ADDCS	DEST <-SRCA + SRCB + C
	IF signed overflow THEN Trap (Out Of Range)

ADDCU	DEST <-SRCA + SRCB + C
	IF unsigned overflow THEN Trap (Out Of Range)

SUB	DEST <-SRCA – SRCB

SUBS	DEST <-SRCA – SRCB
	IF signed overflow THEN Trap (Out Of Range)

SUBU	DEST <-SRCA – SRCB
	IF unsigned underflow THEN Trap (Out Of Range)

SUBC	DEST <-SRCA–SRCB –1 + C

SUBCS	DEST <-SRCA – SRCB –1 + C
	IF signed overflow THEN Trap (Out Of Range)

SUBCU	DEST <-SRCA – SRCB –1 + C
	IF unsigned underflow THEN Trap (Out Of Range)

SUBR	DEST <-SRCB – SRCA

SUBRS	DEST <-SRCB – SRCA
	IF signed overflow THEN Trap (Out Of Range)

SUBRU	DEST <-SRCB – SRCA
	IF unsigned underflow THEN Trap (Out Of Range)

SUBRC	DEST <-SRCB – SRCA –1 + C

SUBRCS	DEST <-SRCB – SRCA –1 + C
	IF signed overflow THEN Trap (Out Of Range)

SUBRCU	DEST <-SRCB – SRCA –1 + C
	IF unsigned underflow THEN Trap (Out Of Range)

MULTIPLU	DEST <-SRCA * SRCB (unsigned)
MULTIPLY	DEST <-SRCA * SRCB (signed)
MUL	Perform 1-bit step of a multiply operation (signed)
MULL	Complete a sequence of multiply steps
MULTM	DEST <-SRCA * SRCB (signed), most-significant bits
MULTMU	DEST <-SRCA * SRCB (unsigned), most-significant bits
MULU	Perform 1-bit step of a multiply operation (unsigned)
DIVIDE	DEST <-(Q//SRCA)/SRCB (signed) Q <-Remainder
DIVIDU	DEST <-(Q//SRCA)/SRCB (unsigned) Q <-Remainder
DIV0	Initialize for a sequence of divide steps (unsigned)
DIV	Perform 1-bit step of a divide operation (unsigned)
DIVL	Complete a sequence of divide steps (unsigned)
DIVREM	Generate remainder for divide operation (unsigned)

Figure 36. Integer Arithmetic Instructions

Mnemonic	Operation Description

CPEQ	IF SRCA = SRCB THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPNEQ	IF SRCA <> SRCB THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPLT	IF SRCA < SRCB THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPLTU	IF SRCA < SRCB (unsigned) THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPLE	IF SRCA <= SRCB THEN DEST <-TRUE
	ELSE DEST <- FALSE

CPLEU	IF SRCA <= SRCB (unsigned) THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPGT	IF SRCA > SRCB THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPGTU	IF SRCA > SRCB (unsigned) THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPGE	IF SRCA >= SRCB THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPGEU	IF SRCA >= SRCB (unsigned) THEN DEST <-TRUE
	ELSE DEST <-FALSE

CPBYTE	IF (SRCA.BYTE0 = SRCB.BYTE0) OR (SRCA.BYTE1 = SRCB.BYTE1) OR (SRCA.BYTE2 = SRCB.BYTE2) OR (SRCA.BYTE3 = SRCB.BYTE3)THEN DEST <-TRUE ELSE DEST <-FALSE

ASEQ	IF SRCA = SRCB THEN Continue
	ELSE Trap (VN)

ASNEQ	IF SRCA <> SRCB THEN Continue
	ELSE Trap (VN)

ASLT	IF SRCA < SRCB THEN Continue
	ELSE Trap (VN)

ASLTU	IF SRCA < SRCB (unsigned) THEN Continue
	ELSE Trap (VN)

ASLE	IF SRCA <= SRCB THEN Continue
	ELSE Trap (VN)

ASLEU	IF SRCA <= SRCB (unsigned) THEN Continue
	ELSE Trap (VN)

ASGT	IF SRCA > SRCB THEN Continue
	ELSE Trap (VN)

ASGTU	IF SRCA > SRCB (unsigned) THEN Continue
	ELSE Trap (VN)

ASGE	IF SRCA >= SRCB THEN Continue
	ELSE Trap (VN)

ASGEU	IF SRCA >= SRCB (unsigned) THEN Continue
	ELSE Trap (VN)

Figure 37. Compare Instructions

Mnemonic	Operation Description
AND	DEST <-SRCA & SRCB
ANDN	DEST <-SRCA & ~ SRCB
NAND	DEST <-~ (SRCA & SRCB)
OR	DEST <-SRCA | SRCB
NOR	DEST <-~ (SRCA | SRCB)
XOR	DEST <-SRCA ^ SRCB
XNOR	DEST <-~ (SRCA ^ SRCB)

Figure 38. Logical Instructions

Mnemonic	Operation Description
SLL	DEST <-SRCA << SRCB (zero fill)
SRL	DEST <-SRCA >> SRCB (zero fill)
SRA	DEST <-SRCA >> SRCB (sign fill)
EXTRACT	DEST <-high-order word of (SRCA//SRCB << FC)

Figure 39. Shift Instructions

Reserved Instructions

Sixteen Am29000 operation codes are reserved for instruction emulation. These instructions cause traps, much like the floating-point instructions, but currently have no specified interpretation. The relevant operation codes and the corresponding trap vectors are:

Operation Codes (hexadecimal)	Trap Vector Numbers (decimal)
D8–DD	24–29
E7–E9	39–41
F8	56
FA–FF	58–63

These instructions are intended for future processor enhancements, and users desiring compatibility with future processor versions should not use them for any purpose.

Mnemonic	Operation Description
LOAD	DEST <-EXTERNAL WORD [SRCB]
LOADL	DEST <-EXTERNAL WORD [SRCB] assert *LOCK output during access
LOADSET	DEST <-EXTERNAL WORD [SRCB] EXTERNAL WORD [SRCB] <-h'FFFFFFFF', assert $\overline{\text{LOCK}}$ output during access
LOADM	DEST.. DEST + COUNT <- EXTERNAL WORD [SRCB] .. EXTERNAL WORD [SRCB + COUNT * 4]
STORE	EXTERNAL WORD [SRCB] <-SRCA
STOREL	EXTERNAL WORD [SRCB] <-SRCA assert $\overline{\text{LOCK}}$ output during access
STOREM	EXTERNAL WORD [SRCB] .. EXTERNAL WORD [SRCB + COUNT * 4] <- SRCA .. SRCA + COUNT
EXBYTE	DEST <-SRCB, with low-order byte replaced by byte in SRCA selected by BP
EXHW	DEST <-SRCB, with low-order half-word replaced by half-word in SRCA selected by BP
EXHWS	DEST <- half-word in SRCA selected by BP, sign-extended to 32 bits
INBYTE	DEST <-SRCA, with byte selected by BP replaced by low-order byte of SRCB
INHW	DEST <-SRCA, with half-word selected by BP replaced by low-order half-word of SRCB
MFSR	DEST <-SPECIAL
MFTLB	DEST <-TLB [SRCA]
MTSR	SPDEST <-SRCB
MTSRIM	SPDEST <- 0I16
MTTLB	TLB [SRCA] <-SRCB

Figure 40. Data Movement Instructions

Mnemonic	Operation Description
CONST	DEST <-0I16
CONSTH	Replace high-order half-word of SRCA by I16
CONSTN	DEST <-1I16

Figure 41. Constant Instructions

Mnemonic	Operation Description
FADD	DEST (single-precision) <-SRCA (single-precision) + SRCB (single-precision)
DADD	DEST (double-precision) <-SRCA (double-precision) + SRCB (double-precision)
FSUB	DEST (single-precision) <-SRCA (single-precision) -SRCB (single-precision)
DSUB	DEST (double-precision) <-SRCA (double-precision) -SRCB (double-precision)
FMUL	DEST (single-precision) <-SRCA (single-precision) * SRCB (single-precision)
FDMUL	DEST (double-precision) <-SRCA (single-precision) * SRCB (single-precision)
DMUL	DEST (double-precision) <-SRCA (double-precision) * SRCB (double-precision)
FDIV	DEST (single-precision) <-SRCA (single-precision)/ SRCB (single-precision)
DDIV	DEST (double-precision) <-SRCA (double-precision)/ SRCB (double-precision)

FEQ	IF SRCA (single-precision) = SRCB (single-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

DEQ	IF SRCA (double-precision) = SRCB (double-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

FGE	IF SRCA (single-precision) >= SRCB (single-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

DGE	IF SRCA (double-precision) >= SRCB (double-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

FGT	IF SRCA (single-precision) > SRCB (single-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

DGT	IF SRCA (double-precision) > SRCB (double-precision)
	THEN DEST <-TRUE
	ELSE DEST <-FALSE

SQRT	DEST (single-precision, double-precision, extended-precision) <-SQRT[SRCA (single-precision, double-precision, extended-precision)]
CONVERT	DEST (integer, single-precision, double-precision) <-SRCA (integer, single-precision, double-precision)
CLASS	DEST (single-precision, double-precision, extended-precision) <-CLASS[SRCA (single-precision, double-precision, extended-precision)]

Figure 42. Floating-Point Instructions

Mnemonic	Operation Description

CALL	DEST <-PC//00 + 8
	PC <-TARGET
	Execute delay instruction

CALLI	DEST <-PC//00 + 8
	PC <-SRCB
	Execute delay instruction

JMP	PC <-TARGET Execute delay instruction
JMPI	PC <-SRCB Execute delay instruction
JMPT	IF SRCA = TRUE THEN PC <-TARGET Execute delay instruction
JMPTI	IF SRCA = TRUE THEN PC <-SRCB Execute delay instruction
JMPF	IF SRCA = FALSE THEN PC <-TARGET Execute delay instruction
JMPFI	IF SRCA = FALSE THEN PC <-SRCB Execute delay instruction

JMPFDEC	IF SRCA = FALSE THEN
	   SRCA <-SRCA -1
	   PC <-TARGET
	ELSE
	   SRCA <-SRCA -1
	Execute delay instruction

Figure 43. Branch Instructions

Mnemonic	Operation Description
CLZ	Determine number of leading zeros in a word
SETIP	Set IPA, IPB, and IPC with operand register numbers
EMULATE	Load IPA and IPB with operand register numbers, and Trap (VN)
INV	Reset all Valid bits in Branch Target Cache to zeros
IRET	Perform an interrupt return sequence
IRETINV	Perform an interrupt return sequence, and reset all Valid bits in Branch Target Cache to zeros
HALT	Enter Halt mode on next cycle

Figure 44. Miscellaneous Instructions

DATA FORMATS AND HANDLING

This section describes the various data types supported by the Am29000, and the mechanisms for accessing data in external devices and memories. The Am29000 includes provisions for the external access of bytes, half-words, unaligned words, and unaligned half-words, as described in this section.

Integer Data Types

Most Am29000 instructions deal directly with word-length integer data; integers may be either signed or unsigned, depending on the instruction. Some instructions (e.g., AND) treat word-length operands as strings of bits. In addition, there is support for character, half-word, and Boolean data types.

Byte Operations

The processor supports character data through load, store, extraction, and insertion operations on word-length operands, and by a compare operation on byte-length fields within words. The format for unsigned and signed characters is shown in Figure 45; for signed characters, the sign bit is the most-significant bit of the character. For sequences of packed characters within words, bytes are ordered either left-to-right or right-to-left, depending on the BO bit of the Configuration Register (see Special Floating-Point Values section).

If the Data Width Enable (DW) bit of the Configuration Register is 1, the Am29000 is enabled to load and store byte data. On a load, an external packed byte is converted to one of the character formats shown in Figure 45. On a store, the low-order byte of a word is packed into every byte of an external word. The External Data Accesses section describes external byte accesses in more detail.

The Extract Byte (EXBYTE) instruction replaces the low-order character of a destination word with an arbitrary byte-aligned character from a source word. For the EXBYTE instruction, the destination word can be a zero word, which effectively zero-extends the character from the source operand.

The Insert Byte (INBYTE) instruction replaces an arbitrary byte-aligned character in a destination word with the low-order character of a source word. For the INBYTE instruction, the source operand can be a character constant specified by the instruction.

The Compare Bytes (CPBYTE) instruction compares two word-length operands and gives a result of True if any corresponding bytes within the operands have equivalent values. This allows programs to detect characters within words without first having to extract individual characters, one at a time, from the word of interest.

Half-Word Operations

The processor supports half-word data through load, store, insertion, and extraction operations on word-length operands. The format for unsigned and signed half-words is shown in Figure 46; for signed half-words, the sign bit is the most-significant bit of the half-word. For sequences of packed half-words within words, half-words are ordered either left-to-right or right-to-left, depending on the Byte Order (BO) bit of the Configuration Register (see Addressing and Alignment section).

If the Data Width Enable (DW) bit of the Configuration Register is 1, the Am29000 is enabled to load and store half-word data. On a load, an external packed half-word is converted to one of the formats shown in Figure 46. On a store, the low-order half-word of a word is packed into every half-word of an external word.

The Extract Half-Word (EXHW) instruction replaces the low-order half-word of a destination word with either the low-order or high-order half-word of a source word. For the EXHW instruction, the destination word can be a zero word, which effectively zero-extends the half-word from the source operand.

The Extract Half-Word, Sign-Extended (EXHWS) instruction is similar to the EXHW instruction, except that it sign-extends the half-word in the destination word (i.e., it replaces the most-significant 16 bits of the destination word with the most-significant bit of the source half-word).

The Insert Half-Word (INHW) instruction replaces either the low-order or high-order half-word in a destination word with the low-order half-word of a source word.

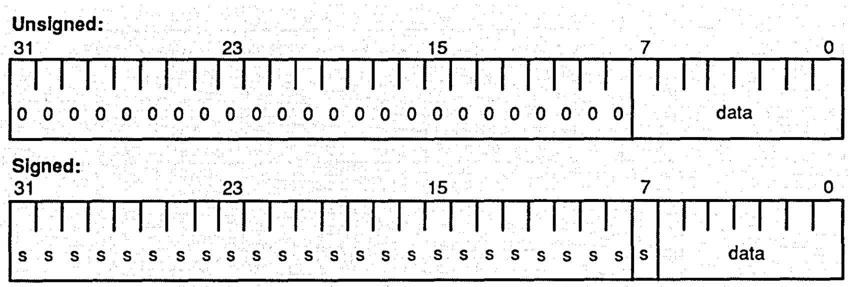


Figure 45. Character Format

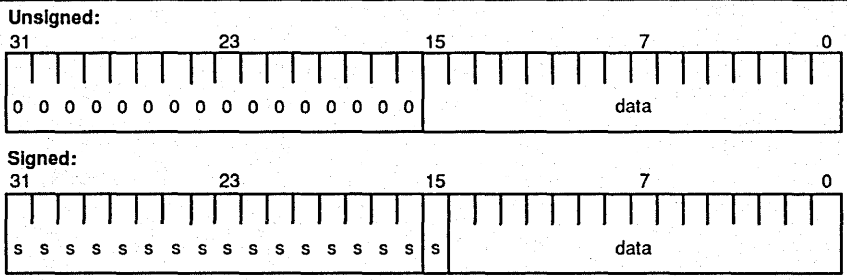


Figure 46. Half-Word Format

Boolean Data

Some instructions in the Compare class generate word-length Boolean results. Also, conditional branches are conditional upon Boolean operands. The Boolean format used by the processor is such that the Boolean values True and False are represented by a 1 or 0, respectively, in the most-significant bit of a word. The remaining bits are unimportant; for the compare instructions, they are reset. Note that twos-complement negative integers are indicated by the Boolean value True in this encoding scheme.

Floating-Point Data Types

The Am29000 defines single- and double-precision floating-point formats that comply with the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985). These data types are not supported directly in processor hardware, but can be implemented by a virtual floating-point interface provided in the Am29000.

In this section, the following nomenclature is used to denote fields in a floating-point value:

- s: sign bit
- bexp: biased exponent
- frac: fraction
- sig: significand

Single-Precision Floating-Point

The format for a single-precision floating-point value is shown in Figure 47.

Typically, the value of a single-precision operand is expressed by:

(–1)**s * 1.frac * 2**(bexp–127).

The encoding of special floating-point values is given in the Special Floating-Point Values section.

Double-Precision Floating-Point

The format for a double-precision floating-point value is shown in Figure 48.

Typically, the value of a double-precision operand is expressed by:

(–1)**s * 1.frac * 2**(bexp–1023).

The encoding of special floating-point values is given in the Special Floating-Point Values section.

In order to be properly referenced by a floating-point instruction, a double-precision floating-point value must be double-word aligned. The absolute register number of the register containing the first word (labeled "0" in Figure 48) must be even. The absolute register number of the register containing the second word (labeled "1" in Figure 48) must be odd. If these conditions are not met, the results of the instruction are unpredictable. Note that the appropriate registers for a double-precision value in the local registers depend on the value of the Stack Pointer.

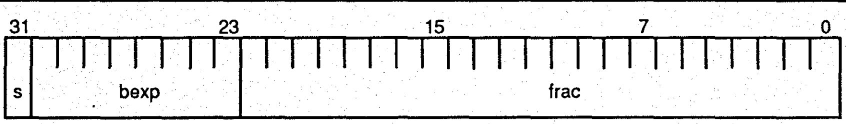


Figure 47. Single-Precision Floating-Point Format

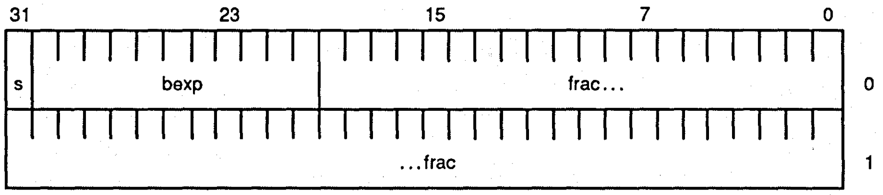


Figure 48. Double-Precision Floating-Point Format

Special Floating-Point Values

The Am29000 defines floating-point values that are encoded for special interpretation. The values are described in this section.

Not-a-Number

A Not-a-Number (NaN) is a symbolic value used to report certain floating-point exceptions. It also can be used to implement user-defined extensions to floating-point operations. A NaN comprises a floating-point number with maximum biased exponent and non-zero fraction. The sign bit can be either 0 or 1 and has no significance. There are two types of NaN: signaling NaNs and quiet NaNs. A signaling NaN causes an Invalid Operation exception if used as an input operand to a floating-point operation; a quiet NaN does not cause an exception. The Am29000 distinguishes signaling and quiet NaNs by the most-significant bit of the fraction: a 1 indicates a quiet NaN, and a 0 indicates 2 signaling NaN.

An operation never generates a signaling NaN as a result. A quiet NaN result can be generated in one of two ways:

- as the result of an invalid operation that cannot generate a reasonable result, or
- as the result of an operation for which one or more input operands are either signaling or quiet NaNs.

In either case, the Am29000 produces a quiet NaN having a fraction of 11000...0; that is, the two most-significant bits of the fraction are 11, and the remaining bits are 0. If desired, the Reserved Operand exception can be enabled to cause a Floating-Point Exception trap. The trap handler in this case can implement a scheme whereby user-defined NaN values appear to pass through operations as results, providing overall status for a series of operations.

Infinity

Infinity is an encoded value used to represent a value that is too large to be represented as a finite number in a given floating-point format. Infinity comprises a floating-point number with maximum biased exponent and zero fraction. The sign bit of an infinity distinguishes $+\infty$ from $-\infty$.

Denormalized Numbers

The IEEE Standard specifies that, wherever possible, a result that is too small to be represented as a normalized number be represented as a denormalized number. A denormalized number may be used as an input operand to any operation. For single- and double-precision formats, a denormalized number comprises a floating-point number with a biased exponent of 0 and a non-zero fraction field; the sign bit can be either 1 or 0. The value of a denormalized number is expressed by:

(–1)**s * 0.frac * 2**(–bias + 1),

where "bias" is the exponent bias for the format in question.

Zero

A zero comprises a floating-point number with a biased exponent of 0 and a zero fraction field. The sign bit of a zero can be either 0 or 1; however, positive and negative zero are both exactly zero, and are considered equal by comparison operations.

External Data Accesses

All processor external accesses occur between general-purpose registers and external devices and memories. Accesses occur as the result of the execution of load and store instructions. The load and store instructions specify which general-purpose register receives the data (for a load) or supplies the data (for a store). The format of the load and store instructions is shown in Figure 49.

Addresses for accesses are given either by the content of a general-purpose register or by a constant value specified by the load or store instruction. The load and store instructions do not perform address computation directly. Any required address computations are performed explicitly by other instructions.

In the load or store instruction, the Coprocessor Enable (CE) bit (bit 23) determines whether or not the access is directed to the coprocessor. If the CE bit is 0, the access is directed to an external device or memory. If the CE bit is 1, data is transferred to or from the coprocessor. The CE bit affects the interpretation of the Control (CNTL) field as well as the channel protocol. This section deals

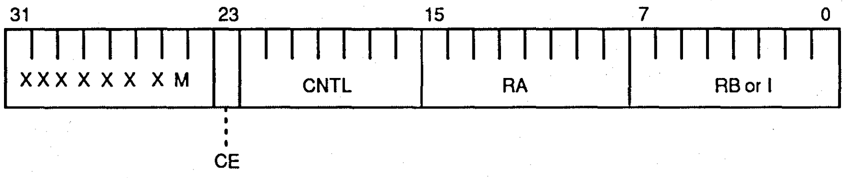


Figure 49. Load/Store Instruction Format

with all external accesses other than coprocessor accesses.

The format of the instructions that do not perform coprocessor data transfers (i.e., in which the CE bit is 0) is shown in Figure 50.

In load and store instructions, the "RB or I" field specifies the address for access. The address is either the content of a general-purpose register, with register number RB, or a constant with a value I (zero-extended to 32 bits). The M bit determines whether the register or the constant is used.

The data for the access is written into the general-purpose register RA for a load, and is supplied by register RA for a store.

The definitions for other fields in the load or store instruction are given below:

Bit 23: Coprocessor Enable (CE)—The CE bit is 0 for a non-coprocessor load or store.

Bit 22: Address Space (AS)—If the AS bit is 0 for an untranslated load or store, the access is directed to instruction/data memory. If the AS bit is 1 for an untranslated load or store, the access is directed to input/output. The AS bit must be 0 for a translated load or store; if the AS bit is 1 for a translated load or store, a Protection Violation trap occurs. The address space for a translated load or store is determined by the Input/Output (IO) bit of the associated TLB entry.

Bit 21: Physical Address (PA)—The PA bit may be used by a Supervisor-mode program to disable address translation for an access. If the PA bit is 1, then address translation is not performed for the access, regardless of the value of the Physical Addressing/Data (PD) bit in the Current Processor Status Register. If the PA bit is 0, address translation depends on the PD bit.

The PA bit may be 1 only for Supervisor-mode instructions. If it is 1 for a User-mode instruction, a Protection Violation trap occurs.

Bit 20: Set Byte Pointer/Sign Bit (SB)—If the Data Width Enable (DW) bit of the Configuration Register is 0 and the SB bit is 1, the Byte Pointer Register is written with the two least-significant bits of the address for the access. These address bits can control subsequent character and half-word operations. If the BP bit is 0, the Byte Pointer Register is not affected.

If the Data Width Enable (DW) bit of the Configuration Register is 1 and the SB bit is 1 for a load, the loaded byte or half-word is sign-extended in the destination register; if the SB bit is 0, the byte or half-word is zero-extended. If the DW bit is 1 and the SB bit is 1 for either a load or store, then each bit of the Byte Pointer Register is written with the complement of the Byte Order bit of the Configuration Register. The Byte Pointer Register is set in this case to provide software compatibility across different types of memory systems. If the SB bit is 0, the Byte Pointer Register is not affected.

Bit 19: User Access (UA)—The UA bit allows programs executing in the Supervisor mode to emulate User-mode accesses. This allows checking of the authorization of an access requested by a User-mode program. It also causes address translation (if applicable) to be performed using the PID field of the MMU Configuration Register, rather than the fixed Supervisor-mode process identifier zero.

If the UA bit is 1 for a Supervisor-mode load or store, the access associated with the instruction is performed in

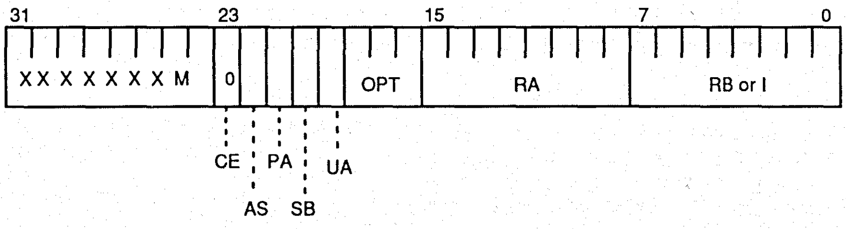


Figure 50. Non-Coprocessor Load/Store Format

the User mode. In this case, the User mode affects only TLB protection checking, the SUP/$\overline{\text{US}}$ output, and the use of the PID field in translation; it has no effect on the registers that can be accessed by the instruction. If the UA bit is 0, the program mode for the access is controlled by the SM bit.

If the UA bit is 1 for a User-mode load or store, a Protection Violation trap occurs.

Bits 18–16: Option (OPT)—This field is placed on the OPT_2–OPT_0 outputs during the address cycle of the access. There is a one-to-one correspondence between the OPT field and the OPT_2–OPT_0 outputs; that is, the most-significant OPT bit is placed on OPT_2, and so on.

The OPT field controls system functions as described below.

Bits 15–8: (RA)—The data for the access is written into the general-purpose register RA for a load, and is supplied by register RA for a store.

Bits 7–0: (RB or I)—In load and store instructions, the "RB or I" field specifies the address for the access. The address is either the content of a general-purpose register with register number RB, or a constant value I (zero-extended to 32 bits). The M bit of the operation code (bit 24) determines whether the register or the constant is used.

Load and store operations are overlapped with the execution of instructions that follow the load or store instruction. Only one load or store may be in progress on any given cycle. If a load or store instruction is encountered while another load or store operation is in progress, the processor enters the Pipeline Hold mode until the first operation is completed. However, the address for the second operation may appear on the address bus if the first operation is to a device or memory that supports pipelined operations (see Pipelined Accesses section).

Load Operations

The processor provides the following instructions for performing load operations: Load (LOAD), Load and Lock (LOADL), Load and Set (LOADSET), and Load Multiple (LOADM). All of these instructions transfer data from an external device or memory into one or more general-purpose registers.

The LOADL instruction supports the implementation of device and memory interlocks in a multiprocessor configuration. It activates the $\overline{\text{LOCK}}$ output during the address cycle of the access.

The LOADSET instruction implements a binary semaphore. It loads a general-purpose register and automatically writes the accessed location with a word that has 1 in every bit position (that is, the write is indivisible from the read). The $\overline{\text{LOCK}}$ output is asserted during both the read and write accesses. Note that, if address translation is enabled for the LOADSET instruction, the TLB memory-protection bits must allow both the read and write accesses. If either the read or write access is not allowed, neither access is performed.

The LOADM loads a specified number of registers from sequential addresses, as explained below.

Load operations are overlapped with the execution of instructions that follow the load instruction. The processor detects any dependencies on the loaded data that subsequent instructions may have, and, if such a dependency is detected, enters the Pipeline Hold mode until the data are returned by the external device or memory. If a register that is the target of an incomplete load is written with the result of a subsequent instruction, the processor does not write the returning data into the register when the load is completed; the Not Needed (NN) bit in the Channel Control Register is set in this case.

Store Operations

The processor provides the following instructions for performing store operations: Store (STORE), Store and Lock (STOREL), and Store Multiple (STOREM). All of these instructions transfer data from one or more general-purpose registers to an external device or memory.

The STOREL instruction supports the implementation of device and memory interlocks in a multiprocessor configuration. It activates the $\overline{\text{LOCK}}$ output during the address cycle of the access.

The STOREM instruction stores a specified number of registers to sequential addresses, as explained below.

Store operations are overlapped with the execution of instructions that follow the store instruction. However, no data dependencies can exist since the store prevents any subsequent accesses until it is completed.

Multiple Accesses

Load Multiple (LOADM) and Store Multiple (STOREM) instructions move contiguous words of data between general-purpose registers and external devices and memories. The number of transfers is determined by the Load/Store Count Remaining Register.

The Load/Store Count Remaining (CR) field in the Load/Store Count Remaining Register specifies the number of transfers to be performed by the next LOADM or STOREM executed in the instruction sequence. The CR field is in the range of 0 to 255 and is zero-based; a count value of 0 represents one transfer, and a count value of 255 represents 256 transfers. The CR field also appears in the Channel Control Register.

Before a LOADM or STOREM is executed, the CR field is set by a Move To Special Register. A LOADM or STOREM uses the most recently written value of the CR field. If an attempt is made to alter the CR field and the Channel Control Register contains information for an external access that has not yet been completed, the processor enters the Pipeline Hold mode until the access is completed. Note that since the CR is set independently of the LOADM and STOREM, the CR field may represent a valid state of an interrupted program even if the Contents Valid (CV) bit of the Channel Control Register is 0.

Because of the pipelined implementation of LOADM and STOREM, at least one instruction (e.g., the instruction that sets the CR field) must separate two successive LOADM and/or STOREM instructions.

After the CR field is set, the execution of a LOADM or STOREM begins the data transfer. As with any other load or store operation, the LOADM or STOREM waits until any pending load or store operation is complete before starting. The LOADM instruction specifies the starting address and starting destination general-purpose register. The STOREM instruction specifies the starting address and the starting source general-purpose register.

During the execution of the LOADM or STOREM instruction, the processor updates the address and register number after every access, incrementing the address by 4 and the register number by 1. This continues until either all accesses are completed or an interrupt or trap is taken.

For a Load Multiple or Store Multiple address sequence, addresses wrap from the largest possible value (hexadecimal FFFFFFFC) to the smallest possible value (hexadecimal 00000000).

The processor increments absolute register numbers during the Load Multiple or Store Multiple sequence. Absolute register numbers wrap from 127 to 128, and from 255 to 128. Thus, a sequence that begins in the global registers may make a transition to the local registers, but a sequence that begins in the local registers remains in the local registers. Also, note that the local registers are addressed circularly.

The normal restrictions on register accesses apply for the Load Multiple and Store Multiple sequences. For example, if a protected general-purpose register is encountered in the sequence for a User-mode program, a Protection Violation trap occurs.

Intermediate addresses are stored in the Channel Address Register, and register numbers are stored in the Target Register (TR) field of the Channel Control Register. For the STOREM instruction, the data for every access is stored in the Channel Data Register (this register also is set during the execution of the LOADM instruction, but has no interpretation in this case). The CR field is updated on the completion of every access so that it indicates the number of accesses remaining in the sequence.

Load Multiple and Store Multiple operations are indicated by the Multiple Operation (ML) bit in the Channel Control Register. This bit may be 1 even though the CR field has a value of 0 (indicating that one transfer remains to be performed). The ML bit is used to restart a multiple operation on an interrupt return; if it is set independently by a Move To Special Register before a load or store instruction is executed, the results are unpredictable.

While a multiple load or store is executing, the processor is in the Pipeline Hold mode, suspending any subsequent instruction execution until the multiple access is completed. If an interrupt or trap is taken, the Channel Address, Channel Data, and Channel Control registers contain the state of the multiple access at the point of interruption. The multiple access may be resumed at this point, at a later time, by an interrupt return.

The processor attempts to complete multiple accesses using the burst-mode capability of the channel (see Burst-Mode Accesses section). For this reason, multiple accesses of individual bytes and half-words are not supported. If the burst-mode access is preempted, the processor retransmits the address at the point of preemption. If the external device or memory cannot support burst-mode accesses, the processor transmits an address for every access. If the address sequence causes a virtual page-boundary crossing, the processor preempts the burst-mode access, translates the address for the new page, and reestablishes the burst-mode access using the new physical address.

The last load or store is executed as a simple access. The processor will preempt burst-mode transfer immediately prior to the last word of the transfer.

Option Bits

The Option field in the load and store instructions supports system functions, such as byte and half-word accesses. The definition of this field for a load or store, depending on the AS bit of the instruction, is as follows:

AS	OPT_2	OPT_1	OPT_0	Meaning
x	0	0	0	Word-length access
x	0	0	1	Byte access
x	0	1	0	Half-word access
0	1	0	0	Instruction ROM access (as data)
0	1	0	1	Cache control
0	1	1	0	ADAPT29K accesses
	-all others -			Reserved

Note that some of these encodings do not affect processor operation, and could have other interpretations in a particular system. For example, the OPT values 000, 001, and 010 affect processor operation only if the DW bit of the Configuration Register is 1. However, nonstandard uses of the OPT field have an implication on the portability of software between different systems.

Addressing and Alignment

Address Spaces

External instructions and data are contained in one of four 32-bit address spaces:

1. Instruction/Data Memory
2. Input/Output
3. Coprocessor
4. Instruction Read-Only Memory (Instruction ROM).

An address in the instruction/data memory address space may be treated as virtual or physical, as determined by the Current Processor Status Register. Address translation for data accesses is enabled separately from address translation for instruction accesses. A program in the Supervisor mode may temporarily disable address translation for individual loads and stores; this permits load-real and store-real operations.

It is possible to partition physical instruction and data addresses into two separate physical address spaces. However, virtual instruction and data addresses appear in the same virtual address space (i.e., instruction/data memory).

The coprocessor address space is not an address space in the strictest sense. The coprocessor address space is defined so that transfers of operands and operation codes to the coprocessor do not interfere with other external devices and memories.

The processor does not directly support the access of the instruction ROM address space using loads and stores; this capability is defined as a system option requiring external hardware.

For untranslated data accesses, bits contained in load and store instructions distinguish between the instruction/data memory, input/output, and coprocessor address spaces. For translated data accesses, the Input/Output bit of the associated TLB entry distinguishes between the instruction/data memory and input/output address spaces.

For instruction fetches, the ROM Enable (RE) bit of the Current Processor Status Register distinguishes between the instruction/data and instruction ROM address spaces.

Byte and Half-Word Addressing

The Am29000 generates word-oriented byte addresses for accesses to external devices and memories. Addresses are word-oriented because loads, stores, and instruction fetches access words. However, addresses are byte addresses because they are sufficient to select bytes packed within accessed words. For load and store operations, the processor provides means for using the least-significant address bits to access bytes and half-words within external words.

The selection of a byte within a word is determined by the two least-significant bits of an address and the Byte Order (BO) bit of the Configuration Register. The selection of a half-word within a word is determined by the next-to-least-significant bit of an address and the BO bit. Figure 51 illustrates the addressing of bytes and half-words when the BO bit is 0, and Figure 52 illustrates the addressing of bytes and half-words when the BO bit is 1. In Figure 51 and Figure 52, addresses are represented in hexadecimal notation.

In the processor, the two least-significant bits of an external address can be reflected in the Byte Pointer (BP) field of the ALU Status Register when the DW bit of the Configuration Register is 0. Alternatively, the two least-significant bits of the address can be used to control byte and half-word accesses when the DW bit is 1. The BO bit affects only the interpretation of the BP field and the two least-significant address bits.

If the BO bit is 0, bytes are ordered within words such that a 00 in the BP field or in the two least-significant address bits selects the high-order byte of a word, and a 11 selects the low-order byte. If the BO bit is 1, a 00 in the BP field or in the two least-significant address bits selects the low-order byte of a word, and a 11 selects the high-order byte.

If the BO bit is 0, half-words are ordered within words such that a 0 in the most-significant bit of the BP field or the next-to-least-significant address bit selects the high-order half-word, and a 1 selects the low-order half-word. If the BO bit is 1, a 0 in the most-significant bit of the BP field or the next-to-least-significant address bit selects the low-order half-word of a word, and a 1 selects the high-order half-word. Note that since the least-significant bit of the BP field or an address does not participate in the selection of half-words, the alignment of half-words is forced to half-word boundaries in this case.

Alignment of Words and Half-Words

Since only byte addressing is supported, it is possible that an address for the access of a word or half-word is not aligned to the desired word or half-word. The Am29000 either ignores or forces alignment in most cases. However, some systems may require that unaligned accesses be supported for compatibility reasons. Because of this, the Am29000 provides an option that creates a trap when a nonaligned access is attempted. This trap allows software emulation of the nonaligned accesses in a manner that is appropriate for the particular system.

The detection of unaligned accesses is activated by a 1 in the Trap Unaligned Access (TU) bit of the Current Processor Status Register. Unaligned access detection is based on the data length as indicated by the OPT field of a load or store instruction, and on the two least-significant bits of the specified address. Only addresses for instruction/data memory accesses are checked; align-

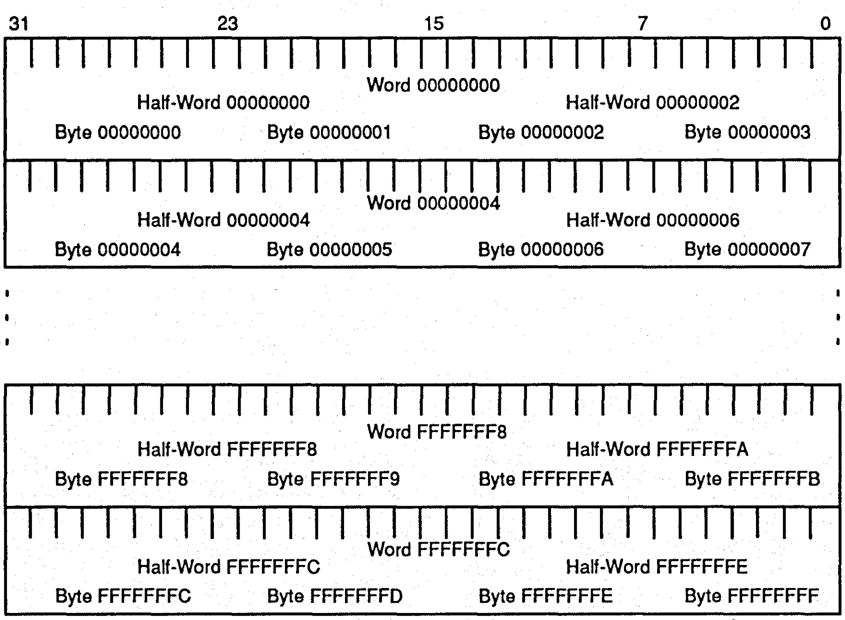


Figure 51. Byte and Half-Word Addressing with BO = 0

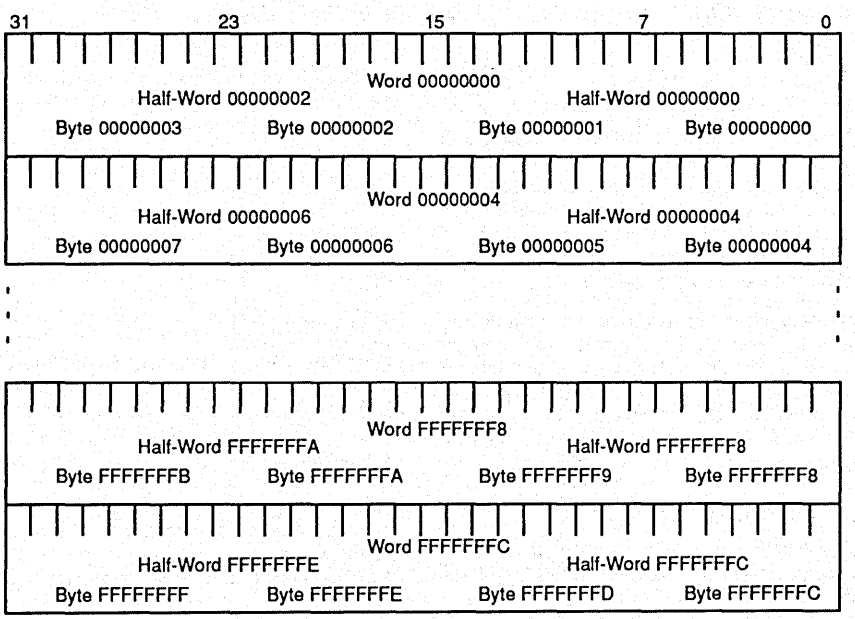


Figure 52. Byte and Half-Word Addressing with BO = 1

ment is ignored for input/output accesses and coprocessor transfers.

An Unaligned Access trap occurs only if the TU bit is 1 and any of the following combinations of OPT field and address bits is detected for a load or store to instruction/data memory:

OPT_2	OPT_1	OPT_0	A_1	A_0	
0	0	0	1	0	Unaligned
0	0	0	0	1	word access
0	0	0	1	1	
0	1	0	0	1	Unaligned
0	1	0	1	1	half-word access

The trap handler for the Unaligned Access trap is responsible for generating the correct sequence of aligned accesses and performing any necessary shifting, masking and/or merging. Note that a virtual page-boundary crossing also may have to be considered.

Alignment of Instructions

In the Am29000, all instructions are 32 bits in length, and are aligned on word-address boundaries. The processor's Program Counter is 30 bits in length, and the least-significant 2 bits of processor-generated instruction addresses are always 00. An unaligned address can be generated by indirect jumps and calls. However, alignment is ignored by the processor in this case, and it expects the system to force alignment (i.e., by interpreting the two least-significant address bits as 00, regardless of their values).

Accessing Instructions as Data

To aid the external access of instructions and data on separate buses, the processor distinguishes between instruction and data accesses. However, it does not support a logical distinction between instruction and data address spaces (except in the case of instruction read-only memory). In particular, address translation in the Memory Management Unit is in no way affected by this distinction (although memory protection is).

In systems where it is necessary to access instructions as data, this function should be performed via the shared address space. The OPT field provides a means for loads to access instructions in the instruction read-only memory (ROM) address space. The Am29000 does not take any action to prevent a store to the instruction ROM address space.

Byte and Half-Word Accesses

The Am29000 can perform byte and half-word accesses in either software or hardware under control of the Data Width Enable (DW) bit of the Configuration Register. Software byte and half-word accesses are selected by a DW bit of 0, and hardware byte and half-word accesses are selected by a DW bit of 1. Software byte and half-word accesses are less efficient than hardware byte and half-word accesses, but hardware accesses require that the system be able to selectively write individual byte and half-word positions within external devices and memories. The software-only technique is compatible with systems designed to provide hardware support for byte and half-word accesses.

This section describes the operation of both software and hardware byte and half-word accesses. Byte and half-word accesses operate as described here for memory and input/output accesses, but not for coprocessor transfers. Coprocessor transfers are unaffected by the DW bit.

The DW bit is cleared by a processor reset. It must explicitly be set to 1 by software before hardware byte and half-word accesses can be performed.

Software Byte and Half-Word Accesses

If the DW bit is 0, the Am29000 allows the Byte Pointer Register to be set with the least-significant bits of an address specified by any load or store instruction, except those that transfer information to and from the coprocessor. Insert and extract instructions can then be used to access the byte or half-word of interest, after the external word has been accessed. This provides a general-purpose mechanism for manipulating external byte and half-word data, without the need for external hardware support.

To load a byte or half-word, a word load is first performed. This load sets the BP field with the two least-significant bits of the address. A subsequent EXBYTE, EXHW, or EXHWS instruction extracts the byte or half-word of interest from the accessed word.

To store a byte or half-word, a load is first performed, setting the BP field with the two least-significant bits of the address. A subsequent INBYTE or INHW instruction inserts the byte or half-word of interest into the accessed word, and the resulting word is then stored.

Software that relies on loads and stores setting the BP field cannot operate correctly when the Freeze (FZ) bit of the Current Processor Status Register is 1, because the ALU Status Register is frozen.

Hardware Byte and Half-Word Accesses

If the DW bit is 1 on a load, the Am29000 selects a byte or half-word from the loaded word depending on the Option (OPT) bits of the load instruction, the Byte Order (BO) bit of the Configuration Register, and the two least-significant bits of the address (for bytes) or the next-to-least-significant bit of the address (for half-words). The selected byte or half-word is right-justified within the destination register. If the SB bit of the load instruction is 0, the remainder of the destination register is zero-extended. If the SB bit is 1, the remainder of the destination register is sign-extended with the sign bit of the selected byte or half-word.

If the DW bit is 1 on a store, the Am29000 replicates the low-order byte or half-word in the source register into every byte and half-word position of the stored word. The system is responsible for generating the appropriate byte and/or half-word strobes, based on the OPT_2–OPT_0 signals and the two least-significant bits of the address, to write the appropriate byte or half-word in the selected device or memory (the system byte order must also be considered). The SB bit does not affect the operation of a store, except for setting the BP field as described below.

If the SB bit is 1 for either a load or store and the DW bit is also 1, both bits of the BP field are set to the complement of the BO bit when the load or store is executed. This does not directly affect the load or store access, but supports compatibility for software developed for word-write-only systems. Hardware byte and half-word accesses—in contrast to software byte and half-word accesses—can be performed when the FZ bit is 1, because these accesses do not rely on the BP field.

System Alternatives and Compatibility

The two mechanisms for performing byte and half-word accesses create the possibility of two types of systems. These are named for convenience:

- Type 1: simple, word-only accesses in external devices and memories; software byte and half-word accesses.
- Type 2: byte/half-word strobes in external devices and memories; hardware byte and half-word accesses by the Am29000.

The provision for hardware byte and half-word accesses encourages Type 2 systems. Software for Type 1 systems can execute on Type 2 systems, but the reverse is not true. Software compatibility is possible primarily because of the DW bit and because the Am29000 sets the BP field with an appropriate byte pointer even when it performs byte and half-word accesses with internal hardware. Also, the system must return a full word in either type of system, regardless of the access data-width. The DW bit must be 0 in Type 1 systems and must be 1 in Type 2 systems. To illustrate compatibility between systems, consider the following steps of an unsigned byte load compiled for a Type 1 system, but executing on a Type 2 system:

1. Perform a load with OPT = 001 and SB = 1.
   - Type 1 system: The addressed word is accessed and placed into the destination register. The BP field is set with the two least-significant bits of the address.
   - Type 2 system: The addressed byte is accessed, aligned, padded, and placed into the destination register. The BP field is set to point to the low-order byte, reflecting the alignment that has been performed (the pointer depends on the value of the BO bit).
2. Perform a byte extract on the loaded word.
   - Type 1 system: The byte selected by the BP field is aligned to the low-order byte of the destination register and the remainder of the word is zero-extended. The selected byte may be in any byte position.
   - Type 2 system: The byte selected by the BP field (set to point to the low-order byte) is aligned to the low-order byte of the destination register and the remainder of the word is zero-extended. (Note that the selected byte was already in the low-order byte position. This operation does not change the program state but merely allows software compatibility.)

The recommended instruction sequences for all types of byte and half-word accesses and for both types of systems are enumerated below. Compatibility between these systems follows the above example, but for brevity, compatibility is not described in detail here.

Byte read, unsigned:

Type 1	Comments
load 0,17,temp,addr	; OPT = 001, SB = 1
exbyte temp,temp,0	; get byte

Type 2	Comments
load 0,1,temp,addr	; OPT = 001, SB = 0

Byte read, signed:

Type 1	Comments
load 0,17,temp,addr	; OPT = 001, SB = 1
exbyte temp,temp,0	; get byte
sll temp,temp,24	; sign extend
sra temp,temp,24	

Type 2	Comments
load 0,17,temp,addr	; OPT = 001, SB = 1 (sign extended)

Byte Write:

Type 1	Comments
load 0,17,temp,addr	; OPT = 001, SB = 1
inbyte temp,temp, data	; insert byte
store 0,1,temp,addr	; store

Type 2	Comments
store 0,1,data,addr	; OPT = 001, SB = 0

Half-word read, unsigned:

Type 1	Comments
load 0,18,temp,addr	; OPT = 010, SB = 1
exhw temp,temp,0	; get half-word unsigned

Type 2	Comments
load 0,2,temp,addr	; OPT = 010, SB = 0

Half-word read, signed:

Type 1	Comments
load 0,18,temp,addr	; OPT = 010, SB = 1
exhws temp,temp	; get half-word sign-extend

Type 2	Comments
load 0,18,temp,addr	; OPT = 010, SB = 1, (sign-extend)

Half-word write:

Type 1	Comments
load 0,18,temp,addr	; OPT = 010, SB = 1
inhw temp,temp,data	; insert half-word
store 0,2,temp,addr	; store

Type 2	Comments
store 0,2,data,addr	; OPT = 010, SB = 0

INTERRUPTS AND TRAPS

Interrupts and traps cause the Am29000 to suspend the execution of an instruction sequence and to begin the execution of a new sequence. The processor may or may not later resume the execution of the original instruction sequence.

The distinction between interrupts and traps is largely one of causation and enabling. Interrupts allow external devices and the Timer Facility to control processor execution, and are always asynchronous to program execution. Traps are intended to be used for certain exceptional events that occur during instruction execution, and are generally synchronous to program execution.

Throughout this manual, a distinction is made between the point at which an interrupt or trap occurs and the point at which it is taken. An interrupt or trap is said to occur when all conditions that define the interrupt or trap are met. However, an interrupt or trap that occurs is not necessarily recognized by the processor, either because of various enables or because of the processor's operational mode (e.g., Halt mode). An interrupt or trap is taken when the processor recognizes the interrupt or trap and alters its behavior accordingly.

Interrupts

Interrupts are caused by signals applied to any of the external inputs $\overline{INTR_3}$–$\overline{INTR_0}$, or by the Timer Facility. The processor may be disabled from taking certain interrupts by the masking capability provided by the Disable All Interrupts and Traps (DA) bit, Disable Interrupts (DI) bit, and Interrupt Mask (IM) field in the Current Processor Status Register.

The DA bit disables all interrupts and most traps. The DI bit disables external interrupts without affecting the recognition of traps and Timer interrupts. The 2-bit IM field selectively enables external interrupts as follows:

IM Value	Result
0 0	$\overline{INTR_0}$ enabled
0 1	$\overline{INTR_1}$–$\overline{INTR_0}$ enabled
1 0	$\overline{INTR_2}$–$\overline{INTR_0}$ enabled
1 1	$\overline{INTR_3}$–$\overline{INTR_0}$ enabled

Note that the $\overline{INTR_0}$ interrupt cannot be disabled by the IM field. Also, note that no external interrupt is taken if either the DA or DI bit is 1. The Interrupt Pending bit in the Current Processor Status indicates that one or more of the signals $\overline{INTR_3}$–$\overline{INTR_0}$ is active, but that the corresponding interrupt is disabled due to the value of either DA, DI, or IM.

Traps

Traps are caused by signals applied to one of the inputs $\overline{TRAP_1}$–$\overline{TRAP_0}$, or by exceptional conditions such as protection violations. Except for the Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps, traps are disabled by the DA bit in the Current Processor Status; a 1 in the DA bit disables traps, and a 0 enables traps. It is not possible to selectively disable individual traps.

Wait Mode

A wait-for-interrupt capability is provided by the Wait mode. The processor is in the Wait mode whenever the Wait Mode (WM) bit of the Current Processor Status is 1. While in Wait mode, the processor neither fetches nor executes instructions and performs no external accesses. The Wait mode is exited when an interrupt or trap is taken.

Note that the processor can take only those interrupts or traps for which it is enabled, even in the Wait mode. For example, if the processor is in the Wait mode with a DA bit of 1, it can leave the Wait mode only via the Reset mode or a $\overline{WARN}$ trap.

Vector Area

Interrupt and trap processing rely on the existence of a user-managed Vector Area in external instruction/data memory or instruction read-only memory (instruction ROM). The Vector Area begins at an address specified by the Vector Area Base Address Register, and provides for as many as 256 different interrupt and trap handling routines. The processor reserves 24 routines for system operation and 40 routines for instruction emulation. The number and definition of the remaining 192 possible routines are system-dependent.

The Vector Area has one of two possible structures as determined by the Vector Fetch (VF) bit in the Configuration Register. The first structure, as described below, requires less external memory than the second, but imposes the performance penalty of the vector-table lookup.

If the VF bit is 1, the structure of the Vector Area is a table of vectors in instruction/data memory. The layout of a single vector is shown in Figure 53. Each vector gives the beginning word-address of the associated interrupt or trap handling routine, and specifies, by the R bit, whether the routine is contained in instruction/data memory (R = 0) or instruction ROM (R = 1).

If the VF bit is 0, the structure of the Vector Area is a segment of contiguous blocks of instructions in instruction/data memory or instruction ROM. The ROM Vector Area (RV) bit of the Configuration Register determines whether the Vector Area is in instruction/data memory (RV = 0) or instruction ROM (RV = 1). A 64-instruction block contains exactly one interrupt or trap handling routine, and blocks are aligned on 64-instruction address boundaries.

Vector Numbers

When an interrupt or trap is taken, the processor determines an 8-bit vector number associated with the interrupt or trap. The vector number gives either the number

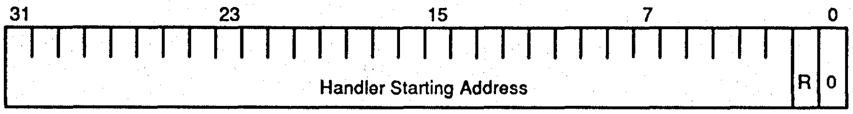


Figure 53. Vector Table Entry

of a vector table entry or the number of an instruction block, depending on the value of the VF bit.

If the VF bit is 1, the physical address of the vector table entry is generated by replacing bits 9–2 of the value in the Vector Area Base Address Register with the vector number.

If the VF bit is 0, the physical address of the first instruction of the handling routine is generated by replacing bits 15–8 of the value in the Vector Table Base Address Register with the vector number.

Vector numbers are either predefined or specified by an instruction causing the trap. The assignment of vector numbers is shown in Figure 54 (vector numbers are in decimal notation). Vector numbers 64 to 255 are for use by trapping instructions; the definition of the routines associated with these numbers is system-dependent.

Interrupt and Trap Handling

Interrupt and trap handling consists of two distinct operations: taking the interrupt or trap, and returning from the interrupt or trap handler. If the interrupt or trap handler returns directly to the interrupted routine, the interrupt or trap handler need not save and restore processor state.

Taking an Interrupt or Trap

The following operations are performed in sequence by the processor when an interrupt or trap is taken:

1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any in-progress load or store operation is completed. Any additional operations are canceled in the case of Load Multiple and Store Multiple.
4. The contents of the Current Processor Status Register are copied into the Old Processor Status Register.
5. The Current Processor Status register is modified as shown in Figure 55 (the value "u" means unaffected). Note that setting the Freeze (FZ) bit freezes the Channel Address, Channel Data, Channel Control, Program Counter 0, Program Counter 1, Program Counter 2, and ALU Status Registers.
6. The address of the first instruction of the interrupt or trap handler is determined. If the VF bit of the Configuration Register is 1, the address is obtained by accessing a vector from instruction/data memory, using the physical address obtained from the Vector Area Base Address Register and the vector number. This access appears on the channel as a data access, and the OPT_2–OPT_0 signals indicate a word-length access. If the VF bit is 0, the instruction address is given directly by the Vector Area Base Address Register and the vector number.
7. If the VF bit is 1, the R bit in the vector fetched in Step 6 is copied into the RE bit of the Current Processor Status Register. If the VF bit is 0, the RV bit of the Configuration Register is copied into the RE bit. This step determines whether or not the first instruction of the interrupt handler is in instruction ROM.
8. An instruction fetch is initiated using the instruction address determined in Step 6. At this point, normal instruction execution resumes.

Note that the processor does not explicitly save the contents of any registers when an interrupt is taken. If register saving is required, it is the responsibility of the interrupt or trap-handling routine. For proper operation, registers must be saved before any further interrupts or traps may be taken. The FZ bit must be reset at least two instructions before interrupts or traps are reenabled to allow the program state to be reflected properly in processor registers if an interrupt or trap is taken.

Returning from an Interrupt or Trap

Two instructions are used to resume the execution of an interrupted program: Interrupt Return (IRET), and Interrupt Return and Invalidate (IRETINV). These instructions are identical except in one respect: the IRETINV instruction resets all Valid bits in the Branch Target Cache, whereas the IRET instruction does not affect the Valid bits.

In some situations, the processor state must be set properly by software before the interrupt return is executed. The following is a list of operations normally performed in such cases:

1. The Current Processor Status is configured as shown in Figure 55 (the value "x" is a "don't care"). Note that setting the FZ bit freezes the registers listed below so that they may be set for the interrupt return.

Number	Type of Trap or Interrupt	Cause
0	Illegal Opcode	executing undefined instruction
1	Unaligned Access	access on unnatural boundary, TU = 1
2	Out of Range	overflow or underflow
3	Coprocessor Not Present	coprocessor access, CP = 0
4	Coprocessor Exception	coprocessor $\overline{DERR}$ response
5	Protection Violation	invalid User-mode operation
6	Instruction Access Exception	$\overline{IERR}$ response
7	Data Access Exception	$\overline{DERR}$ response, not coprocessor
8	User-Mode Instruction TLB Miss	no TLB entry for translation
9	User-Mode Data TLB Miss	"
10	Supervisor-Mode Instruction TLB Miss	"
11	Supervisor-Mode Data TLB Miss	"
12	Instruction TLB Protection Violation	TLB UE/SE = 0
13	Data TLB Protection Violation	TLB UR/SR = 0, UW/SW = 0 on write
14	Timer	Timer Facility
15	Trace	Trace Facility
16	$\overline{INTR_0}$	$\overline{INTR_0}$ input
17	$\overline{INTR_1}$	$\overline{INTR_1}$ input
18	$\overline{INTR_2}$	$\overline{INTR_2}$ input
19	$\overline{INTR_3}$	$\overline{INTR_3}$ input
20	$\overline{TRAP_0}$	$\overline{TRAP_0}$ input
21	$\overline{TRAP_1}$	$\overline{TRAP_1}$ input
22	Floating-Point Exception	unmasked floating-point exception
23	reserved	
24–29	reserved for instruction emulation (op codes D8-DD)	
30	MULTM	MULTM instruction
31	MULTMU	MULTMU instruction
32	MULTIPLY	MULTIPLY instruction
33	DIVIDE	DIVIDE instruction
34	MULTIPLU	MULTIPLU instruction
35	DIVIDU	DIVIDU instruction
36	CONVERT	CONVERT instruction
37	SQRT	SQRT instruction
38	CLASS	CLASS instruction
39–41	reserved for instruction emulation (op codes E7–E9)	
42	FEQ	FEQ instruction
43	DEQ	DEQ instruction
44	FGT	FGT instruction
45	DGT	DGT instruction
46	FGE	FGE instruction
47	DGE	DGE instruction
48	FADD	FADD instruction
49	DADD	DADD instruction
50	FSUB	FSUB instruction
51	DSUB	DSUB instruction
52	FMUL	FMUL instruction
53	DMUL	DMUL instruction
54	FDIV	FDIV instruction
55	DDIV	DDIV instruction
56	reserved for instruction emulation (op code F8)	
57	FDMUL	FDMUL instruction
58–63	reserved for instruction emulation (op codes FA-FF)	
64–255	Assert and EMULATE instruction traps (vector number specified by instruction)	

Figure 54. Vector Number Assignments

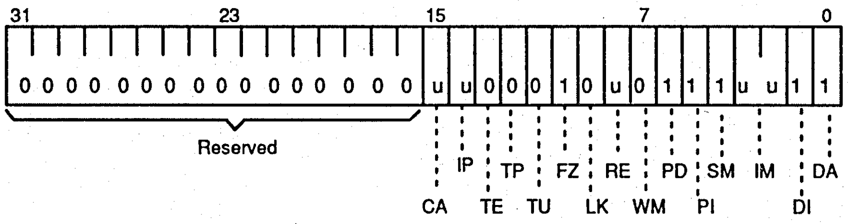

31
23
15
7
0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
u u 0 0 0 1 0 u 0 1 1 1 u u 1 1
Reserved
CA IP TE TP TU FZ LK RE WM PD PI SM IM DI DA

Figure 55. Current Processor Status after an Interrupt or Trap

2. The Old Processor Status is set to the value of the Current Processor Status for the target routine.

3. The Channel Address, Channel Data, and Channel Control registers are set to restart or resume uncompleted channel operations of the target routine.

4. The Program Counter 1 and Program Counter 0 registers are set to the addresses of the first and second instructions, respectively, to be executed in the target routine.

5. Other registers are set as required. These may include registers such as the ALU Status, Q, and so forth, depending on the particular situation. Some of these registers are unaffected by the FZ bit, so they must be set in such a manner that they are not modified unintentionally before the interrupt return.

Once the processor registers are configured properly, as described above, an interrupt return instruction (IRET or IRETINV) performs the remaining steps necessary to return to the target routine. The following operations are performed by the interrupt return instruction:

1. Any in-progress load or store operation is completed. If a Load Multiple or Store Multiple sequence is in progress, the interrupt return is not executed until the sequence is completed.

2. Interrupts and traps are disabled, regardless of the settings of the DA, DI, and IM fields of the Current Processor Status, for Steps 3 through 10.

3. If the interrupt return instruction is an IRETINV, all Valid bits in the Branch Target Cache are reset.

4. The contents of the Old Processor Status Register are copied into the Current Processor Status Register. This normally resets the FZ bit allowing the Program Counter 0, 1, 2, Channel Address, Data, Control, and ALU Status registers to update normally. Since certain bits of the Current Processor Status Register always are updated by the processor, this copy operation may be irrelevant for certain bits (e.g., the Interrupt Pending bit).

5. If the Contents Valid (CV) bit of the Channel Control Register is 1, and the Not Needed (NN) and Multiple Operation (ML) bits are both 0, an external access is started. This operation is based on the contents of the Channel Address, Channel Data, and Channel Control registers. The Current Processor Status Register conditions the access—as is normally the case. Note that Load Multiple and Store Multiple operations are not restarted at this point.

6. The address in Program Counter 1 is used to fetch an instruction. The Current Processor Status Register conditions the fetch. This step is treated as a branch in the sense that the proces-

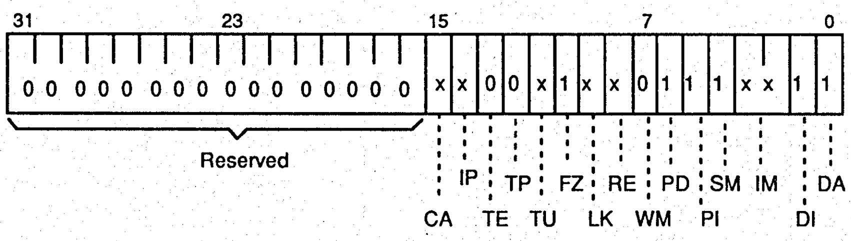

31
23
15
7
0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
x x 0 0 x 1 x x 0 1 1 1 x x 1 1
Reserved
CA IP TE TP TU FZ LK RE WM PD PI SM IM DI DA

Figure 56. Current Processor Status Before Interrupt Return

sor searches the Branch Target Cache for the target of the fetch.

7. The instruction fetched in Step 6 enters the decode stage of the pipeline.
8. The address in Program Counter 0 is used to fetch an instruction. The Current Processor Status Register conditions the fetch. This step is treated as a branch in the sense that the processor searches the Branch Target Cache for the target of the fetch.
9. The instruction fetched in Step 6 enters the execute stage of the pipeline, and the instruction fetched in Step 8 enters the decode stage.
10. If the CV bit in the Channel Control Register is a 1, the NN bit is 0, and the ML bit is 1, a Load Multiple or Store Multiple sequence is started, based on the contents of the Channel Address, Channel Data, and Channel Control registers.
11. Interrupts and traps are enabled per the appropriate bits in the Current Processor Status Register.
12. The processor resumes normal operation.

Fast Interrupt Processing

The registers affected by the FZ bit of the Current Processor Status Register are those that are modified by almost any usual sequence of instructions. Since the FZ bit is set by an interrupt or trap, the interrupt or trap handler is able to execute while not disturbing the state of the interrupted routine, though its execution is somewhat restricted. Thus, it is not necessary in many cases for the interrupt or trap handler to save the registers that are affected by the FZ bit.

The processor provides an additional benefit if the Program Counter 0 and Program Counter 1 registers are not modified by the interrupt or trap handler. If Program Counters 0 and 1 contain the addresses of sequential instructions when an interrupt or trap is taken, and if they are not modified before an interrupt return is executed, Step 8 of the interrupt return sequence above occurs as a sequential fetch—instead of a branch—for the interrupt return. The performance impact of a sequential fetch is normally less than that of a nonsequential fetch.

Because the registers affected by the FZ bit are sometimes required for instruction execution, it is not possible for the interrupt or trap handler to execute all instructions unless the required registers are first saved elsewhere (e.g., in one or more global registers). Most of the restrictions due to register dependencies are obvious (e.g., the Byte Pointer for byte extracts), and will not be discussed here. Other less obvious restrictions are listed below:

1. Load Multiple and Store Multiple. The Channel Address, Channel Data, and Channel Control registers are used to sequence Load Multiple and Store Multiple operations, so these instructions cannot be executed while the registers are frozen. However, note that other external accesses may occur; the Channel Address, Channel Data, and Channel Control registers are required only to restart an access after an exception, and the interrupt or trap handler is not expected to encounter any exceptions.
2. Loads and stores that set the Byte Pointer. If the Set Byte Pointer (SB) of a load or store instruction is 1 and the FZ bit is also 1, there is no effect on the Byte Pointer. Thus, the execution of external byte and half-word accesses using this mechanism is not possible.
3. Extended arithmetic. The Carry bit of the ALU Status Register is not updated while the FZ bit is 1.
4. Divide step instructions. The Divide Flag of the ALU Status Register is not updated when the FZ bit is 1.

If the interrupt or trap handler does not save the state of the interrupted routine, it cannot allow additional interrupts and traps. Also, the operation of the interrupt or trap handler cannot depend on any trapping instructions (e.g., Floating-Point instructions, illegal operation codes, arithmetic overflow, etc.) since these are disabled. There are certain cases, however, where traps are unavoidable; these are discussed in the Arithmetic Exceptions section.

$\overline{\text{WARN}}$ Trap

The processor recognizes a special trap, caused by the activation of the $\overline{\text{WARN}}$ input, that cannot be masked. The $\overline{\text{WARN}}$ trap is intended to be used for severe system-error or deadlock conditions. It allows the processor to be placed in a known, operable state, while preserving much of its original state for error reporting and possible recovery. Therefore, it shares some features in common with the Reset mode as well as features common to other traps described in this section.

The major differences between the $\overline{\text{WARN}}$ trap and other traps are:

1. The processor does not wait for an in-progress external access to be completed before taking the trap, since this access might not be completed. However, the information related to any outstanding access is retained by the Channel Address, Channel Data, and Channel Control registers when the trap is taken.
2. The vector-fetch operation is not performed, regardless of the VF bit of the Configuration Register, when the $\overline{\text{WARN}}$ trap is taken. Instead, the ROM Enable (RE) bit in the Current Processor Status is set, and instruction fetching begins immediately at Address 16 in the instruction ROM.

The trap handler executes directly from the instruction ROM without the need to access external (and possibly nonfunctional or invalid) instruction/data memory.

Note that $\overline{\text{WARN}}$ trap may disrupt the state of the routine that is executing when it is taken, prohibiting this routine from being restarted.

Sequencing of Interrupts and Traps

On every cycle, the processor decides either to execute instructions or to take an interrupt or trap. Since there are multiple sources of interrupts and traps, more than one interrupt or trap may be pending on a given cycle.

To resolve conflicts, interrupts and traps are taken according to the priority shown in Figure 57. In this table, interrupts and traps are listed in order of decreasing priority. This section discusses the first three columns of Figure 57. The last two columns are discussed in the Exception Reporting and Restarting section.

In Figure 57, interrupts and traps fall into one of two categories depending on the timing of their occurrence relative to instruction execution. These categories are indicated in the third column by the labels "inst" and "async." These labels have the following meanings:

1. Inst—Generated by the execution or attempted execution of an instruction.
2. Async—Generated asynchronous to and independent of the instruction being executed, although it may be a result of an instruction executed previously.

The principle for interrupt and trap sequencing is that the highest priority interrupt or trap is taken first. Other interrupts and traps remain active until they can be taken, or are regenerated when they can be taken. This is accomplished, depending on the type of interrupt or trap, as follows:

1. All traps in Figure 57 with Priority 13 or 14 are regenerated by the re-execution of the causing instruction.
2. Most of the interrupts and traps of Priorities 4 through 12 must be held by external hardware until they are taken. The exceptions to this are listed in (3) below.
3. The exceptions to (2) above are the Data Access Exception trap, the Coprocessor Exception trap, the Timer interrupt, and the Trace trap. These are caused by bits in various registers in the processor and are held by these registers until taken or cleared. The relevant bits are: the Transaction Faulted (TF) bit of the Channel Control Register for Data Access Exception and Coprocessor Exception traps, the Interrupt (IN) bit of the Timer Reload Register for Timer interrupts, and the Trace Pending (TP) bit of the Current Processor Status Register for Trace traps.
4. All traps of Priorities 2 and 3 in Figure 57, except for the Unaligned Access trap, are not regenerated. These traps are mutually exclusive and are given high priority because they cannot be regenerated; they must be taken if they occur. If one of these traps occurs at the same time as a reset or $\overline{\text{WARN}}$ trap, it is not taken, and its occurrence is lost.
5. The Unaligned Access trap is regenerated internally when an external access is restarted by the Channel Address, Channel Data, and Channel Control registers. Note that this trap is not necessarily exclusive to the traps discussed in (4) above.

Note that the Channel Address, Channel Data, and Channel Control registers are set for a $\overline{\text{WARN}}$ trap only if an external access is in progress when the trap is taken.

Exception Reporting and Restarting

When an instruction encounters an exceptional condition, the Program Counter 0, Program Counter 1, and Program Counter 2 registers report the relevant instruction address(es), and allow the instruction sequence to be restarted once the exceptional condition has been remedied (if possible). Similarly, when an external access or coprocessor transfer encounters an exceptional condition, the Channel Address, Channel Data, and Channel Control registers report information on the access or transfer, and allow it to be restarted. This section describes the interpretation and use of these registers.

The "PC1" column in Figure 57 describes the value held in the Program Counter 1 Register (PC1) when the interrupt or trap is taken. For traps in the "inst" category, PC1 contains either the address of the instruction causing the trap, indicated by "curr," or the address of the instruction following the instruction causing the trap, indicated by "next."

For interrupts and traps in the "async" category, PC1 contains the address of the first instruction, which was not executed due to the taking of the interrupt or trap. This is the next instruction to be executed upon interrupt return, as indicated by "next" in the PC1 column.

Instruction Exceptions

For traps caused by the execution of an instruction (e.g., the Out of Range trap), the Program Counter 2 Register contains the address of the instruction causing the trap. In all of these cases, PC1 is in the "next" category. The Exception Opcode Register contains the operation code of the instruction causing the trap.

The traps associated with instruction fetches (i.e., those of Priority 13) occur only if the processor attempts the execution of the associated instruction. An exception

Priority	Type Of Interrupt Or Trap	Inst/Async	PC1	Channel Regs
1 (highest)	$\overline{\text{WARN}}$	async	next	see Note 1
2	User-Mode Data TLB Miss	inst	next	all
	Supervisor-Mode Data TLB Miss	inst	next	all
	Data TLB Protection Violation	inst	next	all
3	Unaligned Access	inst	next	all
	Coprocessor not Present	inst	next	all
	Out of Range	inst	next	N/A
	Floating-Point Exceptions	inst	next	N/A
	Assert Instructions	inst	next	N/A
	Floating-Point Instructions	inst	next	N/A
	MULTIPLY	inst	next	N/A
	MULTM	inst	next	N/A
	DIVIDE	inst	next	N/A
	MULTIPLU	inst	next	N/A
	MULTMU	inst	next	N/A
	DIVIDU	inst	next	N/A
	EMULATE	inst	next	N/A
4	Data Access Exception	async	next	all
	Coprocessor Exception	async	next	all
5	$\overline{\text{TRAP}}_0$	async	next	multiple
6	$\overline{\text{TRAP}}_1$	async	next	multiple
7	$\overline{\text{INTR}}_0$	async	next	multiple
8	$\overline{\text{INTR}}_1$	async	next	multiple
9	$\overline{\text{INTR}}_2$	async	next	multiple
10	$\overline{\text{INTR}}_3$	async	next	multiple
11	Timer	async	next	multiple
12	Trace	async	next	multiple
13	User-Mode Instruction TLB Miss	inst	curr	N/A
	Supervisor-Mode Instr. TLB Miss	inst	curr	N/A
	Instruction TLB Protection Violation	inst	curr	N/A
	Instruction Access Violation	inst	curr	N/A
14 (lowest)	Illegal Opcode	inst	curr	N/A
	Protection Violation	inst	curr	N/A

Note: The Channel Address, Channel Data, and Channel Control registers are set for a $\overline{\text{WARN}}$ trap only if an external access is in progress when the trap is taken.

Figure 57. Interrupt and Trap Priority Table

may be detected during an instruction prefetch, but the associated trap does not occur if a nonsequential fetch occurs before the processor attempts the execution of the invalid instruction. This prevents the spurious indication of instruction exceptions.

Data Exceptions

The "Channel Regs" column of Figure 57 indicates the cases for which the Channel Address, Channel Data, and Channel Control registers contain information related to an external access or coprocessor transfer (these registers collectively are termed "channel registers" in the following discussion). For the cases indicated, the access or transfer was not completed because of some exceptional condition. Note that the Channel Data Register contains relevant information only in the case of a store.

For the $\overline{\text{WARN}}$ trap, the channel registers are valid only if a load or store were in progress when the trap was taken. Recall that the $\overline{\text{WARN}}$ trap does not wait for any in-progress access to be completed.

For the traps with an "all" in the "Channel Regs" column of Figure 57, the channel registers contain information relevant to the trap in all cases. These traps are associated with exceptional events during external accesses or coprocessor transfers.

For the traps with a "multiple" in the "Channel Regs" column, the channel registers might contain information for restarting an interrupted Load Multiple or Store Multiple operation. In these cases, the operation did not encounter an exception, but was simply canceled for latency considerations.

The information contained in the channel registers allows the processor to restart the related operation during an interrupt return sequence, without any special assistance by software. Software must only ensure that the relevant information is retained in, or restored to, the channel registers before an interrupt return is executed.

Arithmetic Exceptions

Integer and floating-point instructions can cause Out of Range or Floating-Point Exception traps, respectively, if an exception is detected during the arithmetic operation. This section describes the conditions under which these traps occur and the additional operations performed beyond those described in the Interrupt and Trap Handling section.

Integer Exceptions

Some integer add and subtract instructions—ADDS, ADDU, ADDCS, ADDCU, SUBS, SUBU, SUBCS, SUBCU, SUBRS, SUBRU, SUBRCS, and SUBRCU—cause an Out of Range trap upon overflow or underflow of a 32-bit signed or unsigned result, depending on the instruction.

Two integer multiply instructions—MULTIPLY and MULTIPLU—cause an Out of Range trap upon overflow of a 32-bit signed or unsigned result, respectively, if the MO bit of the Integer Environment Register is 0. If the MO bit is 1, these multiply instructions cannot cause an Out of Range trap.

Two integer divide instructions—DIVIDE and DIVIDU—take the Out of Range trap upon overflow of a 32-bit signed or unsigned result, respectively, if the DO bit of the Integer Environment Register is 0. If the DO bit is 1, the divide instructions cannot cause an Out of Range trap unless the divisor is 0. If the divisor is 0, an Out of Range trap always occurs, regardless of the DO bit.

In addition to the operations described in the Interrupt and Trap Handling section, the following operations are performed when an Out of Range trap is taken:

1. The operation code of the instruction causing the exception is placed in the IOP field of the Exception Opcode Register.
2. For the MULTIPLY, MULTIPLU, DIVIDE, and DIVIDU instructions, the absolute register numbers of the excepting instruction's source and destination registers are placed into the Indirect Pointer A, Indirect Pointer B, and Indirect Pointer C registers.
3. For the MULTIPLY, MULTIPLU, DIVIDE, and DIVIDU instructions, the destination register or registers are unchanged.

Floating-Point Exceptions

A Floating-Point Exception trap occurs when an exception is detected during a floating-point operation, and the exception is not masked by the corresponding bit of the Floating-Point Mask Register. In this context, a floating-point operation is defined as any operation that accepts a floating-point number as a source operand, that produces a floating-point result, or both. Thus, for example, the CONVERT instruction may create an exception while attempting to convert a floating-point value to an integer value.

In addition to the operations described in the Interrupt and Trap Handling section, the following operations are performed when a Floating-Point Exception trap is taken:

1. The operation code of the instruction causing the exception is placed in the IOP field of the Exception Opcode Register.
2. The status of the trapping operation is written into the trap status bits of the Floating-Point Status Register. The status bits that are written do not depend on the values of the corresponding mask bits in the Floating-Point Environment Register.
3. The absolute register numbers of the excepting instruction's source and destination registers are placed into the Indirect Pointer A, Indirect Pointer B, and Indirect Pointer C registers. If the RB or RC fields specify a function code, that code is transferred to the corresponding indirect pointer. Note that if the most-significant bit of the this function code is 1, the value of the Stack

Pointer has been added to the RB field and must be subtracted to recover the original field.

4. The destination register or registers are left unchanged.

Exceptions During Interrupt and Trap Handling

In most cases, interrupt and trap handling routines are executed with the DA bit in the Current Processor Status having a value of 1. It is assumed that these routines do not create many of the exceptions possible in most other processor routines, so most of these are ignored.

If the assumption of no exceptions is not valid for a particular interrupt or trap handler, it is important that the handler save the state of the processor and reset the FZ bit of the Current Processor Status, so that the handler itself may be restarted properly. This must be accomplished before any interrupts or traps can be taken. In this case, the state (or the state of some other process) must be restored before an interrupt return is executed.

It is possible that errors reported via the $\overline{\text{IERR}}$ and $\overline{\text{DERR}}$ signals are associated with hardware errors, independent of any routine being executed. For this reason, the Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps cannot be disabled by the DA bit, and the processor may take one of these traps even while handling another interrupt or trap.

If the processor does take an unmaskable trap while handling another interrupt or trap, and the state of the interrupt or trap handler is not reflected in processor registers, it is not possible to return to the point at which the unmaskable trap is taken. When the unmaskable trap is taken, the processor state saved is that state associated with the original interrupt or trap, not with the unmaskable trap; however, the Old Processor Status Register is modified to reflect the Current Processor Status Register of the interrupt or trap handler. This situation, indicated by the DA bit being 1 in the Old Processor Status Register, may not be recoverable.

MEMORY MANAGEMENT

The Am29000 incorporates a Memory Management Unit (MMU) for performing virtual-to-physical address translation and memory access protection. This section describes the logical operation of the Memory Management Unit.

Address translation can be performed only for instruction/data memory accesses. No address translation is performed for instruction ROM, input/output, coprocessor, or interrupt/trap vector accesses. However, an instruction/data memory access can be redirected to input/output by the address-translation process.

Translation Look-Aside Buffer

The MMU stores the most recently performed address translations in a special cache, the Translation Look-Aside Buffer (TLB). All virtual addresses generated by the processor are translated by the TLB. Given a virtual address, the TLB determines the corresponding physical address.

The TLB reflects information in the processor system page tables, except that it specifies the translation for many fewer pages; this restriction allows the TLB to be incorporated on the processor chip where the performance of address translation is maximized.

A diagram of the TLB is shown in Figure 58. The TLB is a table of 64 entries, divided into two equal sets, called Set 0 and Set 1. Within each set, entries are numbered 0 to 31. Entries in different sets that have equivalent entry numbers are grouped into a unit called a line; there are thus 32 lines in the TLB, numbered 0 to 31.

Each TLB entry is 64 bits long and contains mapping and protection information for a single virtual page. TLB entries may be inspected and modified by processor instructions executed in the Supervisor mode. The layout of TLB entries is described in the Register Description section.

The TLB stores information about the ownership of the TLB entries in an 8-bit Task Identifier (TID) field in each entry. This makes it possible for the TLB to be shared by several independent processes without the need for invalidation of the entire TLB as processes are activated. It also increases system performance by permitting processes to warm-start (i.e., to start execution on the

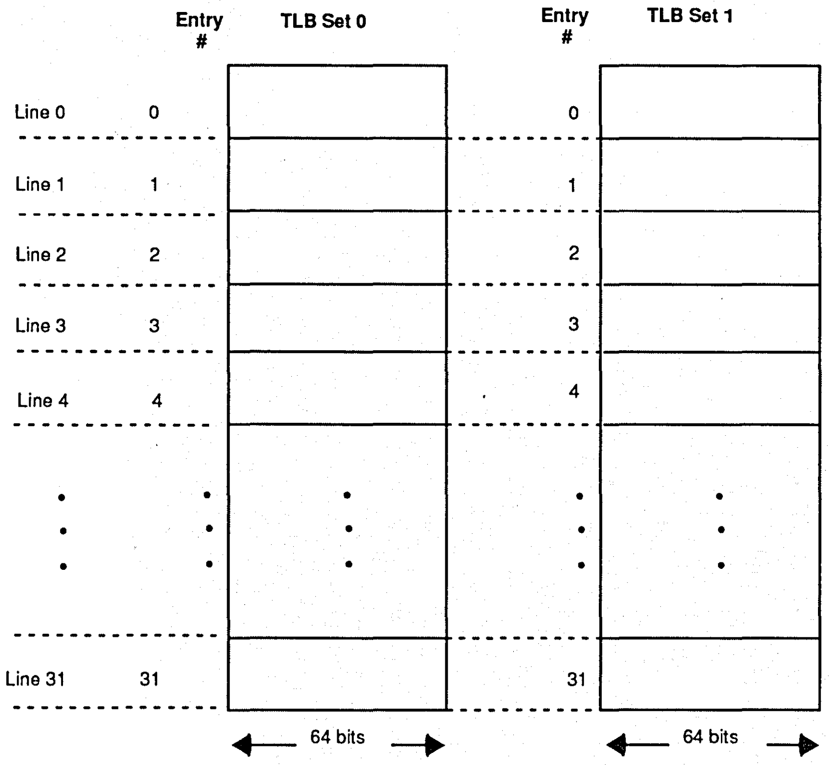


Figure 58. Translation Look-Aside Buffer Organization

processor with a certain number of TLB entries remaining in the TLB from a previous execution).

Each TLB entry contains a Usage bit to assist management of the TLB entries. The Usage bit indicates which set of the entry within a given line was least recently used to perform an address translation. Usage bits for two entries in the same line are equivalent.

The TLB contains other fields, described in the following sections.

Address Translation

For the purpose of address translation, the virtual instruction/data address space of a process is partitioned into regions of fixed size, called pages. Pages are mapped by the address-translation process into equivalent-sized regions of physical memory, called page frames. All accesses to instructions or data contained within a given page use the same virtual-to-physical address translation.

Virtual addresses are partitioned into three fields for the address-translation process, as shown in Figure 59. The partitioning of the virtual address is based on the page size. Page sizes may be of 1, 2, 4, or 8 kb, as specified by the MMU Configuration Register. The fields shown in Figure 59 are described in the following discussion.

Address Translation Controls

The processor attempts to perform address translation for the following external accesses:

1. Instruction accesses, if the Physical Addressing/ Instructions (PI) and ROM Enable (RE) bits of the Current Processor Status are both 0.
2. User-mode accesses to instruction/data memory if the Physical Addressing/Data (PD) bit of the Current Processor Status is 0.
3. Supervisor-mode accesses to instruction/data memory if the Physical Address (PA) bit of the load or store instruction performing the access is 0, and the PD bit of the Current Processor Status is 0.

Address translation also is controlled by the MMU Configuration Register. This register specifies the virtual page size and contains an 8-bit Process Identifier (PID) field. The PID field specifies the process number associated with the currently running program, if this is a User-mode program. Supervisor-mode programs are assigned a fixed process number of 0. The process number is compared with Task Identifier (TID) fields of the TLB entries during address translation. The TID field of a TLB entry must match the process number for the translation to be valid.

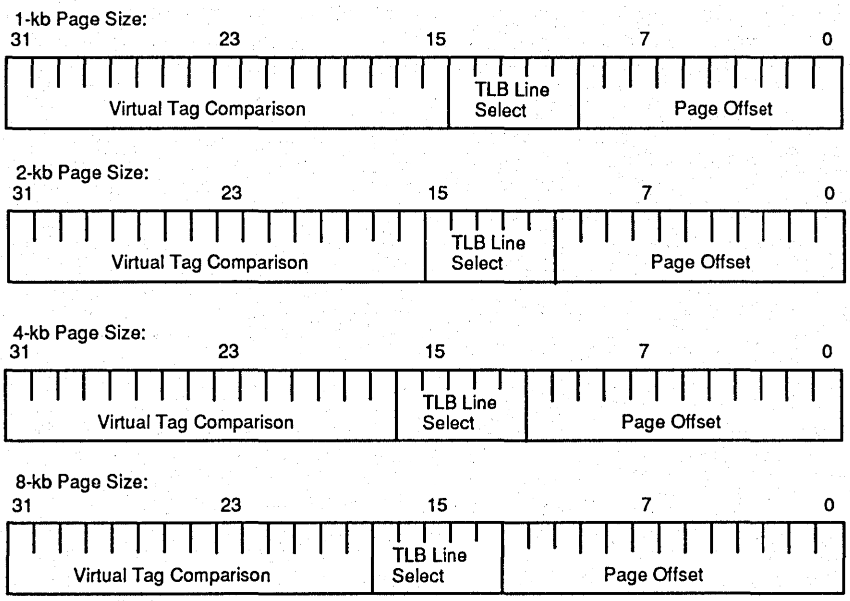


Figure 59. Virtual Address for 1-, 2-, 4-, and 8-kb Pages

Address Translation Process

The address-translation process is diagrammed in Figure 60. Address translation is performed by the following fields in the TLB entry: the Virtual Tag (VTAG), the Task Identifier (TID), the Valid Entry (VE) bit, the Real Page Number (RPN) field, and the Input/Output (IO) bit. To perform an address translation, the processor accesses the TLB line whose number is given by certain bits in the virtual address. The bits used depend on the page size as follows:

Page Size	Virtual Address Bits (for Line Access)
1 kb	14–10
2 kb	15–11
4 kb	16–12
8 kb	17–13

The accessed line contains two TLB entries, which in turn contain two VTAG fields. The VTAG fields are both compared to bits in the virtual address. This comparison depends on the page size as follows (note that VTAG bit-numbers are relative to the VTAG field, not the TLB entry):

Page Size	Virtual Address Bits	VTAG Bits
1 kb	31–15	16–0
2 kb	31–16	16–1
4 kb	31–17	16–2
8 kb	31–18	16–3

Certain bits of the VTAG field do not participate in the comparison for page sizes larger than 1 kb. These bits of the VTAG field are required to be 0.

For an address translation to be valid, the following conditions must be met:

1. The virtual address bits match corresponding bits of the VTAG field as specified above.
2. For a User-mode access, the TID field in the TLB entry matches the PID field in the MMU Configu-

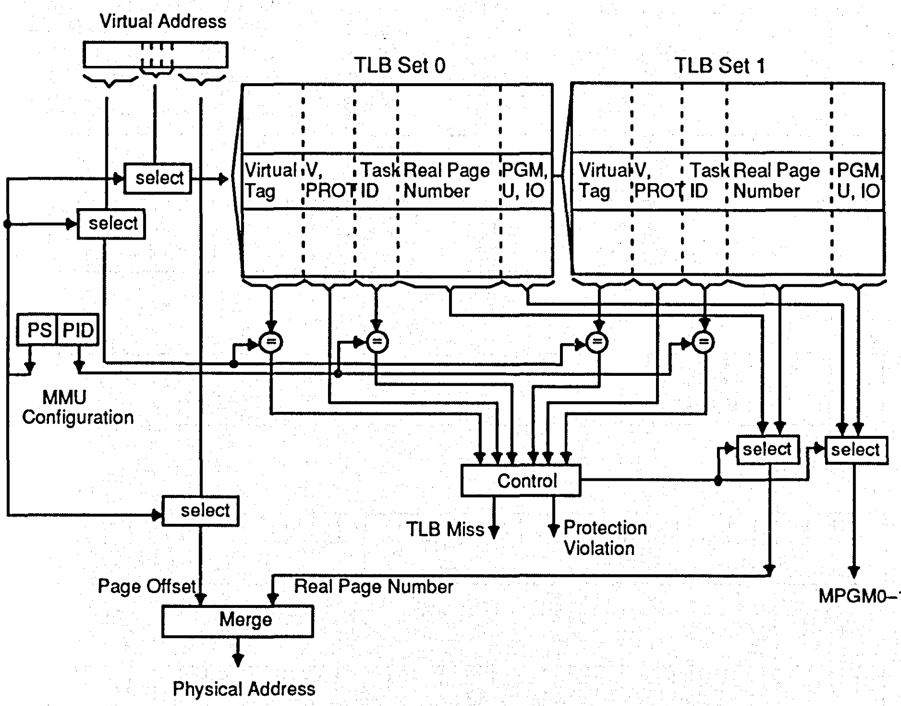


Figure 60. Address Translation Process

ration Register. For a Supervisor-mode access, the TID field is 0.

3. The VE bit in the TLB entry is 1.
4. Only one entry in the line meets conditions 1, 2, and 3 above. If this condition is not met, the results of the translation may be treated as valid by the processor, but the results are unpredictable.

If the address translation is valid for one TLB entry in the selected line, the RPN field in this entry is used to form the physical address of the access. The RPN field gives the portion of the physical address that depends on the translation; the remaining portion of the virtual address, called the Page Offset, is invariant with address translation.

The Page Offset comprises the low-order bits of the virtual address, and gives the location of a byte (because of byte addressing) within the virtual page. This byte is located at the same position in the physical page frame, so the Page Offset also comprises the low-order bits of the physical address.

The 32-bit physical address is the concatenation of certain bits of the RPN field and Page Offset, where the bits from each depend on the page size as follows (note that RPN bit numbers are relative to the RPN field, not the TLB entry):

Page Size	RPN Bits	Virtual Address Bits for Page Offset
1 kb	21–0	9–0
2 kb	21–1	10–0
4 kb	21–2	11–0
8 kb	21–3	12–0

Note that certain bits of the RPN field are not used in forming the physical address for page sizes greater than 1 kb. These bits of the RPN are required to be 0. In addition, for certain instruction accesses, the Page Offset is incremented by 16.

The address space of the physical address is determined by the Input/Output (IO) bit of the TLB entry. If the IO bit is 0, the address is in the instruction/data memory address space. If the IO bit is 1, the address is in the input/output address space.

Successful and Unsuccessful Translations

If an address translation is successful, the TLB entry is further used to perform protection checking for the access. Bits in the TLB make it possible to restrict accesses—independently for Supervisor-mode and User-mode accesses—to any combination of load, store, and instruction accesses, or to no access.

If the address translation is valid and no protection violation is detected, the physical address from the translation is placed on the processor's address bus and the access is initiated. If the translation is not valid or a protection violation is detected, a trap occurs. Depending on the state of the channel interface, the access request may be placed on the address bus with the signal $\overline{BINV}$ asserted, even though the trap occurs.

Also, if the address translation is successful and there is no protection violation, the PGM bits from the TLB entry used for translation are placed on the $MPGM_1$–$MPGM_0$ outputs during the address cycle for the access. If address translation is not performed, these pins are both Low for the address cycle.

If the TLB cannot translate an address, a TLB miss occurs. The MMU causes a trap if either a TLB miss occurs, or the translation is successful and a protection violation is detected. The processor distinguishes between traps caused by instruction and data accesses, and between traps caused by User and Supervisor-mode accesses, as follows:

Trap Vector Number	Type of Trap
8	User-Mode Instruction TLB Miss
9	User-Mode Data TLB Miss
10	Supervisor-Mode Instruction TLB Miss
11	Supervisor-Mode Data TL Miss
12	Instruction TLB Protection Violation
13	Data TLB Protection Violation

The distinction between the above traps is made to assist trap handling, particularly the routines that load TLB entries.

Reload

So that the MMU may support a large variety of memory-management architectures, it does not directly load TLB entries that are required for address translation. It simply causes a TLB miss trap when address translation is unsuccessful. The trap causes a program—called the TLB reload routine—to execute. The TLB reload routine is defined according to the structure and access method of the page table contained in an external device or memory.

When a TLB miss trap occurs, the LRU Recommendation Register is written with the TLB register number for Word 0 of the TLB entry to be used by the TLB reload routine. For instruction accesses, the Program Counter 1 Register contains the instruction address that was not successfully translated. For data accesses, the Channel Address Register contains the data address that was not successfully translated.

The TLB reload routine determines the translation for the address given by the Program Counter 1 Register or Channel Address Register, as appropriate. The TLB reload routine uses an external page table to determine the required translation, and loads the TLB entry indicated by the LRU Recommendation Register so that the entry may perform this translation. In a demand-paged environment, the TLB reload routine may additionally invoke a page-fault handler when the translation cannot be performed.

TLB entries are written by the Move To TLB (MTTLB) instruction, which copies the contents of a general-purpose register into a TLB register. The TLB register number is specified by bits 6–0 of a general-purpose register. TLB entries are read by the Move From TLB (MFTLB) instruction, which copies the contents of a TLB register into a general-purpose register. Again, the TLB register number is specified by a general-purpose register.

Entry Invalidation

There are two methods for invalidating TLB entries that are no longer required at a given point in program execution. The first involves resetting the Valid Entry bit of a single entry (this is done by a Move To TLB instruction). The second involves changing the value of the Process Identifier (PID) field of the MMU Configuration Register; this invalidates all entries whose Task Identifier (TID) fields do not match the new value.

If an entry is invalidated by changing the PID field, the TLB entry still remains valid in some sense. If the PID field is changed again to match the TID field, the entry may once again participate in address translation. This ability can be used to reduce the number of TLB misses in a system during process switching. However, it is important to manage TLB entries so that an invalid match cannot occur between the PID field and the TID field of an old TLB entry.

Protection

If an address translation is performed successfully, the TLB entry used in address translation is used to perform protection checking for the access. There are 6 bits in the TLB entry for this purpose: Supervisor Read (SR), Supervisor Write (SW), Supervisor Execute (SE), User Read (UR), User Write (UW), and User Execute (UE). These bits restrict accesses, depending on the program mode of the access, as shown in Figure 61 (the value "x" is a "don't care").

Note that for the Load and Set (LOADSET) instruction, the protection bits must be set to allow both the load and store access. If this condition does not hold, neither access is performed.

If protection checking indicates that a given access is not allowed, a Data TLB Protection Violation or Instruction TLB Protection Violation trap occurs. The cause of the trap is determined by inspection of the Program Counter 1 Register for an Instruction TLB Protection Violation, or by inspection of the contents of the Channel Address and Channel Control registers for a Data TLB Protection Violation.

SR	SW	SE	UR	UW	UE	Type of Access Allowed
x	x	x	0	0	0	No user access
x	x	x	0	0	1	User instruction
x	x	x	0	1	0	User store
x	x	x	0	1	1	User store or instruction
x	x	x	1	0	0	User load
x	x	x	1	0	1	User load or instruction
x	x	x	1	1	0	User load or store
x	x	x	1	1	1	Any user access
0	0	0	x	x	x	No supervisor access
0	0	1	x	x	x	Supervisor instruction
0	1	0	x	x	x	Supervisor store
0	1	1	x	x	x	Supervisor store or instruction
1	0	0	x	x	x	Supervisor load
1	0	1	x	x	x	Supervisor load or instruction
1	1	0	x	x	x	Supervisor load or store
1	1	1	x	x	x	Any supervisor access

Figure 61. TLB Access Protection

CHANNEL DESCRIPTION

The processor channel provides the bandwidth required for performance, while permitting the connection of many different types of devices. This section describes the channel and methods of connecting devices and memories to the processor.

The channel consists of three 32-bit synchronous buses with associated control and status signals: the Address Bus, Data Bus, and Instruction Bus. The Address Bus transfers addresses and control information to devices and memories. The Data Bus transfers data to and from devices and memories. The Instruction Bus transfers instructions to the processor from instruction memories. In addition, a set of signals allows control of the channel to be relinquished to an external master.

There are five logical groups of signals performing five distinct functions, as follows (since some signals perform more than one function, a signal may appear in more than one group):

1. Instruction Address Transfer and Instruction Access Requests: A_{31}–A_0, SUP/$\overline{\text{US}}$, MPGM_1–MPGM_0, $\overline{\text{PEN}}$, $\overline{\text{IREQ}}$, IREQT, $\overline{\text{PIA}}$, $\overline{\text{BINV}}$
2. Instruction Transfer: I_{31}–I_0, $\overline{\text{IBREQ}}$, $\overline{\text{IRDY}}$, $\overline{\text{IERR}}$, $\overline{\text{IBACK}}$
3. Data Address Transfer and Data Access Requests: A_{31}–A_0, R/$\overline{\text{W}}$, SUP/$\overline{\text{US}}$, $\overline{\text{LOCK}}$, MPGM_1–MPGM_0, $\overline{\text{PEN}}$, $\overline{\text{DREQ}}$, DREQT_1–DREQT_0, OPT_2–OPT_0, $\overline{\text{PDA}}$, $\overline{\text{BINV}}$
4. Data Transfer: D_{31}–D_0, $\overline{\text{DBREQ}}$, $\overline{\text{DRDY}}$, $\overline{\text{DERR}}$, $\overline{\text{DBACK}}$, $\overline{\text{CDA}}$
5. Arbitration: $\overline{\text{BREQ}}$, $\overline{\text{BGRT}}$, $\overline{\text{BINV}}$

User-Defined Signals

There are two types of user-defined outputs on the processor to control devices and memories directly in a system-dependent manner. Each of these outputs is valid simultaneously with—and for the same duration as—the address for an access.

The first set of user-defined signals, MPGM_1–MPGM_0, is determined by the PGM bits in the Translation Look-Aside Buffer entry used in address translation. If address translation is not performed, these outputs are both Low.

The second set of signals, OPT_2–OPT_0, is determined by bits 18–16 of the load or store instruction that initiates an access. These signals are valid only for data accesses, and have a predefined interpretation for coprocessor data transfers.

Standard interpretations of OPT_2–OPT_0 are given in the Pin Description section. Since the OPT_2–OPT_0 signals are determined by instructions, they have an impact on application-software compatibility, and system hardware should use the given definitions of OPT_2–OPT_0.

The OPT_2–OPT_0 signals are used to encode byte and half-word accesses. However, for a load, the system should return an entire aligned word, regardless of the indicated data width.

Note that the standard interpretations of OPT_2–OPT_0 apply only to accesses to instruction/data memory and input/output. Other interpretations may be used for coprocessor transfers.

For interrupt and trap vector fetches, the MPGM_1–MPGM_0 and OPT_2–OPT_0 outputs are all Low.

Instruction Accesses

Instruction accesses occur to one of two address spaces: instruction/data memory and instruction read-only memory (instruction ROM). The distinction between these address spaces is made by the IREQT signal, which is in turn derived from the ROM Enable (RE) bit of the Current Processor Status Register. These are truly distinct address spaces; each may be populated independently based on the needs of a particular system.

Instruction/data memory contains both instructions and data. Although the channel supports separate instruction and data memories, the Memory Management Unit does not. In certain systems, it may be required to access instructions via loads and stores, even though instructions may be contained in physically separate memories. For example, this requirement might be imposed because of the need to load instructions into memory. Note also that the OPT_2–OPT_0 signals may be used to allow the access of instructions in instruction ROM, using loads; the Am29000 does not prevent a store to the instruction ROM, and protection against stores to the instruction ROM must be provided externally, if required.

All processor instruction fetches are read accesses, and the R/$\overline{\text{W}}$ signal is High for all instruction fetches.

Data Accesses

Data accesses occur to one of three address spaces: instruction/data memory, input/output (I/O), and the coprocessor. The distinction between these spaces is made by the DREQT_1–DREQT_0 signals, which are in turn determined by the load or store instruction that initiates a data access. Each of these address spaces is distinct from the others.

The protocol for data transfers to and from the coprocessor is slightly different than the protocol for instruction/data memory and I/O accesses.

Data accesses may occur either from a slave device or memory to the processor (for a load), or from the processor to a slave device or memory (for a store). The direction of transfer is determined by the R/$\overline{\text{W}}$ signal. In the case of a load, the processor requires that data on the data bus be held valid only for a short time before the end of a cycle. In the case of a store, the processor drives the data bus as soon as the bus is available and holds the data valid until the slave device or memory signals that the access is complete.

Reporting Errors

The successful completion of an instruction access is indicated by an active level on the IRDY input, and the successful completion of a data access is indicated by an active level on the DRDY input. If there are exceptional conditions for which an instruction or data access cannot be completed successfully, the unsuccessful completion is indicated by an active level on the IERR or DERR input, as appropriate.

If the processor receives an IERR or DERR in response to an instruction or data access, it ignores the content of the instruction or data bus and the value of IRDY or DRDY. An IERR response causes an Instruction Access Exception trap, unless it is associated with an instruction that the processor does not ultimately execute (because of a nonsequential instruction fetch). A DERR response always causes either a Data Access Exception trap or a Co-processor Exception Trap.

The processor supports the restarting of unsuccessful accesses upon an interrupt return. In the case of an unsuccessful instruction access, the restart is performed by the Program Counter 0 and Program Counter 1 registers. In the case of an unsuccessful data access, the restart is performed by the Channel Address, Channel Data, and Channel Control registers. In any event, the control program must determine whether or not an access can and/or should be restarted.

The Instruction Access Exception and Data Access Exception traps cannot be masked. If one of these traps occurs within an interrupt or trap handler, the processor state may not be recoverable.

Access Protocols

Figure 62 shows a control flowchart for accesses performed by the Am29000. This control flow applies independently to both instruction and data accesses. Since the processor performs concurrent instruction and data accesses, these accesses may be at different points in the control flow at any given point in time.

Note that the items on the flowchart of Figure 62 do not represent actual states and have no particular relationship to processor cycles. The flowchart provides only a high-level understanding of the control flow. Also, exceptions and error conditions are not shown.

The channel supports three protocols for accesses: simple, pipelined, and burst-mode. These are described in the following sections. The various protocols are defined to accommodate minimum-latency accesses as well as maximum-transfer-rate accesses. The protocols allow an access to complete in a single cycle, although they support accesses requiring arbitrary numbers of cycles. Address transfers for accesses may be independent of instruction or data transfers.

Simple Accesses

For a simple access, the processor holds the address valid throughout the entire access. This protocol is used for single-cycle accesses, and for accesses to simple devices and memories.

On any cycle before the completion of the access, a simple access may be converted to a pipelined access (by the assertion of PEN) or to a burst-mode access (by the assertion of IBACK or DBACK, if the processor is asserting IBREQ or DBREQ). Thus, the protocol for simple accesses also may be used during the initial cycles of pipelined and/or burst-mode accesses. This is advantageous, for example, in cases where the slave device or memory either requires the address to be held for multiple cycles at the beginning of the pipelined or burst-mode access, or cannot respond to the pipelined or burst-mode request within one cycle.

Pipelined Accesses

A pipelined access is one that starts before an earlier in-progress accesses completed. The in-progress access is called a primary access and the second access is called a pipelined access. A pipelined access is of the same type as the primary access. For example, an instruction access that begins before the completion of a data access is not considered to be a pipelined access, whereas a second data access is.

The Am29000 allows only one pipelined access at any given time.

Tradeoffs

For accesses that require more than one cycle to complete, pipelined accesses perform better than simple accesses because they allow the overlap of portions of two accesses. In addition, the ability to latch addresses in support of pipelined accesses reduces utilization of the address bus, thereby reducing contention between instruction and data accesses. However, devices and memories that support pipelined accesses are somewhat more complex than devices and memories that support only simple accesses.

Support for pipelined operations is required for both the primary access and the pipelined access. The slave performing the primary access must contain some means for storing the address and other information about the access. The slave performing the pipelined access must be able to restrict its use of the instruction bus or data Bus, and must be prepared to cancel the access (as explained below).

Pipelined Operation

Pipelined accesses are controlled by the signals PEN, PIA, and PDA. Because of internal data-flow constraints, the Am29000 does not perform a pipelined store operation while a load is in progress. However, the protocol does not restrict pipelined operations. Other channel masters may perform a pipelined store during a load.

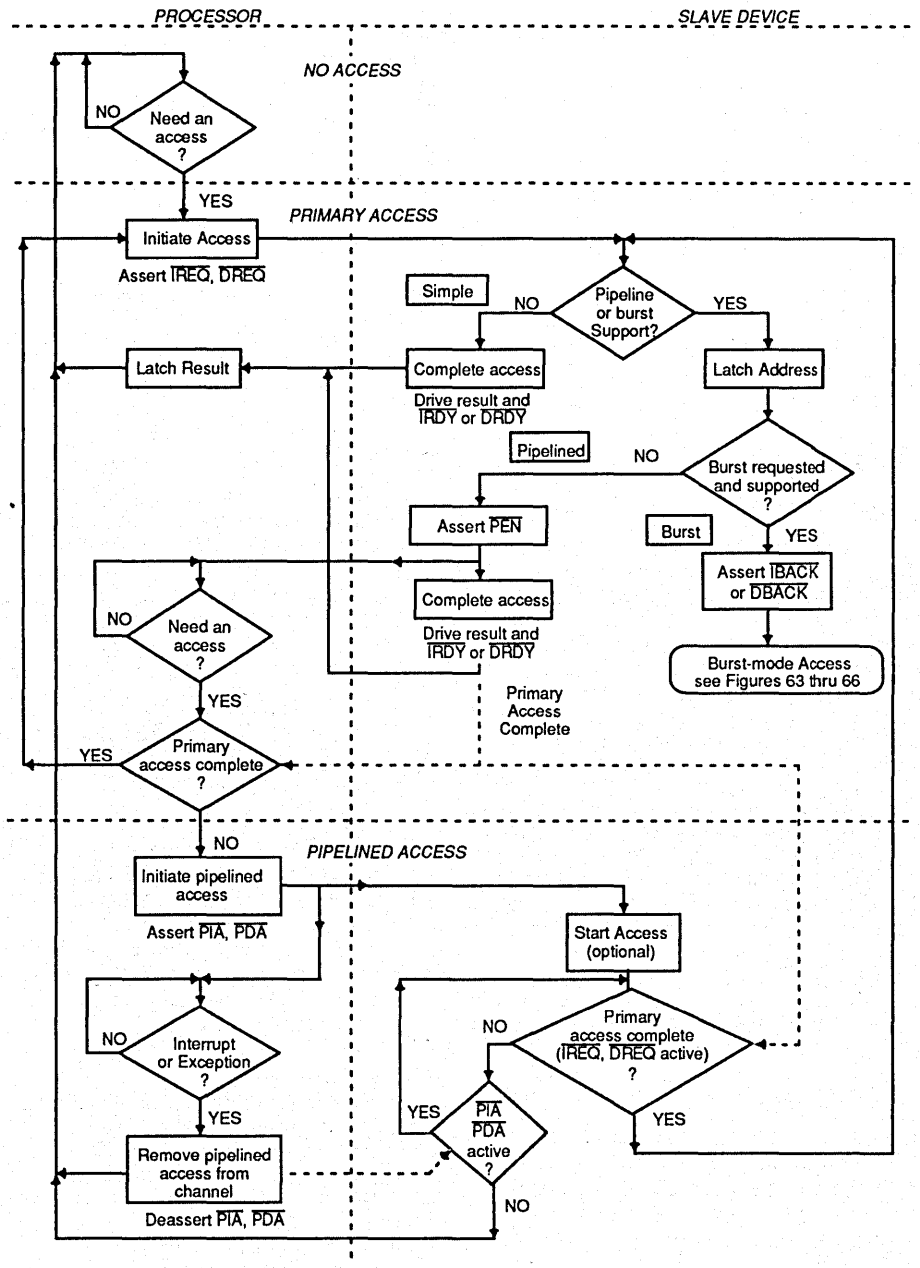

PROCESSOR
SLAVE DEVICE
NO ACCESS
Need an access ?
NO
YES
PRIMARY ACCESS
Initiate Access
Assert $\overline{\text{IREQ}}$, $\overline{\text{DREQ}}$
Simple
NO
Pipeline or burst Support?
YES
Latch Result
Complete access
Drive result and $\overline{\text{IRDY}}$ or $\overline{\text{DRDY}}$
Latch Address
Pipelined
NO
Burst requested and supported ?
Assert $\overline{\text{PEN}}$
Burst
YES
Assert $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$
Complete access
Drive result and $\overline{\text{IRDY}}$ or $\overline{\text{DRDY}}$
Burst-mode Access see Figures 63 thru 66
NO
Need an access ?
YES
Primary Access Complete
YES
Primary access complete ?
NO
PIPELINED ACCESS
Initiate pipelined access
Assert $\overline{\text{PIA}}$, $\overline{\text{PDA}}$
Start Access (optional)
NO
Primary access complete ($\overline{\text{IREQ}}$, $\overline{\text{DREQ}}$ active) ?
NO
Interrupt or Exception ?
YES
YES
$\overline{\text{PIA}}$ $\overline{\text{PDA}}$ active ?
YES
Remove pipelined access from channel
Deassert $\overline{\text{PIA}}$, $\overline{\text{PDA}}$
NO

Figure 62. Channel Flowchart

Except as noted above, the processor attempts to perform pipelining for every access; the input $\overline{\text{PEN}}$ indicates whether or not pipelining is supported for a given access. The $\overline{\text{PEN}}$ input can be driven by individual devices, or can be tied active or inactive to enable or disable system-wide pipelined accesses. The processor ignores the value of $\overline{\text{PEN}}$ unless it is performing an access.

The processor samples $\overline{\text{PEN}}$ on every cycle during a primary access. If $\overline{\text{PEN}}$ is active on any cycle, the processor ceases to drive the address and associated controls for the primary access in the next cycle. If the processor requires another access before the primary access is completed, it drives the address and controls for the second access, asserting $\overline{\text{PIA}}$ or $\overline{\text{PDA}}$ to indicate that the second access is a pipelined access.

The output $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$, as appropriate, is not asserted for a pipelined access. Devices and memories that cannot support pipelined accesses should therefore ignore $\overline{\text{PIA}}$ and/or $\overline{\text{PDA}}$, and base their operation upon $\overline{\text{IREQ}}$ and/or $\overline{\text{DREQ}}$.

A device or memory that receives a request for a pipelined access may treat it as any other access, with one exception: the pipelined access cannot use the Instruction and data buses or the associated controls (e.g., $\overline{\text{IRDY}}$ or $\overline{\text{DRDY}}$). In the case of a data read or instruction access, the results of the pipelined access cannot be driven on the appropriate bus. In the case of a data write, the data do not appear on the data bus. Any other operations for the access, such as address decoding, can occur.

When the primary access is completed (as indicated by $\overline{\text{IRDY}}$ or $\overline{\text{DRDY}}$), the pipelined access becomes a primary access. The processor indicates this by asserting $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$, depending on the type of access. The device or memory performing the pipelined access may complete the access as soon as $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ is asserted (possibly in the same cycle). When the access becomes a primary access, it controls the channel as any other primary access. For example, it may determine whether or not another pipelined access can be performed.

When the pipelined access becomes a primary access, the output $\overline{\text{PIA}}$ or $\overline{\text{PDA}}$ remains asserted for one cycle to ensure continuity of control within the slave device or memory. In the cycle after $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ is asserted, $\overline{\text{PIA}}$ or $\overline{\text{PDA}}$ is deasserted unless the processor initiates another pipelined access, in which case $\overline{\text{PIA}}$ or $\overline{\text{PDA}}$ remains asserted for the new access.

Cancellation of Pipelined Accesses

If the processor takes an interrupt or trap before a pipelined access becomes a primary access, the request for the pipelined access is removed from the channel. This may occur, for example, when $\overline{\text{IERR}}$ or $\overline{\text{DERR}}$ is signaled for the primary access.

If the pipelined access is removed from the channel, the slave device or memory does not receive an $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ for the pipelined access. Hence, the pipelined access does not become a primary access, and cannot be completed. A pipelined access may be canceled in this manner at any time before it becomes a primary access. Because of this, a pipelined access should not change the state of a slave device or memory until the pipelined access becomes a primary access.

Burst-Mode Accesses

A burst-mode access allows multiple instructions or data words at sequential addresses to be accessed with a single address transfer. The number of accesses performed and the timing of each access within the sequence are controlled dynamically by the burst-mode protocol. Burst-mode accesses take advantage of sequential addressing patterns, and provide several benefits over simple and pipelined accesses:

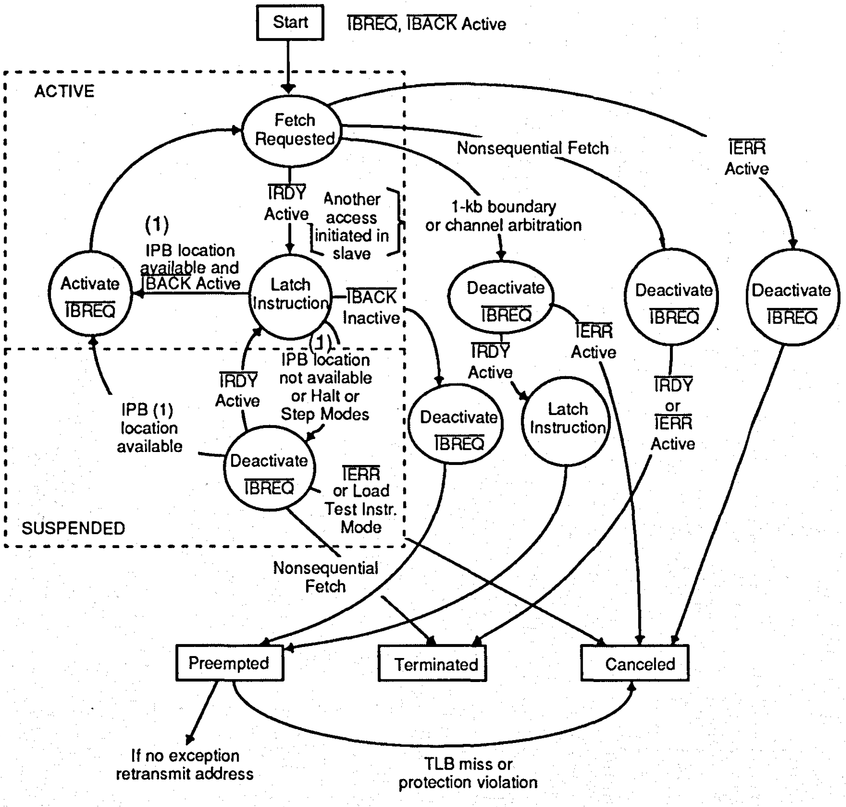
1. Simultaneous instruction and data acçesses. Burst-mode accesses reduce the utilization of the address bus. This is especially important for instruction accesses, which are normally sequential. Burst-mode instruction accesses eliminate most of the address transfers for instructions, allowing the address bus to be used for simultaneous data accesses.

2. Faster access times. By eliminating the address-transfer cycle, burst-mode accesses allow addresses to be generated in a manner that improves access times.

3. Faster memory access modes. Many memories have special high-bandwidth access modes (e.g., fast page mode DRAM). These modes generally require a sequential addressing pattern, even though addresses may not be presented explicitly to the memory for all accesses. Burst-mode accesses allow the use of these access modes without hardware to detect sequential addressing patterns.

Burst-Mode Overview

The control-flow diagrams in Figure 63 and Figure 64 illustrate the operation of the processor and an instruction memory during a burst-mode instruction access. The control-flow diagrams in Figure 65 and Figure 66 illustrate the operation of the processor and a data memory or device during a burst-mode data access. These diagrams are for illustration only; nodes on these diagrams do not necessarily correspond to processor or slave states, and transitions on these diagrams do not necessarily correspond to processor cycles.



(1) IPB = Instruction Prefetch Buffer

Figure 63. Processor Burst-Mode Instruction Accesses: Control Flow

A burst-mode access is in one of the following operational conditions at any given time:

1. Established: The processor and slave device have successfully initiated the burst-mode access. A burst-mode access that has been established is either active or suspended. An established burst-mode access may become preempted, terminated or canceled.

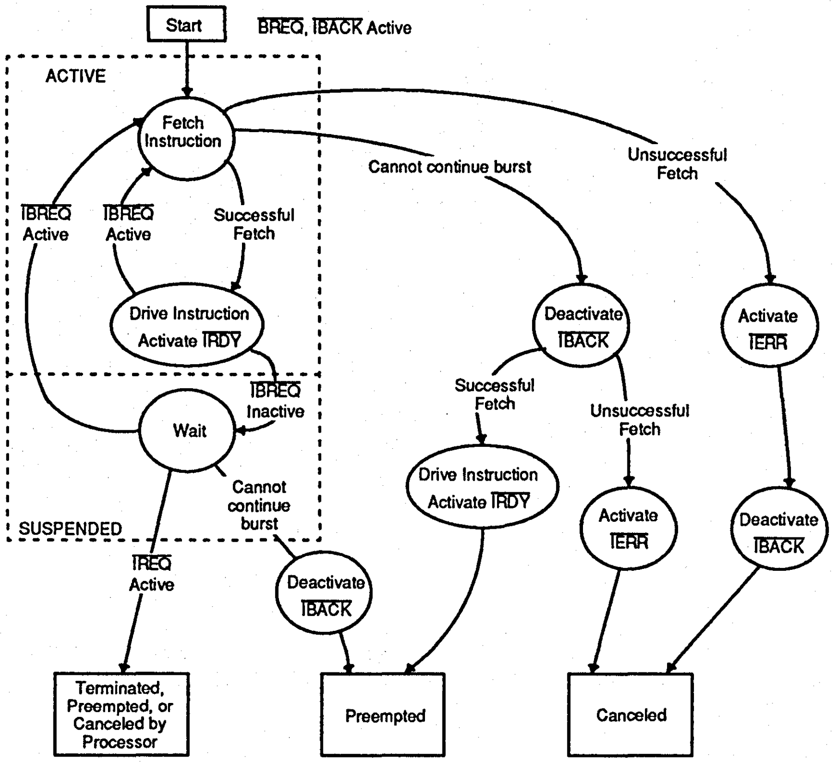
2. Active: Instruction or data accesses and transfers are being performed as the result of the burst-mode access. An active burst-mode access may become suspended.

3. Suspended: No accesses or transfers are being performed as the result of the burst-mode access, but the burst-mode access remains established. Additional accesses and transfers may occur at some later time (i.e., the burst-mode access may become active) without the retransmission of the address for the access.

4. Preempted: The burst-mode access can no longer continue because of some condition, but the burst-mode access can be re-established within a short amount of time.

5. Terminated: All required accesses have been performed.

6. Canceled: The burst-mode access can no longer continue because of


Start
$\overline{\text{BREQ}}$, $\overline{\text{IBACK}}$ Active
ACTIVE
Fetch Instruction
$\overline{\text{IBREQ}}$ Active
$\overline{\text{IBREQ}}$ Active
Successful Fetch
Drive Instruction Activate $\overline{\text{IRDY}}$
$\overline{\text{IBREQ}}$ Inactive
Wait
Cannot continue burst
SUSPENDED
$\overline{\text{IREQ}}$ Active
Terminated, Preempted, or Canceled by Processor
Deactivate $\overline{\text{IBACK}}$
Preempted
Cannot continue burst
Unsuccessful Fetch
Deactivate $\overline{\text{IBACK}}$
Successful Fetch
Drive Instruction Activate $\overline{\text{IRDY}}$
Unsuccessful Fetch
Activate $\overline{\text{IERR}}$
Activate $\overline{\text{IERR}}$
Deactivate $\overline{\text{IBACK}}$
Canceled

Note: A similar state transition may be used to support suspended burst-mode data accesses or a channel master other than the processor.

Figure 64. Slave Burst-Mode Instruction Accesses: Control Flow

some exceptional condition. The access may be re-established only after the exceptional condition has been corrected, if possible.

Each of the above conditions, except for the terminated condition, is under the control of both the processor and slave device or memory. The terminated condition is determined by the processor, because only the processor can determine that all required accesses have been performed. The following sections discuss each of the above conditions with respect to the burst-mode protocol.

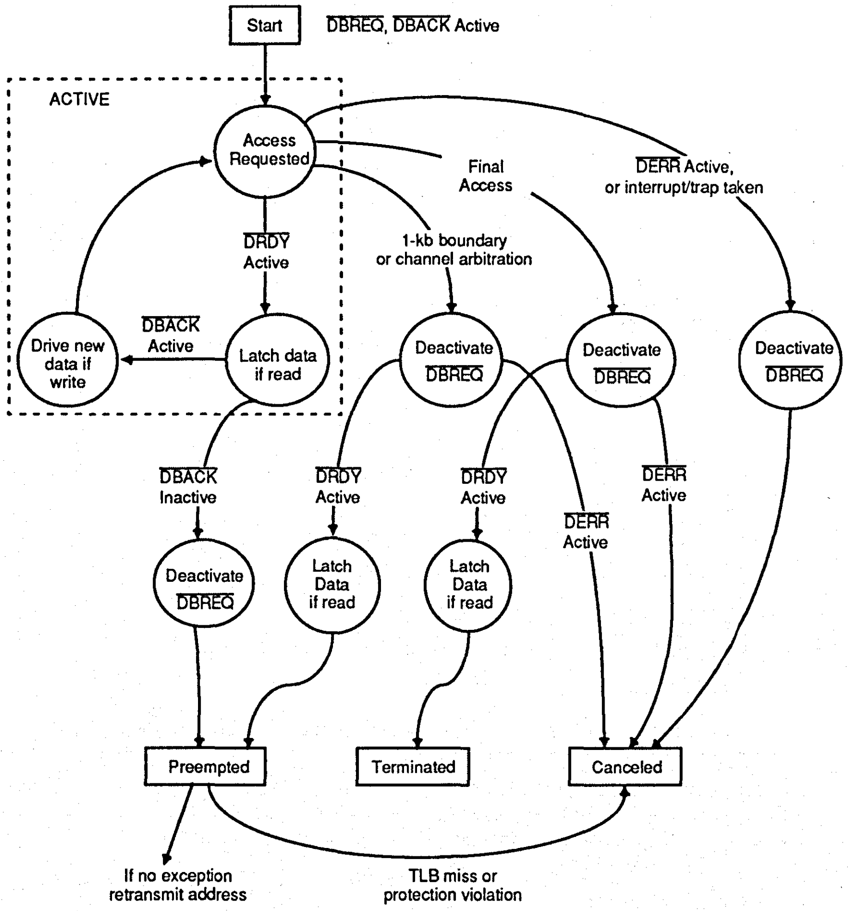
Establishing Burst-Mode Accesses

The Am29000 attempts to perform all instruction prefetches using burst-mode accesses, except for instruction fetches at the last word before a 1-kb address boundary. For data accesses, the processor attempts to perform Load Multiple and Store Multiple operations using burst-mode accesses. The processor indicates that it desires a burst-mode access by asserting $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ during the cycle in which the initial address is placed on the address bus (however, note that these signals become valid later in the cycle than the address).

The inputs $\overline{\text{IBACK}}$ and $\overline{\text{DBACK}}$ indicate that a requested burst-mode access is supported. The processor ignores the value of $\overline{\text{IBACK}}$ unless $\overline{\text{IBREQ}}$ is asserted, and it ignores the value of $\overline{\text{DBACK}}$ unless $\overline{\text{DBREQ}}$ is asserted.

When it desires a burst-mode access, the processor continues to drive $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ on every cycle for which the address is valid on the address bus. During this time, the device or memory involved in the access may assert $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ to indicate that it can perform the burst-mode access. If $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ (as appropriate) is asserted while the initial address appears on the address bus, the burst-mode access is established. In the following cycle, the processor removes the request address and deasserts $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$. However, it continues to assert $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$.

If the burst-mode access is not established on the first access, the processor attempts to establish a burst-



Note: The Am29000 does not suspend burst-mode data accesses.

Figure 65. Processor Burst-Mode Data Accesses: Control Flow

mode access on each subsequent address transfer, as long as there are more accesses yet to be performed. During any subsequent access, the addressed device or memory may establish a burst-mode access by asserting $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$. If the burst-mode access is never established, the default behavior is to have the processor transmit an address for every access.

Active and Suspended Burst-Mode Accesses

After the burst-mode access is established, $\overline{\text{IBREQ}}$ and $\overline{\text{DBREQ}}$ are used during subsequent accesses to indicate that the processor requires at least one more access. If $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ is active at the end of the cycle in which an access is successfully completed (i.e., when $\overline{\text{IRDY}}$ or $\overline{\text{DRDY}}$ is active), the processor requires another access. If the slave device or memory previously has not preempted the burst-mode access, and does not preempt (by deasserting $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$) or cancel (by asserting $\overline{\text{IERR}}$ or $\overline{\text{DERR}}$) the burst-mode access in the cycle that the access completes, the additional access must be performed.

The execution rate of instructions is known only dynamically, so that in certain situations, a burst-mode instruction access must be suspended. If $\overline{\text{IBREQ}}$ is inactive during the cycle in which an instruction access is completed, the burst-mode access is suspended (if it is neither preempted nor canceled at the same time). The burst-mode access remains suspended unless the processor requests a new instruction access (in which case $\overline{\text{IREQ}}$ is asserted), or unless the instruction memory preempts the burst-mode access.

A suspended burst-mode instruction access becomes active whenever the processor can accept more instruc-

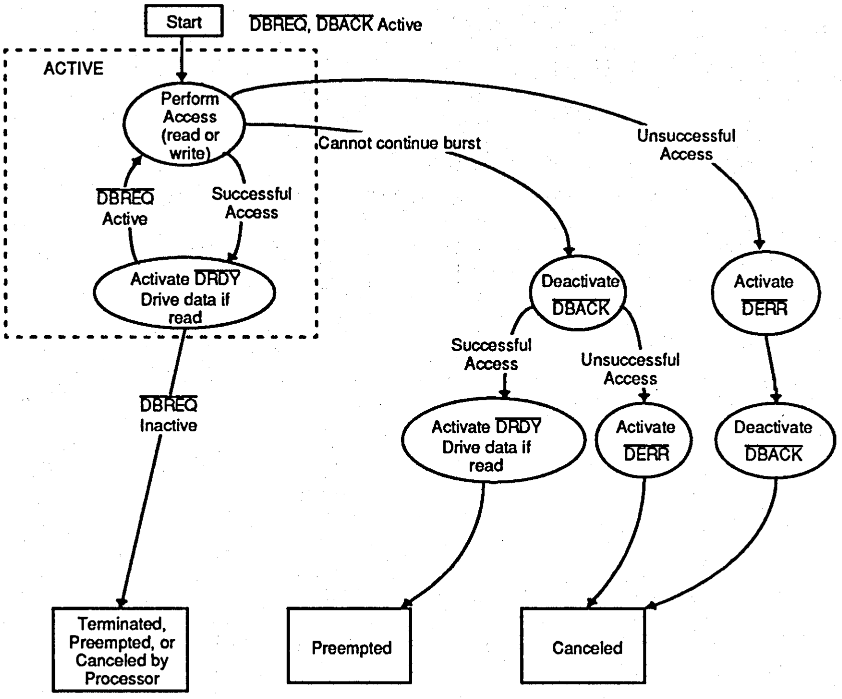


Figure 66. Slave Burst-Mode Data Accesses: Control Flow

tions. The processor activates the burst-mode access by asserting $\overline{\text{IBREQ}}$. If the instruction memory does not preempt the burst-mode access during this cycle, an instruction access must be performed.

When a suspended burst-mode instruction access is activated, the resulting instruction access is not permitted to be completed in the cycle in which $\overline{\text{IBREQ}}$ is asserted, but may be completed in the next cycle. The reason for this restriction is that the burst-mode protocol is defined such that the combination of an active level on $\overline{\text{IBREQ}}$ and $\overline{\text{IRDY}}$ causes an instruction access (as previously discussed). If the instruction access is completed immediately in the cycle where a suspended burst-mode access is activated, there is an ambiguity in the protocol: it is possible to interpret a single-cycle assertion of $\overline{\text{IBREQ}}$ as a request for two instructions.

The above ambiguity is resolved by delaying the instruction access resulting from a reactivated burst-mode access for a cycle. Since this restriction applies only when the Instruction Prefetch Buffer is full and the instruction memory is capable of a very fast access, the delayed instruction response has no performance impact.

The Am29000 does not suspend burst-mode data accesses because the data transfers occur to and from general-purpose registers, which are always available. However, other channel masters may suspend burst-mode data accesses (during direct memory accesses, for example). The principles for suspending burst-mode accesses are the same as those for instruction accesses discussed above.

Processor Preemption, Termination, and Cancellation

The processor may preempt, terminate or cancel a burst-mode access by deasserting $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ and asserting $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ at some later point. Normally, the processor receives one more instruction or data word after $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ is deasserted. However, this access may be completed in the same cycle that $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ is deasserted. During the period after $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ is deasserted and before $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ is asserted, the burst-mode access is in a suspended condition.

The slave device or memory cannot distinguish between preempted, terminated, and canceled burst-mode accesses, when these are caused by the processor, until the processor asserts $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$. If the slave continues to assert $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ after $\overline{\text{IBREQ}}$ or $\overline{\text{DBREQ}}$ is deasserted, the slave should be prepared to accept any new request during the cycle in which $\overline{\text{IREQ}}$ or $\overline{\text{DREQ}}$ is asserted to begin the new access. The reason for this is that the processor may attempt to establish a burst-mode access for the new access: if the slave is asserting $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ because of a previously preempted, terminated, or canceled burst-mode access, the processor interprets the active $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ as establishing the new burst-mode access and removes the request in the following cycle.

The processor preempts a burst-mode access when an external channel master arbitrates for the channel, or when a burst-mode fetch crosses a potential virtual-page boundary. Since the minimum page size is 1 kb, burst-mode instruction and data accesses are preempted whenever the address sequence crosses a 1-kb address boundary. The burst is reestablished as soon as a new address translation is performed (if required). A new physical address is transmitted when the burst-mode access is reestablished.

Note that the preemption resulting from page boundaries is advantageous for devices or memories that require counters to follow the burst-mode address sequence. Since all burst-mode accesses are word accesses and the processor retransmits an address at every 1-kb address boundary, an 8-bit counter in the slave device or memory is sufficient to follow the burst-mode address sequence. Additional address bits are simply latched.

The processor terminates a burst-mode access whenever all required instructions or data have been accessed. In the case of instruction accesses, the burst-mode access is terminated when a nonsequential fetch occurs. In the case of data accesses, the burst-mode access is terminated when the count indicates a single load or store remains. The last load or store is executed as a simple access.

The processor cancels a burst-mode access when an interrupt or trap is taken. Note that a trap may be caused by the burst-mode access, for example when a Translation Look-Aside Buffer miss occurs on an address in the burst-mode sequence. If the processor cancels a burst-mode access when an access in the sequence remains to be completed, this access must be completed in spite of the cancellation.

Canceled burst-mode data accesses may be restarted at some (possibly much later) point in execution via the Channel Address, Channel Data, and Channel Control registers. In this case, the burst-mode access is restarted at the point at which it was canceled, rather than at the beginning of the original address sequence.

Slave Preemption and Cancellation

The slave device or memory involved in a burst-mode access may preempt the access by deasserting $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$. The processor samples $\overline{\text{IBACK}}$ and $\overline{\text{DBACK}}$ when $\overline{\text{IRDY}}$ and $\overline{\text{DRDY}}$ are active so that $\overline{\text{IBACK}}$ and $\overline{\text{DBACK}}$ may be deasserted as the last supported access is completed. However, $\overline{\text{IBACK}}$ and $\overline{\text{DBACK}}$ also may be deasserted in any cycle before the access is completed. If $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ is deasserted when the processor is in a state where it expects an access, the access must be completed.

In general, the slave device or memory preempts the burst-mode access whenever it cannot support any further accesses in the burst-mode sequence. This normally occurs whenever an implementation-dependent address boundary is encountered (e.g., a cache-block boundary), but may occur for any reason. By preempting the burst-mode access, the slave receives a new request with the address of the next instruction or data word required by the processor.

The slave device or memory may cancel a burst-mode access by asserting $\overline{\text{IERR}}$ or $\overline{\text{DERR}}$ in response to a requested access. The signals $\overline{\text{IBACK}}$ or $\overline{\text{DBACK}}$ need not be deasserted at this time, but should be deasserted in the next cycle.

Note that the $\overline{\text{IERR}}$ and $\overline{\text{DERR}}$ signals cause non-maskable traps, except in the case where $\overline{\text{IERR}}$ is asserted for an instruction that the processor does not execute.

Arbitration

External masters can gain access to the address, data, and instruction buses by asserting the $\overline{\text{BREQ}}$ input. The processor completes any pending access, preempts any burst-mode access, and asserts the $\overline{\text{BGRT}}$ output. At this time, the processor places all channel outputs associated with the address, data, and instruction buses in the high-impedance state.

For the first cycle in which $\overline{\text{BGRT}}$ is asserted, the output $\overline{\text{BINV}}$ is also asserted. If the external master cannot control the address bus and associated controls in the cycle where $\overline{\text{BGRT}}$ is asserted, the active level on $\overline{\text{BINV}}$ may be used to define an idle cycle for the channel (i.e., any spurious access requests are ignored). The $\overline{\text{BINV}}$ signal is asserted only for a single cycle, so the external master must take control of the channel in the cycle after $\overline{\text{BGRT}}$ is asserted.

While the $\overline{\text{BREQ}}$ input remains asserted, the processor continues to assert $\overline{\text{BGRT}}$. The external master has control over the channel during this time.

To release the channel to the processor, the external master deasserts $\overline{\text{BREQ}}$, but must continue to control the channel for the first cycle in which $\overline{\text{BREQ}}$ is deasserted. In the cycle after $\overline{\text{BREQ}}$ is deasserted, the processor asserts $\overline{\text{BINV}}$ and deasserts $\overline{\text{BGRT}}$; the external master should release control of the channel at this time. On the following cycle, the processor deasserts $\overline{\text{BINV}}$ and is able to use the channel. The processor reestablishes any burst-mode access preempted by arbitration.

The processor does not relinquish the channel when the $\overline{\text{LOCK}}$ signal is active. This prevents external masters from interfering with exclusive accesses.

Use of $\overline{\mathrm{BINV}}$ to Cancel an Access

Besides using the $\overline{\mathrm{BINV}}$ signal to transfer control of the channel from one master to another, the Am29000 uses the $\overline{\mathrm{BINV}}$ signal to cancel accesses after they have been initiated. To cancel an access, $\overline{\mathrm{BINV}}$ is asserted during a cycle in which $\overline{\mathrm{IREQ}}$ or $\overline{\mathrm{DREQ}}$ also is asserted. If an access is canceled, the accompanying response (using $\overline{\mathrm{IRDY}}$, $\overline{\mathrm{IERR}}$, $\overline{\mathrm{DRDY}}$ or $\overline{\mathrm{DERR}}$) is ignored during the cycle where $\overline{\mathrm{BINV}}$ is asserted; thereafter, the system should not respond to the canceled access.

The $\overline{\mathrm{BINV}}$ signal is used to cancel an instruction access in the following situations:

- when an interrupt or trap is taken
- when an instruction fetch-ahead is canceled because a target block is only partially present in the Branch Target Cache
- when an instruction TLB miss or protection violation occurs on an instruction access
- when a branch instruction is the delay instruction of another branch, and the targets of both branches are in the Branch Target Cache (in this case, the external fetch for the target of the first branch is not required)
- when the processor enters the Load Test Instruction Mode, and there is an active instruction request on the channel

The $\overline{\mathrm{BINV}}$ signal is used to cancel a data access in the following situations:

- when a data TLB miss or protection violation occurs on the data access
- when an interrupt or trap is taken in the cycle where a pipelined data access becomes a primary access

If, for data accesses, address translation is not performed and pipelined accesses are not implemented, the $\overline{\mathrm{BINV}}$ signal can be ignored by the system during the access.

When a LOADSET instruction encounters a protection violation because store access is not permitted, the processor cancels the load access with $\overline{\mathrm{BINV}}$.

Bus Sharing—Electrical Considerations

When buses are shared among multiple masters and slaves, it is important to avoid situations where these devices are driving a bus at the same time. This may occur when more than one master or slave is allowed to drive a bus in the same cycle if bus arbitration is incompletely or incorrectly performed. However, it also occurs when a master or slave releases a bus in the same cycle that another master or slave gains control, and the first master or slave is slow in disabling its bus drivers, compared to the point at which the second master or slave begins to drive the bus. The latter situation is called a bus collision in the following discussion.

In addition to the logical errors that can occur when multiple devices drive a bus simultaneously, such situations may cause bus drivers to carry large amounts of electrical current. This can have a significant impact on driver reliability and power dissipation. Since bus collisions usually occur for a small amount of time, they are of less concern, but may contribute to high-frequency electromagnetic emissions.

The Am29000 channel is defined to prevent all situations where multiple drivers are driving a bus simultaneously. However, bus collisions may be allowed to occur, depending on the system design.

In the case of the Am29000 channel, arbitration for the channel prevents the processor from driving the address and data buses at the same time as another channel master. If there is more than one external master, the system design must include some means for ensuring that only one external master gains control of the channel, and that no external master gains control of the channel at the same time as the processor.

When the processor relinquishes control of the channel to an external master, bus collisions may be prevented by not allowing the external master to drive any bus while $\overline{\mathrm{BINV}}$ is active. This ensures that all processor outputs are disabled by the time the external master takes control of the channel. However, there is nothing in the channel protocol to prevent the external master from taking control as soon as $\overline{\mathrm{BGRT}}$ is asserted.

Slave devices and memories are prevented from simultaneously driving the instruction bus or data bus by allowing only the device or memory performing a primary access to drive the appropriate bus. When a pipelined access becomes a primary access, it may drive the instruction or data bus immediately, so there is a potential bus collision if the pipelined access is performed by a slave other than the slave performing the original primary access. This bus collision may be prevented by restricting all slaves to driving the instruction and data buses in the second half-cycle (using SYSCLK, for example). Since the processor samples data only at the end of a cycle, this restriction does not affect performance.

When the processor performs a store immediately following a load, it drives the data bus for the store in the second cycle following the cycle in which the data for the load appears on the data bus. This provides a complete cycle for the slave involved in the load to disable its data drivers. The processor continues to drive the data bus until it receives a $\overline{\mathrm{DRDY}}$ or $\overline{\mathrm{DERR}}$ in response to the store; it ceases to drives the data bus in the cycle following the response.

Channel Behavior for Interrupts and Traps

If an interrupt or trap is taken, any burst-mode accesses are canceled. If a request for a pipelined access is on the address bus, this request is removed. Any other accesses are completed and no new accesses are started, other than those required for the interrupt or trap. Note that any accesses that the processor expects to complete must be completed, even though burst-mode and pipelined accesses are canceled.

When interrupt or trap processing is complete, any canceled burst-mode access transactions are reestablished using the address of the access that was to be performed next when the interrupt or trap was taken. Uncompleted pipelined accesses are restarted, either by the interrupt return sequence in the case of an instruction access, or by restarting the initiating instruction in the case of a data access.

Note that the restarting of a pipelined access is not performed by the Channel Address, Channel Data, and Channel Control registers, since these registers may be required to restart the primary access. The instruction initiating the pipelined access is not allowed to be completed until the primary access is completed, so that the Program Counter 1 (PC1) register contains the address of the initiating instruction when a pipelined access is canceled. The address in PC1 can restart this instruction on interrupt return.

Effect of the $\overline{\text{LOCK}}$ Output

The $\overline{\text{LOCK}}$ output provides synchronization and exclusion of accesses in a multiprocessor environment. $\overline{\text{LOCK}}$ has no predefined effect for a system, other than the fact that the Am29000 does not grant the channel to an external master while $\overline{\text{LOCK}}$ is active.

The $\overline{\text{LOCK}}$ output is asserted for the address cycle of the Load-and-Lock and Store-and-Lock instructions, and is asserted for both the read and write accesses of a Load and Set instruction. $\overline{\text{LOCK}}$ may also be active for an extended period of time under control of the Lock bit in the Current Processor Status Register (this capability is available only to Supervisor-mode programs).

$\overline{\text{LOCK}}$ may be defined to provide any level of resource locking for a particular system. For example, it may lock the channel, an individual device or memory, or a location within a device or memory.

When a resource is locked, it is available for access only by the processor with the appropriate access privilege. The mechanisms for restricting accesses and the methods for reporting attempted violations of the restrictions are system-dependent.

Initialization and Reset

When power is first applied to the processor, it is in an unknown state and must be placed in a known state. Also, under certain circumstances, it may be necessary to place the processor in a defined state. This is accomplished by the Reset mode, which is invoked by activating the $\overline{\text{RESET}}$ pin for the required duration. The Reset mode configures the processor state as follows:

1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any interrupt or trap conditions are ignored.
4. The Current Processor Status Register is set as shown in Figure 67.
5. The Cache Disable bit of the Configuration Register is set.
6. The Data Width Enable bit of the Configuration Register is reset.
7. The Contents Valid bit of the Channel Control Register is reset.

Except as previously noted, the contents of all general-purpose registers, special-purpose registers, and TLB registers are undefined. The contents of the Branch Target Cache are also undefined.

The Reset mode also configures the processor to initiate an instruction fetch using an address of 0. Since the ROM enable (RE) bit of the Current Processor Status is 1, this fetch is directed to external instruction read-only memory. This fetch occurs when the Reset mode is exited (i.e., when the $\overline{\text{RESET}}$ input is deasserted).

The Reset mode is invoked by asserting the $\overline{\text{RESET}}$ input and can be entered only if the SYSCLK pin is operating normally, whether or not the SYSCLK pin is being

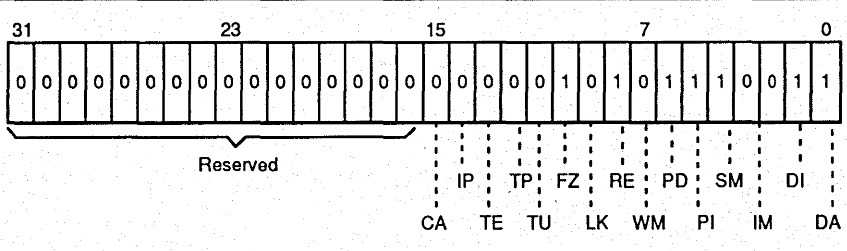


Figure 67. Current Processor Status Register In Reset Mode

driven by the processor. The Reset mode is entered within four processor cycles after $\overline{\text{RESET}}$ is asserted. The $\overline{\text{RESET}}$ input must be asserted for at least four processor cycles to accomplish a processor reset.

The Reset mode can be entered from any other processor mode (e.g., the Reset mode can be entered from the Halt mode). If the $\overline{\text{RESET}}$ input is asserted at the time that power is first applied to the processor, the processor enters the Reset mode only after four cycles have occurred on the SYSCLK pin.

The Reset mode is exited when the $\overline{\text{RESET}}$ input is deasserted. Either three or four cycles after $\overline{\text{RESET}}$ is deasserted (depending on internal synchronization time), the processor performs an initial instruction access on the channel. The initial instruction access is directed to Address 0 in the instruction read-only memory (instruction ROM). If instruction ROM is not implemented in a particular system, another device or memory must respond to this instruction fetch.

If the $CNTL_1$–$CNTL_0$ inputs are 10 or 01 when $\overline{\text{RESET}}$ is deasserted, the processor enters the Halt or Step mode, respectively. If the processor enters the Halt mode immediately after reset, the protection checking that normally applies to the Halt instruction is disabled so that the Halt instruction can be used as an instruction breakpoint in a User-mode program. The Load Test Instruction mode cannot be directly entered from the Reset mode. If the $CNTL_1$–$CNTL_0$ inputs are 00 immediately after $\overline{\text{RESET}}$ is deasserted, the effect on processor operation is unpredictable. If the $CNTL_1$–$CNTL_0$ inputs are 11, the processor enters the Executing mode.

The processor samples the $STAT_0$ output internally when $\overline{\text{RESET}}$ is asserted. A High level on $STAT_0$ in this case is used to enable a special test configuration and causes the processor to be inoperable. When $\overline{\text{RESET}}$ is asserted, the processor drives $STAT_0$ Low in order to disable this test configuration. However, if processor outputs are disabled by the Test mode, the processor is not able to drive $STAT_0$. Thus, if $\overline{\text{RESET}}$ is asserted when the processor is in the Test mode, the $STAT_0$ pin must be driven Low externally. (In a master/slave configuration, $STAT_0$ is driven Low by the master processor when $\overline{\text{RESET}}$ is asserted.)

ABSOLUTE MAXIMUM RATINGS

Storage Temperature	–65 to +150°C
Voltage on any Pin with Respect to GND	–0.5 to V_{CC} +0.5 V

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices	
Case Temperature (T_C)	0 to +85°C
Supply Voltage (V_{CC})	+4.75 to +5.25 V
Military Devices	
Case Temperature (T_C)*	–55 to +125°C
Supply Voltage (V_{CC})	+4.5 to +5.5 V

Operating ranges define those limits between which the functionality of the device is guaranteed.

*measured "instant on"

DC CHARACTERISTICS over COMMERCIAL and MILITARY operating ranges

Parameter Symbol	Parameter Description	Test Conditions	Min.	Max.	Unit
V_{IL}	Input Low Voltage		–0.5	0.8	V
V_{IH}	Input High Voltage		2.0	V_{CC}+0.5	V
$V_{ILINCLK}$	INCLK Input Low Voltage		–0.5	0.8	V
$V_{IHINCLK}$	INCLK Input High Voltage		2.0	V_{CC}+0.5	V
$V_{ILSYSCLK}$	SYSCLK Input Low Voltage		–0.5	0.8	V
$V_{IHSYSCLK}$	SYSCLK Input High Voltage		V_{CC}–0.8	V_{CC} +0.5	V
V_{OL}	Output Low Voltage for All Outputs except SYSCLK	I_{OL} = 3.2 mA		0.45	V
V_{OH}	Output High Voltage for All Outputs except SYSCLK	I_{OH} = –400 µA	2.4		V
I_{LI}	Input Leakage Current	0.45V ≤ V_{IN} ≤ V_{CC} –0.45V		±10	µA
I_{LO}	Output Leakage Current	0.45V ≤ V_{OUT} ≤ V_{CC} –0.45V		±10	µA
I_{CCOP}	Operating Power-Supply Current	V_{CC} = 5.25V, Outputs Floating; Holding $\overline{RESET}$ active with externally supplied SYSCLK		22 for Commercial 25 for Military	mA/MHz
V_{OLC}	SYSCLK Output Low Voltage	I_{OLC} = 20 mA		0.6	V
V_{OHC}	SYSCLK Output High Voltage	I_{OHC} = 20 mA	V_{CC}–0.6		V
I_{OSGND}	SYSCLK GND Short Circuit Current	V_{CC} = 5.0 V	100		mA
I_{OSVCC}	SYSCLK V_{CC} Short Circuit Current	V_{CC} = 5.0 V	100		mA

CAPACITANCE

Parameter Symbol	Parameter Description	Test Conditions	Min.	Max.	Unit
C_{IN}	Input Capacitance	fC = 1 MHz (Note 1)		15	pF
C_{INCLK}	INCLK Input Capacitance			20	pF
C_{SYSCLK}	SYSCLK Capacitance			90	pF
C_{OUT}	Output Capacitance			20	pF
$C_{I/O}$	I/O Pin Capacitance			20	pF

Note: 1. Not 100% tested.

SWITCHING CHARACTERISTICS over COMMERCIAL operating range

No.	Parameter Description	Test Conditions	33 MHz Min.	33 MHz Max.	25 MHz Min.	25 MHz Max.	Unit
1	System Clock (SYSCLK) Period (T)	Note 1			40	1000	ns
1A	SYSCLK at 1.5V to $\overline{SYSCLK}$ at 1.5V when used as an output	Note 13			0.5T−1	0.5T+1	ns
2	SYSCLK High Time when used as input	Note 13			19		ns
3	SYSCLK Low Time when used as input	Note 13			17		ns
4	SYSCLK Rise Time	Note 2				5	ns
5	SYSCLK Fall Time	Note 2				5	ns
6	Synchonous SYSCLK Output Valid Delay	Notes 3, 12			3	14	ns
6A	Synchronous SYSCLK Output Valid Delay for D_{31}–D_0	Note 12			4	18	ns
7	Three-State Synchronous SYSCLK Output Invalid Delay	Notes 4, 14, 15			3	30	ns
8	Synchronous $\overline{SYSCLK}$ Output Valid Delay	Notes 5, 12			3	14	ns
8A	Three-State $\overline{SYSCLK}$ Synchronous Output Invalid Delay	Notes 5, 14, 15			3	30	ns
9	Synchronous Input Setup Time	Note 7			12		ns
9A	Synchronous Input Setup Time for D_{31}–D_0, I_{31}–I_0				6		ns
9B	Synchronous Input Setup Time for $\overline{DRDY}$				13		ns
10	Synchronous Input Hold Time	Note 6			2		ns
11	Asynchronous Input Minimum Pulse Width	Note 8			T+10		ns
12	INCLK Period				20	500	ns
12A	INCLK to SYSCLK Delay				2	10	ns
12B	INCLK to $\overline{SYSCLK}$ Delay				2	10	ns
13	INCLK Low Time				8		ns
14	INCLK High Time				8		ns
15	INCLK Rise Time					5	ns
16	INCLK Fall Time					5	ns
17	INCLK to Deassertion of $\overline{RESET}$ (for phase synchronization of SYSCLK)	Note 9			0	5	ns
18	$\overline{WARN}$ Asynchronous Deassertion Hold Minimum Pulse Width	Note 10			4T		ns
19	$\overline{BINV}$ Synchronous Output Valid Delay from $\overline{SYSCLK}$	Note 12			1	7	ns
20	Three-State synchronous SYSCLK output invalid delay for D_{31}–D_0	Notes 11, 14, 15			3	20	ns

SWITCHING CHARACTERISTICS over COMMERCIAL operating range

No.	Parameter Description	Test Conditions	20 MHz Min.	20 MHz Max.	16 MHz Min.	16 MHz Max.	Unit
1	System Clock (SYSCLK) Period (T)	Note 1	50	1000	60	1000	ns
1A	SYSCLK at 1.5V to $\overline{\text{SYSCLK}}$ at 1.5V when used as an output	Note 13	0.5T−1	0.5T+1	0.5T−2	0.5T+2	ns
2	SYSCLK High Time when used as input	Note 13	22		27		ns
3	SYSCLK Low Time when used as input	Note 13	19		22		ns
4	SYSCLK Rise Time	Note 2		5		5	ns
5	SYSCLK Fall Time	Note 2		5		5	ns
6	Synchonous SYSCLK Output Valid Delay	Notes 3, 12	3	16	3	16	ns
6A	Synchronous SYSCLK Output Valid Delay for D_{31}–D_0	Note 12	4	20	4	20	ns
7	Three-State Synchronous SYSCLK Output Invalid Delay	Notes 4, 14, 15	3	30	3	30	ns
8	Synchronous $\overline{\text{SYSCLK}}$ Output Valid Delay	Notes 5, 12	3	16	3	16	ns
8A	Three-State $\overline{\text{SYSCLK}}$ Synchronous Output Invalid Delay	Notes 5, 14, 15	3	30	3	30	ns
9	Synchronous Input Setup Time	Note 7	15		15		ns
9A	Synchronous Input Setup Time for D_{31}–D_0, I_{31}–I_0		8		8		ns
9B	Synchronous Input Setup Time for $\overline{\text{DRDY}}$		16		16		ns
10	Synchronous Input Hold Time	Note 6	2		2		ns
11	Asynchronous Input Minimum Pulse Width	Note 8	T+10		T+10		ns
12	INCLK Period		25	500	30	500	ns
12A	INCLK to SYSCLK Delay		2	12	2	15	ns
12B	INCLK to $\overline{\text{SYSCLK}}$ Delay		2	12	2	15	ns
13	INCLK Low Time		10		12		ns
14	INCLK High Time		10		12		ns
15	INCLK Rise Time			5		5	ns
16	INCLK Fall Time			5		5	ns
17	INCLK to Deassertion of $\overline{\text{RESET}}$ (for phase synchronization of SYSCLK)	Note 9	0	5	0	5	ns
18	$\overline{\text{WARN}}$ Asynchronous Deassertion Hold Minimum Pulse Width	Note 10	4T		4T		ns
19	$\overline{\text{BINV}}$ Synchronous Output Valid Delay from $\overline{\text{SYSCLK}}$	Note 12	1	8	1	9	ns
20	Three-State synchronous SYSCLK output invalid delay for D_{31}–D_0	Notes 11, 14, 15	3	25	3	25	ns

SWITCHING CHARACTERISTICS over MILITARY operating range

No.	Parameter Description	Test Conditions	20 MHz Min.	20 MHz Max.	16 MHz Min.	16 MHz Max.	Unit
1	System Clock (SYSCLK) Period (T)	Note 1	50	1000	60	1000	ns
1A	SYSCLK at 1.5V to $\overline{\text{SYSCLK}}$ at 1.5V when used as an output	Note 13	0.5T−1	0.5T+1	0.5T−2	0.5T+2	ns
2	SYSCLK High Time when used as input	Note 13	22		27		ns
3	SYSCLK Low Time when used as input	Note 13	19		22		ns
4	SYSCLK Rise Time	Note 2		5		5	ns
5	SYSCLK Fall Time	Note 2		5		5	ns
6	Synchonous SYSCLK Output Valid Delay	Notes 3, 12	3	16	3	16	ns
6A	Synchronous SYSCLK Output Valid Delay for D_{31}–D_0	Note 12	4	20	4	20	ns
7	Three-State Synchronous SYSCLK Output Invalid Delay	Notes 4, 14, 15	3	30	3	30	ns
8	Synchronous $\overline{\text{SYSCLK}}$ Output Valid Delay	Notes 5, 12	3	16	3	16	ns
8A	Three-State $\overline{\text{SYSCLK}}$ Synchronous Output Invalid Delay	Notes 5, 14, 15	3	30	3	30	ns
9	Synchronous Input Setup Time	Note 7	15		15		ns
9A	Synchronous Input Setup Time for D_{31}–D_0, I_{31}–I_0		8		8		ns
9B	Synchronous Input Setup Time for $\overline{\text{DRDY}}$		16		16		ns
10	Synchronous Input Hold Time	Note 6	2		2		ns
11	Asynchronous Input Minimum Pulse Width	Note 8	T+10		T+10		ns
12	INCLK Period		25	500	30	500	ns
12A	INCLK to SYSCLK Delay			12		15	ns
12B	INCLK to $\overline{\text{SYSCLK}}$ Delay			12		15	ns
13	INCLK Low Time		10		12		ns
14	INCLK High Time		10		12		ns
15	INCLK Rise Time			5		5	ns
16	INCLK Fall Time			5		5	ns
17	INCLK to Deassertion of $\overline{\text{RESET}}$ (for phase synchronization of SYSCLK)	Note 9	0	5	0	5	ns
18	$\overline{\text{WARN}}$ Asynchronous Deassertion Hold Minimum Pulse Width	Note 10	4T		4T		ns
19	$\overline{\text{BINV}}$ Synchronous Output Valid Delay from $\overline{\text{SYSCLK}}$	Note 12	1	8	1	9	ns
20	Three-State synchronous SYSCLK output invalid delay for D_{31}–D_0	Notes 11, 14, 15	4	25	4	25	ns

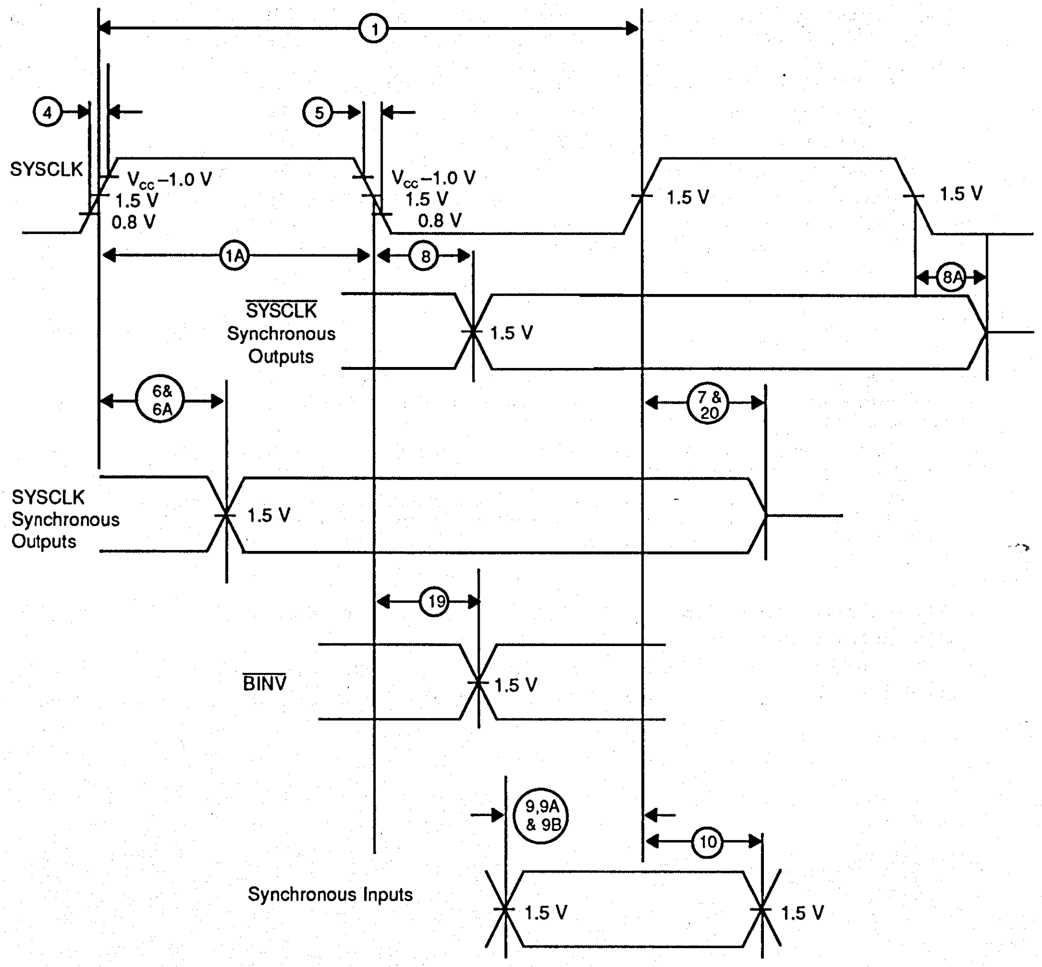
Notes:

1. AC measurements made relative to 1.5 V, except where noted.
2. SYSCLK rise and fall times measured between 0.8 V and (V_{CC} – 1.0 V).
3. Synchronous Outputs relative to SYSCLK rising edge include: A_{31}–A_0, $\overline{\text{BGRT}}$, R/$\overline{\text{W}}$, SUP/$\overline{\text{US}}$, $\overline{\text{LOCK}}$, MPGM_1–MPGM_0, $\overline{\text{IREQ}}$, IREQT, $\overline{\text{PIA}}$, $\overline{\text{DREQ}}$, DREQT_1–DREQT_0, $\overline{\text{PDA}}$, OPT_2–OPT_0, STAT_2–STAT_0, and MSERR.
4. Three-state Synchronous Outputs relative to SYSCLK rising edge include: A_{31}–A_0, R/$\overline{\text{W}}$, SUP/$\overline{\text{US}}$, $\overline{\text{LOCK}}$, MPGM_1–MPGM_0, $\overline{\text{IREQ}}$, IREQT, $\overline{\text{PIA}}$, $\overline{\text{DREQ}}$, DREQT_1–DREQT_0, $\overline{\text{PDA}}$, and OPT_2–OPT_0.
5. Synchronous Outputs relative to SYSCLK falling edge ($\overline{\text{SYSCLK}}$): $\overline{\text{IBREQ}}$, $\overline{\text{DBREQ}}$.
6. Synchronous Inputs include: $\overline{\text{BREQ}}$, $\overline{\text{PEN}}$, $\overline{\text{IRDY}}$, $\overline{\text{IERR}}$, $\overline{\text{IBACK}}$, $\overline{\text{DERR}}$, $\overline{\text{DBACK}}$, $\overline{\text{CDA}}$, I_{31}–I_0, $\overline{\text{DRDY}}$, and D_{31}–D_0.
7. Synchronous Inputs include: $\overline{\text{BREQ}}$, $\overline{\text{PEN}}$, $\overline{\text{IRDY}}$, $\overline{\text{IERR}}$, $\overline{\text{IBACK}}$, $\overline{\text{DERR}}$, $\overline{\text{DBACK}}$, and $\overline{\text{CDA}}$.
8. Asynchronous Inputs include: $\overline{\text{WARN}}$, $\overline{\text{INTR}}_3$–$\overline{\text{INTR}}_0$, $\overline{\text{TRAP}}_3$–$\overline{\text{TRAP}}_0$, and CNTL_1–CNTL_0.
9. $\overline{\text{RESET}}$ is an asynchronous input on assertion/deassertion. As an option to the user, $\overline{\text{RESET}}$ deassertion can be used to force the state of the internal divide-by-two flip-flop to synchronize the phase of SYSCLK (if internally generated) relative to $\overline{\text{RESET}}$/INCLK.
10. $\overline{\text{WARN}}$ has a minimum pulse width requirement upon deassertion.
11. To guarantee Store/Load with one-cycle memories, D_{31}–D_0 must be asserted relative to SYSCLK falling edge from an external drive source.
12. Refer to Capacitive Output Delay table when capacitive loads exceed 80 pF.
13. When used as an input, SYSCLK presents a 90-pF max. load to the external driver. When SYSCLK is used as an output, timing is specified with an external load capacitance of ≤ 200 pF.
14. Three-State Output Inactive Test Load. Three-State Synchronous Output Invalid Delay is measured as the time to a ±500 mV change from prior output level.
15. When a three-state output makes a synchronous transition from a valid logic level to a high-impedance state, data is guaranteed to be held valid for an amount of time equal to the lesser of the minimum Three-State Synchronous Output Invalid Delay and the minimum Synchronous Output Valid Delay.

Conditions:

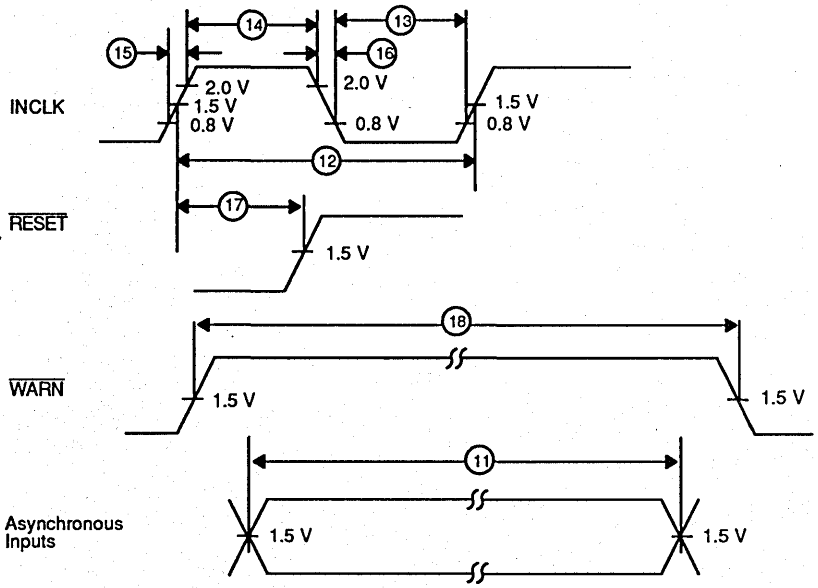
a. All inputs/outputs are TTL compatible for V_{IH}, V_{IL}, V_{OH}, and V_{OL} unless otherwise noted.
b. All output timing specifications are for 80 pF of loading.
c. All setup, hold, and delay times are measured relative to SYSCLK or INCLK unless otherwise noted.
d. All input Low levels must be driven to 0.45 V and all input High levels must be driven to 2.4 V except SYSCLK.

SWITCHING WAVEFORMS



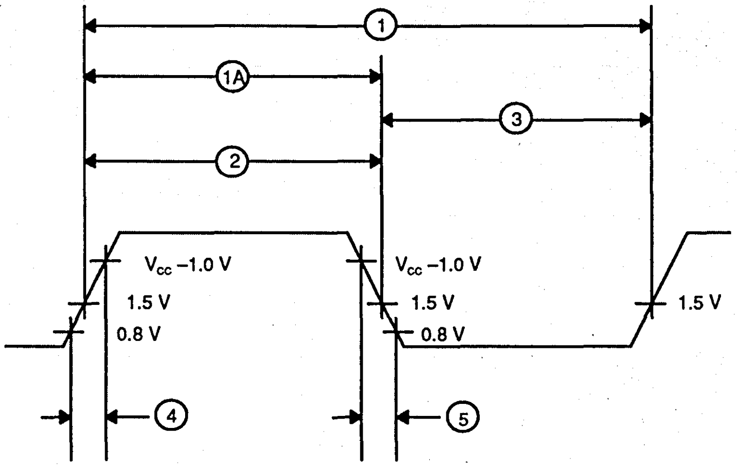
Relative to SYSCLK

SWITCHING WAVEFORMS

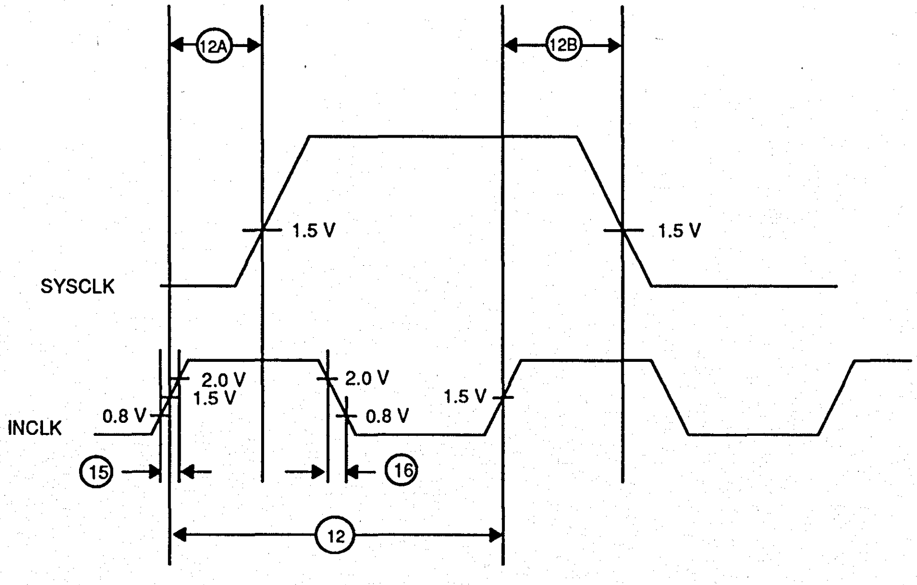


INCLK and Asynchronous Inputs

SWITCHING WAVEFORMS



SYSCLK Definition



INCLK to SYSCLK Delay

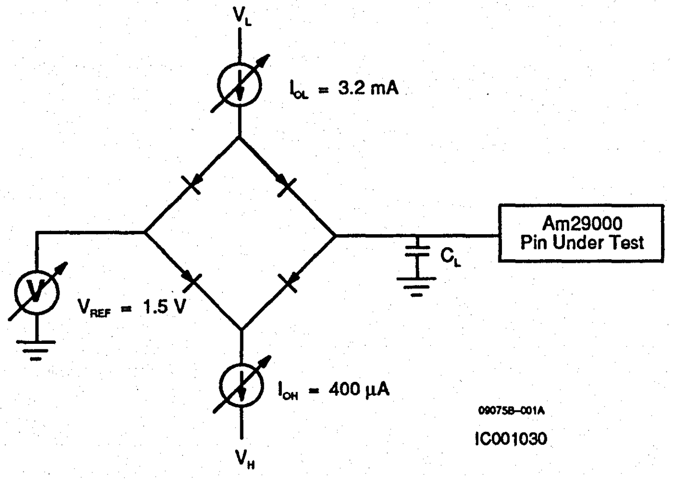
Capacitive Output Delays

For loads greater than 80 pF

This table describes the additional output delays for capacitive loads greater than 80 pF. Values in the Maximum Additional Delay column should be added to the value listed in the Switching Characteristics table. For loads less than or equal to 80 pF, refer to the delays listed in the Switching Characteristics table.

No.	Parameter Description	Total External Capacitance	Maximum Additional Delay
6	Synchronous SYSCLK Output Valid Delay	100 pF	+1 ns
		150 pF	+2 ns
		200 pF	+4 ns
		250 pF	+6 ns
		300 pF	+8 ns
6A	Synchronous SYSCLK Output Valid Delay for D_{31}–D_0	100 pF	+1 ns
		150 pF	+6 ns
		200 pF	+10 ns
		250 pF	+15 ns
		300 pF	+19 ns
8	Synchronous $\overline{\text{SYSCLK}}$ Output Valid Delay	100 pF	+1 ns
		150 pF	+2 ns
		200 pF	+4 ns
		250 pF	+6 ns
		300 pF	+8 ns
19	$\overline{\text{BINV}}$ Synchronous Output Valid Delay from $\overline{\text{SYSCLK}}$	100 pF	+1 ns
		150 pF	+3 ns
		200 pF	+4 ns
		250 pF	+6 ns
		300 pF	+7 ns

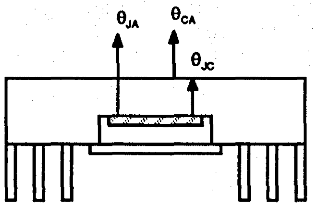
SWITCHING TEST CIRCUIT



C_L is guaranteed to 80 pF. For capacitive loading greater than 80 pF, refer to the Capacitive Output Delay table.

Am29000 Thermal Characteristics

Pin-Grid-Array Package

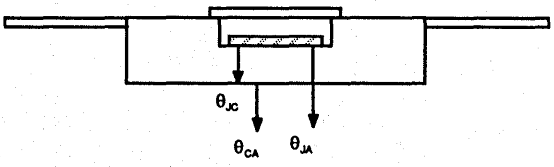


$\theta_{JA} = \theta_{JC} + \theta_{CA}$

Thermal Resistance – °C/Watt

Parameter	Airflow—ft./min. (m/sec) 0 (0)	150 (0.76)	300 (1.53)	480 (2.45)	700 (3.58)	900 (4.61)
θ_{JC} Junction-to-Case	2	2	2	2	2	2
θ_{CA} Case-to-Ambient (no Heatsink)	18	16	14	13	11	10
θ_{CA} Case-to-Ambient (with omnidirectional 4-Fin Heatsink, Thermalloy 0417261)	10	6	3	2	2	2
θ_{CA} Case-to-Ambient (with unidirectional Pin Fin Heatsink, Wakefield 840-20)	10	6	3	2	2	2

Ceramic-Quad-Flat-Pack Package



$\theta_{JA} = \theta_{JC} + \theta_{CA}$

IC001040

Thermal Resistance – °C/Watt

Parameter	Airflow—ft./min. (m/sec) 0 (0)	150 (0.76)	300 (1.53)	480 (2.45)	700 (3.58)	900 (4.61)
θ_{JC} Junction-to-Case						
θ_{CA} Case-to-Ambient						

Note: This is for reference only.

Preliminary

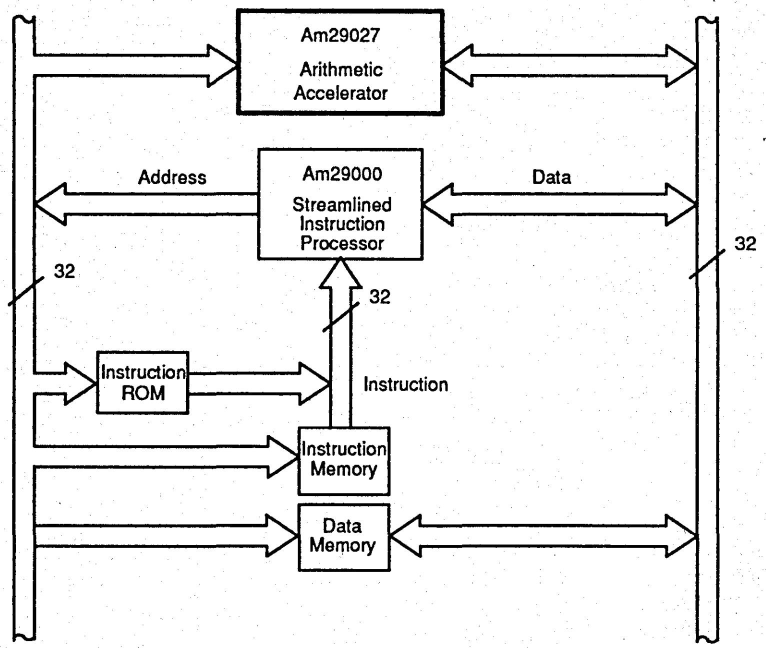
Advanced Micro Devices

Am29027
Arithmetic Accelerator

DISTINCTIVE CHARACTERISTICS

- **High-speed floating-point accelerator for the Am29000™ processor**
- **Comprehensive floating-point and integer instruction sets, including addition, subtraction, and multiplication**
- **Single-, double-, and mixed-precision operations**
- **Performs conversions between precisions and between data formats**
- **Complies with seven industry-standard floating-point formats:**
  - –IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE std 754-1985), single- and double-precision
  - –DEC™ F, DEC D, and DEC G Standards
  - –IBM® System/370 single- and double-precision
- **Exact IEEE compliance for denormalized numbers with no speed penalty**
- **Simple interface requires no glue logic between Am29000 and Am29027™**
- **Eight-deep register file for intermediate results and on-chip 64-bit data path facilitate compound operations, for example, Newton-Raphson division, sum-of-products, and transcendentals**
- **Supports pipelined or flow-through operation**
- **Full compiler and assembler support for IEEE format**
- **Fabricated with Advanced Micro Devices' 1.2-micron CMOS process**

SIMPLIFIED SYSTEM DIAGRAM



09114-001C

Publication # 09114 Rev. C Amendment /0
Issue Date: October 1989

TABLE OF CONTENTS

GENERAL DESCRIPTION

The Am29027 Arithmetic Accelerator is a high-performance computational unit intended for use with the Am29000 Streamlined Instruction Processor. When added to an Am29000-based system, the Am29027 improves floating-point performance by an order of magnitude or more.

The Am29027 implements an extensive floating-point and integer instruction set, and can perform operations on single-, double-, or mixed-precision operands. The three most widely used floating-point formats—IEEE, DEC, and IBM—are supported. IEEE operations fully comply with the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE standard 754-1985), with direct implementation of special features such as gradual underflow and exception handling.

The Am29027 consists of a 64-bit ALU, a 64-bit data path, and a control unit. The ALU has three data input ports, and can perform operations requiring one, two, or three input operands. The data path comprises two 64-bit input operand registers, an 8-by-64-bit register file for storage of intermediate results, three operand selection multiplexers that provide for orthogonal selection of input operands, and an output multiplexer that allows access to the result data, the operation status, the flags, or the accelerator state. The control unit interprets transaction requests from the Am29000, and sequences the ALU and data path.

Operations can be performed in either of two modes: flow-through or pipeline. In flow-through mode, the ALU is completely combinatorial; this mode is best suited to scalar operations. Pipeline mode divides the ALU into two or three pipelined stages for use in vector operations, such as those found in graphics or signal processing.

The Am29027 connects directly to Am29000 system buses and requires no additional interface circuitry.

Fabricated with AMD's 1.2-micron CMOS technology, the Am29027 is housed in two packages: a 169-lead pin-grid-array (PGA) package, and a 164-lead ceramic-quad-flat-pack (CQFP) package for military applications.

Related AMD Products

Part No.	Description
Am29000	Streamlined Instruction Processor

29K™ Family Development Support Products

Contact your local AMD representative for information on the complete set of development support tools.

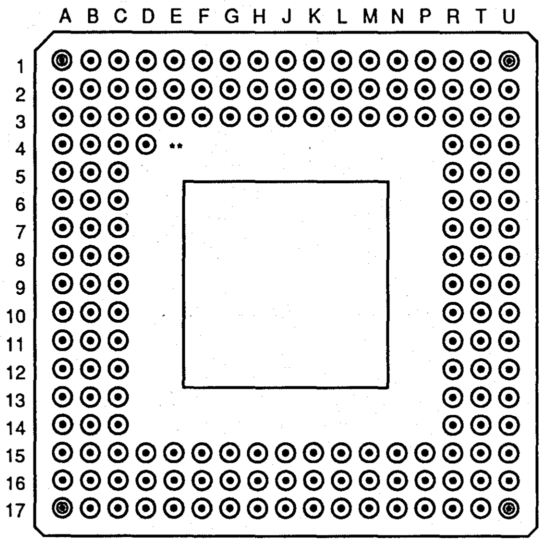
Software development products on several hosts:

- Optimizing compilers for common high-level languages
- Assembler and utility packages
- Source- and assembly-level software debuggers
- Target-resident development monitors
- Simulators

Hardware Development:

- ADAPT29K™ Advanced Development and Prototyping Tool

CONNECTION DIAGRAMS
169-Lead PGA*
Bottom View



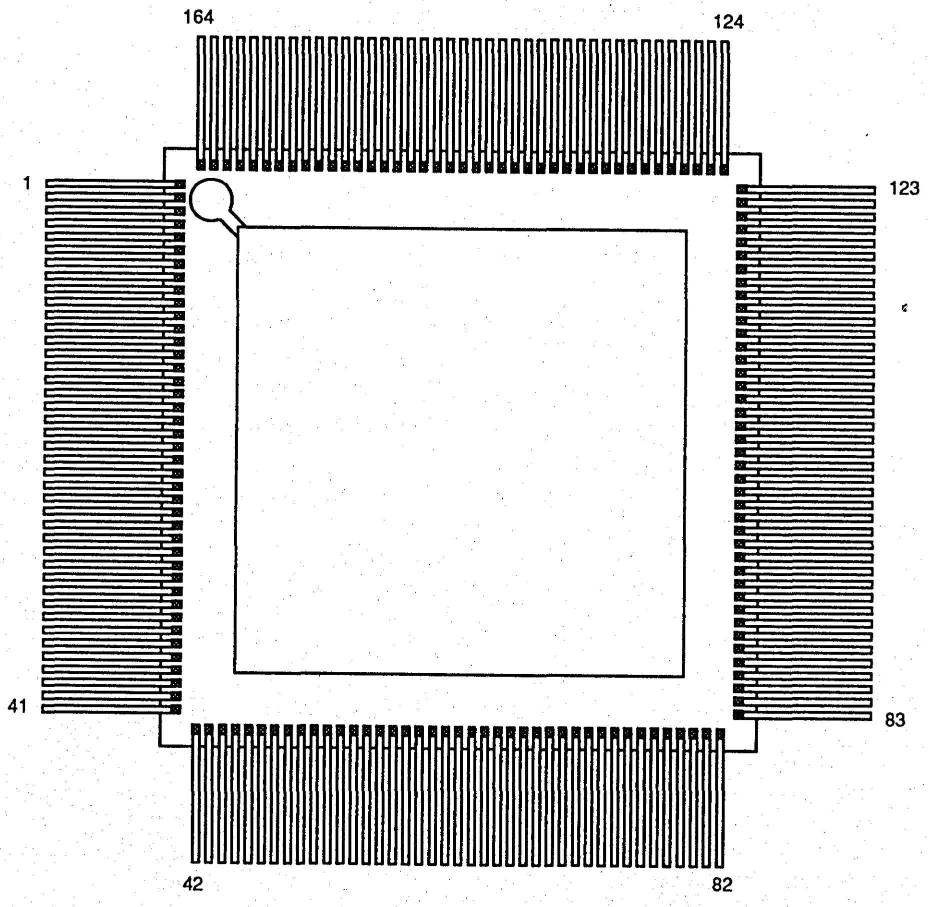
* Pinout observed from pin side of package.
**Alignment pin (not connected internally).

CD009761

CONNECTION DIAGRAMS (continued)

164-Lead CQFP*

Top View
(Lid Facing Viewer)



PGA PIN DESIGNATIONS (sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
A-1	S_{31}	C-10	F_{20}	J-16	I_{16}	R-12	DREQT$_0$
A-2	F_4	C-11	Vcco	J-17	I_{18}	R-13	$\overline{\text{RESET}}$
A-3	F_6	C-12	GNDO	K-1	S_9	R-14	$\overline{\text{DREQ}}$
A-4	F_8	C-13	F_{29}	K-2	S_{10}	R-15	I_{29}
A-5	F_{10}	C-14	GNDO	K-3	GND	R-16	I_{27}
A-6	F_{12}	C-15	Vcco	K-15	I_{21}	R-17	I_{24}
A-7	F_{14}	C-16	I_2	K-16	I_{20}	T-1	R_{28}
A-8	F_{16}	C-17	I_6	K-17	I_{19}	T-2	R_{23}
A-9	F_{18}	D-1	S_{24}	L-1	S_8	T-3	R_{21}
A-10	F_{21}	D-2	S_{25}	L-2	S_7	T-4	R_{18}
A-11	F_{22}	D-3	S_{29}	L-3	S_6	T-5	R_{16}
A-12	F_{24}	D-4	(see note)	L-15	GNDO	T-6	R_{13}
A-13	F_{27}	D-15	I_0	L-16	I_{23}	T-7	R_{10}
A-14	F_{28}	D-16	I_3	L-17	I_{22}	T-8	R_7
A-15	F_{31}	D-17	I_8	M-1	S_5	T-9	R_5
A-16	$\overline{\text{SLAVE}}$	E-1	S_{21}	M-2	S_4	T-10	R_3
A-17	I_1	E-2	S_{23}	M-3	S_2	T-11	R_0
B-1	S_{30}	E-3	S_{26}	M-15	Vcco	T-12	OPT$_1$
B-2	F_1	E-15	I_4	M-16	$\overline{\text{DRDY}}$	T-13	DREQT$_1$
B-3	F_3	E-16	I_7	M-17	$\overline{\text{CDA}}$	T-14	$\overline{\text{BINV}}$
B-4	F_5	E-17	I_9	N-1	S_3	T-15	I_{31}
B-5	F_7	F-1	S_{18}	N-2	S_1	T-16	I_{28}
B-6	F_9	F-2	S_{20}	N-3	R_{30}	T-17	I_{25}
B-7	F_{13}	F-3	S_{22}	N-15	NC	U-1	R_{25}
B-8	F_{15}	F-15	Vcc	N-16	$\overline{\text{EXCP}}$	U-2	R_{22}
B-9	F_{17}	F-16	I_{10}	N-17	$\overline{\text{DERR}}$	U-3	R_{19}
B-10	F_{19}	F-17	I_{12}	P-1	S_0	U-4	R_{17}
B-11	F_{23}	G-1	S_{15}	P-2	R_{29}	U-5	R_{15}
B-12	F_{25}	G-2	S_{17}	P-3	R_{26}	U-6	R_{14}
B-13	F_{26}	G-3	S_{19}	P-15	I_{26}	U-7	R_{11}
B-14	F_{30}	G-15	GND	P-16	NC	U-8	R_9
B-15	GND	G-16	I_{11}	P-17	NC	U-9	R_6
B-16	MSERR	G-17	I_{14}	R-1	R_{31}	U-10	R_4
B-17	I_5	H-1	S_{13}	R-2	R_{27}	U-11	R_2
C-1	S_{27}	H-2	S_{14}	R-3	R_{24}	U-12	R_1
C-2	S_{28}	H-3	S_{16}	R-4	R_{20}	U-13	OPT$_0$
C-3	F_0	H-15	GND	R-5	Vcc	U-14	OPT$_2$
C-4	F_2	H-16	I_{13}	R-6	GND	U-15	R/$\overline{\text{W}}$
C-5	Vcco	H-17	I_{15}	R-7	R_{12}	U-16	$\overline{\text{OE}}$
C-6	GNDO	J-1	S_{11}	R-8	R_8	U-17	I_{30}
C-7	F_{11}	J-2	S_{12}	R-9	GND		
C-8	GNDO	J-3	Vcc	R-10	Vcc		
C-9	Vcco	J-15	I_{17}	R-11	CLK		

Note: Pin Number D-4 = Alignment Pin.
V_{CCO} and GNDO are power and ground pins for the output buffers.
V_{CC} and GND are power and ground pins for the rest of the logic.

PGA PIN DESIGNATIONS (sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
T-14	$\overline{BINV}$	G-15	GND	B-16	MSERR	P-1	S_0
M-17	$\overline{CDA}$	H-15	GND	N-15	NC	N-2	S_1
R-11	CLK	K-3	GND	P-16	NC	M-3	S_2
N-17	$\overline{DERR}$	R-6	GND	P-17	NC	N-1	S_3
M-16	$\overline{DRDY}$	R-9	GND	U-16	$\overline{OE}$	M-2	S_4
R-14	$\overline{DREQ}$	C-6	GNDO	U-13	OPT_0	M-1	S_5
R-12	$DREQT_0$	C-8	GNDO	T-12	OPT_1	L-3	S_6
T-13	$DREQT_1$	C-12	GNDO	U-14	OPT_2	L-2	S_7
N-16	$\overline{EXCP}$	C-14	GNDO	T-11	R_0	L-1	S_8
C-3	F_0	L-15	GNDO	U-12	R_1	K-1	S_9
B-2	F_1	D-15	I_0	U-11	R_2	K-2	S_{10}
C-4	F_2	A-17	I_1	T-10	R_3	J-1	S_{11}
B-3	F_3	C-16	I_2	U-10	R_4	J-2	S_{12}
A-2	F_4	D-16	I_3	T-9	R_5	H-1	S_{13}
B-4	F_5	E-15	I_4	U-9	R_6	H-2	S_{14}
A-3	F_6	B-17	I_5	T-8	R_7	G-1	S_{15}
B-5	F_7	C-17	I_6	R-8	R_8	H-3	S_{16}
A-4	F_8	E-16	I_7	U-8	R_9	G-2	S_{17}
B-6	F_9	D-17	I_8	T-7	R_{10}	F-1	S_{18}
A-5	F_{10}	E-17	I_9	U-7	R_{11}	G-3	S_{19}
C-7	F_{11}	F-16	I_{10}	R-7	R_{12}	F-2	S_{20}
A-6	F_{12}	G-16	I_{11}	T-6	R_{13}	E-1	S_{21}
B-7	F_{13}	F-17	I_{12}	U-6	R_{14}	F-3	S_{22}
A-7	F_{14}	H-16	I_{13}	U-5	R_{15}	E-2	S_{23}
B-8	F_{15}	G-17	I_{14}	T-5	R_{16}	D-1	S_{24}
A-8	F_{16}	H-17	I_{15}	U-4	R_{17}	D-2	S_{25}
B-9	F_{17}	J-16	I_{16}	T-4	R_{18}	E-3	S_{26}
A-9	F_{18}	J-15	I_{17}	U-3	R_{19}	C-1	S_{27}
B-10	F_{19}	J-17	I_{18}	R-4	R_{20}	C-2	S_{28}
C-10	F_{20}	K-17	I_{19}	T-3	R_{21}	D-3	S_{29}
A-10	F_{21}	K-16	I_{20}	U-2	R_{22}	B-1	S_{30}
A-11	F_{22}	K-15	I_{21}	T-2	R_{23}	A-1	S_{31}
B-11	F_{23}	L-17	I_{22}	R-3	R_{24}	A-16	$\overline{SLAVE}$
A-12	F_{24}	L-16	I_{23}	U-1	R_{25}	F-15	V_{CC}
B-12	F_{25}	R-17	I_{24}	P-3	R_{26}	J-3	V_{CC}
B-13	F_{26}	T-17	I_{25}	R-2	R_{27}	R-5	V_{CC}
A-13	F_{27}	P-15	I_{26}	T-1	R_{28}	R-10	V_{CC}
A-14	F_{28}	R-16	I_{27}	P-2	R_{29}	C-5	V_{CCO}
C-13	F_{29}	T-16	I_{28}	N-3	R_{30}	C-9	V_{CCO}
B-14	F_{30}	R-15	I_{29}	R-1	R_{31}	C-11	V_{CCO}
A-15	F_{31}	U-17	I_{30}	R-13	$\overline{RESET}$	C-15	V_{CCO}
B-15	GND	T-15	I_{31}	U-15	$R/\overline{W}$	M-15	V_{CCO}

Note: Pin Number D-4 = Alignment Pin.
V_{CCO} and GNDO are power and ground pins for the output buffers.
V_{CC} and GND are power and ground pins for the rest of the logic.

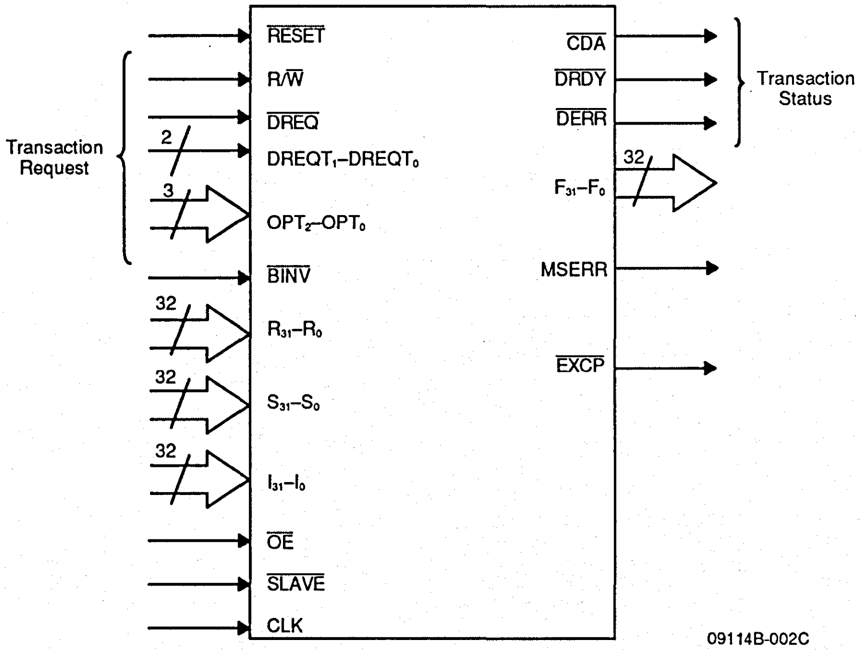
CQFP PIN DESIGNATIONS (sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
1	F_0	42	V_{CC}	83	I_{29}	124	R_{25}
2	F_1	43	GND	84	I_{28}	125	R_{26}
3	F_2	44	I_0	85	I_{31}	126	R_{27}
4	F_3	45	I_1	86	$\overline{DREQ}$	127	R_{28}
5	F_4	46	I_2	87	$\overline{OE}$	128	R_{29}
6	V_{CCO}	47	I_3	88	$\overline{BINV}$	129	R_{30}
7	GNDO	48	I_4	89	$\overline{RESET}$	130	R_{31}
8	F_5	49	I_5	90	$R/\overline{W}$	131	S_0
9	F_6	50	I_6	91	$DREQT_1$	132	S_1
10	F_7	51	I_7	92	$DREQT_0$	133	S_2
11	F_8	52	I_8	93	OPT_2	134	S_3
12	F_9	53	I_9	94	OPT_1	135	S_4
13	F_{10}	54	I_{10}	95	OPT_0	136	S_5
14	F_{11}	55	I_{11}	96	CLK	137	S_6
15	F_{12}	56	I_{12}	97	R_0	138	S_7
16	F_{13}	57	I_{13}	98	R_1	139	S_8
17	F_{14}	58	GND	99	R_2	140	S_9
18	F_{15}	59	I_{14}	100	R_3	141	S_{10}
19	GNDO	60	I_{15}	101	R_4	142	S_{11}
20	V_{CCO}	61	I_{16}	102	V_{CC}	143	GND
21	F_{16}	62	I_{17}	103	GND	144	V_{CC}
22	F_{17}	63	I_{18}	104	R_5	145	S_{12}
23	F_{18}	64	I_{19}	105	R_6	146	S_{13}
24	F_{19}	65	I_{20}	106	R_7	147	S_{14}
25	F_{20}	66	I_{21}	107	R_8	148	S_{15}
26	F_{21}	67	I_{22}	108	R_9	149	S_{16}
27	F_{22}	68	I_{23}	109	R_{10}	150	S_{17}
28	F_{23}	69	$\overline{CDA}$	110	R_{11}	151	S_{18}
29	F_{24}	70	$\overline{DRDY}$	111	R_{12}	152	S_{19}
30	F_{25}	71	$\overline{DERR}$	112	R_{13}	153	S_{20}
31	F_{26}	72	GNDO	113	R_{14}	154	S_{21}
32	V_{CCO}	73	V_{CCO}	114	R_{15}	155	S_{22}
33	GNDO	74	$\overline{EXCP}$	115	R_{16}	156	S_{23}
34	F_{27}	75	NC	116	R_{17}	157	S_{24}
35	F_{28}	76	NC	117	R_{18}	158	S_{25}
36	F_{29}	77	NC	118	R_{19}	159	S_{26}
37	F_{30}	78	I_{24}	119	R_{20}	160	S_{27}
38	F_{31}	79	I_{25}	120	R_{21}	161	S_{28}
39	GND	80	I_{26}	121	R_{22}	162	S_{29}
40	$\overline{SLAVE}$	81	I_{27}	122	R_{23}	163	S_{30}
41	MSERR	82	I_{30}	123	R_{24}	164	S_{31}

CQFP PIN DESIGNATIONS (sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
88	$\overline{BINV}$	39	GND	41	MSERR	130	R_{31}
69	$\overline{CDA}$	43	GND	75	NC	40	$\overline{SLAVE}$
96	CLK	58	GND	76	NC	131	S_0
71	$\overline{DERR}$	103	GND	77	NC	132	S_1
86	$\overline{DREQ}$	143	GND	87	$\overline{OE}$	133	S_2
92	$\overline{DREQT}_0$	7	GNDO	95	OPT_0	134	S_3
91	$\overline{DREQT}_1$	19	GNDO	94	OPT_1	135	S_4
70	$\overline{DRDY}$	33	GNDO	93	OPT_2	136	S_5
74	$\overline{EXCP}$	72	GNDO	89	$\overline{RESET}$	137	S_6
1	F_0	44	I_0	90	$R/\overline{W}$	138	S_7
2	F_1	45	I_1	97	R_0	139	S_8
3	F_2	46	I_2	98	R_1	140	S_9
4	F_3	47	I_3	99	R_2	141	S_{10}
5	F_4	48	I_4	100	R_3	142	S_{11}
8	F_5	49	I_5	101	R_4	145	S_{12}
9	F_6	50	I_6	104	R_5	146	S_{13}
10	F_7	51	I_7	105	R_6	147	S_{14}
11	F_8	52	I_8	106	R_7	148	S_{15}
12	F_9	53	I_9	107	R_8	149	S_{16}
13	F_{10}	54	I_{10}	108	R_9	150	S_{17}
14	F_{11}	55	I_{11}	109	R_{10}	151	S_{18}
15	F_{12}	56	I_{12}	110	R_{11}	152	S_{19}
16	F_{13}	57	I_{13}	111	R_{12}	153	S_{20}
17	F_{14}	59	I_{14}	112	R_{13}	154	S_{21}
18	F_{15}	60	I_{15}	113	R_{14}	155	S_{22}
21	F_{16}	61	I_{16}	114	R_{15}	156	S_{23}
22	F_{17}	62	I_{17}	115	R_{16}	157	S_{24}
23	F_{18}	63	I_{18}	116	R_{17}	158	S_{25}
24	F_{19}	64	I_{19}	117	R_{18}	159	S_{26}
25	F_{20}	65	I_{20}	118	R_{19}	160	S_{27}
26	F_{21}	66	I_{21}	119	R_{20}	161	S_{28}
27	F_{22}	67	I_{22}	120	R_{21}	162	S_{29}
28	F_{23}	68	I_{23}	121	R_{22}	163	S_{30}
29	F_{24}	78	I_{24}	122	R_{23}	164	S_{31}
30	F_{25}	79	I_{25}	123	R_{24}	42	Vcc
31	F_{26}	80	I_{26}	124	R_{25}	102	Vcc
34	F_{27}	81	I_{27}	125	R_{26}	144	Vcc
35	F_{28}	84	I_{28}	126	R_{27}	6	Vcco
36	F_{29}	83	I_{29}	127	R_{28}	20	Vcco
37	F_{30}	82	I_{30}	128	R_{29}	32	Vcco
38	F_{31}	85	I_{31}	129	R_{30}	73	Vcco

LOGIC SYMBOL



09114B-002C

ORDERING INFORMATION
Standard Products

AMD standard products are available in several packages and operating ranges. The ordering number (Valid Combination) is formed by a combination of:

a. Device Number
b. Speed Option (if applicable)
c. Package Type
d. Temperature Range
e. Optional Processing

AM29027 -25 G C B

e. OPTIONAL PROCESSING
Blank = Standard Processing
B = Burn-in

d. TEMPERATURE RANGE
C = Commercial (0 to +85°C)

c. PACKAGE TYPE
G = 169-Lead Pin Grid Array without Heatsink (CGX169)

b. SPEED OPTION
–25 = 25 MHz
–20 = 20 MHz
–16 = 16 MHz

a. DEVICE NUMBER/DESCRIPTION
Am29027
Arithmetic Accelerator

Valid Combinations	
AM29027-25	GC, GCB
AM29027-20	
AM29027-16	

Valid Combinations
Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations, to check on newly released combinations, and to obtain additional data on AMD's standard military grade products.

MILITARY ORDERING INFORMATION
APL Products

AMD products for Aerospace and Defense applications are available in several packages and operating ranges. APL (Approved Products List) products are fully compliant with MIL-STD-883C requirements. The order number (Valid Combination) is formed by a combination of

a. Device Number
b. Speed Option (if applicable)
c. Device Class
d. Package Type
e. Lead Finish

AM29027 -20 /B Z C

e. LEAD FINISH
C = Gold

d. PACKAGE TYPE
Z = 169-Lead Pin Grid Array without Heatsink (CGX169)
Y = 164-Lead Ceramic Quad Flat Pack without Heatsink

c. DEVICE CLASS
/B = Class B

b. SPEED OPTION
–20 = 20 MHz
–16 = 16 MHz

a. DEVICE NUMBER/DESCRIPTION
Am29027
Arithmetic Accelerator

Valid Combinations	
AM29027-20	/BZC, /BYC
AM29027-16	

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations or to check on newly released valid combinations.

Group A Tests

Group A tests consist of Subgroups 1, 2, 3, 7, 8, 9, 10, 11.

PIN DESCRIPTION

$\overline{\text{BINV}}$

Bus Invalid (Synchronous Input)

A logic Low indicates that the Am29000 address bus and related control signals are invalid. The Am29027 will ignore signal DREQT$_1$ when $\overline{\text{BINV}}$ is Low.

$\overline{\text{CDA}}$

Coprocessor Data Accept (Three-State Output)

A logic Low indicates that the Am29027 is ready to accept data from the Am29000. This signal is normally driven by the Am29027, and assumes a high-impedance state only if input signal $\overline{\text{OE}}$ is High or input signal $\overline{\text{SLAVE}}$ is Low.

CLK

Clock (Input)

$\overline{\text{DERR}}$

Data Error (Three-State Output)

A logic Low indicates that an unmasked exception occurred during or preceding the current transaction request. This signal is normally driven by the Am29027, and assumes a high-impedance state only if input signal $\overline{\text{OE}}$ is High or input signal $\overline{\text{SLAVE}}$ is Low.

$\overline{\text{DRDY}}$

Data Ready (Three-State Output)

A logic Low indicates that data is available on Port F. This signal is normally driven by the Am29027, and assumes a high-impedance state only if input signal $\overline{\text{OE}}$ is High or input signal $\overline{\text{SLAVE}}$ is Low.

$\overline{\text{DREQ}}$

Data Request (Synchronous Input)

A logic Low indicates that the Am29000 is making a data access. The Am29027 will ignore signal DREQT$_1$ when $\overline{\text{DREQ}}$ is High.

DREQT$_0$

Start Instruction/Suppress Errors (Synchronous Input)

This signal, when accompanied by a valid write operand R, write operand S, write operands R, S, or write instruction transaction request, commands the Am29027 to begin a new operation. When accompanying a valid read result LSBs, read result MSBs, read flags, or read status transaction request, DREQT$_0$ suppresses the reporting of operation errors. DREQT$_0$ also modifies the action of the write status transaction request to retime an operation in flow-through mode, or to invalidate the ALU pipeline in pipeline mode.

DREQT$_1$

Accelerator Transaction Request (Synchronous Input)

A logic High indicates that the Am29000 is making an accelerator transaction request. This signal is considered valid only when signal $\overline{\text{BINV}}$ is High and signal $\overline{\text{DREQ}}$ is Low.

$\overline{\text{EXCP}}$

Exception (Three-State Output)

Indicates that the status register contains one or more unmasked exception bits. This signal can be used as an interrupt or trap signal by the Am29000. $\overline{\text{EXCP}}$ is normally driven by the Am29027, and assumes a high-impedance state only if input signal $\overline{\text{OE}}$ is High or input signal $\overline{\text{SLAVE}}$ is Low.

F$_{31}$–F$_0$

F Output Bus (Three-State Output)

I$_{31}$–I$_0$

Instruction Bus (Synchronous Input)

Used to specify the operation to be performed by the accelerator.

MSERR

Master/Slave Error (Output)

Reports the result of the comparison of processor outputs with the signals provided internally to the off-chip drivers. If there is a difference for any enabled driver, MSERR assumes the logic High state.

$\overline{\text{OE}}$

Output Enable (Asynchronous Input)

A logic High forces all accelerator outputs except MSERR to assume a high-impedance state unconditionally; master/slave comparison circuitry is also disabled. This signal is provided for test purposes.

OPT$_2$–OPT$_0$

Transaction Type (Synchronous Input)

These signals, in conjunction with R/$\overline{\text{W}}$, specify the type of accelerator transaction, if any, currently being requested by the Am29000.

R$_{31}$–R$_0$

R Data Bus (Synchronous Input)

$\overline{\text{RESET}}$

Reset (Asynchronous Input)

Resets the Am29027. When $\overline{\text{RESET}}$ is a logic Low, the state of internal sequencing circuitry is initialized, and the status register is cleared. $\overline{\text{RESET}}$ must be connected to the signal line used to reset the Am29000.

R/$\overline{\text{W}}$

Read/Write (Synchronous Input)

Determines the direction of a transaction. When R/$\overline{\text{W}}$ is High, data is transferred from the Am29027 to the Am29000; when R/$\overline{\text{W}}$ is Low, data is transferred from the Am29000 to the Am29027.

S_{31}–S_0
S Data Bus (Synchronous Input)

$\overline{\text{SLAVE}}$
Master/Slave Mode Select (Synchronous Input)

A logic Low selects Slave mode; in this mode all outputs except MSERR assume a high-impedance state. A logic High selects Master mode.

FUNCTIONAL DESCRIPTION

Overview

The Am29027 is a high-performance, single-chip arithmetic accelerator for the Am29000 Streamlined Instruction Processor.

Architecture

The Am29027 comprises a high-speed ALU, a 64-bit data path, and control circuitry.

The core of the Am29027 is a 64-bit floating-point/integer ALU. The ALU takes operands from three 64-bit input ports and performs the selected operation, placing the result on a 64-bit output port. Seven ALU flags report operation status. The ALU is completely combinatorial for minimum latency; optional pipelining is available to boost throughput for vector operations.

The data path consists of two 32-bit input buses, R and S; two 64-bit input registers; two 64-bit temporary input registers; a 64-bit result register; an 8-word-by-64-bit register file for storage of intermediate results; three operand selection multiplexers that provide for orthogonal selection of input operands; an output multiplexer that selects data, operation flags, operation status, or other accelerator state; and a 32-bit output bus, F. Input operands enter the floating-point accelerator through the R and S buses, and are then demultiplexed and buffered for subsequent storage in the input registers. The operand selection multiplexers route the operands to the ALU; operation results and status leave the device on Output Bus F. Operation results also can be stored in the register file for use in subsequent operations.

On-board control circuitry sequences the ALU and data path during operations, and manages the transfer of data between the accelerator and the Am29000. A 32-bit instruction register and a 32-bit temporary instruction register hold the instruction words for current and pending operations.

Instruction Set

The Am29027 implements 57 arithmetic and logical instructions. Thirty-five instructions operate on floating-point numbers; these instructions fall into the following categories:

- addition/subtraction
- multiplication
- multiplication-accumulation
- comparison
- selecting the maximum or minimum of two numbers
- rounding to integral value
- absolute value, negation, pass
- reciprocal seed generation
- conversion between any of the supported floating-point formats, including conversions between precisions
- conversion of a floating-point number to an integer format, with an optional scale factor

By concatenating these operations, the user can also perform division, square-root extraction, polynomial evaluation, and other functions not implemented directly.

Twenty-two instructions operate on integers, and belong to the following general categories;

- addition/subtraction
- multiplication
- comparison
- selecting the maximum or minimum of two numbers
- absolute value, negation, pass
- logical operations, e.g., AND, OR, XOR, NOT
- arithmetic, logical, and funnel shifts
- conversion between single- and double-precision integer formats
- conversion of an integer number to a floating-point format, with an optional scale factor
- pass operand

One special instruction is provided to move data.

Performance

The Am29027 provides operation speeds several times greater than conventional floating-point processors by virtue of its extensive use of combinatorial, rather than sequential, logic. Most floating-point operations, whether single, double, or mixed precision, can be performed in as few as six system clock cycles. Performance is further enhanced by the presence of the on-board register file that can be used to hold intermediate results, thus reducing the amount of time needed to transfer operands between the Am29027 and the Am29000. The input operand registers and the instruction register are double-buffered, so that a new operation can be specified while the current operation is being completed.

Interface

The Am29027 connects directly to the Am29000 system buses. Am29027 operations are specified by a series of operand and instruction transactions issued by the Am29000. Eight input signals specify the transaction to be performed; three output signals report transaction status.

Master/Slave

The Am29027 contains special comparison hardware to allow the operation of two accelerators in parallel, with one accelerator (the slave) checking the results produced by the other (the master). This feature is of particular importance in the design of high-reliability systems.

Support

The Am29027 IEEE format is fully supported by those hardware and software tools available for the Am29000, including:

- HighC29K Cross-Development Toolkit
- ASM29K Cross-Development Toolkit
- ADAPT29K, a general-purpose hardware development system. The ADAPT29K permits single-step operation, break-point insertion, and other standard debugging techniques.

Block Diagram Description

A block diagram of the Am29027 is shown in Figure 1. The Am29027 comprises the input registers, the operand selection multiplexers, the instruction register, the ALU, the output register/register file, the flag register, the status register, the output multiplexer, the mode register, the control unit, and the master/slave comparator.

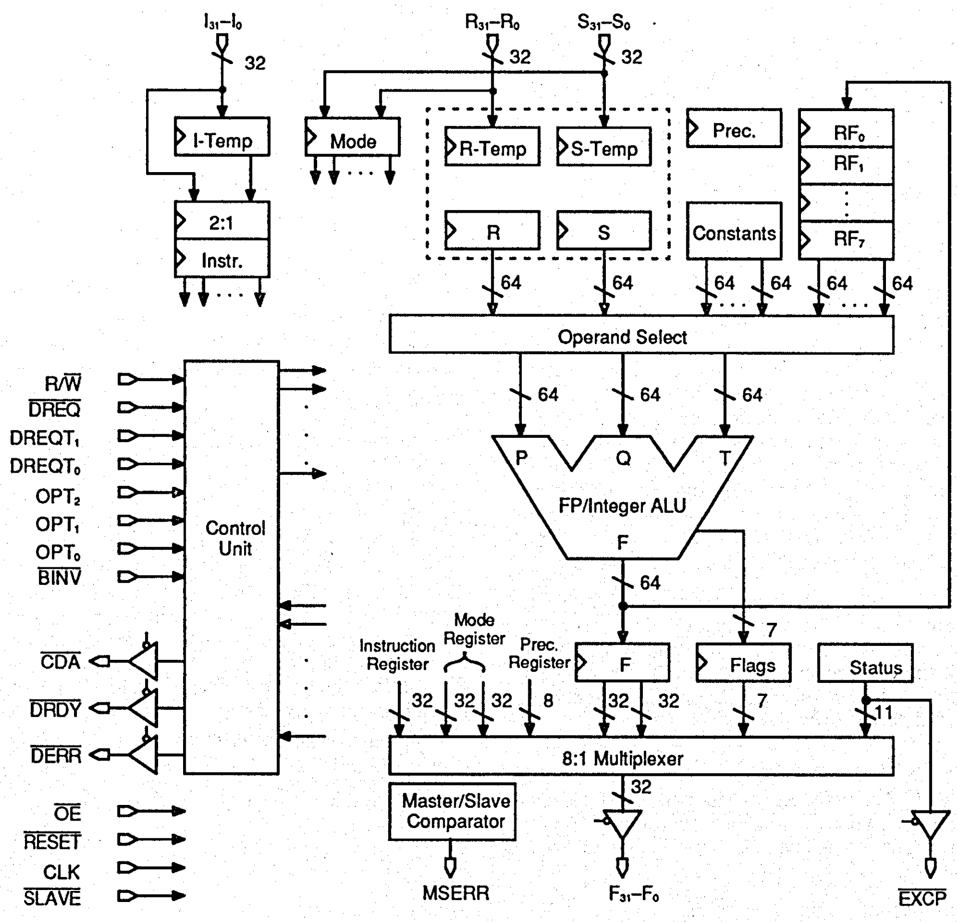


Figure 1. Am29027 Block Diagram

09114-003C

Input Registers

Operands are loaded into the accelerator via the 32-bit R and S buses, and are demultiplexed and buffered for subsequent storage in 64-bit registers R and S; input operands may be either single-precision (32-bit) or double-precision (64-bit). Two single-precision or one double-precision operand may be written to the input registers in a single system clock cycle. Accompanying the input registers are two 64-bit temporary registers, R-Temp and S-Temp, that permit the overlapping of operand transfers and ALU operations.

Operand Selection Multiplexers

The operand selection multiplexers route operands to the ALU. These multiplexers, as well as selecting operands from input registers R and S and register file locations RF_7–RF_0, also have access to a set of floating-point and integer constants. These constants are double-precision preprogrammed numbers for use in ALU operations, and are automatically provided in the appropriate format.

Instruction Register

The instruction register stores a 32-bit word specifying the current accelerator operation. Included in the instruction word are fields that specify the core operation to be performed by the ALU, operand format (integer or floating-point), sign-change selects for ALU input and result operands, operand precisions, operand sources, and register file controls. The instruction register is preceded by the 32-bit temporary register, I-Temp, permitting the overlapping of instruction transfers and ALU operations. Instructions enter the accelerator via 32-bit Instruction Bus I.

ALU

The ALU is a combinatorial arithmetic/logic unit that performs a large repertoire of floating-point and integer operations. The ALU has three operand inputs. Some operations require a single input operand, for example, conversion operations. Others, such as addition or multiplication, require two input operands. The multiplication-accumulation and funnel shift operations require three input operands. Most ALU operations allow the user to modify operand signs, thus greatly increasing the number of arithmetic expressions that can be evaluated in a single ALU pass.

The ALU can be configured in either flow-through mode, for which the ALU is completely combinatorial, or pipeline mode, for which ALU operations are divided into one or two pipeline stages.

Output Register/Register File

Operation results are stored in 64-bit output register F; results can also be stored in the 8-by-64-bit register file for use in subsequent operations. A precision register, part of the register file, contains bits indicating the precisions of the operands stored in each register file location, thus permitting the ALU to correctly process these operands in later operations.

Flag Register

The 32-bit flag register stores flags pertaining to the most recently performed operation. The flags indicate error conditions, such as underflow or overflow, and also report results for operations that produce result flags, such as comparisons.

Status Register

The 32-bit status register contains information regarding the status of past, current, and pending operations.

Six exception bits report operation error conditions. These exception bits are individually latched; once a given bit is set, it remains set until reset by the Am29000 or by system reset. The exception bits indicate error conditions of overflow, underflow, zero result, reserved operand, invalid operation, and inexact result. At the user's option, the presence of an exception can be used to report a data error to the Am29000, or to halt Am29027 operation; exception bits can be individually enabled or disabled by programming the corresponding mask bit in the mode register.

Exception bit activity is summarized by a seventh bit, Exception Status, which indicates that one or more unmasked status bits are set. If desired, the state of this bit can be placed on signal $\overline{\text{EXCP}}$, which can be used to interrupt the Am29000.

The status register contains four additional bits—R-Temp Valid, S-Temp Valid, I-Temp Valid, and Operation Pending—that pertain to the state of pending operands and operations.

Output Multiplexer

The output multiplexer routes operation results and accelerator's internal state to the Am29000 through the 32-bit F bus. This multiplexer can select Register F, the flag register, status register, instruction register, mode register, or precision register.

Mode Register

The 64-bit mode register contains accelerator control parameters that change infrequently or not at all, such as floating-point format, round mode, and operation timing information. These parameters are initialized by the Am29000 during system start-up, and are modified as required during operation.

Control Unit

The control unit manages the transfer of data between the Am29000 and the Am29027, as well as the timing of operation execution. The Am29000 oversees operation of the Am29027 by issuing one of thirteen commands, or transaction requests, to the control unit via eight signal lines. Each transaction request specifies an action on the part of the Am29027, such as writing an operand to an input register or returning a result to the Am29000. The control unit interprets the transaction request and sequences the Am29027 to produce the desired action. Three transaction status lines are generated by the control unit to indicate transaction completion, or to indicate the existence of an accelerator error condition.

Master/Slave Comparator

Each Am29027 output signal has associated logic that compares that signal with the signal that the accelerator provides internally to the output driver; any discrepancies are indicated by assertion of signal MSERR.

For a single accelerator, this output comparison detects short circuits in output signals or defective output drivers, but does not detect open circuits. It is possible to connect a second accelerator in parallel with the first, with the second accelerator's outputs disabled by assertion of signal $\overline{\text{SLAVE}}$. The second accelerator detects open-circuit signals, and provides a check of the outputs of the first accelerator.

System Interface

Am29000/Am29027 signal interconnects are depicted in Figure 2.

Three Am29027 buses—R_{31}–R_0, I_{31}–I_0, and F_{31}–F_0—are connected to Am29000 Data Bus D_{31}–D_0; the remaining Am29027 bus, S_{31}–S_0, is connected to Am29000 Address Bus A_{31}–A_0. Through these connections, the Am29000 can transfer to the Am29027 a 32-bit instruction, two 32-bit operands, or a 64-bit operand in a single cycle, or can receive a 32-bit result from the Am29027 in a single cycle.

Twelve additional signals govern communication between the Am29000 and Am29027. Eight Am29000 output signals—R/$\overline{\text{W}}$, $\overline{\text{DREQ}}$, DREQT_1, DREQT_0, OPT_2–OPT_0, and $\overline{\text{BINV}}$—are connected to the corresponding Am29027 signals and are used to issue transaction requests to the Am29027. Three Am29027 signals—$\overline{\text{CDA}}$, $\overline{\text{DRDY}}$, and $\overline{\text{DERR}}$—report transaction status. $\overline{\text{CDA}}$ is directly connected to the corresponding input of the Am29000, while $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ must be ORed with like signals from other resources. A fourth Am29027 signal, $\overline{\text{EXCP}}$, may be connected to an Am29000 trap or interrupt input to signal the presence of Am29027 operation exceptions at the user's option.

The Am29027 takes its clock input from the Am29000 SYSCLK system clock output.

The signal used to reset the Am29000 must also be connected to the Am29027 $\overline{\text{RESET}}$ input.

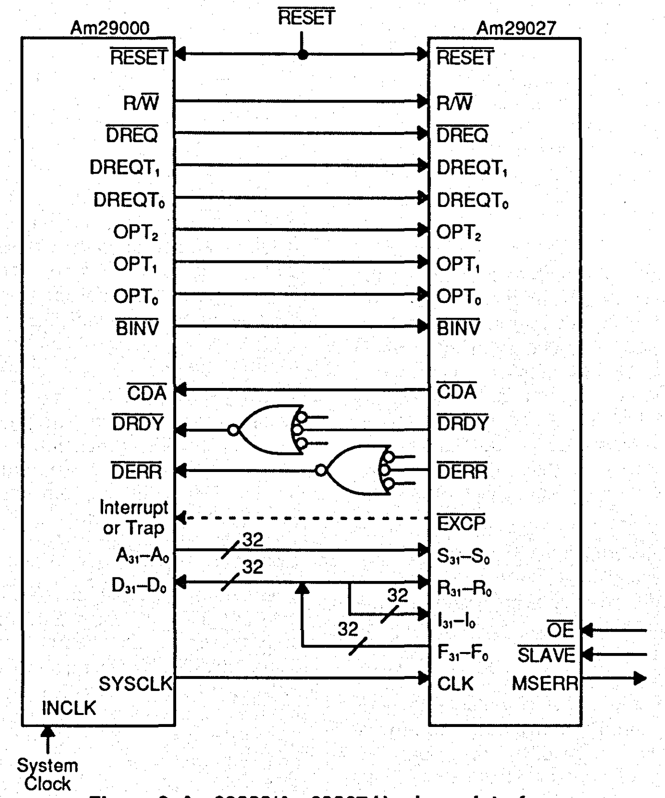


Figure 2. Am29000/Am29027 Hardware Interface

Special-Purpose Registers

The Am29027 contains six special-purpose registers: the mode register, status register, flag register, precision register, instruction register, and I-Temp register.

Mode Register

The 64-bit mode register stores 24 infrequently changed parameters pertaining to accelerator operation; its format is shown in Figure 3. The Am29000 modifies the accelerator parameter set by issuing a write mode register transaction request.

The mode register should be initialized after hardware reset, and may be written with new parameters when a new mode of accelerator operation is required; mode changes take effect immediately. The Am29027 does not alter the contents of the mode register in the course of operation.

Bits 63–47—Reserved for future use. This field must be set to 0 to assure future compatibility.

Bit 46—$\overline{\text{EXCP}}$ Enable (EX): When EX is High, reporting of unmasked exceptions via signal $\overline{\text{EXCP}}$ is enabled. When EX is Low, signal $\overline{\text{EXCP}}$ is forced inactive (logic High).

Bit 45—Halt On Error Enable (HE): When HE is High, the Am29027 will halt operation in the presence of an unmasked exception.

Bit 44—Advance $\overline{\text{DRDY}}$ (AD): When AD is High, signal $\overline{\text{DRDY}}$ is advanced one cycle in flow-through mode. This bit has no effect in pipeline mode.

Bits 43–40—Timer Count for the MOVE P Operation (MVTC): In flow-through mode, MVTC specifies the number of clock cycles needed for data to traverse the ALU for base operation code MOVE P; in pipeline mode, it has no effect. This field can assume values between 3 and 15, inclusive.

Bits 39–36—Timer Count for the Multiply-Accumulate Operation (MATC): In flow-through mode, MATC specifies the number of clock cycles needed for data to traverse the ALU for base operation code $F' = (P' \times Q') + T'$; in pipeline mode, it has no effect. This field can assume values between 3 and 15, inclusive.

Bits 35–32—Pipeline Timer Count (PLTC): In flow-through mode, PLTC specifies the number of clock cycles needed for data to traverse the ALU for any base operation code except $F' = (P' \times Q') + T'$ or MOVE P; in pipeline mode, it specifies the number of cycles needed for data to traverse a single pipeline stage for any base operation code. This field can assume values between 3 and 15, inclusive, in flow-through mode, and between 2 and 15, inclusive, in pipeline mode.

Bits 31–28—Reserved for future use. This field must be set to 0 to assure future compatibility.

Bit 27—Zero Result Exception Mask (ZMSK): When ZMSK is High, the status register zero result exception bit is masked and will not contribute to the detection of an error condition.

Bit 26—Inexact Result Exception Mask (XMSK): When XMSK is High, the status register inexact result exception bit is masked and will not contribute to the detection of an error condition.

Bit 25—Underflow Exception Mask (UMSK): When UMSK is High, the status register underflow exception bit is masked and will not contribute to the detection of an error condition.

Bit 24—Overflow Exception Mask (VMSK): When VMSK is High, the status register overflow exception bit is masked and will not contribute to the detection of an error condition.

Bit 23—Reserved Operand Exception Mask (RMSK): When RMSK is High, the status register reserved operand exception bit is masked and will not contribute to the detection of an error condition.

Bit 22—Invalid Operation Exception Mask (IMSK): When IMSK is High, the status register invalid operation exception bit is masked and will not contribute to the detection of an error condition.

Bit 21—Reserved for future use. This bit must be set to 0 to assure future compatibility.

Bit 20—Pipeline Mode Select (PL): When PL is High, pipeline mode is selected; when PL is Low, flow-through (unpipelined) mode is selected.

Bits 19–17—Reserved for future use. This field must be set to 0 to assure future compatibility.

Bits 16–14—Round Mode Select (RMS): Selects one of six rounding modes as follows:

RMS	Round Mode
0 0 0	Round to nearest (IEEE)
0 0 1	Round to minus infinity
0 1 0	Round to plus infinity
0 1 1	Round to zero
1 0 0	Round to nearest (DEC)
1 0 1	Round away from zero
1 1 X	Illegal value

Additional information on round modes can be found in Appendix B.

Bits 13–12—Integer Multiplication Format Adjust (MF): Selects the output format for integer multiplication. The user may select either the MSBs or the LSBs of an integer multiplication result, with optional format adjust. MF is encoded as follows:

MF	Output Format
0 0	LSBs
0 1	LSBs, format-adjusted
1 0	MSBs
1 1	MSBs, format-adjusted

"Format-adjusted" indicates that the product is shifted left one place before the MSBs or LSBs are selected.

Bit 11—Integer Multiplication Signed/Unsigned Select (MS): If MS is High, input operands for integer multiplication operations are treated as two's complement numbers. If MS is Low, the input operands are treated as unsigned numbers.

Bit 10—Reserved for future use. This bit must be set to 0 to assure future compatibility.

Bit 9—IBM Underflow Mask Enable (BU): If BU is High, certain underflowed IBM operations will produce a normalized result with a biased exponent increased by 128. If BU is Low, these operations will produce a final result of true zero. BU affects only those operations that produce a result in IBM format and that use the following base operation codes:

$F' = P' + T'$
$F' = P' \times Q'$
Compare P, T
$F' = (P' \times Q') + T'$

Convert T to Alternate F.P. Format
Convert T from Alternate F.P. Format
Scale T to Floating-point by Q

Bit 8—IBM Significance Mask Enable (BS): If BS is High, certain IBM operations having intermediate results of 0 will produce a final result of 0 with the biased exponent unchanged. If BS is Low, these operations will produce a final result of true zero. BS affects only those operations that produce a result in IBM format and that use the $F' = P' + Q'$ and COMPARE P, T base operation codes.

Bit 7—IEEE Sudden Underflow Enable (SU): If SU is High, all IEEE denormalized results are replaced by a 0 of the same sign; if SU is Low, the appropriate denormalized number will be produced. If IEEE traps are enabled (mode register bit TRP High), sudden underflow is disabled.

Bit 6—IEEE Trap Enable (TRP): If TRP is High, IEEE trapped operation is enabled; the Saturate Enable (SAT) and Sudden Underflow (SU) bits are ignored. For an underflowed result, the biased exponent is increased by 192 (single precision) or 1536 (double precision), with the significand unchanged. For an overflowed result, the biased exponent is decreased by a like amount with the significand unchanged. If TRP is Low, IEEE trapped operation is disabled. This bit affects only those operations that produce a result in IEEE floating-point format.

Bit 5—IEEE Affine/Projective Select (AP): If AP is High, IEEE addition or subtraction operations having infinite input operands are performed in affine mode; if AP is Low, these operations are performed in projective mode. In affine mode, it is permissible to add infinities of like sign or subtract infinities of opposite sign, producing an infinite result with the appropriate sign. In projective mode these operations will produce an invalid operation exception. This bit affects only those operations that produce a result in IEEE floating-point format.

Bit 4—Saturate Enable (SAT): If SAT is High, overflowed results are replaced by the largest representable value in the selected format of the same sign as the overflowed result; if SAT is Low, the result produced depends on the overflow conventions for the selected floating-point format. If IEEE traps are enabled (mode register bit TR High), saturation is disabled for any operation that produces a result in IEEE floating-point format.

Bits 1–0 Primary Floating-Point Format (PFF), Bits 3–2 Alternate Floating-Point Format (AFF): The primary format is used as the source and destination format for all floating-point operations except conversions, and as the source or destination format for operations that convert between floating-point and integer formats. The alternate format is used as a source or destination format in operations that convert one floating-point format to another. Both the PFF and AFF fields are encoded as follows:

High Bit	Low Bit	Format
0	0	IEEE
0	1	DEC F (Single), DEC D (Double)
1	0	DEC F (Single), DEC G (Double)
1	1	IBM

Floating-point formats are discussed in further detail in Appendix A.

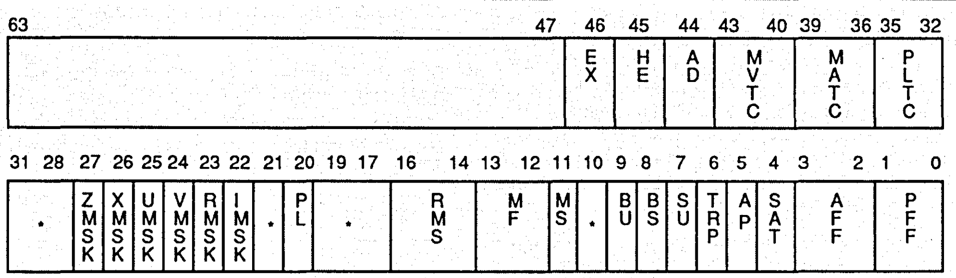


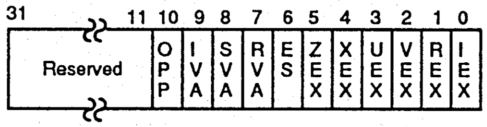
Figure 3. Mode Register

09114-005C

Status Register

The status register contains operation exception status, as well as the status of pending operands and operations; its format is shown in Figure 4. The Am29000 can initialize or modify the contents of the status register by issuing a write status transaction request, and can read current status register contents by issuing a read status transaction request or as part of a save state sequence.

All status register bits are initialized to a logic Low after hardware reset.



09114-006C

Figure 4. Status Register

Bits 31–11—Reserved for future use. This field must be set to 0 when written to assure future compatibility.

Bit 10—Operation Pending (OPP): A logic High indicates that an operation awaits execution.

Bit 9—I-Temp Valid (IVA): A logic High indicates that register I-Temp contains an instruction for a pending operation.

Bit 8—S-Temp Valid (SVA): A logic High indicates that register S-Temp contains an operand for a pending operation.

Bit 7—R-Temp Valid (RVA): A logic High indicates that register R-Temp contains an operand for a pending operation.

Bit 6—Exception Status (ES): A logic High indicates that status register bits 0–5 contain an unmasked exception.

Bit 5—Zero Result Flag (ZEX): A logic High indicates that an operation produced a zero result. Latches until cleared.

Bit 4—Inexact Result Bit (XEX): A logic High indicates that an operation result had to be rounded to fit the destination format. Latches until cleared.

Bit 3—Underflow Exception Bit (UEX): A logic High indicates that an operation result has underflowed the destination format. Latches until cleared.

Bit 2—Overflow Exception Bit (VEX): A logic High indicates that an operation result overflowed the destination format. Latches until cleared.

Bit 1—Reserved Operand Exception Bit (REX): A logic High indicates that a reserved operand appeared as an input operand to an operation or was generated as a result. Latches until cleared.

Bit 0—Invalid Operation Exception Bit (IEX): A logic High indicates that input operands are unsuitable for the operation performed (e.g., $\infty \times 0$). Latches until cleared.

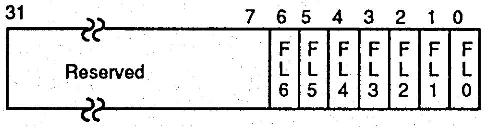
Flag Register

The flag register contains 7 flag bits that report exception or Boolean results for the most recently performed operation; its format is shown in Figure 5. The remaining 25 register bits are reserved for future use. The Am29000 can read the current flag register contents by issuing a read flags transaction request.

Flag register bits 6–0 correspond to Flag 6–Flag 0 (FL_6–FL_0).

These flags assume a meaning that is operation-dependent, as discussed in the Operation Flags section.

The flag register is made transparent in flow-through mode.



09114-007C

Figure 5. Flag Register

Precision Register

The precision register contains 8 bits that report the precision of operands stored in the register file; its format is shown in Figure 6. Bit 0 (PR_0) reports the precision of register file location 0 (RF_0), bit 1 the precision of location 1 (RF_1), and so on. A logic High indicates a single-precision value, logic Low a double-precision value.

The precision register also contains the Accelerator Release Level (ARL), an 8-bit, read-only identification number that specifies the accelerator version. The ARL field occupies bits 31–24.

The remaining 16 bits of the precision word are reserved for future use, and must be set to 0 when written to assure future compatibility.

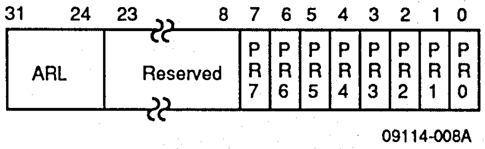


Figure 6. Precision Register

Instruction Register, I-Temp Register

The instruction register contains a 32-bit instruction word that specifies the ALU operation; its format is shown in Figure 7.

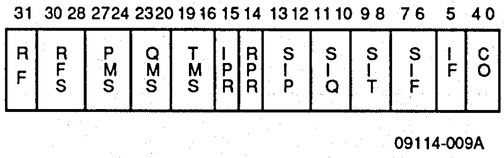


Figure 7. Instruction Register

Bit 31—Register File Enable (RF): Enables a write to the register file. When RF is High, the operation result is written to the register file location specified by RFS and the resulting precision is written to the corresponding bit of the precision register. When RF is Low, no write is performed either to the register file or the precision register.

Bits 30–28—Register file select (RFS): Selects the register file location (RF_7–RF_0) to which the operation result is to be written. If bit RF is Low, the value of RFS is a "don't care."

Bits 27–24—Select for P Operand Multiplexer (PMS): Selects the data input for the ALU P port.

Bits 23–20—Select for Q Operand Multiplexer (QMS): Selects the data input for the ALU Q port.

Bits 19–16—Select for T Operand Multiplexer (TMS): Selects the data input for the ALU T port.

Bit 15—Input Precision (IPR): Precision of the operands in Registers R and S; single precision when High, double precision when Low.

Bit 14—Result Precision (RPR): Precision of the ALU output; single precision when High, double precision when Low.

Bits 13–12—Sign P (SIP): Sign-change control for the ALU P input.

Bits 11–10—Sign Q (SIQ): Sign-change control for the ALU Q input.

Bits 9–8—Sign T (SIT): Sign-change control for the ALU T input.

Bits 7–6—Sign F (SIF): Sign-change control for the ALU output.

Bit 5—Integer/Floating-point Select (IF): A logic Low selects a floating-point operation, a logic High an integer operation.

Bits 4–0—Core Operation (CO): Specifies the core operation to be performed by the ALU.

The function of the instruction word fields is discussed in further detail in the Accelerator Instruction Set section.

The I-Temp register has a format identical to that of the instruction register; this register is used to temporarily buffer instructions for pending operations, thus allowing the overlap of operation specification and execution.

The Am29000 can write to the instruction and I-Temp registers by issuing the write instruction transaction request, and can read the contents of these registers as part of the save state sequence.

Operand Registers

The Am29027 holds operands in thirteen 64-bit registers. Four registers—R, S, R-Temp, and S-Temp—store ALU input operands; a fifth register, F, stores ALU results. Eight remaining registers, RF_7–RF_0, are arranged as a file into which operation results can be written, and from which operands can be taken for use in subsequent operations.

All operand registers share common data formats; any register can hold a single- or double-precision floating-point number, or a single- or double-precision integer.

Floating-point numbers are stored with the sign bit in the most significant bit (bit 63) of the operand register. For single-precision numbers, the 32 LSBs of the register are unused; the value of these unused bits is a "don't care."

Integer numbers are stored with the least significant bit placed in the least significant bit (bit 0) of the operand register. For single-precision numbers, the 32 MSBs of the register are unused; the value of these unused bits is a "don't care." Floating-point and integer formats are described in further detail in Appendix A.

Accelerator Transaction Requests

The Am29000 controls the Am29027 with 13 transaction requests. Transaction request type is indicated by the state of four signals: $R/\overline{W}$ and OPT_2–OPT_0. Table 1 lists the transaction types and corresponding signal states.

Transaction requests are conditioned by signal $DREQT_1$ (which when High indicates an accelerator transaction) and signals $\overline{BINV}$ and $\overline{DREQ}$. The Am29027 will recognize a transaction request only if $DREQT_1$ and $\overline{BINV}$ are High and $\overline{DREQ}$ is Low.

Signal $DREQT_0$ modifies the execution of most transaction requests. For transaction requests that transfer operands or instructions to the Am29027, asserting $DREQT_0$ will start the execution of an accelerator operation. For transaction requests that transfer operation results, status, or flags to the Am29000, asserting $DREQT_0$ will suppress the reporting of unmasked exceptions via signal $\overline{DERR}$. For the write status transaction request, asserting $DREQT_0$ either retimes the operation currently described by the instruction register (flow-through mode) or invalidates the ALU pipeline (pipeline mode).

Write Transaction Requests

Write transactions transfer data from the Am29000 to the Am29027, or cause the Am29027 to transfer data internally. To perform a write request, the Am29000:

- Issues the appropriate transaction request on signals OPT_2–OPT_0, and asserts signal $R/\overline{W}$ Low
- Places the data to be transferred, if any, on output signals D_{31}–D_0 and A_{31}–A_0

The Am29027 responds to the request by asserting one (and only one) of two status signals:

- $\overline{CDA}$ indicates that the Am29027 will take the specified action and clock in the data accompanying the transaction request, if any, on the next rising edge of clock.
- $\overline{DERR}$ indicates that the Am29027 is unable to accept the data, due to the presence of an unmasked exception.

Timing for write transactions is illustrated in Appendix D.

Table 1. Transaction Requests

$R/\overline{W}$	OPT_2	OPT_1	OPT_0	Request Type
0	0	0	0	Write Operand R
0	0	0	1	Write Operand S
0	0	1	0	Write Operands R, S
0	0	1	1	Write Mode
0	1	0	0	Write Status
0	1	0	1	Write RF Precisions
0	1	1	0	Write Instruction
0	1	1	1	Advance Temp Registers
1	0	0	0	Read Results MSBs
1	0	0	1	Read Results LSBs
1	0	1	0	Read Flags
1	0	1	1	Read Status
1	1	0	0	Save State

There are eight write transactions:

Write Operand R: An operand is written to Input Register R and/or R-Temp. The most significant half of the 64-bit operand to be written is placed on Input Bus R, the least significant half on Input Bus S. The action taken depends on signal $DREQT_0$ and on whether an accelerator operation will be in progress during the next clock cycle.

$DREQT_0$ asserted	Operation in progress next clock cycle	Data written to	R-Temp valid bit	Operation pending bit
No	X	R-Temp	Set	Unchanged
Yes	No	R-Temp, R	Reset	Reset
Yes	Yes	R-Temp	Set	Set

If $DREQT_0$ is asserted and no accelerator operation will be in progress during the next clock cycle, a new operation will be started on the next rising edge of CLK.

If mode register bit HE (Halt On Error Enable) is High and an unmasked exception has been detected, the Am29027 will respond to a write operand R request by asserting signal $\overline{DERR}$; the contents of Registers R and R-Temp will not be changed, and the R-Temp Valid and Operation Pending bits will retain their current values.

Write Operand S: An operand is written to Input Register S and/or S-Temp. The most significant half of the 64-bit operand to be written is placed on Input Bus R, the least significant half on Input Bus S. The action taken depends on signal $DREQT_0$ and on whether an accelerator operation will be in progress during the next clock cycle.

DREQT$_0$ asserted	Operation in progress next clock cycle	Data written to	S-Temp valid bit	Operation pending bit
No	X	S-Temp	Set	Unchanged
Yes	No	S-Temp, S	Reset	Reset
Yes	Yes	S-Temp	Set	Set

If DREQT$_0$ is asserted and no accelerator operation will be in progress during the next clock cycle, a new operation will be started on the next rising edge of CLK.

If mode register bit HE (Halt On Error Enable) is High and an unmasked exception has been detected, the Am29027 will respond to a write operand S request by asserting signal $\overline{\text{DERR}}$; the contents of Registers S and S-Temp will not be changed, and the S-Temp Valid and Operation Pending bits will retain their current values.

Write Operands R, S: Two 32-bit operands are written to Registers R and S and/or Registers R-Temp and S-Temp. The 32-bit operand to be written to Registers R or R-Temp is placed on Input Bus R; the 32-bit operand to be written to Registers S or S-Temp is placed on Input Bus S. Each 32-bit word is written to both the upper and lower halves of the target register. The action taken depends on signal DREQT$_0$ and on whether an accelerator operation will be in progress during the next clock cycle.

DREQT$_0$ asserted	Operation in progress next clock cycle	Data written to	R-, S-Temp valid bits	Operation pending bit
No	X	R-Temp S-Temp	Set	Unchanged
Yes	No	R-Temp S-Temp R, S	Reset	Reset
Yes	Yes	R-Temp S-Temp	Set	Set

If DREQT$_0$ is asserted and no accelerator operation will be in progress during the next clock cycle, a new operation will be started on the next rising edge of CLK.

If mode register bit HE (Halt On Error Enable) is High and an unmasked exception has been detected, the Am29027 will respond to a write operands R, S request by asserting signal $\overline{\text{DERR}}$; the contents of Registers R, R-Temp, S, and S-Temp will not be changed, and the R-Temp Valid, S-Temp Valid, and Operation Pending bits will retain their current values.

Write Mode: A 64-bit word is written to the mode register. The least significant half of the mode word is placed on Input Bus R, the most significant half on Input Bus S. The state of signal DREQT$_0$ is a "don't care" for this transaction request.

Write Status: A 32-bit word is written to the status register and the status word to be written is placed on Input Bus R. Asserting signal DREQT$_0$ will produce an additional action that is mode-dependent. In flow-through mode, asserting DREQT$_0$ will cause the operation currently specified by the instruction register to be retimed; operation results will not be written to the status register or the register file. In pipeline mode, asserting DREQT$_0$ will invalidate the ALU pipeline.

Write Register File Precisions: A 32-bit word indicating the precisions of register file locations RF_7–RF_0 is written to the precision register; the precision word to be written is placed on Input Bus R. The state of signal DREQT$_0$ is a "don't care" for this transaction request.

Write Instruction: A 32-bit accelerator instruction is written to the instruction register and/or Register I-Temp. The 32-bit instruction is taken from input signals I_{31}–I_0. The action taken depends on signal DREQT$_0$, and on whether an accelerator operation will be in progress during the next clock cycle.

DREQT$_0$ asserted	Operation in progress next clock cycle	Data written to	I-Temp valid bit	Operation pending bit
No	X	I-Temp	Set	Unchanged
Yes	No	I-Temp instruction register	Reset	Reset
Yes	Yes	I-Temp	Set	Set

If DREQT$_0$ is asserted and no accelerator operation will be in progress during the next clock cycle, a new operation will be started on the next rising edge of CLK.

If mode register bit HE (Halt On Error Enable) is High and an unmasked exception has been detected, the Am29027 will respond to a write instruction transaction request by asserting signal $\overline{\text{DERR}}$; the contents of Register I-Temp and the instruction register will not be changed, and the I-Temp Valid and Operation Pending bits will retain their current values.

Advance Temp Registers: The contents of the R-Temp, S-Temp, and I-Temp registers are transferred to Register R, Register S, and the instruction register, respectively. The state of signal DREQT$_0$ is a "don't care" for this transaction request. The advance temp registers transaction request is used during restoration of accelerator state.

Read Transaction Requests

Read transactions transfer data from the Am29027 to the Am29000. When data is to be transferred, the Am29000:

- Issues the appropriate transaction request on signals OPT_2–OPT_0, and asserts signal R/$\overline{W}$ High.
- Places its data bus drivers in a high-impedance state.

The Am29027 then places the requested data on signals F_{31}–F_0 and issues two status signals:

- $\overline{DRDY}$ indicates that the data requested is available on Output Bus F_{31}–F_0.
- $\overline{DERR}$ indicates that the Am29027 has detected an unmasked exception; the exception may or may not be related to the data requested.

$\overline{DRDY}$ and $\overline{DERR}$ may both be active at the same time; if so, the Am29000 will respond to $\overline{DERR}$ and ignore $\overline{DRDY}$.

Timing for read transactions is illustrated in Appendix D.

There are five read transactions:

Read Result MSBs: The 32 MSBs of Register F are placed on output bus F. Asserting signal $DREQT_0$ will suppress the reporting of unmasked exceptions.

Read Result LSBs: The 32 LSBs and 32 MSBs of Register F are placed on Output Bus F in consecutive clock cycles. Asserting signal $DREQT_0$ will suppress the reporting of unmasked exceptions. The read result LSBs request must always be followed by a read result MSBs request.

Read Flags: The flag register contents are placed on Output Bus F; bits F_{31}–F_7 will be logic Low. Asserting signal $DREQT_0$ will suppress the reporting of unmasked exceptions.

Read Status: The status register contents are placed on Output Bus F; bits F_{31}–F_{11} will be logic Low. Asserting signal $DREQT_0$ will suppress the reporting of unmasked exceptions.

Save State: The contents of the instruction register, mode register, status register, register file, precision register, and Registers R, R-Temp, S, S-Temp, and I-Temp are transferred to the Am29000 via Output Bus F. Exception reporting via signal $\overline{DERR}$ is suppressed; the state of signal $DRETQ_0$ is a "don't care." Further details on the use of this request appear in the Saving and Restoring State sections.

Coprocessor Data Accept

The Coprocessor Data Accept ($\overline{CDA}$) signal indicates to the Am29000 that the Am29027 is able to accept new operands or instructions. $\overline{CDA}$ is normally Low (active), but will go High if:

- The Am29027 has an operation currently in progress and a completely specified pending operation waiting in the temporary registers,

or

- The Am29027 has halted in response to an unmasked exception (Halt On Error mode enabled).

If the Am29027 issues any write transaction request and $\overline{CDA}$ is active Low, the transaction request will complete in a single cycle. If $\overline{CDA}$ is High, response to a write transaction request depends on request type:

- For the write operand R, write operand S, write operands R, S, and write instruction transaction requests, the Am29027 will assert $\overline{CDA}$ active when it is able to accept new data. If it is not able to accept new data indefinitely due to presence of an unmasked exception (Halt On Error mode enabled), it will respond to the transaction request by asserting signal $\overline{DERR}$.
- For the write mode, write status, write register file precisions, and advance temp registers transaction requests, the Am29027 will temporarily assert $\overline{CDA}$ during the cycle after the request is issued, regardless of whether an operation is in progress or an unmasked exception has halted the accelerator.

$\overline{CDA}$ pertains only to write transaction requests; for read transaction requests, the Am29000 ignores the state of $\overline{CDA}$.

Data Ready

The Data Ready ($\overline{DRDY}$) signal indicates to the Am29000 that the Am29027 is placing data on the F output bus. The Am29027 generates $\overline{DRDY}$ in response to the read result MSBs, read result LSBs, read status, read flags, and save state transaction requests.

For the read result MSBs, read result LSBs, read flags, and read status transaction requests, there is usually a minimum of one cycle delay between the time the request is issued and the time that $\overline{DRDY}$ is asserted. The only exception to this rule is when a read result LSBs request is immediately followed by a read result MSBs request, in which case the Am29027 responds to the second request in a single cycle. If the Am29027 is unable to respond immediately to a read transaction request, as may be the case when an operation is in progress, the $\overline{DRDY}$ signal will be held inactive until such a time as the requested data can be output. For the save state transaction request, the delay between the issuance of the transaction request and the $\overline{DRDY}$ response varies according to the specific data requested.

$\overline{DRDY}$ pertains only to read transaction requests; for write transaction requests, $\overline{DRDY}$ remains inactive.

Data Error

The Data Error ($\overline{DERR}$) signal indicates to the Am29000 that the Am29027 is unable to respond to a transaction request normally, due to the presence of an unmasked exception bit in the status register.

For read transaction requests, read result LSBs, read result MSBs, read flags, and read status, the Am29027 asserts $\overline{DERR}$ if the status register contains an unmasked exception bit. The Am29000 may suppress error reporting for these requests by issuing them with signal DREQT$_0$ asserted.

For write transaction requests, write operand R, write operand S, write operands R, S, and write instruction, $\overline{\text{DERR}}$ is issued in the presence of an unmasked exception if Halt On Error Mode is enabled in such an event, the contents of the target registers are left unchanged.

$\overline{\text{DERR}}$ is never issued in response to transaction requests write mode, write status, write register file precisions, advance temp registers, and save state.

Accelerator Instruction Set

The ALU performs 57 arithmetic and logic instructions. Input operands for these instructions can be taken from Input Registers R and S, register file locations RF$_7$–RF$_0$, and on-board constant stores. At the user's option, results can be stored in register file locations RF$_7$–RF$_0$.

Instruction Word

The 32-bit instruction word, IN$_{31}$–IN$_0$, specifies the operation to be performed by the ALU. The instruction word is stored in the instruction register; instruction register format is shown in Figure 7. In flow-through mode, the instruction word specifies the operation to be performed by the entire ALU. In pipeline mode, the instruction word specifies the operation to be performed by the first pipeline stage; the remaining pipeline stage or stages are controlled by their respective pipeline registers. The instruction word also specifies input operand sources, result destination, and operand precisions.

An instruction word comprises five sections: base operation code, sign-change selects, operand precision selects, operand source selects, and register file controls.

Base Operation Code

The base operation code consists of the core operation field (CO), which specifies the type of operation to be performed, and the integer/floating-point select bit (IF), which specifies whether the operation is integer or floating-point. Available base operation codes and the corresponding values for CO and IF are listed in Table 2. Note that the value of IF is a "don't care" for base operation code MOVE P.

Sign-Change Selects

Each ALU input and output port has associated hardware that can be used to modify operand signs (see Figure 8). These sign-change blocks, when applied to base operations, greatly increase the number of available operations. The base operation code $F' = P' + T'$, for example, can be used to perform operations such as $P - T$, $ABS(P) + ABS(T)$, $ABS(P + T)$, and others, simply by modifying the signs of the input and output operands. The SIP, SIQ, and SIT instruction word fields control the sign-change blocks for the P, Q, and T input operands, respectively; the SIQ and SIF fields control the sign change block for output operand F.

Using the sign-change blocks, the sign of an input operand may be left unchanged, inverted, set Low, or set High; the sign of the output operand may be left unchanged, inverted, set Low, set High, set to the sign of the P input operand, or set to the sign of the T input operand. Select codes for the P, Q, T, and F sign-change blocks are shown in Tables 3, 4, 5, and 6, respectively.

Operand Precision Selects

The Am29027 supports mixed-precision operations; it is possible, for example, to perform an operation having single-precision inputs and a double-precision output, or one single- and one double-precision input, or any other combination.

The precision of the operands in Registers R and S is specified by instruction bit IPR, which is logic High for single-precision operands and logic Low for double-precision operands. Note that the operands in the R and S registers must have the same precision if they are to be used in the same operation. This restriction does not preclude performing an operation with mixed-precision input operands, as there are no restrictions on the precisions of operands stored in the register file. The precision of each operand stored in the register file is recorded in the precision register; this precision information is automatically supplied to the ALU when a register file location is specified as an input operand to an operation.

The precision of an operation result is specified by instruction bit RPR, which is set High for a single-precision result, and Low for a double-precision result. Should the instruction word specify that the result is to be written to the register file (instruction bit RF High), the resulting precision will be written to the appropriate precision register bit when the result is written to the register file.

Table 2. Operation Codes

IF	CO					
IN_5	IN_4	IN_3	IN_2	IN_1	IN_0	**Base Operation Code (Floating-Point)**
0	0	0	0	0	0	$F' = P$
0	0	0	0	0	1	$F' = P' + T'$
0	0	0	0	1	0	$F' = P' \times Q'$
0	0	0	0	1	1	Compare P, T
0	0	0	1	0	0	Max, P, T
0	0	0	1	0	1	Min P, T
0	0	0	1	1	0	Convert T to Integer
0	0	0	1	1	1	Scale T to Integer by Q
0	0	1	0	0	0	$F' = (P' \times Q') + T'$
0	0	1	0	0	1	Round T to Integral Value
0	0	1	0	1	0	Reciprocal Seed of P
0	0	1	0	1	1	Convert T to Alternate F. P. Format
0	0	1	1	0	0	Convert T from Alternate F. P. Format
IN_5	IN_4	IN_3	IN_2	IN_1	IN_0	**Base Operation Code (Integer)**
1	0	0	0	0	0	$F = P$
1	0	0	0	0	1	$F = P + T$
1	0	0	0	1	0	$F = P \times Q$
1	0	0	0	1	1	Compare P, T
1	0	0	1	0	0	Max P, T
1	0	0	1	0	1	Min P, T
1	0	0	1	1	0	Convert T to Floating-Point
1	0	0	1	1	1	Scale T to Floating-Point by Q
1	1	0	0	0	0	F = P OR T
1	1	0	0	0	1	F = P AND T
1	1	0	0	1	0	F = P XOR T
1	1	0	0	1	1	Shift P Logical Q Places
1	1	0	1	0	0	Shift P Arithmetic Q Places
1	1	0	1	0	1	Funnel Shift PT Logical Q Places
IN_5	IN_4	IN_3	IN_2	IN_1	IN_0	**Base Operation Code (Special)**
X	1	1	0	0	0	MOVE P

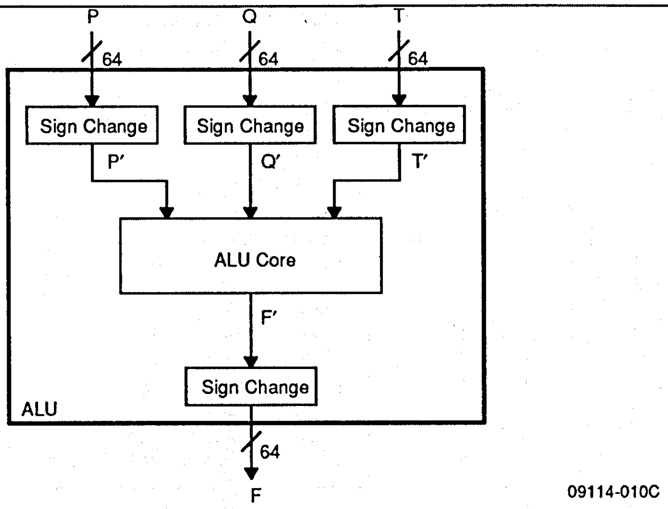


Figure 8. ALU Sign-Change Blocks

Table 3. Select Codes for P Operand Sign-Change Block

SIP		
IN_{13}	IN_{12}	SIGN (P′)
0	0	SIGN(P)
0	1	$\overline{\text{SIGN}}$ ($\overline{\text{P}}$)
1	0	0
1	1	1

Table 4. Select Codes for Q Operand Sign-Change Block

SIQ		
IN_{11}	IN_{10}	SIGN (Q′)
0	0	SIGN(Q)
0	1	$\overline{\text{SIGN}}$ ($\overline{\text{Q}}$)
1	0	0
1	1	1

Table 5. Select Codes for T Operand Sign-Change Block

SIT		
IN_9	IN_8	SIGN (T′)
0	0	SIGN(T)
0	1	$\overline{\text{SIGN}}$ ($\overline{\text{T}}$)
1	0	0
1	1	1

Table 6. Select Codes for F Operand Sign-Change Block

	SIQ		SIF		
Base Operation	IN_{11}	IN_{10}	IN_7	IN_6	SIGN(F)
F′ = P (Floating-Point)	0	X	0	0	SIGN(F′)
F = P (Integer)	0	X	0	1	$\overline{\text{SIGN}}$(F′)
<u>OR</u>	0	X	1	0	0
Maximum P, T	0	X	1	1	1
<u>OR</u>	1	0	X	X	SIGN(P)
Minimum P, T	1	1	X	X	SIGN(T)
	X	X	0	0	SIGN(F′)
<u>All Other</u> Base	X	X	0	1	$\overline{\text{SIGN}}$(F′)
Operations	X	X	1	0	0
	X	X	1	1	1

Operand Source Selects

Instruction fields PMS, QMS, and TMS specify the select codes for the P, Q, and T operand multiplexers, respectively; these codes are summarized in Table 7.

The P, Q, and T operand multiplexers can independently select Register R, Register S, register file locations RF_7–RF_0, or one of six predefined constants. For operations with floating-point inputs, constants 0, 0.5, 1, 2, 3, and pi are available; for operations with integer inputs, constants 0, –1, 1, 2, 3, and $-(2^{63})$ are available. These constants are supplied to the ALU as double-precision numbers, independent of the precisions specified for other input and result operands. Hexadecimal values for the constants are listed in Table 8.

Register File Controls

Instruction fields RF and RFS control the storing of operation results in the register file. If register file enable bit RF is High, the result of the operation specified by the instruction word will be stored in register file location RFS, where RFS is a number from 7 to 0; the precision of the result, as specified by the RPR bit, will be written to the appropriate bit in the precision register. If RF is Low, the operation result is written to neither the register file nor the precision register.

Accelerator Operations

Table 9 illustrates a number of possible ALU instructions and corresponding values for instruction word fields SIP, SIQ, SIT, SIF, IF, and CO. Note that the remaining instruction fields—RF, RFS, PMS, QMS, TMS, IPR, and RPR—can be specified independently.

The user may create additional instructions using instruction words other than those listed in Table 9. For some base operation codes, sign-change control settings SIP, SIQ, SIT, and SIF are completely arbitrary; for others, only the sign-change field values shown in Table 9 are valid. Table 10 summarizes permissible sign-change field values for each base operation code.

Table 7. Select Codes for P, Q, and T Operand Multiplexers

PMS	IN_{27}	IN_{26}	IN_{25}	IN_{24}	P
QMS	IN_{23}	IN_{22}	IN_{21}	IN_{20}	**Q**
TMS	IN_{19}	IN_{18}	IN_{17}	IN_{16}	**T**
	0	0	0	0	Register R
	0	0	0	1	Register S
	0	0	1	0	0 (Zero)
	0	0	1	1	0.5 (F.P.) – 1(integer)
	0	1	0	0	1
	0	1	0	1	2
	0	1	1	0	3
	0	1	1	1	π (F.P.) – 2^{63}(integer)
	1	0	0	0	RF_0
	1	0	0	1	RF_1
	1	0	1	0	RF_2
	1	0	1	1	RF_3
	1	1	0	0	RF_4
	1	1	0	1	RF_5
	1	1	1	0	RF_6
	1	1	1	1	RF_7

Table 8. Hexadecimal Values for On-Chip Constants

IEEE Floating-Point Constant	Hexadecimal Representation
0	0000000000000000
0.5	3FE0000000000000
1	3FF0000000000000
2	4000000000000000
3	4008000000000000
π	400921FB54442D18

DEC D Floating-Point Constant	Hexadecimal Representation
0	0000000000000000
0.5	4000000000000000
1	4080000000000000
2	4100000000000000
3	4140000000000000
π	41490FDAA22168C2

DEC G Floating-Point Constant	Hexadecimal Representation
0	0000000000000000
0.5	4000000000000000
1	4010000000000000
2	4020000000000000
3	4028000000000000
π	402921FB54442D18

IBM Floating-Point Constant	Hexadecimal Representation
0	0000000000000000
0.5	4080000000000000
1	4110000000000000
2	4120000000000000
3	4130000000000000
π	413243F6A8885A31

Integer Constant	Hexadecimal Representation
0	0000000000000000
–1	FFFFFFFFFFFFFFFF
1	0000000000000001
2	0000000000000002
3	0000000000000003
-2^{63}	8000000000000000

Table 9. Instruction Words for Typical ALU Operations

Operation	SIP	SIQ	SIT	SIF	IF	CO
FP P	00	00	XX	00	0	00000
FP – P	00	00	XX	01	0	00000
FP ABS(P)	00	00	XX	10	0	00000
FP Sign(T) × ABS(P)	00	11	XX	XX	0	00000
FP P + T	00	XX	00	00	0	00001
FP P – T	00	XX	01	00	0	00001
FP T – P	01	XX	00	00	0	00001
FP –P – T	01	XX	01	00	0	00001
FP ABS(P + T)	00	XX	00	10	0	00001
FP ABS(P – T)	00	XX	01	10	0	00001
FP ABS(P) + ABS(T)	10	XX	10	00	0	00001
FP ABS(P) – ABS(T)	10	XX	11	00	0	00001
FP ABS[ABS(P) – ABS(T)]	10	XX	11	10	0	00001
FP P × Q	00	00	XX	00	0	00010
FP (–P) × Q	01	00	XX	00	0	00010
FP ABS(P × Q)	00	00	XX	10	0	00010
FP Compare P, T	00	XX	01	00	0	00011
FP Max P, T	00	00	01	00	0	00100
FP Max ABS(P), ABS(T)	10	00	11	00	0	00100
FP Min P, T	01	00	00	00	0	00101
FP Min ABS(P), ABS(T)	11	00	10	00	0	00101
FP Limit P to Magnitude T	11	10	10	XX	0	00101
FP Convert T to Integer	XX	XX	00	00	0	00110
FP Scale T to Integer by Q	XX	00	00	00	0	00111
FP T + P × Q	00	00	00	00	0	01000
FP T – P × Q	01	00	00	00	0	01000
FP –T + P × Q	00	00	01	00	0	01000
FP –T – P × Q	01	00	01	00	0	01000
FP ABS(T) + ABS(P × Q)	10	10	10	00	0	01000
FP ABS(T) – ABS(P × Q)	11	10	10	00	0	01000
FP ABS(P × Q) – ABS(T)	10	10	11	00	0	01000
FP Round T to Integral Value	XX	XX	00	00	0	01001
FP Reciprocal Seed (P)	00	XX	XX	00	0	01010
FP Convert T to Alternate Floating-Point Format	XX	XX	00	00	0	01011
FP Convert T from Alternate Floating-Point Format	XX	XX	00	00	0	01100
int P	00	00	00	00	1	00000
int –P	00	00	00	01	1	00000
int ABS(P)	00	00	00	10	1	00000
int Sign(T) × ABS(P)	00	11	00	XX	1	00000
int P + T	00	XX	00	00	1	00001
int P – T	00	XX	01	00	1	00001
int T – P	01	XX	00	00	1	00001
int ABS(P + T)	00	XX	00	10	1	00001
int ABS(P – T)	00	XX	01	10	1	00001
int P × Q	00	00	XX	00	1	00010
int Compare P, T	00	XX	01	00	1	00011
int Max P, T	00	00	01	00	1	00100
int Min P, T	01	00	00	00	1	00101

Table 9. Instruction Words for Typical ALU Operations (continued)

Operation	SIP	SIQ	SIT	SIF	IF	CO
int Convert T to Float	XX	XX	00	00	1	00110
int Scale T to Float by Q	XX	00	00	00	1	00111
int P OR T	XX	XX	XX	XX	1	10000
int P AND T	XX	XX	XX	XX	1	10001
int P XOR T	XX	XX	XX	XX	1	10010
int NOT T (see Note 1)	XX	XX	XX	XX	1	10010
int Shift P Logical Q Places	00	00	XX	00	1	10011
int Shift P Arithmetic Q Places	00	00	XX	00	1	10100
int Funnel Shift PT Q Places	00	00	00	00	1	10101
MOVE P	XX	XX	XX	XX	X	11000

Note 1. NOT T is performed by XORing T with a word containing all 1s (integer – 1). When invoking NOT T the user must set instruction field PMS to 0011_2, thus selecting integer constant –1.

Table 10. Allowable Sign-Change Combinations

IF	CO	Operation	SIP	SIQ	SIT	SIF
0	00000	FP F′ = P	F	V	X	V
0	00001	FP F′ = P′ + T′	V	X	V	V
0	00010	FP F′ = P′ × Q′	V	V	X	V
0	00011	FP Compare P, T	F	X	F	F
0	00100	FP Max P, T	F	F	F	F
0	00101	FP Min P, T	F	F	F	F
0	00110	FP Convert T to Integer	X	X	F	F
0	00111	FP Scale T to Integer	X	F	F	F
0	01000	FP F′ = (P′ × Q′) + T	V	V	V	V
0	01001	FP Round T	X	X	F	F
0	01010	FP Reciprocal Seed P	F	X	X	F
0	01011	FP Convert T to Alt Format	X	X	F	F
0	01100	FP Convert T from Alt Format	X	X	F	F
1	00000	int F = P	F	F	F	F
1	00001	int F = P + T	F	X	F	F
1	00010	int F = P × Q	F	F	X	F
1	00011	int Compare P, T	F	X	F	F
1	00100	int Max P, T	F	F	F	F
1	00101	int Min P, T	F	F	F	F
1	00110	int Convert T to F.P.	X	X	F	F
1	00111	int Scale T to F.P.	X	F	F	F
1	10000	int F = P OR T	X	X	X	X
1	10001	int F = P AND T	X	X	X	X
1	10010	int F = P XOR T	X	X	X	X
1	10011	int Shift P Logical	F	F	X	F
1	10100	int Shift P Arithmetic	F	F	X	F
1	10101	int Funnel Shift PT	F	F	F	F
X	11000	MOVE P	X	X	X	X

Key: V = Variable; user can specify arbitrary sign change.
F = Fixed; user is restricted to sign-change combinations shown in Table 9.
X = Don't care; this field does not affect the operation or its result.

Base Operation Code Description

F′ = P (Floating-Point): The P-operand is passed through the ALU unchanged, except for any specified precision conversions. If the user specifies different input and output precisions, the operation may be used to perform single-to-double or double-to-single conversions. Instructions such as negation, absolute value extraction and sign transfer may be executed by setting the sign-change controls appropriately while executing this base operation.

F′ = P′ + T′ (Floating-Point): The two operands P′ and T′ are added, taking into account any specified precision conversions. Instructions such as subtraction, sum-of-absolute-values, difference-of-absolute-values, absolute-value-of-sum, and absolute-value-of-difference may be executed by setting the sign-change controls appropriately while executing this base operation.

F′ = P′ × Q′ (Floating-Point): The operands P′ and Q′ are multiplied, taking into account any specified precision conversions. Instructions such as negative-product and absolute-value-of-product may be executed by setting the sign-change controls appropriately while executing this base operation.

Compare P, T (Floating-Point): The two operands P and T are compared, taking into account any specified precision conversions. The output of the operation is the result of the subtraction (P – T). The flags are set appropriately to indicate the result of the comparison, conforming to the relevant parts of the floating-point standards. For IEEE and DEC operations, one of four flags (greater than, less than, equal to, or unordered) is set for any given compare operation. For IBM operations, the unordered flag does not apply since the format does not support reserved operands.

Maximum P, T (Floating-Point): The two operands P and T are compared, taking into account any specified precision conversions. The most positive operand is selected as the output. The Winner flag indicates which of the operands is selected. Additionally, the operation maximum-of-absolute-value may be performed by setting the appropriate sign-change controls.

Minimum P, T (Floating-Point): The two operands P and T are compared, taking into account any specified precision conversions. The most negative operand is selected as the output. The Winner flag indicates which of the two operands is selected. Additionally, the operations minimum-of-absolute-values and limit-P-to-magnitude-T may be performed by setting the appropriate sign-change controls. The limit-P-to-magnitude-T operation is useful for clipping a sequence of operands to ensure that their magnitude never exceeds a preset limit.

Convert T to Integer (Floating-Point): The operand T is converted from floating-point representation to two's complement integer representation, taking into account the specified precision of the floating-point operand. If the output precision is specified as single, the result is a 32-bit integer. If the output precision is specified as double, the result is a 64-bit integer.

Scale T to Integer by Q (Floating-Point): The operand T is converted from floating-point representation to two's complement integer representation, using the exponent of the floating-point operand Q as a scale factor and taking into account the specified precision of the floating-point operands. The unbiased exponent of the operand Q is added to the exponent of the operand T, permitting IEEE and DEC operands to be multiplied by any power of 2, and IBM operands by any power of 16, before the conversion is performed. If the output precision is specified as single, the result is a 32-bit integer. If the output precision is specified as double, the result is a 64-bit integer.

F′ = (P′ × Q′) + T′ (Floating-Point): The operands P′ and Q′ are multiplied, producing a double-precision product. This product is added to the operand T′, taking into account any specified precision conversions. Instructions such as P × Q – T, T – P × Q, ABS (P × Q) + ABS(T) and ABS(P × Q + T) may be executed by setting the sign-change controls appropriately while executing this base operation.

Round T to Integral Value (Floating-Point): The floating-point operand T is rounded to an integer-valued floating-point operand, using the specified rounding mode and taking into account any specified precision conversions. As an example, the operation converts a floating-point representation of Pi (3.14159 . . .) to a floating-point representation of 3.0 or 4.0, depending on the rounding mode selected. The final result of the operation is a floating-point number.

Reciprocal Seed of P (Floating-Point): An approximation to the reciprocal of the operand P is evaluated, taking into account any specified precision conversions. The reciprocal seed comprises an accurate sign, a fully-accurate exponent and a mantissa that is accurate to only one place. This operation can be used as the initial step in performing Newton-Raphson division; optionally, an external seed look-up table can be used for faster convergence.

Convert T to Alternate Floating-Point Format (Floating-Point): The floating-point operand T, assumed to be in the primary floating-point format, is converted to a floating-point operand in the alternate floating-point format, taking into account any specified precision conversions.

Convert T from Alternate Floating-Point Format (Floating-Point): The floating-point operand T, assumed to be in the alternate floating-point format, is converted to a floating-point operand in the primary floating-point format, taking into account any specified precision conversions.

F = P (Integer): The P-operand is passed through the ALU unchanged except for any specified precision conversions. If the user specifies different input and output precisions, the operation may be used to perform single-to-double or double-to-single conversions. Instructions such as negation, absolute value extraction, and sign transfer may be performed by setting the sign-change control appropriately while executing this base operation.

F = P + T (Integer): The two operands P and T are added, taking into account any specified precision conversions. Instructions such as subtraction, absolute-value-of-sum, and absolute-value-of-difference may be performed by setting the sign-change controls appropriately while executing this base operation.

F = P × Q (Integer): The two operands P and Q are multiplied, taking into account any specified precision conversions. Either 32-bit multiplication or 64-bit multiplication may be performed, and the user may select either the MSBs or the LSBs of the product as the final result. In addition, format-adjusting may be implemented if required, and the operands may be considered as signed (two's complement) or unsigned.

Compare P, T (Integer): The two operands P and T are compared, taking into account any specified precision conversions. The output of the operation is the result of the subtraction (P – T). The flags are set appropriately to indicate the result of the comparison, one of three flags (greater than, less than, or equal to) being set for any given compare operation.

Maximum P, T (Integer): The two operands P and T are compared, taking into account any specified precision conversions. The most positive operand is selected as the output. The Winner flag indicates which of the two operands is selected.

Minimum P, T (Integer): The two operands P and T are compared, taking into account any specified precision conversions. The most negative operand is selected as the output. The Winner flag indicates which of the two operands is selected.

Convert T to Floating-Point (Integer): The operand T is converted from two's complement integer representation to floating-point representation, taking into account the specified precision of the integer operand. If the output precision is specified as single, the result is a 32-bit floating-point operand. If the output precision is specified as double, the result is a 64-bit floating-point operand.

Scale T to Floating-Point by Q (Integer): The operand T is converted from two's complement integer representation to floating-point representation, using the exponent of the floating-point operand Q as a scale factor and taking into account the specified precision of the integer operand. The unbiased exponent of the operand Q is added to the exponent of the floating-point result, permitting IEEE and DEC operands to be multiplied by any power of 2, and IBM operands by any power of 16 after the conversion is performed. If the output precision is specified as single, the result is a 32-bit floating-point operand. If the output precision is specified as double, the result is a 64-bit floating-point operand.

F = P OR T (Integer): The operand P is logically ORed with the operand T. Before the operation is performed, the inputs, if 32-bit, are sign-extended to 64 bits.

F = P AND T (Integer): The operand P is logically ANDed with the operand T. Before the operation is performed, the inputs, if 32-bit, are sign-extended to 64 bits.

F = P XOR T (Integer): The operand P is logically exclusive-ORed with the operand T. Before the operation is performed, the inputs, if 32-bit, are sign-extended to 64 bits. This operation may be used to invert an operand by selecting the second operand to be the integer constant, –1, so that all bits of this second operand are 1. Exclusive-ORing an operand with –1 is equivalent to inverting each bit in the operand.

Shift P Logical Q Places (Integer): This operation cannot be performed in mixed-precision mode. The precision of the result is the same as the precision of the input operand P. A two's-complement shift length in the range –64 to +63 (double-precision) or –32 to +31 (single-precision) is extracted from the LSBs of the operand Q. The operand P is logically right-shifted by the number of places specified by the shift length. A negative shift length therefore produces a left-shift. If a right-shift is performed, 0s fill vacated bit positions to the left of the input operand. If a left-shift is performed, 0s fill vacated bit positions to the right of the input operand.

Shift P Arithmetic Q Places (Integer): This operation cannot be performed in mixed-precision mode. The precision of the result is the same as the precision of the input operand P. A two's-complement shift length in the range –64 to +63 (double-precision) or –32 to +31 (single-precision) is extracted from the LSBs of the operand Q. The operand P is arithmetically right-shifted by the number of places specified by the shift length. A negative shift length therefore produces a left-shift. If a right-shift is performed, the MSB (bit 63 or 31) is replicated to fill vacated bit positions to the left of the input operand. If a left-shift is performed, 0s fill vacated bit positions to the right of the input operand.

Funnel Shift PT Q Places (Integer): This operation cannot be performed in mixed-precision mode. The operand T is interpreted as having the same precision as the input operand P, and the precision of the result is also the same as the precision of the input operand P. A two's-complement shift length in the range –64 to +63 (double-precision) or –32 to +31 (single-precision) is extracted from the LSBs of the operand Q. A triple-width operand (96-bit or 192-bit) is formed by concatenating the input operands into the arrangement P-T-P, with the 32-bit or 64-bit result field initially aligned with the T-operand. The triple-width operand is logically right-shifted by the number of places specified by the shift length. A negative shift length therefore produces a left-shift.

Move P (Floating-Point or Integer): The 64-bit operand P is passed unchanged through the ALU. No exceptions are detected or signaled.

Primary and Alternate Floating-Point Formats

Two mode register fields, PFF and AFF, specify the primary and alternate floating-point formats used by the ALU. All floating-point operations except format conversions are performed in the format specified by PFF. For format conversion operations, either primary floating-point format PFF or alternate floating-point format AFF are used as follows:

- For conversions between floating-point and integer formats (base operation codes Convert T to integer, Convert T to floating-point, Scale T to integer by Q, Scale T to floating-point by Q), the floating-point source or destination format is specified by PFF; for the scale operations, the format of operand Q is also specified by PFF.
- When converting from the primary floating-point format to the alternate floating-point format (base operation code Convert T to alternate F. P. format), an operand in format PFF is converted to format AFF.
- When converting from the alternate floating-point format to the primary floating-point format (base operation code Convert T to primary F.P. format), an operand in format AFF is converted to format PFF.

Operation Precision

The ALU performs all operations in double-precision format. All single-precision input operands are converted to double-precision equivalents by the ALU at the start of an operation. If the operation is to report a single-precision result, the ALU converts the double-precision internal result to single-precision at the end of the operation.

Note that operation flags and exception bits pertain to the source and destination precisions. If, for example, an operation produces a single-precision overflowed result, an overflow is indicated regardless of whether that result overflows the double-precision internal format.

Operation Flags

For each operation, the ALU produces thirteen flags. Of these, a maximum of seven are relevant to any given operation. The relevant flags are placed in the flag register in the manner shown in Table 11. All flags are active High. In flow-through mode the flag register is made transparent, and the selected flags are presented directly to the output multiplexer.

The ALU flags are:

C—CARRY: Carry-out bit produced by integer addition, subtraction, or comparison.

I—INVALID OPERATION: Indicates that the input operands are unsuitable for the operation performed (e.g., $\infty \times 0$).

R—RESERVED OPERAND: Indicates that the operation result is a reserved operand. Reserved operands include signaling or quiet NaNs in IEEE format, and DEC reserved operands in DEC D or G formats.

S—SIGN: Result sign; Low for a non-negative result, High for a negative result.

U—UNDERFLOW: Indicates that the operation result underflowed the destination format.

V—OVERFLOW: Indicates that the operation result overflowed the destination format.

W—WINNER: Indicates which of two input operands is reported as the result of the MAX P, T and MIN P, T operations. A logic High indicates that operand T is reported as the result, a logic Low operand P.

X—INEXACT RESULT: Indicates that the operation result had to be rounded to fit the destination format.

Z—ZERO RESULT: Indicates that the operation produced a zero result. Note that the result is exactly zero only if the Z flag is High and the X flag is Low.

>, =, <, #—GREATER THAN, EQUAL TO, LESS THAN, UNORDERED: Used to report the result of an operation with the Compare P, T base operation code. The Greater Than flag indicates that $P > T$, the Equal To flag that $P = T$, and the Less Than flag that $P < T$. The Unordered flag indicates that one or both input operands are reserved operands and cannot be compared. Note that the Unordered flag cannot arise when comparing IBM floating-point operands or integers. Exactly one comparison flag will be active per comparison operation.

Table 11. Organization of Flags

			Flag Register						
Format	**Operation**	**CO IN$_4$–IN$_0$**	**FL6**	**FL5**	**FL4**	**FL3**	**FL2**	**FL1**	**FL0**
IEEE	F′ = P′	00000	S	Z	X	U	V	R	I
IEEE	F′ = P′ + T′	00001	S	Z	X	U	V	R	I
IEEE	F′ = P′ × Q′	00010	S	Z	X	U	V	R	I
IEEE	Compare P, T	00011	S	=	>	<	#	R	I
IEEE	Maximum P, T	00100	S	Z		W		R	I
IEEE	Minimum P, T	00101	S	Z		W		R	I
IEEE	Convert T to Integer	00110	S	Z	X		V	R	I
IEEE	Scale T to Integer	00111	S	Z	X		V	R	I
IEEE	F′ = (P′ × Q′) + T′	01000	S	Z		U	V	R	I
IEEE	Round T to Integral Value	01001	S	Z	X		V	R	I
IEEE	Reciprocal Seed of P	01010	S	Z		U	V	R	I
IEEE	Convert T to Alt F.P. Format	01011	S	Z	X	U	V	R	I
IEEE	Convert T from Alt F.P. Format	01100	S	Z	X	U	V	R	I
DEC D	F′ = P′	00000	S	Z	X		V	R	
DEC D	F = P′ + T′	00001	S	Z	X	U	V	R	
DEC D	F′ = P′ × Q′	00010	S	Z	X	U	V	R	
DEC D	Compare P, T	00011	S	=	>	<	#	R	
DEC D	Maximum P, T	00100	S	Z		W		R	
DEC D	Minimum P, T	00101	S	Z		W		R	
DEC D	Convert T to Integer	00110	S	Z	X		V	R	I
DEC D	Scale T to Integer	00111	S	Z	X		V	R	I
DEC D	F′ = (P′ × Q′) + T′	01000	S	Z		U	V	R	
DEC D	Round T to Integral Value	01001	S	Z	X		V	R	
DEC D	Reciprocal Seed of P	01010	S	Z		U	V	R	I
DEC D	Convert T to Alt F.P. Format	01011	S	Z	X	U	V	R	I
DEC D	Convert T from Alt F.P. Format	01100	S	Z	X	U	V	R	I
DEC G	F′ = P′	00000	S	Z	X	U	V	R	
DEC G	F = P′ + T′	00001	S	Z	X	U	V	R	
DEC G	F′ = P′ × Q′	00010	S	Z	X	U	V	R	
DEC G	Compare P, T	00011	S	=	>	<	#	R	
DEC G	Maximum P, T	00100	S	Z		W		R	
DEC G	Minimum P, T	00101	S	Z		W		R	
DEC G	Convert T to Integer	00110	S	Z	X		V	R	I
DEC G	Scale T to Integer	00111	S	Z	X		V	R	I
DEC G	F′ = (P′ × Q′) + T′	01000	S	Z		U	V	R	
DEC G	Round T to Integral Value	01001	S	Z	X		V	R	
DEC G	Reciprocal Seed of P	01010	S	Z		U	V	R	I
DEC G	Convert T to Alt F.P. Format	01011	S	Z	X	U	V	R	I
DEC G	Convert T from Alt F.P. Format	01100	S	Z	X	U	V	R	I
IBM	F′ = P′	00000	S	Z	X		V		
IBM	F = P′ + T′	00001	S	Z	X	U	V		
IBM	F′ = P′ × Q′	00010	S	Z	X	U	V		
IBM	Compare P, T	00011	S	=	>	<			
IBM	Maximum P, T	00100	S	Z		W			
IBM	Minimum P, T	00101	S	Z		W			
IBM	Convert T to Integer	00110	S	Z	X		V		
IBM	Scale T to Integer	00111	S	Z	X		V		
IBM	F′ = (P′ × Q′) + T′	01000	S	Z		U	V		
IBM	Round T to Integral Value	01001	S	Z	X		V		
IBM	Reciprocal Seed of P	01010	S	Z			V		I
IBM	Convert T to Alt F.P. Format	01011	S	Z	X	U	V	R	
IBM	Convert T from Alt F.P. Format	01100	S	Z	X	U	V	R	I

Table 11. Organization of Flags (continued)

Format	Operation	CO IN_4–IN_0	Flag Register FL6	FL5	FL4	FL3	FL2	FL1	FL0
Integer	F = P	00000	S	Z			V		
Integer	F = P + T	00001	S	Z			V		C
Integer	F = P × Q	00010	S	Z			V		
Integer	Compare P, T	00011	S	=	>	<	V		C
Integer	Maximum P, T	00100	S	Z		W			
Integer	Minimum P, T	00101	S	Z		W			
Integer	Convert T to Floating-Point	00110	S	Z	X				
Integer	Scale T to Floating-Point	00111	S	Z	X	U	V	R	
Integer	F = P OR T	10000	S	Z					
Integer	F = P AND T	10001	S	Z					
Integer	F = P XOR T	10010	S	Z					
Integer	Logical Shift P by Q Places	10011	S	Z					
Integer	Arithmetic Shift P by Q Places	10100	S	Z			V		
Integer	Funnel Shift P T by Q Places	10101	S	Z					
	MOVE P	11000	S						

Note: Unused flags assume the Low state.

Updating the Status Register

The status register exception bits are updated at the conclusion of each operation in flow-through mode, and at the start of each operation in pipeline mode. An exception bit is updated only if the operation reports that exception with a flag. For example, an IEEE floating-point addition operation produces an overflow flag and would therefore update the overflow exception bit; an IEEE floating-point comparison operation, on the other hand, does not produce an overflow flag and would therefore leave the overflow exception bit unchanged.

The mode register exception mask bits do not affect the updating of the status register exception bits—masked exceptions still appear in the status register. However, a masked exception will not set the exception status bit (ES).

Operation Sequencing

The Am29027 can be configured for either pipelined or flow-through (unpipelined) operation. Flow-through mode is normally selected for performing scalar operations; pipeline mode provides high throughput for vector operations. The manner in which operations are sequenced depends on the mode currently invoked.

Operation in Flow-Through Mode

Flow-through mode is invoked by setting mode register bit PL (Pipeline Mode Select) to logic Low.

Programmer's Model

A programmer's model of the Am29027 in flow-through mode is shown in Figure 9. Note that Output Register F and the flag register are made transparent in this mode.

Performing Operations

Flow-through mode operations are performed by:

- Storing instructions and/or operands in the Am29027 and starting the operation
- Loading the result

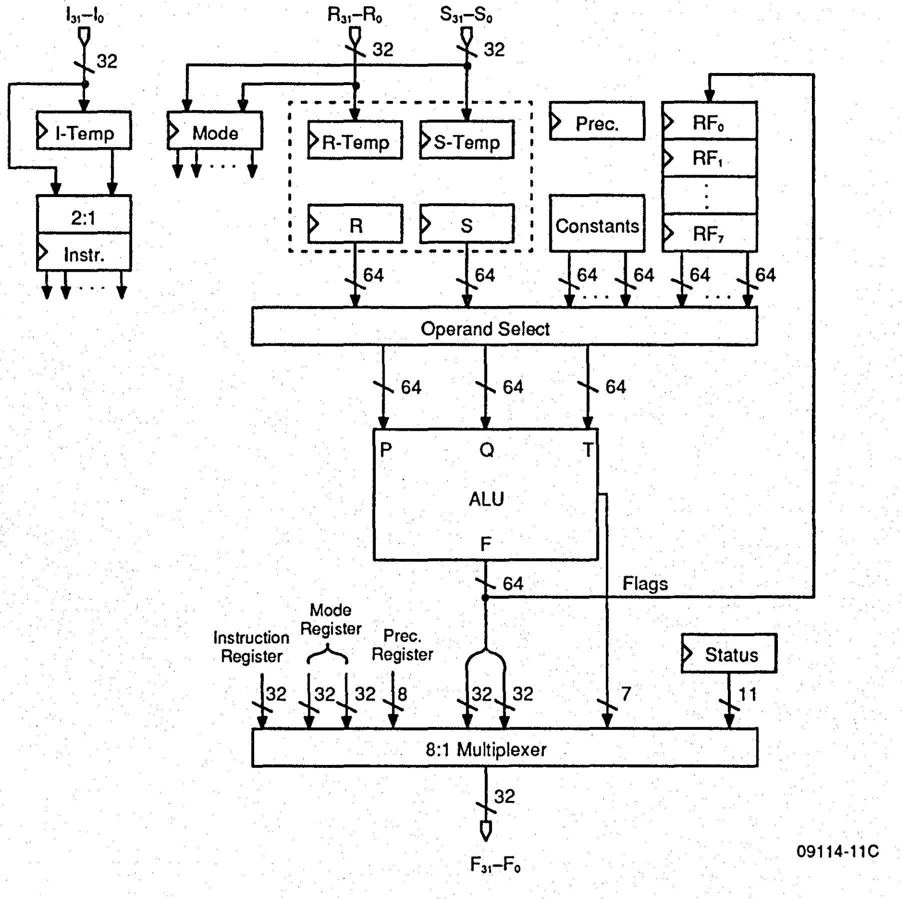


Figure 15. Programmer's Model for Flow-Through Mode

Storing instructions and operands can be done in any of three ways:

- **Writing the instruction only, and starting the operation**: This is appropriate when all necessary operands are already present in the Am29027, as is sometimes the case when using on-board constants or the results of previous operations stored in the register file.
- **Writing the operands only, and starting the operation**: This is appropriate when the desired instruction is already present in the Am29027, as is the case when performing the second of two identical operations.
- **Writing the instruction and operands, and starting the operation**: This is appropriate whenever the next operation requires both a new instruction and new operands.

Operands and instructions are written using the write operand R, write operand S, write operands R, S, and write instruction transaction requests. Operands and instructions can be written to the Am29027 in any order, with the operation start bit ($DREQT_0$ High) accompanying the last of the transaction requests.

Loading an operation result is performed using the read result MSBs, read result LSBs, and read flags transaction requests. The specific request used depends on whether the result of an operation is a flag or flags (as is the case with comparison operations) or data (as is the case with most other operations). In cases where the operation result is stored in the register file, the user may elect not to read the result but to proceed with the next operation.

Operation Timing

The Am29027 will usually start a flow-through operation during the first cycle following the receipt of a write operand R, write operand S, write operands R, S, or write instruction transaction request having signal $DREQT_0$ set High.

Operation execution begins with the transfer of the contents of the R-Temp, S-Temp, and I-Temp registers to Register R, Register S, and the instruction register, respectively; only those temporary registers written to as part of the operation specification will be transferred. The operand or instruction accompanying the transaction request that starts the operation (that is, the transaction request for which signal $DREQT_0$ is High) is written directly to the appropriate working register, that is, Register R, Register S, or the instruction register.

Once started, an operation will proceed for the number of cycles specified by mode register fields MATC, MVTC, and PLTC; MATC specifies the number of cycles for base operation code $(P \times Q) + T$, MVTC the number of cycles for base operation code MOVE P, and PLTC the number of cycles for all other base operation codes. At the end of the last operation cycle, the status register exception bits and exception status bit will be updated and, optionally, the operation result will be written to the register file and precision register.

There are two conditions for which the Am29027 will not start an operation immediately. The first condition is when an operation is already in progress. In this case the new operation is kept pending in the I-Temp, R-Temp, and S-Temp registers until the current operation is completed, at which time the new operation begins. The second condition is when a previous operation creates an unmasked exception in Halt On Error mode (mode register bit HE High). In this case the new operation is kept in the I-Temp, R-Temp, and S-Temp registers until the exception is cleared, at which time the new operation begins.

Timing for typical accelerator operations in the flow-through mode is illustrated in Appendix D.

Availability of Operation Results

In order to directly read the result of an operation, the operation specification should be followed by the appropriate read transaction request. Should the Am29000 attempt to read an operation result before the operation is completed, the Am29027 will withhold acknowledging the transaction request by holding signals $\overline{DRDY}$ and $\overline{DERR}$ inactive until the operation has been completed. All read transaction requests, including save state, will be held off in this manner.

Overlapping Operations

Due to the presence of the R-Temp, S-Temp, and I-Temp registers, it is possible to partially or completely specify a new operation while the previously specified operation is being performed. Execution of the new operation will begin immediately after the previous operation is completed. Execution begins with the transfer of the contents of the R-Temp, S-Temp, and I-Temp registers to the corresponding working registers; only those temporary registers that have been written to as part of the operation specification are transferred.

It is important to note that, once the new operation is completely specified, any attempt to read a result will be held off until the new operation is completed. This means that it is not possible to directly read the result of an operation if another operation is completely specified before the results of the first operation are read. If, for example, specification of operation $2.0 + 3.0$ is immediately followed by specification of operation 4.0×5.0, subsequent read result LSBs and read result MSBs transaction requests will return value 20.0, the result of the second operation. Similarly, a read flags transaction request will return flags for the second operation, and a read status transaction request will return status reflecting the completion of the second operation. This delayed read feature is provided to eliminate ambiguity in the correspondence between operations and results.

Should two operations be overlapped, and should the first operation have as its target a register file location, the second operation can be completely specified before the first operation is completed. If the first operation produces a result that is to be read directly by the Am29000, the second operation can be partially specified before the result of the first operation is read. A partial operation specification is one that includes all but the last operand or instruction.

Timing for typical overlapped operations in flow-through mode is illustrated in Appendix D.

Saving and Restoring State

In flow-through mode, the complete state of the Am29027 can be saved and restored with the save state transaction request. The first save state transaction request will return the contents of the instruction register; subsequent requests will return the contents of Registers I-Temp, R, S, R-Temp, S-Temp, the status register, the precision register, register file locations RF_7–RF_0, and the mode register. The user has the option of saving only part of the state by issuing only the number of save state transaction requests needed to save registers of interest. When issuing a series of save state transaction requests, data is returned in the following order:

Request	Data Returned
1	Instruction
2	I-Temp
3	R LSBs
4	R MSBs
5	S LSBs
6	S MSBs
7	R-Temp LSBs
8	R-Temp MSBs
9	S-Temp LSBs
10	S-Temp MSBs
11	Status
12	Precision
13	RF_0 LSBs
14	RF_0 MSBs
.	.
.	.
.	.
27	RF_7 LSBs
28	RF_7 MSBs
29	Mode LSBs
30	Mode MSBs

Sequencing for the save state transaction request is reinitialized when the Am29000 issues any transaction request other than save state. If, for example, the Am29000 issues a write operand R transaction request after a series of save state requests, the next save state request will return the contents of the instruction register.

It should be noted that the process of saving state alters the contents of the instruction register and Registers R and S.

Error reporting via signal $\overline{\text{DERR}}$ is suppressed for the save state transaction request.

Accelerator state is restored using transaction requests in concert with the MOVE P base operation code. Before restoring state, all status register bits should be set to logic Low using the write status transaction request to prevent the possibility of an unmasked exception bit inhibiting the restore sequence. The accelerator operand and instruction registers can then be restored, followed by restoration of the status register using the write status transaction request, with signal $DREQT_0$ asserted to indicate the end of the restore sequence. When state restoration is complete, the Am29027 will retime the operation specified by current instruction register contents.

Accelerator state is restored in the following order:

Register to be restored	Procedure for restoring
Status	Set all bits in the status register to a logic Low using the write status transaction request.
Mode	Write using write mode transaction request.
RF_0	Write "Move R to RF_0" instruction using write instruction transaction request.
	Write RF_0 value to Register R using write operand R transaction request, start operation.
	.
	.
	.
RF_7	Write "Move R to RF_7" instruction using write instruction transaction request.
	Write RF_7 value to Register R using write operand R transaction request, start operation.
Precision	Guarantee that "Move R to RF_7" operation has been completed by performing a read result MSBs transaction request.
	Write precisions using write register file precisions transaction request.
R, S, Instruction	Write R value to Register R-Temp using the write operand R transaction request.
	Write S value to Register S-Temp using the write operand S transaction request.
	Write instruction value to Register I-Temp using write instruction transaction request.
	Transfer contents of Registers R-Temp, S-Temp, and I-Temp to Register R, Register S, and the instruction register, respectively, using the advance temp registers transaction request.
R-Temp, S-Temp, I-Temp	Write R-Temp value to Register R-Temp using the write operand R transaction request.
	Write S-Temp value to Register S-Temp using the write operand S transaction request.
	Write I-Temp value to Register I-Temp using the write instruction transaction request.
Status	Write status to status register using the write status transaction request, with signal $DREQT_0$ asserted to indicate that the restore sequence is complete.

The user may elect to restore only those registers relevant to a particular application by omitting parts of the state restoration sequence. The only mandatory portions of state restoration are the initial clearing of the status register, and restoration of the status register with signal $DREQT_0$ asserted to indicate completion of the restore sequence.

Error Recovery

Six exception bits—invalid operation, reserved operand, overflow, underflow, inexact result, and zero result—are maintained in the status register; these bits are updated upon completion of an operation. Exception bits can be masked individually by programming the appropriate bits in the mode register; if the corresponding mask bit is inactive (logic Low), the exception bit is said to be unmasked and contributes to error reporting. The Am29027 provides three mechanisms with which unmasked exceptions can be handled.

Reporting Errors Upon Read

If an unmasked status register exception bit is set, the Am29027 will signal an error by asserting signal $\overline{DERR}$ when the Am29000 performs a read result LSBs, read result MSBs, read flags, or read status transaction request. Error reporting can be suppressed by issuing any of these transaction requests with signal $DREQT_0$ asserted.

Halt On Error Mode

Should the application require, the Am29027 can be configured to halt operation upon detection of an unmasked exception; this mode is invoked by setting mode register bit HE (Halt On Error) High. Once configured this way, the Am29027 will respond to an unmasked exception as follows:

- Signal $\overline{CDA}$ will become inactive upon completion of the operation producing the unmasked exception.
- Should the operation producing the unmasked exception specify that the operation result be stored on-chip, that is, in the register file, the result will not be written to its destination.
- A pending operation will not be started; the operands and/or instruction for that operation will remain in the appropriate temporary registers.
- If the Am29000 attempts to start a new operation during the last cycle of the operation that produces the unmasked exception by issuing a write operand R, write operand S, write operands R, S, or write instruction transaction request with $DREQT_0$ asserted, and if no other operation is pending, the operand or instruction will be written to the appropriate temporary register rather than to the R, S, or instruction register.
- Once $\overline{CDA}$ is deasserted, the Am29027 will respond to the write operand R, write operand S, write operands R, S, and write instruction transaction requests by asserting signal $\overline{DERR}$ one cycle after the request is issued; the contents of the target register or registers will remain unchanged.

Through these measures, the Am29027 will retain the input operands and instructions for the operation causing the exception. The input operands will be retained in the R register, S register, or register file locations, and the instructions will be retained in the instruction register. Additionally, the R-Temp, S-Temp, and I-Temp registers may contain the operands and instructions for a partially or fully specified pending operation. The Am29000 can recover these operands and instructions with the save state transaction request; this information can then be given to an error-handling routine for resolution.

The error halt condition is removed by clearing the status register exception status (ES) bit and the exception bit or bits responsible for producing the halt.

Reporting Errors via $\overline{\text{EXCP}}$

Signal $\overline{\text{EXCP}}$ will go active Low in the presence of an unmasked exception. This signal can be connected to an Am29000 trap or exception input signal, and is enabled or disabled independent of other exception handling mechanisms with mode register bit EX.

Writing to the Mode, Status, and Precision Registers

Unlike the R, S, and instruction registers, the mode, status, and precision registers are not preceded by temporary registers. Accordingly, writing to these registers may produce undesirable or unpredictable side effects if an accelerator operation is in progress at the time. To avoid such side effects, a write to any of these registers should be preceded by a read transaction request, which will guarantee that any current or pending accelerator operations will have been completed before the write transaction request is issued.

Writing to the Register File

The numerical result of any operation may be written to the register file by specifying the desired destination in instruction field RFS and setting instruction bit RF High. The result can then be used as an input operand for subsequent operations.

It is permissible for an operation result to be placed in a register file location that previously contained an input operand for that operation. In such a case, however, it is not permissible for the Am29000 to directly read the result, status, or flags for that operation, as the writing of the result modifies the operation performed by the ALU.

Determining Timer Counts

To provide optimum accelerator performance over a range of possible system clock frequencies, the timing of Am29027 operations is programmable. Three mode register fields—pipeline timer count (PLTC), timer count for the Multiply-Accumulate Operation (MATC), and timer count for the MOVE P Operation (MVTC)—must be programmed according to system clock frequency and accelerator speed.

PLTC

PLTC specifies the number of cycles allotted to operations other than those using base operation codes $(P \times Q) + T$ or MOVE P. This count can assume values between 3 and 15, inclusive, and must be given a value that satisfies the relationship:

$$[8] \leq \text{PLTC} \times [1],$$

where

[8] = Operation time, flow-through mode, all other base operation codes

and [1] = CLK period,

as described in the Switching Characteristics table.

MATC

MATC specifies the number of cycles allotted to operations that use base operation code $F' = (P' \times Q') + T'$. This count can assume values between 3 and 15, inclusive, and must be given a value that satisfies the relationship:

$$[6] \leq \text{MATC} \times [1],$$

where

[6] = Operation time, flow-through mode, $F' = (P' \times Q') + T'$

and [1] = CLK period,

as described in the Switching Characteristics table.

MVTC

MVTC specifies the number of cycles allotted to operations that use the MOVE P base operation code. This count can assume values between 3 and 15, inclusive, and must be given a value that satisfies the relationship:

$$[7] \leq \text{MVTC} \times [1],$$

where

[7] = Operation time, flow-through mode, MOVE P

and [1] = CLK period,

as described in the Switching Characteristics table.

ADVANCING $\overline{\text{DRDY}}$

Normally, an operation result produced by the Am29027 in flow-through mode is read by the Am29000 no sooner than the clock cycle following operation completion. Depending on the system clock frequency used, it may be advantageous to overlap the reading of the result with the last cycle of the operation. Consider, for example, a system with a 45-ns clock cycle and an Am29027 that performs an operation in 240 ns. The pipeline timer count PLTC will have to be set to a minimum of 6 for such a system, and the Am29000 will read a result no sooner than during the seventh clock cycle after the start of an operation.

Mode register bit DA, $\overline{\text{DRDY}}$ Advance, can be used to advance transaction status signals $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ by a full clock cycle, thus allowing the Am29000 to read data one clock cycle earlier than would otherwise be possible. For the example given above PLTC remains at 6, but the Am29000 can read data during the sixth clock cycle after the operation starts rather than the seventh, thus saving a clock cycle.

In order to advance $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$, the following system timing conditions must be met:

$$[19] \leq (\text{MATC} \times [1]) - [\times 9B] - [\text{gate}]$$
$$[20] \leq (\text{MVTC} \times [1]) - [\times 9B] - [\text{gate}]$$
$$[21] \leq (\text{PLTC} \times [1]) - [\times 9B] - [\text{gate}]$$

where [19] = Data operation-start-to-output valid delay, $F' = P' \times Q' + T'$
[20] = Data operation-start-to-output valid delay, MOVE P
[21] = Data operation-start-to-output valid delay, all other operations
and [1] = CLK period

as described in the Switching Characteristics table and

[×9] = Synchronous input setup time

as described in the Switching Characteristics table of the Am29000 Preliminary Data Sheet (order #09075).

The term [gate] represents the delay of the external gate through which the $\overline{\text{DERR}}$ signal passes.

Timing for a typical accelerator operation with $\overline{\text{DRDY}}$ advanced is illustrated in Appendix D.

Operation in Pipeline Mode

Pipeline mode is invoked by setting mode register bit PL (Pipeline Mode Select) to logic High.

Programmer's Model

A programmer's model of the Am29027 in pipeline mode is shown in Figure 10. Note that Output Register F and the flag register are non-transparent in this mode, thus permitting the overlap of the current operation(s) with the reading of the result for a previous operation.

Pipeline Delays

When placed in pipeline mode, the ALU is divided into three pipeline stages for multiply-accumulate operations, and into two stages for all other operations. The ALU configuration for pipeline mode is shown in Figure 11. Note that for multiplication-accumulation operations, multiplicand P and multiplier Q enter the first pipeline stage, while addend T enters the second pipeline stage. As a consequence, the source for operands P and Q must be specified in the corresponding multiply-accumulate instruction, while the source for operand T must be specified in the following instruction.

Pipeline Advance

The ALU pipeline is advanced whenever a new operation begins. One consequence of this advance criterion is that data does not fall through the pipe but instead is "pushed" through. If, for example, an addition is performed in pipeline mode, the pipe must be advanced twice (by starting two operations) before the result of the addition appears in Register F, the flag register, the status register, and, optionally, a register file location.

Performing Operations

Pipeline mode operations are performed by:

- Storing instructions and/or operands in the Am29027, and starting the operation
- Loading the result of a previous operation

Storing instructions and operands can be done in any of three ways:

- **Writing the instructions only, and starting the operation**: This is appropriate when all necessary operands are already present in the Am29027, as is sometimes the case when using on-board constants or the results of previous operations stored in the register file.
- **Writing the operands only, and starting the operation**: This is appropriate when the desired instructions are already present in the Am29027, as is the case when performing the second of two identical operations.
- **Writing the instructions and operands, and starting the operation**: This is appropriate whenever the next operation requires both new instructions and new operands.

Operands and instructions are written using the write operand R, write operand S, write operands R, S, and write instruction transaction requests. Operands and instructions can be written to the Am29027 in any order, with the operation start bit (DREQT$_0$ High) accompanying the last of the transaction requests.

Loading the result of a previous operation is performed using the read result MSBs, read result LSBs, and read flags transaction requests. The specific request used depends on whether the result is a flag or flags (as is the case with comparison operations) or data (as is the case with most other operations). In cases where the operation result is stored in the register file, the user may elect not to read the result, but to proceed with the next operation.

Operation Timing

The Am29027 will usually start a pipelined operation during the first cycle following the receipt of a write operand R, write operand S, write operands R, S, or write instruction transaction request having signal DREQT$_0$ set High.

Operation execution begins with the transfer of the contents of the R-Temp, S-Temp, and I-Temp registers to Register R, Register S, and the instruction register, respectively; data is transferred only from those temporary registers written to as part of the operation specification. The operand or instruction accompanying the

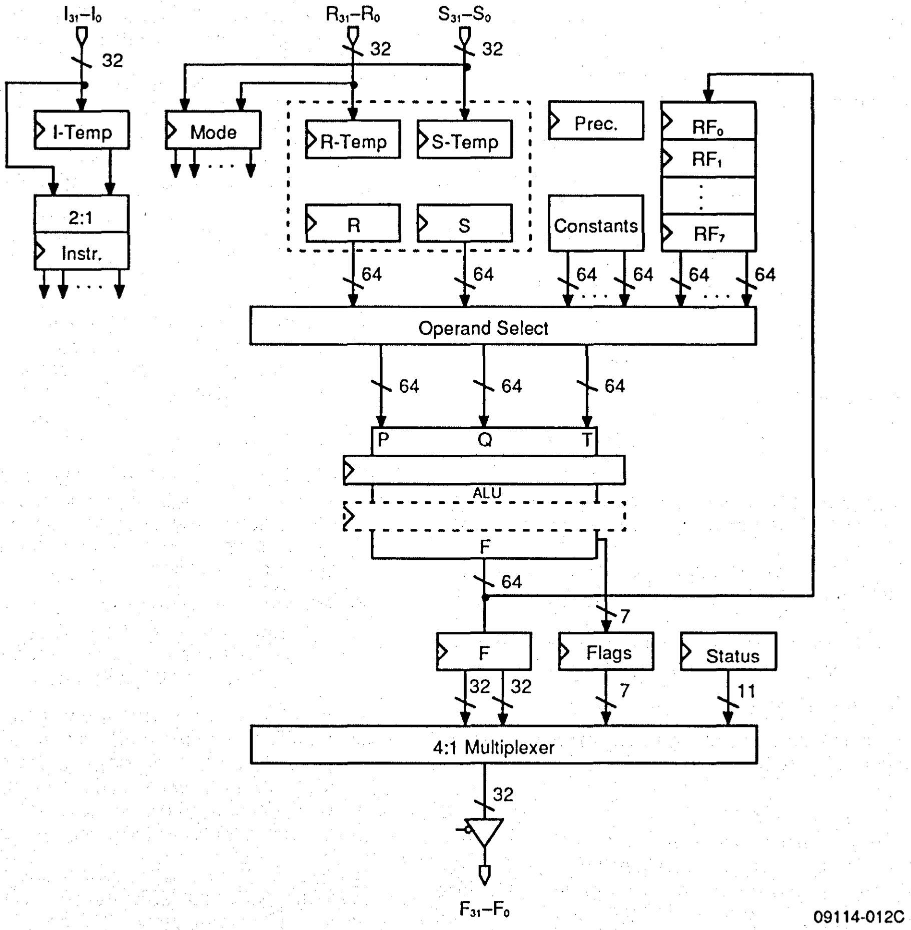


Figure 16. Programmer's Model for Pipeline Mode

transaction request that starts the operation (that is, the transaction request for which signal $DREQT_0$ is High) is written directly to the appropriate working register, that is, Register R, Register S, or the instruction register. At the start of the operation, the output of the last ALU pipeline stage is transferred to Register F, the flag register, and, optionally, to a register file location; the status register exception status and exception bits are updated. The outputs of all other ALU pipeline stages are written to their respective pipeline registers.

Once started, an operation will proceed for the number of cycles specified by mode register field PLTC, which denotes the number of cycles needed for data to traverse a single pipeline stage.

There are two conditions for which the Am29027 will not start an operation immediately. The first condition is when an operation has been started recently and has not yet had time to settle at the output of the first pipeline stage. In this case the new operation is kept pending in the I-Temp, R-Temp, and S-Temp registers until the previous operation completes the first pipeline stage. The second condition is when a previous operation creates an unmasked exception in Halt On Error mode (mode register bit HE High). In this case the new operation is kept in the I-Temp, R-Temp, and S-Temp registers until the exception is cleared, at which time the new operation will begin.

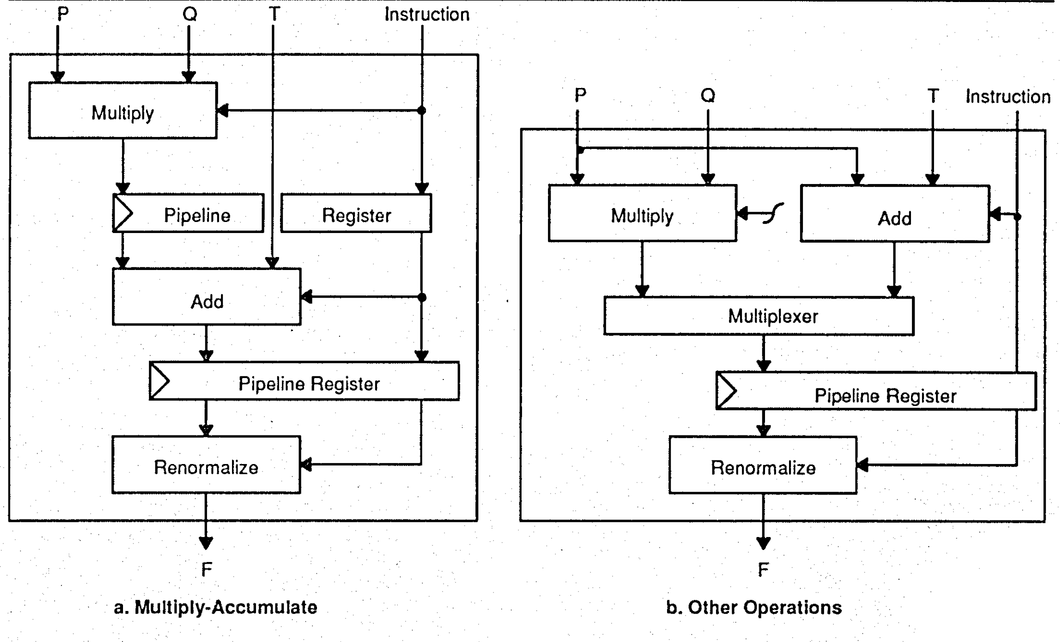


Figure 17. ALU Configuration for Pipeline Mode

Timing for typical accelerator operations in the pipeline mode is illustrated in Appendix D.

Availability of Operation Results

Because Register F, the flag register, and the status register are updated at the beginning of an operation, these registers can be read at any time after an operation begins.

Overlapping Operations

Due to the presence of the R-Temp, S-Temp, and I-Temp registers, it is possible to partially or completely specify a new operation while the previously specified operation is propagating through the first ALU pipeline stage. Execution of the new operation will begin immediately after the previous operation completes the first pipeline stage. Execution begins with the transfer of the contents of the R-Temp, S-Temp, and I-Temp registers to the corresponding working registers; only those temporary registers that have been written to as part of operation specification are transferred.

It is important to note that, once the new operation is completely specified, any attempt to read a result will be held off until the new operation begins; this means that it is not possible to read the result that is placed in the output registers when the first operation begins. If, for example, result X is placed in Register F when an operation starts and if another operation is completely specified thereafter, subsequent read result MSBs and read result LSBs transaction requests will return not X, but the result placed in the F register when the second operation begins; the read flags and read status transaction requests will behave in like manner. This delayed read feature is provided to eliminate ambiguity in the correspondence between operations and results.

Saving and Restoring State

Due to the presence of ALU pipeline registers, it is not possible to save the complete state of the Am29027 in pipeline mode. Pipeline operations may therefore be interrupted only under special circumstances, such as:

- If the interrupting routine does not use the floating-point accelerator

or

- If the current series of pipelined operations has been completed, and any operands needed for future operations have already been transferred to the Am29000

The save state transaction request is disabled in pipeline mode. It is permissible to switch to flow-through mode and use the save state transaction request, but doing so does not permit the saving of Register F, the flag register, or the ALU pipeline registers.

Error Recovery

As for flow-through mode, the Am29027 provides three mechanisms with which unmasked exceptions can be handled.

Reporting Errors Upon Read

If an unmasked status register exception bit is set, the Am29027 will signal an error by asserting signal $\overline{\text{DERR}}$ when the Am29000 performs a read result LSBs, read result MSBs, read flags, or read status transaction request. Error reporting can be suppressed by issuing any of these transaction requests with signal DREQT_0 asserted.

Halt On Error Mode

Should the application require it, the Am29027 can be configured to halt operation upon detection of an unmasked exception; this mode is invoked by setting mode register bit HE (Halt On Error) High. Once configured this way, the Am29027 will respond to an unmasked exception as follows:

- Signal $\overline{\text{CDA}}$ will become inactive when the results of the operation producing the unmasked exception are transferred from the last pipeline stage to Register F, the flag register, and the status register.
- Once $\overline{\text{CDA}}$ is deasserted, the Am29027 will respond to the write operand R, write operand S, write operands R, S, and write instruction transaction requests by asserting signal $\overline{\text{DERR}}$ one cycle after the request is issued; the contents of the target register or registers will remain unchanged.

Through these measures, the Am29027 will retain the input operands and instructions for the most recently started operation. The input operands for that operation will be retained in the R register, S register, or register file locations, and the instructions will be retained in the instruction register. Additionally, the R-Temp, S-Temp, and I-Temp registers may contain the operands and instructions for a partially or fully specified pending operation. Note that the input operands and instructions words for the operation causing the exception, as well as for operations currently in the ALU pipeline, will not be available. At the user's option, this information can be stored in a circular queue in the Am29000 register file so that full recovery from a pipelined exception is possible.

The Am29000 can read the contents of Am29027 operand and instruction registers by invoking flow-through mode and using the save state transaction request. Note that the contents of Register F, the flag register, and the ALU pipeline registers will be lost. This information can then be given to an error-handling routine for resolution.

The error halt condition is removed by clearing the status register exception status (ES) bit and the exception bit or bits responsible for producing the halt.

Reporting Errors via $\overline{\text{EXCP}}$

Same as for the flow-through mode.

Pipeline Invalidation

There are several situations for which the ALU pipeline stages may contain invalid data. The Am29027 recognizes these situations and invalidates results automatically; results marked as invalid will not update the status register, register file locations RF_7–RF_0, or the precision register. Results are invalidated for the following conditions:

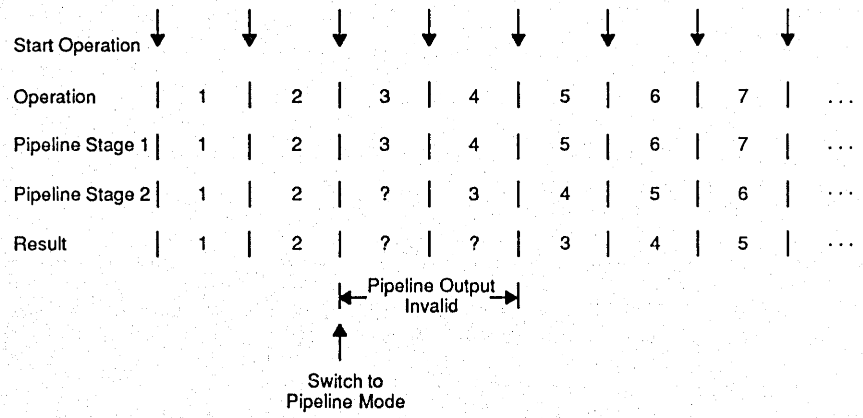
- The Am29027 is switched from flow-through mode to pipeline mode. Any data present in the ALU at the time of the switch is marked as invalid. This invalidation is illustrated in Figure 12a.
- The Am29027 performs a multiply-accumulate operation that is preceded by an operation other than multiply-accumulate. The multiply-accumulate operation result and the result that precedes it will be separated by a spurious result, due to the insertion of an additional pipeline stage for the multiply-accumulate operation. The spurious result is marked invalid. This invalidation is illustrated in Figure 12b.

The pipeline may also be invalidated manually by issuing a write status transaction request with signal DREQT_0 asserted High; this request invalidates all current pipeline contents. Pipeline invalidation does not apply to operation in flow-through mode.

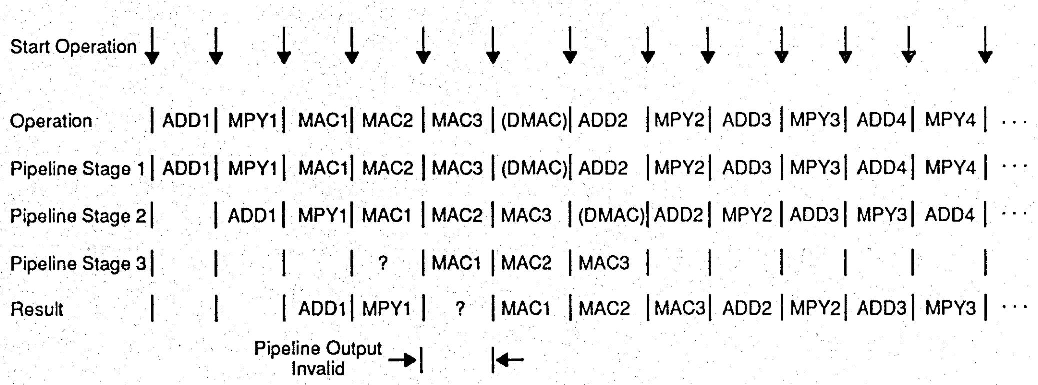
Writing to the Mode, Status, and Precision Registers

Unlike the R, S, and instruction registers, the mode, status, and precision registers are not preceded by temporary registers. Accordingly, writing to these registers may produce undesirable or unpredictable side effects if an accelerator operation is pending at the time. To avoid such side effects, a write to any of these registers should be preceded by a read transaction request, which will guarantee that any pending accelerator operation will have started before the write transaction request is issued.

The mode register outputs are not pipelined in the ALU, that is, all pipeline stages receive mode information directly from the mode register. Accordingly, writing to the mode register may produce undesirable or unpredictable side effects for operations currently in the ALU pipeline. To avoid such side effects, a write to the mode register should be performed only if the contents of the ALU pipeline are a "don't care," that is, only after the last operation result of interest has been written to Register F, the flag register, or a register file location. If, for exam-


Start Operation
Operation | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ...
Pipeline Stage 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ...
Pipeline Stage 2 | 1 | 2 | ? | 3 | 4 | 5 | 6 | ...
Result | 1 | 2 | ? | ? | 3 | 4 | 5 | ...
Pipeline Output Invalid
Switch to Pipeline Mode

a. Pipeline invalidation timing for switch from flow-through to pipeline mode. Operations shown incur two pipe-line delays in pipeline mode [all base operations except F′ = (P′ × Q′) + T′].


Start Operation
Operation | ADD1 | MPY1 | MAC1 | MAC2 | MAC3 | (DMAC) | ADD2 | MPY2 | ADD3 | MPY3 | ADD4 | MPY4 | ...
Pipeline Stage 1 | ADD1 | MPY1 | MAC1 | MAC2 | MAC3 | (DMAC) | ADD2 | MPY2 | ADD3 | MPY3 | ADD4 | MPY4 | ...
Pipeline Stage 2 | | ADD1 | MPY1 | MAC1 | MAC2 | MAC3 | (DMAC) | ADD2 | MPY2 | ADD3 | MPY3 | ADD4 | ...
Pipeline Stage 3 | | | | ? | MAC1 | MAC2 | MAC3 | | | | | |
Result | | | ADD1 | MPY1 | ? | MAC1 | MAC2 | MAC3 | ADD2 | MPY2 | ADD3 | MPY3 | ...
Pipeline Output Invalid

b. Pipeline invalidation timing for multiply-accumulate operations in pipeline mode.

Notes:		
ADDx	=	addition operation
MPYx	=	multiplication operation
MACx	=	multiply-accumulate operation
(DMAC)	=	dummy multiply-accumulate operation

09114-014C

Figure 18. Pipeline Invalidation Timing

ple, the last in a series of addition operations has just been started, the mode register should not be written until the pipeline is advanced twice, placing that operation's results in the F register, flag register, and, optionally, a register file location.

Writing to the Register File

The numerical result of any operation may be written to the register file by specifying the desired destination in instruction field RFS and setting instruction bit RF High. The result may then be used as an input operand in subsequent operations. Because all ALU operations incur one or more pipeline delays, the result of an operation will not be available for use by the very next operation.

It is permissible for an operation result to be placed in a register file location that previously contained an input operand for that operation.

Multiplication-Accumulation Operations

The pipeline structure of the Am29027 permits the evaluation of sum-of-products expressions in a canonically efficient manner by interleaving the evaluation of two sum-of-product expressions. Operation sequencing is described in Figure 13.

Determining Timer Counts

As for flow-through mode, the timing of operations in pipeline mode is programmable to accommodate variations in system timing. A single mode register field—pipeline timer count (PLTC)—specifies the timing of all pipelined operations; fields MATC and MVTC are not used.

PLTC specifies the number of cycles allotted for data to traverse a single pipeline stage. This count can assume values between 2 and 15, inclusive, and must be given a value that satisfies the relationship:

[9] ≤ PLTC × [1],

where

[9] = Operation time, pipeline mode, all operations

and [1] = CLK period,

as described in the Switching Characteristics table.

Advancing $\overline{\text{DRDY}}$

Because the Am29027 F register and flag register are non-transparent in pipeline mode, it is not possible (nor advantageous) to advance $\overline{\text{DRDY}}$. Accordingly, mode register bit M44 has no effect in pipeline mode.

Master/Slave Operation

Two Am29027 accelerators can be tied together in master/slave configuration, with the slave checking the results produced by the master. All input and output signals of the slave, with the exception of $\overline{\text{SLAVE}}$ and MSERR, are connected directly to the corresponding signals of the master. The master is selected by asserting signal $\overline{\text{SLAVE}}$ Low, the slave by asserting signal $\overline{\text{SLAVE}}$ High.

The slave accelerator, by comparing its outputs to the outputs of the master accelerator, performs a comprehensive check of master accelerator logic. In addition, if the slave accelerator is connected at the proper position on the Am29000 buses, it may detect open circuits and other faults in the electrical path between the master accelerator and the Am29000.

Note that the master accelerator also performs a comparison between its outputs and its own internally generated results, and is therefore able to detect faults in its output drivers, which it reports with its MSERR signal.

Initialization and Reset

The accelerator is in an unknown state when power is first applied and must be initialized before processing can begin. This is accomplished by asserting the $\overline{\text{RESET}}$ signal, which initializes accelerator state as follows:

- All bits in the status register are cleared
- The accelerator is placed in flow-through mode
- Signal $\overline{\text{CDA}}$ is active; signals $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ are inactive
- All internal circuitry controlling operation timing is initialized

The $\overline{\text{RESET}}$ signal does not initialize the operand and instruction registers and may corrupt existing register contents. It is the responsibility of the user to initialize these registers, if needed.

Applications

Suggestions for Power and Ground Pin Connections

The Am29027 operates in an environment of fast signal rise times and substantial switching currents. Therefore, care must be exercised during circuit board design and layout, as with any high-performance component. The following is a suggested layout, but since systems vary widely in electrical configuration, an empirical evaluation of the intended layout is recommended.

The V_{CCO} and GNDO pins carry output driver switching currents and can be electrically noisy. The V_{CC} and GND pins, which supply the logic core of the device, tend to produce less noise and the circuits they supply may be adversely affected by noise spikes on the V_{CC} plane. For this reason, it is best to provide isolation between the V_{CC} and V_{CCO} pins as well as independent decoupling for each. Isolating the GND and GNDO pins is not required.

Printed Circuit-Board Layout Suggestions

1. Use of a multilayer PC board with separate power, ground, and signal planes is highly recommended.
2. All V_{CC} and V_{CCO} pins should be connected to the V_{CC} plane. V_{CCO} pins should be isolated from V_{CC} pins by means of an isolation slot which is cut in the V_{CC} plane (see Figure 14). By physically separating the V_{CC} and V_{CCO} pins, coupled noise will be reduced.
3. All GND and GNDO pins should be connected directly to the ground plane.
4. The V_{CCO} pins should be decoupled to ground with a 0.1-µF ceramic capacitor and a 10-µF electrolytic capacitor, placed as closely to the Am29027 as is practical. V_{CC} pins should be decoupled to ground in a similar manner.

A suggested layout is shown in Figure 14.

Operation	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC	MAC*		
Register R	a11	a21	a12	a22	a13	a23	a14	a24	a31	a41	a32	a42	a33	a43	a34	a44			
Register S	b1	b1	b2	b2	b3	b3	b4	b4	b1	b1	b2	b2	b3	b3	b4	b4			
Pipeline Stage 1	a11×b1	a21×b1	a12×b2	a22×b2	a13×b3	a23×b3	a14×b4	a24×b4	a31×b1	a41×b1	a32×b2	a42×b2	a33×b3	a43×b3	a34×b4	a44×b4			
Pipeline Stage 2		a11×b1	a21×b1	a12×b2+ (c1)	a22×b2+ (c2)	a13×b3+ (c1)	a23×b3+ (c2)	a14×b4+ (c1)	a24×b4+ (c2)	a31×b1	a41×b1	a32×b2+ (c3)	a42×b2+ (c4)	a33×b3+ (c3)	a43×b3+ (c4)	a34×b4+ (c3)	a44×b4+ (c4)		
Pipeline Stage 3			a11×b1	a21×b1	a12×b2+ (c1)	a22×b2+ (c2)	a13×b3+ (c1)	a23×b3+ (c2)	a14×b4+ (c1)	a24×b4+ (c2)	a31×b1	a41×b1	a32×b2+ (c3)	a42×b2+ (c4)	a33×b3+ (c3)	a43×b3+ (c4)	a34×b4+ (c3)	a44×b4+ (c4)	
RF_0				(c1)	(c2)	(c1)	(c2)	(c1)	(c2)	c1	c2	(c3)	(c4)	(c3)	(c4)	(c3)	(c4)	c3	c4
Register F				(c1)	(c2)	(c1)	(c2)	(c1)	(c2)	c1	c2	(c3)	(c4)	(c3)	(c4)	(c3)	(c4)	c3	c4

Calculate matrix product C = A × B, where:

$$A = \begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} \quad B = \begin{bmatrix} b1 \\ b2 \\ b3 \\ b4 \end{bmatrix} \quad C = \begin{bmatrix} c1 \\ c2 \\ c3 \\ c4 \end{bmatrix}$$

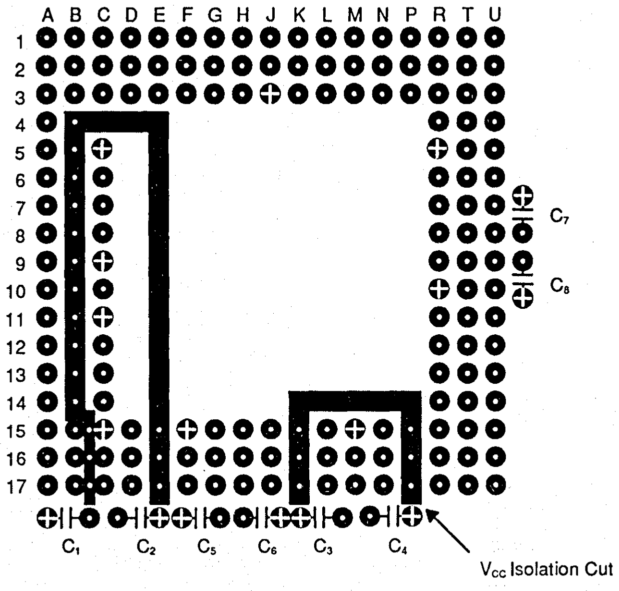
$$\begin{aligned} c1 &= a11 \times b1 + a12 \times b2 + a13 \times b3 + a14 \times b4 \\ c2 &= a21 \times b1 + a22 \times b2 + a23 \times b3 + a24 \times b4 \\ c3 &= a31 \times b1 + a32 \times b2 + a33 \times b3 + a34 \times b4 \\ c4 &= a41 \times b1 + a42 \times b2 + a43 \times b3 + a44 \times b4 \end{aligned}$$

09114-015C

Notes: 1. Register file location RF_0 is used as the accumulator.
2. Parentheses are used to indicate partial sums of products.

*Additional MAC operation needed to terminate sequence.

Figure 13. Canonically Efficient Sum-of-Products Evaluation in Pipeline Mode



● = Through Hole

⊕ = V_{CC} Plane Connection

CD011711

$C_1 = C_3 = C_5 = C_7 = 0.1\ \mu F$ (ceramic or monolithic capacitor)

$C_2 = C_4 = C_6 = C_8 = 10\ \mu F$ (electrolytic or tantalum capacitor)

Figure 20. Suggested Printed Circuit-Board Layout (power and ground connections)

ABSOLUTE MAXIMUM RATINGS

Storage Temperature	–65 to +150°C
(Ambient) Temperature Under Bias	–55 to +125°C
Supply Voltage to Ground Potential Continuous	–0.3 V to +7.0 V
DC Voltage Applied to Outputs for High Output State	–0.3 V to +V_{CC} +0.3 V
DC Input Voltage	–0.3 V to +V_{CC} +0.3 V
DC Output Current, Into Low Outputs	30 mA
DC Input Current	–10 mA to +10 mA

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices

Case Temperature (T_C)	0 to +85°C
Supply Voltage (V_{CC})	+4.75 V to +5.25 V

Military* (M) Devices

Case Temperature (T_C)	–55 to +125°C
Supply Voltage (V_{CC})	+4.5 V to +5.5 V

Operating ranges define those limits between which the functionality of the device is guaranteed.

*Military Product 100% tested at T_C=+25°C, +125°C, and –55°C.

DC CHARACTERISTICS over COMMERCIAL operating range unless otherwise specified (for APL Products, Group A, Subgroups 1, 2, and 3 are tested unless otherwise noted)

Parameter Symbol	Parameter Description	Test Conditions (Note 1)			Min.	Max.	Unit
V_{OH}	Output High Voltage	V_{CC} = Min. V_{IN} = V_{IH} or V_{IL}	I_{OH} = −4.0 mA		2.4		V
V_{OL}	Output Low Voltage	V_{CC} = Min. V_{IN} = V_{IH} or V_{IL}	I_{OL} = 4.0 mA			0.45	V
V_{IH}	Guaranteed Input Logical High Voltage (Note 2)				2.0		V
V_{IL}	Guaranteed Input Logical Low Voltage (Note 2)					0.8	V
V_{IH}(F)	Guaranteed Input Logical High Voltage (Notes 2, 6)	F Bus, Slave Operation Only			V_{CC} −0.5		V
V_{IL}(F)	Guaranteed Input Logical Low Voltage (Notes 2, 6)	F Bus, Slave Operation Only				0.5	V
I_{IL}	Input Leakage Current	$0.450 \le V_{IN} \le V_{CC} - 0.450$, V_{CC} = Max.				±10	µA
I_{LO}	Output Leakage Current	$0.450 \le V_{OUT} \le V_{CC} - 0.450$, V_{CC} = Max.				±10	µA
I_{CC} Static	Static Power Supply Current	V_{CC} = Max. I_O = 0 µA	Com'l T_C = 0 to +85°C	(Note 3) CMOS V_{IN} = V_{CC} or GND		240	mA
				(Note 3) TTL V_{IN} = 0.5 V or 2.4 V		275	
			MIL T_C = −55 to +125°C	(Note 3) CMOS V_{IN} = V_{CC} or GND			
				(Note 3) TTL V_{IN} = 0.5 V or 2.4 V			
I_{CCOP}	Operating Power Supply Current	V_{CC} = Max. Outputs floating				9.0	mA/MHz

Notes: 1. V_{CC} conditions shown as Min. or Max. refer to ±5% V_{CC} (commercial) and ±10% V_{CC} (military).
2. These input levels provide zero noise immunity and should only be statically tested in a noise-free environment (not functionally tested).
3. Use CMOS I_{CC} when the device is driven by CMOS circuits and TTL I_{CC} when the device is driven by TTL circuits.
4. I_{CC} (Total) = I_{CC} (Static) + I_{CCOP} x f, where f is in MHz. This is tested on a sample basis only.
5. Tested on a sample basis only.
6. These levels guaranteed compatible with F bus output levels.

CAPACITANCE

Parameter Symbol	Parameter Description	Test Conditions	Min.	Max.	Unit
C_{IN}	Input Capacitance	fc = 1 MHz (Note 5)		12	pF
C_{OUT}	Output Capacitance			20	pF
$C_{I/O}$	I/O Pin Capacitance			20	pF

SWITCHING CHARACTERISTICS over COMMERCIAL operating range

No.	Parameter Description	Test Conditions	25 MHz		20 MHz		16 MHz		Unit
			Min.	Max.	Min.	Max.	Min.	Max.	
1	CLK Period	(Note 1)	40	DC	50	DC	60	DC	ns
2	CLK Low Time		18		20		22		ns
3	CLK High Time		18		20		22		ns
4	CLK Rise Time	(Note 2)		5		5		5	ns
5	CLK Fall Time	(Note 2)		5		5		5	ns
	Operation Time, Low-Latency								
6	Mode, F′ = (P′ × Q′) + T′			280		300		360	ns
7	MOVE P			120		150		180	ns
8	(All Other Base Operation Codes)			200		250		300	ns
	Operation Time, Pipeline Mode								
9	All Operations			150		150		180	ns
10	Transaction Request Setup Time	(Note 3)	20		24		26		ns
11	Transaction Request Hold Time	(Note 3)	0		0		0		ns
12	$\overline{\text{BINV}}$ Setup Time		11		13		15		ns
13	$\overline{\text{BINV}}$ Hold Time		2		2		2		ns
14	Data Setup Time		18		22		24		ns
15	Data Hold Time	(Note 4)	2		2		2		ns
16	Instruction Setup Time		18		22		24		ns
17	Instruction Hold Time	(Note 5)	2		2		2		ns
18	$\overline{\text{CDA}}$ CLK-to-Output-Valid Delay			20		24		26	ns
19	F_{31}–F_0 CLK-to-Output-Valid Delay			30		35		37	ns
20	F_{31}–F_0 Three-State CLK-to-Output-Inactive Delay	(Note 6)		22		25		27	ns
	Data Operation-Start-to-Output-Valid Delay								
21	F′ = (P′ × Q′) + T′			270		285		340	ns
22	MOVE P			110		135		160	ns
23	(All Other Base Operation Codes)			190		235		280	ns
24	$\overline{\text{DRDY}}$ CLK-to-Output-Valid Delay			18		21		23	ns
25	$\overline{\text{DERR}}$ CLK-to-Output-Valid Delay			18		21		23	ns
26	$\overline{\text{EXCP}}$ CLK-to-Output-Valid Delay			18		21		23	ns
27	MSERR CLK-to-Output-Valid Delay			20		25		30	ns

Notes: 1. CLK switching characteristics are made relative to 1.5 V.

2. CLK rise time/fall time measured between 0.8 V and (V_{CC} –1.0 V). Tested on a sample basis only.

3. Transaction request signals include R/$\overline{\text{W}}$, $\overline{\text{DREQ}}$, DREQT_1–DREQT_1, and OPT_2–OPT_0.

4. Data signals include R_{31}–R_0 and S_{31}–S_0.

5. Instruction signals include I_{31}–I_0.

6. Three-State Output Inactive Test Load. Three-State CLK-to-Output-Inactive Delay is measured as the time to a ±500 mV change from prior output level.

Conditions: A. All inputs/outputs are TTL-compatible for V_{IH}, V_{IL}, and V_{OL} unless otherwise noted.

B. All outputs are driving 80 pF unless otherwise noted.

C. All setup, hold, and delay times are measured relative to CLK at 1.5 V unless otherwise noted.

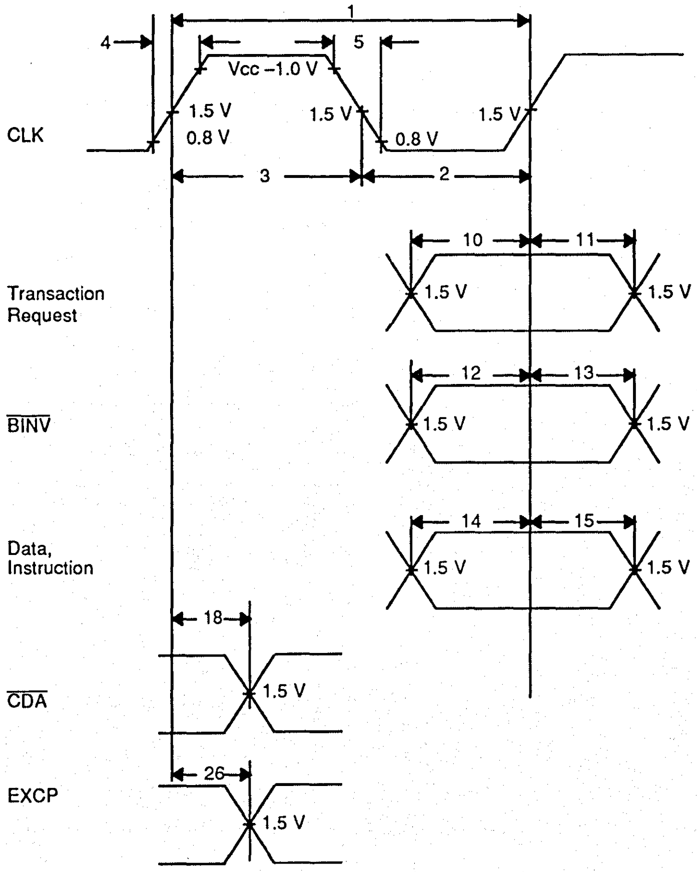
SWITCHING CHARACTERISTICS over MILITARY operating range

No.	Parameter Description	Test Conditions	20 MHz		16 MHz		Unit
			Min.	Max.	Min.	Max.	
1	CLK Period	(Note 1)	50	DC	60	DC	ns
2	CLK Low Time		20		22		ns
3	CLK High Time		20		22		ns
4	CLK Rise Time	(Note 2)		5		5	ns
5	CLK Fall Time	(Note 2)		5		5	ns
	Operation Time, Low-Latency						
6	Mode, $F' = (P' \times Q') + T'$			300		360	ns
7	MOVE P			150		180	ns
8	(All Other Base Operation Codes)			250		300	ns
	Operation Time, Pipeline Mode						
9	All Operations			150		180	ns
10	Transaction Request Setup Time	(Note 3)	24		26		ns
11	Transaction Request Hold Time	(Note 3)	0		0		ns
12	$\overline{BINV}$ Setup Time		14		16		ns
13	$\overline{BINV}$ Hold Time		2		2		ns
14	Data Setup Time		22		24		ns
15	Data Hold Time	(Note 4)	2		2		ns
16	Instruction Setup Time		22		24		ns
17	Instruction Hold Time	(Note 5)	2		2		ns
18	$\overline{CDA}$ CLK-to-Output-Valid Delay			24		26	ns
19	F_{31}–F_0 CLK-to-Output-Valid Delay			35		40	ns
20	F_{31}–F_0 Three-State CLK-to-Output-Inactive Delay	(Note 6)		26		30	ns
	Data Operation-Start-to-Output-Valid Delay						
21	$F' = (P' \times Q') + T'$			285		340	ns
22	MOVE P			135		160	ns
23	(All Other Base Operation Codes)			235		280	ns
24	$\overline{DRDY}$ CLK-to-Output-Valid Delay			21		23	ns
25	$\overline{DERR}$ CLK-to-Output-Valid Delay			21		23	ns
26	$\overline{EXCP}$ CLK-to-Output-Valid Delay			21		23	ns
27	MSERR CLK-to-Output-Valid Delay			25		30	ns

Notes:
1. CLK switching characteristics are made relative to 1.5 V.
2. CLK rise time/fall time measured between 0.8 V and (V_{CC}–1.0 V). Tested on a sample basis only.
3. Transaction request signals include R/$\overline{W}$, $\overline{DREQ}$, $DREQT_1$–$DREQT_0$, and OPT_2–OPT_0.
4. Data signals include R_{31}–R_0 and S_{31}–S_0.
5. Instruction signals include I_{31}–I_0.
6. Three-State Output Inactive Test Load. Three-State CLK-to-Output-Inactive Delay is measured as the time to a ±500 mV change from prior output level.

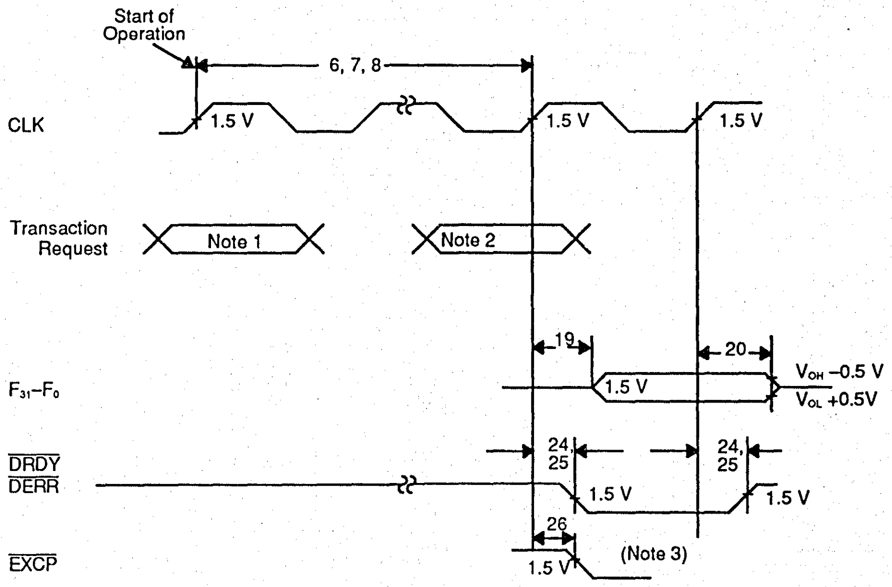
Conditions:
A. All inputs/outputs are TTL-compatible for V_{IH}, V_{IL}, and V_{OL} unless otherwise noted.
B. All outputs are driving 80 pF unless otherwise noted.
C. All setup, hold, and delay times are measured relative to CLK at 1.5 V unless otherwise noted.

SWITCHING WAVEFORMS



Input Signal Timing; $\overline{\text{CDA}}$, $\overline{\text{EXCP}}$ Timing

SWITCHING WAVEFORMS (continued)



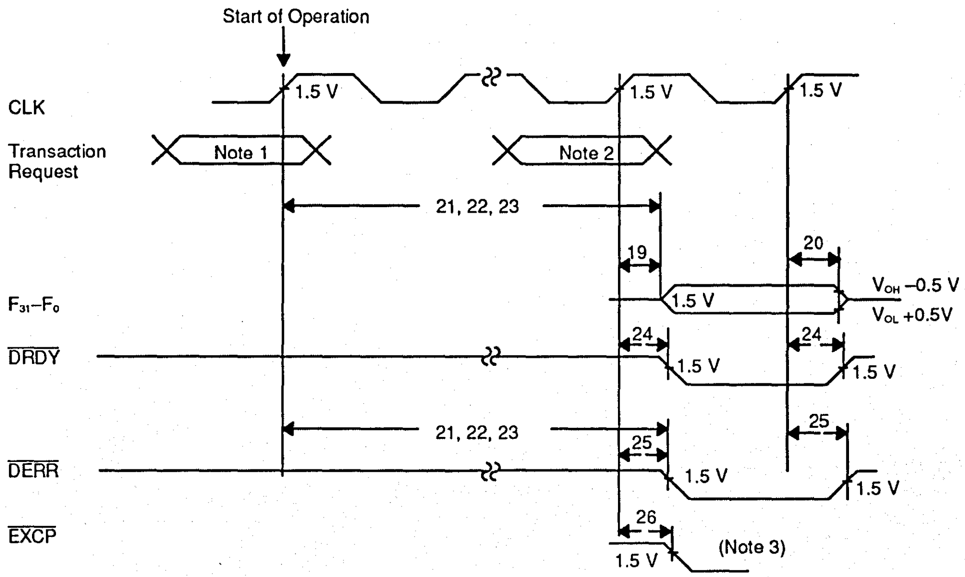
Operation Timing for Flow-Through Mode, $\overline{DRDY}$, $\overline{DERR}$ Not Advanced (Mode Register Bit AD = 0)

Notes: 1. Transaction request Write Operand R; Write Operand S; Write Operands R, S; or Write Instruction with Signal DREQT$_0$ asserted.

2. Transaction Request Read Result MSBs, Read Result LSBs, Read Flags, Read Status, or Save State. If request Read Result LSBs is issued, the Am29027 produces two data outputs in two consecutive cycles, with $\overline{DRDY}$ or $\overline{DERR}$ active for both cycles.

3. Signal $\overline{EXCP}$ is asserted in the presence of unmasked exception.

SWITCHING WAVEFORMS (continued)



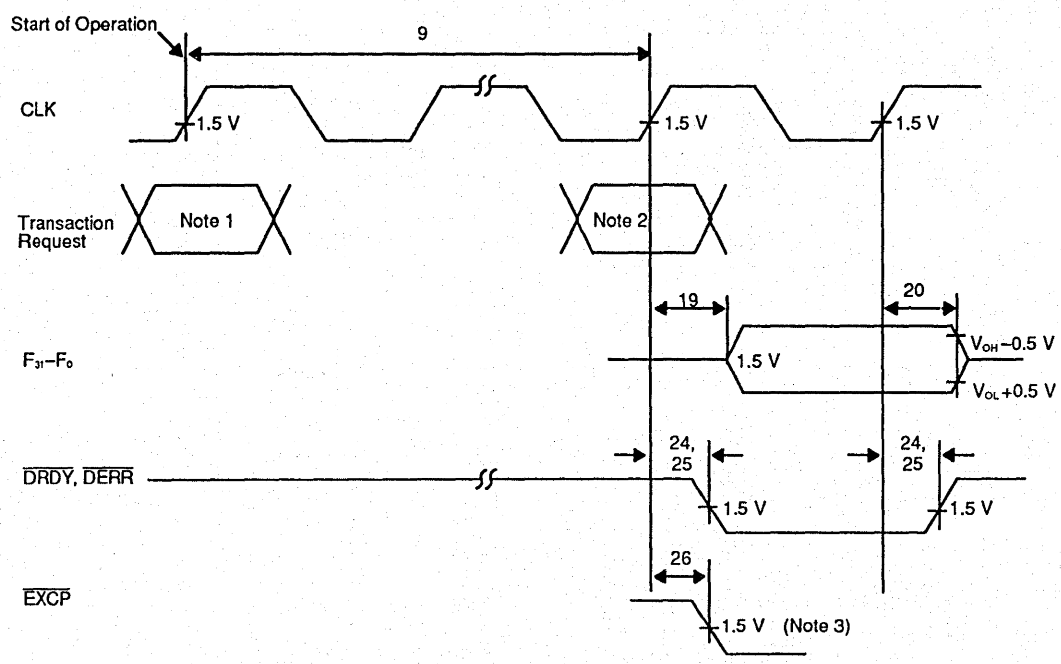
Operation Timing for Flow-Through Mode, $\overline{DRDY}$, $\overline{DERR}$ Advanced (Mode Register Bit AD = 1)

Notes: 1. Transaction request Write Operand R; Write Operand S; Write Operands R, S; or Write Instruction with Signal $DREQT_0$ asserted.

2. Transaction Request Read Result MSBs, Read Result LSBs, Read Flags, Read Status, or Save State. If request Read Result LSBs is issued, the Am29027 produces two data outputs in consecutive cycles, with $\overline{DRDY}$ or $\overline{DERR}$ active for both cycles.

3. Signal $\overline{EXCP}$ is asserted in the presence of an unmasked exception.

SWITCHING WAVEFORMS (continued)

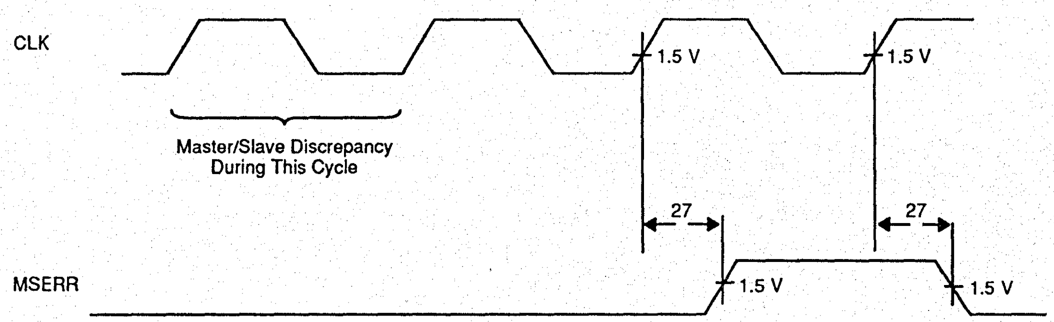


Operation Timing for Pipeline Mode

Notes: 1. Transaction request Write Operand R; Write Operand S; Write Operands R, S; or Write Instruction with signal $DREQT_0$ asserted.

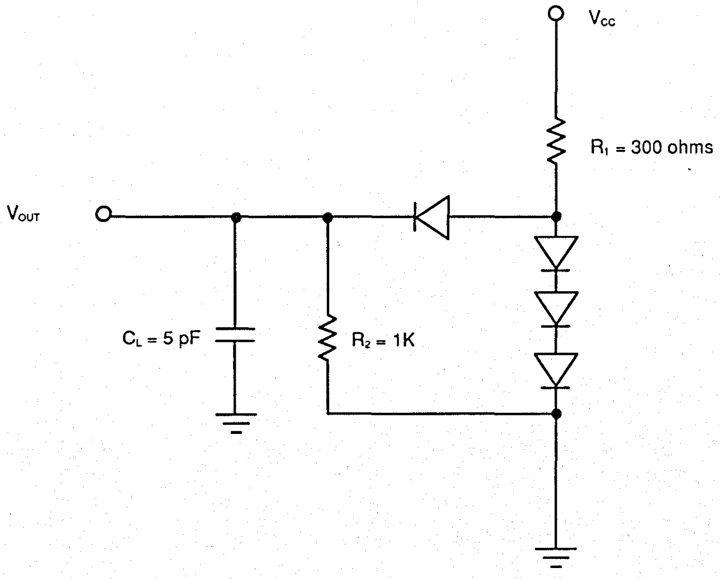
2. Transaction Request Read Result MSBs, Read Result LSBs, Read Flags, Read Status, or Save State. If request Read Result LSBs is issued, the Am29027 produces two data outputs in consecutive cycles, with $\overline{DRDY}$ or $\overline{DERR}$ for both cycles.

3. Signal $\overline{EXCP}$ is asserted in the presence of an unmasked exception.

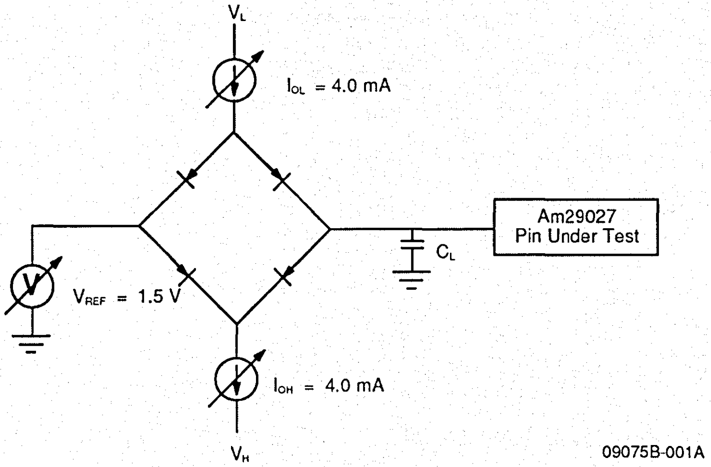


Master/Slave Timing

SWITCHING TEST CIRCUIT



Three-State Output Inactive Test



C_L is guaranteed to 80 pF.

TEST PHILOSOPHY AND METHODS

The following nine points describe AMD's philosophy for high-volume, high-speed automatic testing.

1. Ensure that the part is adequately decoupled at the test head. Large changes in V_{CC} current as the device switches may cause erroneous function failures due to V_{CC} changes.

2. Do not leave inputs floating during any tests, as they may start to oscillate at high frequency.

3. Do not attempt to perform threshold tests at high speed. Following an output transition, ground current may change by as much as 400 mA in 5–8 ns. Inductance in the ground cable may allow the ground pin at the device to rise by hundreds of millivolts momentarily.

4. Use extreme care in defining point input levels for AC tests. Many inputs may be changed at once, so there will be significant noise at the device pins and they may not actually reach V_{IL} or V_{IH} until the noise has settled. AMD recommends using $V_{IL} \leq 0$ V and $V_{IH} \geq 3.0$ V for AC tests.

5. To simplify failure analysis, programs should be designed to perform DC, Function, and AC tests as three distinct groups of tests.

6. Capacitive Loading for AC Testing.

   Automatic testers and their associated hardware have stray capacitance that varies from one type of tester to another, but is generally around 50 pF. This, of course, makes it impossible to make direct measurements of parameters that call for smaller capacitive load than the associated stray capacitance. Typical examples of this are the so-called float delays, which measure the propagation delays into the high-impedance state and are usually specified at a load capacitance of 5.0 pF. In these cases, the test is performed at the higher load capacitance (typically 50 pF), and engineering correlations based on data taken with a bench setup are used to predict the result at the lower capacitance.

   Similarly, a product may be specified at more than one capacitive load. Since the typical automatic tester is not capable of switching loads in mid-test, it is impossible to make measurements at both capacitances even though they may both be greater than the stray capacitance. In these cases, a measurement is made at one of the two capacitances. The result at the other capacitance is predicted from engineering correlations based on data taken with a bench setup and the knowledge that certain DC measurements (I_{OH}, I_{OL}, for example) have already been taken and are within spec. In some cases, special DC tests are performed in order to facilitate this correlation.

7. Threshold Testing

   The noise associated with automatic testing (due to the long, inductive cables) and the high gain of the tested device when in the vicinity of the actual device threshold, frequently give rise to oscillations when testing high-speed circuits. These oscillations are not indicative of a reject device, but instead of an overtaxed test system. To minimize this problem, thresholds are tested at least once for each input pin. Thereafter, hard high and low levels are used for other tests. Generally this means that function and AC testing are performed at hard input levels rather than at V_{IL} Max. and V_{IH} Min.

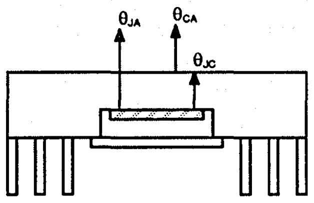
8. AC Testing

   Occasionally, parameters are specified that cannot be measured directly on automatic testers because of tester limitations. Data input hold times often fall into this category. In these cases, the parameter in question is guaranteed by correlating these tests with other AC tests that have been performed. These correlations are arrived at by the cognizant engineer by using precise bench measurements in conjunction with the knowledge that certain DC parameters have already been measured and are within spec.

   In some cases, certain AC tests are redundant, since they can be shown to be predicted by some other tests that have already been performed. In these cases, the redundant tests are not performed.

Am29027 Thermal Characteristics

Pin-Grid-Array Package



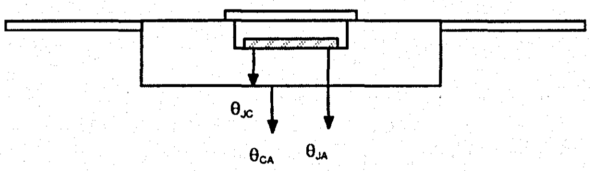
$\theta_{JA} = \theta_{JC} + \theta_{CA}$

Thermal Resistance – °C/Watt

Parameter	Airflow—ft./min. (m/sec)					
	0 (0)	150 (0.76)	300 (1.53)	480 (2.45)	700 (3.58)	900 (4.61)
θ_{JC} Junction-to-Case	4	4	4	4	4	4
θ_{CA} Case-to-Ambient (no Heatsink)	16	14	12	11	9	8
θ_{CA} Case-to-Ambient (with omnidirectional 4-Fin Heatsink, Thermalloy 0417261)	10	6	3	2	2	2
θ_{CA} Case-to-Ambient (with unidirectional Pin Fin Heatsink, Wakefield 840-20)	10	6	3	2	2	2

Am29027 Thermal Characteristics

Ceramic Quad-Flat-Pack Package



$\theta_{JA} = \theta_{JC} + \theta_{CA}$

Thermal Resistance – °C/Watt

Parameter	Airflow—ft./min. (m/sec)					
	0 (0)	150 (0.76)	300 (1.53)	480 (2.45)	700 (3.58)	900 (4.61)
θ_{JC} Junction-to-Case						
θ_{CA} Case-to-Ambient (no Heatsink)						

Note: This is for reference only.

APPENDIX A—DATA FORMATS

The following data formats are supported: 32-bit integer, 64-bit integer, IEEE single-precision, IEEE double-precision, DEC F, DEC D, DEC G, IBM single-precision, and IBM double-precision.

The primary and alternate floating-point formats are selected by mode register fields PFF and AFF. The user may select between floating-point operations and integer operations by means of instruction bit IN_5.

The nine supported formats are described below:

Integer Formats

32-Bit Integer

The 32-bit integer word is arranged as follows:

Bit	31	30	29	28	27	26	25	. . . . .	7	6	5	4	3	2	1	0
	-2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}	. . . . .	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

TB001030

The 32-bit word is interpreted as a two's-complement integer. For integer multiplications, the user has the option of interpreting integers as unsigned. An unsigned single-precision integer has a format similar to that of the two's-complement integer, but with an MSB weight of 2^{31}.

64-Bit Integer

The 64-bit integer word is arranged as follows:

Bit	63	62	61	60	59	58	57	. . . . .	7	6	5	4	3	2	1	0
	-2^{63}	2^{62}	2^{61}	2^{60}	2^{59}	2^{58}	2^{57}	. . . . .	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

TB001040

The 64-bit word is interpreted as a two's-complement integer. For integer multiplications, the user has the option of interpreting integers as unsigned. An unsigned double-precision integer has a format similar to that of the two's-complement integer, but with an MSB weight of 2^{63}.

IEEE Formats

IEEE Single Precision

The IEEE single-precision word is 32 bits wide and is arranged in the format shown below:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	. . .	3	2	1	0
s	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	. . .	2^{-20}	2^{-21}	2^{-22}	2^{-23}
sign	biased exponent (e)								fraction (f)									

TB001050

The floating-point word is divided into three fields: a single-bit sign, an 8-bit biased exponent, and a 23-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers. 0 may have either sign.

The biased exponent is an 8-bit unsigned integer representing a multiplicative factor of some power of 2. The bias value is 127. If, for example, the multiplicative value for a floating-point number is to be 2^a, the value of the biased exponent is a + 127, where "a" is the true exponent.

The fraction is a 23-bit unsigned fractional field containing the 23 least significant bits of the floating-point number's 24-bit mantissa. The weight of the fraction's most significant bit is 2^{-1}. The weight of the least significant bit is 2^{-23}.

An IEEE floating-point number is evaluated or interpreted as follows:

If $e = 255$ and $f \neq 0$	value = NaN	Not a Number
If $e = 255$ and $f = 0$	value = $(-1)^s\infty$	Infinity
If $0 < e < 255$	value = $(-1)^s2^{e-127}$ (1.f)	Normalized number
If $e = 0$ and $f \neq 0$	value = $(-1)^s2^{-126}$ (0.f)	Denormalized number
If $e = 0$ and $f = 0$	value = $(-1)^s0$	Zero

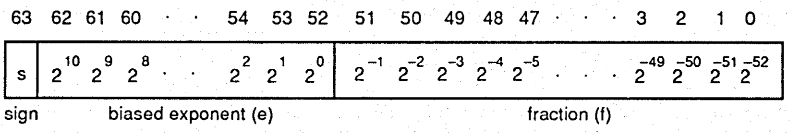
Infinity: Infinity can have either a positive or negative sign. The interpretation of infinities is determined by mode register bit AP.

NaN: A NaN is interpreted as a signal or symbol. NaNs are used to indicate invalid operations and as a means of passing process status through a series of calculations. They arise in two ways: either generated by the Am29027 to indicate an invalid operation, or provided by the user as an input. A signaling NaN has the MSB of its fraction set to 0 and at least one of the remaining fraction bits set to 1. A quiet NaN has the MSB of its fraction set to 1.

The IEEE format is fully described in ANSI/IEEE Standard 754-1985.

IEEE Double Precision

The IEEE double-precision word is 64 bits wide and is arranged in the format shown below:



TB001060

The floating-point word is divided into three fields: a single-bit sign, an 11-bit biased exponent, and a 52-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; 0 may have either sign.

The biased exponent is an 11-bit unsigned integer representing a multiplicative factor of some power of 2. The bias value is 1023. If, for example, the multiplicative value for a floating-point number is to be 2^a, the value of the biased exponent is $a + 1023$, where "a" is the true exponent.

The fraction is a 52-bit unsigned fractional field containing the 52 least significant bits of the floating-point number's 53-bit mantissa. The weight of the fraction's most significant bit is 2^{-1}. The weight of the least significant bit is 2^{-52}.

An IEEE floating-point number is evaluated or interpreted as follows:

If $e = 2047$ and $f \neq 0$	value = Reserved operand	Not a Number
If $e = 2047$ and $f = 0$	value = $(-1)^s\infty$	Infinity
If $0 < e < 2047$	value = $(-1)^s2^{e-1023}$ (1.f)	Normalized number
If $e = 0$ and $f \neq 0$	value = $(-1)^s2^{-1022}$ (0.f)	Denormalized number
If $e = 0$ and $f = 0$	value = $(-1)^s0$	Zero

Infinity: Infinity can have either a positive or negative sign. The interpretation of infinities is determined by mode register bit AP.

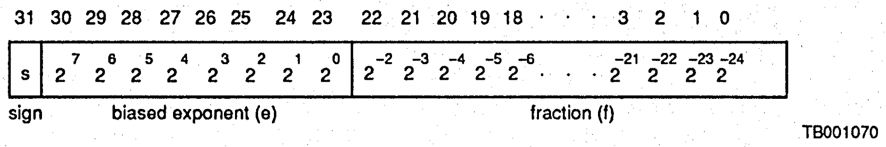
NaN: A NaN is interpreted as a signal or symbol. NaNs are used to indicate invalid operations and as a means of passing process status through a series of calculations. They arise in two ways: either generated by the Am29027 to indicate an invalid operation, or provided by the user as an input. A signaling NaN has the MSB of its fraction set to 0 and at least one of the remaining fraction bits set to 1. A quiet NaN has the MSB of its fraction set to 1.

The IEEE format is fully described in ANSI/IEEE Standard 754-1985.

DEC Formats

DEC F

The DEC F word is 32 bits wide and is arranged in the format shown below:



The floating-point word is divided into three fields: a single-bit sign, an 8-bit biased exponent, and a 23-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; 0 has a positive sign.

The biased exponent is an 8-bit unsigned integer representing a multiplicative factor of some power of 2. The bias value is 128. If, for example, the multiplicative value for a floating-point number is to be 2^a, the value of the biased exponent is a + 128, where "a" is the true exponent.

The fraction is a 23-bit unsigned fractional field containing the 23 least significant bits of the floating-point number's 24-bit mantissa. The weight of the fraction's most significant bit is 2^{-2}. The weight of the least significant bit is 2^{-24}.

A DEC F floating-point number is evaluated or interpreted as follows:

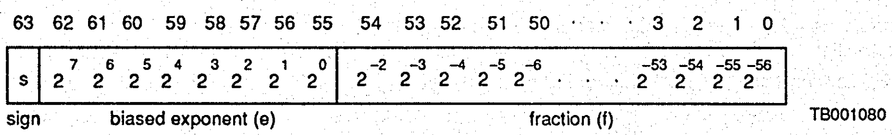
If $e \neq 0$ .... value $\neq (-1)^S 2^{e-128}$ (0.1f)
If s = 0 and e = 0 ...... value = 0
If s = 1 and e = 0 .... value = DEC-Reserved Operand

DEC-Reserved Operand: A DEC-Reserved Operand is interpreted as a signal or symbol. DEC-Reserved Operands are used to indicate invalid operations and operations whose results have overflowed the destination format. They may also be used to pass symbolic information from one calculation to another.

The DEC formats are fully described in the VAX™ Architecture Manual.

DEC D

The DEC D word is 64 bits wide and is arranged in the format shown below:



The floating-point word is divided into three fields: a single-bit sign, an 8-bit biased exponent, and a 55-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; 0 has a positive sign.

The biased exponent is an 8-bit unsigned integer representing a multiplicative factor of some power of 2. The bias value is 128. If, for example, the multiplicative value for a floating-point number is to be 2^a, the value of the biased exponent is a + 128, where "a" is the true exponent.

The fraction is a 55-bit unsigned fractional field containing the 55 least significant bits of the floating-point number's 56-bit mantissa. The weight of the fraction's most significant bit is 2^{-2}. The weight of the least significant bit is 2^{-56}.

A DEC D floating-point number is evaluated or interpreted as follows:

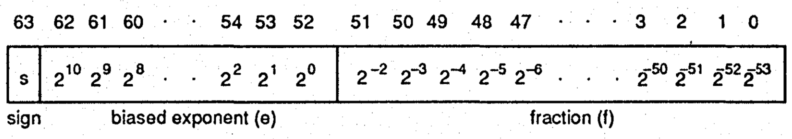
If $e \neq 0$ .... value $= (-1)^S 2^{e-128}$ (0.1f)
If s = 0 and e = 0 .... value = 0
If s = 1 and e = 0 .... value = DEC-Reserved Operand

DEC-Reserved Operand: A DEC-Reserved Operand is interpreted as a signal or symbol. DEC-Reserved Operands are used to indicate invalid operations and operations whose results have overflowed the destination format. They may also be used to pass symbolic information from one calculation to another.

The DEC formats are fully described in the VAX Architecture Manual.

DEC G

The DEC G word is 64 bits wide and is arranged in the format shown below:



TB001090

The floating-point word is divided into three fields: a single-bit sign, an 11-bit biased exponent, and a 52-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; 0 has a positive sign.

The biased exponent is an 11-bit unsigned integer representing a multiplicative factor of some power of 2. The bias value is 1024. If, for example, the multiplicative value for a floating-point number is to be 2^a, the value of the biased exponent is a + 1024, where "a" is the true exponent.

The fraction is a 52-bit unsigned fractional field containing the 52 least significant bits of the floating-point number's 53-bit mantissa. The weight of the fraction's most significant bit is 2^{-2}. The weight of the least significant bit is 2^{-53}.

A DEC G floating-point number is evaluated or interpreted as follows:

If $e \neq 0$	value = $(-1)^S 2^{e-1024}$ (0.1f)
If s = 0 and e = 0	value = 0
If s = 1 and e = 0	value = DEC-Reserved Operand

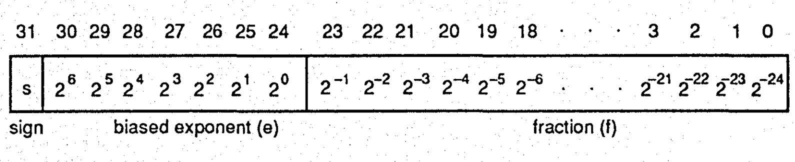
DEC-Reserved Operand: A DEC-Reserved Operand is interpreted as a signal or symbol. DEC-Reserved Operands are used to indicate invalid operations and operations whose results have overflowed the destination format. They may also be used to pass symbolic information from one calculation to another.

The DEC formats are fully described in the VAX Architecture Manual.

IBM Formats

IBM Single Precision

The IBM single-precision word is 32 bits wide and is arranged in the format shown below:



TB001080

The floating-point word is divided into three fields: a single-bit sign, a 7-bit biased exponent, and a 24-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; a true 0 has a positive sign.

The biased exponent is a 7-bit unsigned integer representing a multiplicative factor of some power of 16. The bias value is 64. If, for example, the multiplicative value for a floating-point number is to be 16^a, the value of the biased exponent is a + 64, where "a" is the true exponent.

The fraction is a 24-bit unsigned fractional field containing the 24 least significant bits of the floating-point number's 25-bit mantissa. The weight of the fraction's most significant bit is 2^{-1}. The weight of the least significant bit is 2^{-24}.

An IBM floating-point number is evaluated or interpreted as follows:

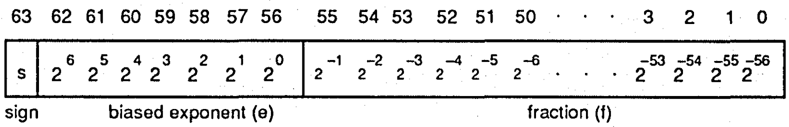
$$\text{Value} = (-1)^S\, 16^{e-64}(0.f)$$

Zero: There are two classes of zero. If the sign, biased exponent, and fraction are all zero, the operand is known as a "True Zero." If the fraction is zero, but the sign and biased exponent are not both zero, the operand is known as a "Floating-point Zero."

The IBM format is fully described in the IBM System/370 Principles of Operation Manual.

IBM Double Precision

The IBM double-precision word is 64 bits wide and is arranged in the format shown below:



TB00110

The floating-point word is divided into three fields: a single-bit sign, a 7-bit biased exponent, and a 56-bit fraction.

The sign bit is 0 for positive numbers and 1 for negative numbers; a true 0 has a positive sign.

The biased exponent is a 7-bit unsigned integer representing a multiplicative factor of some power of 16. The bias value is 64. If, for example, the multiplicative value for a floating-point number is to be 16^a, the value of the biased exponent is a + 64, where "a" is the true exponent.

The fraction is a 56-bit unsigned fractional field containing the 56 least significant bits of the floating-point number's 57-bit mantissa. The weight of the fraction's most significant bit is 2^{-1}. The weight of the least significant bit is 2^{-56}. An IBM floating-point number is evaluated or interpreted as follows:

$$\text{Value} = (-1)^S 16^{e-64}(0.f)$$

Zero: There are two classes of zero. If the sign, biased exponent, and fraction are all zero, the operand is known as a "True Zero." If the fraction is zero, but the sign and biased exponent are not both zero, the operand is known as a "Floating-point Zero."

The IBM format is fully described in the IBM System/370 Principles of Operation Manual.

APPENDIX B—ROUNDING MODES

The round mode is selected by mode register field RMS as follows:

RMS	Round Mode
000	Round to Nearest (IEEE)
001	Round to Minus Infinity (IEEE)
010	Round to Plus Infinity (IEEE)
011	Round to Zero (IEEE)
100	Round to Nearest (DEC)
101	Round Away from Zero
11X	Illegal Value

Round to Nearest (IEEE)

The infinitely precise result of an operation is rounded to the closest representable value in the destination format. If the infinitely precise result is exactly halfway between two representations, it is rounded to the representation having a least significant bit of 0.

Round to Minus Infinity (IEEE)

The infinitely precise result of an operation is rounded to the closest representable value in the destination format that is less than or equal to the infinitely precise result.

Round to Plus Infinity (IEEE)

The infinitely precise result of an operation is rounded to the closest representable value in the destination format that is greater than or equal to the infinitely precise result.

Round to Zero (IEEE)

The infinitely precise result of an operation is rounded to the closest representable value in the destination format whose magnitude is less than or equal to the infinitely precise result.

Round to Nearest (DEC)

The infinitely precise result of an operation is rounded to the closest representable value in the destination format. If the infinitely precise result is exactly halfway between two representations, it is rounded to the representation having the greater magnitude.

Round Away from Zero

The infinitely precise result of an operation is rounded to the closest representable value in the destination format whose magnitude is greater than or equal to the infinitely precise result.

A graphical representation of these round modes is shown in Figures B1 and B2.

The IEEE standard specifies that all four "IEEE" modes be available so that the user may select the mode most appropriate for the algorithm being executed. The DEC standard specifies that two rounding modes be available—Round-to-Nearest (DEC) and Round-to-Zero. The IBM standard specifies that all operations be performed using the Round-to-Zero mode.

It should be noted, however, that the Am29027 permits *any* of the supported rounding modes to be selected, regardless of the format of the operation. It is permissible to use one of the IEEE rounding modes with an IBM operation, or DEC rounding with an IEEE operation, or any other possible combination. For those integer operations where rounding is performed, any rounding mode may be chosen. This flexibility allows the user to select the mode most appropriate for the arithmetic environment in which the processor is operating.

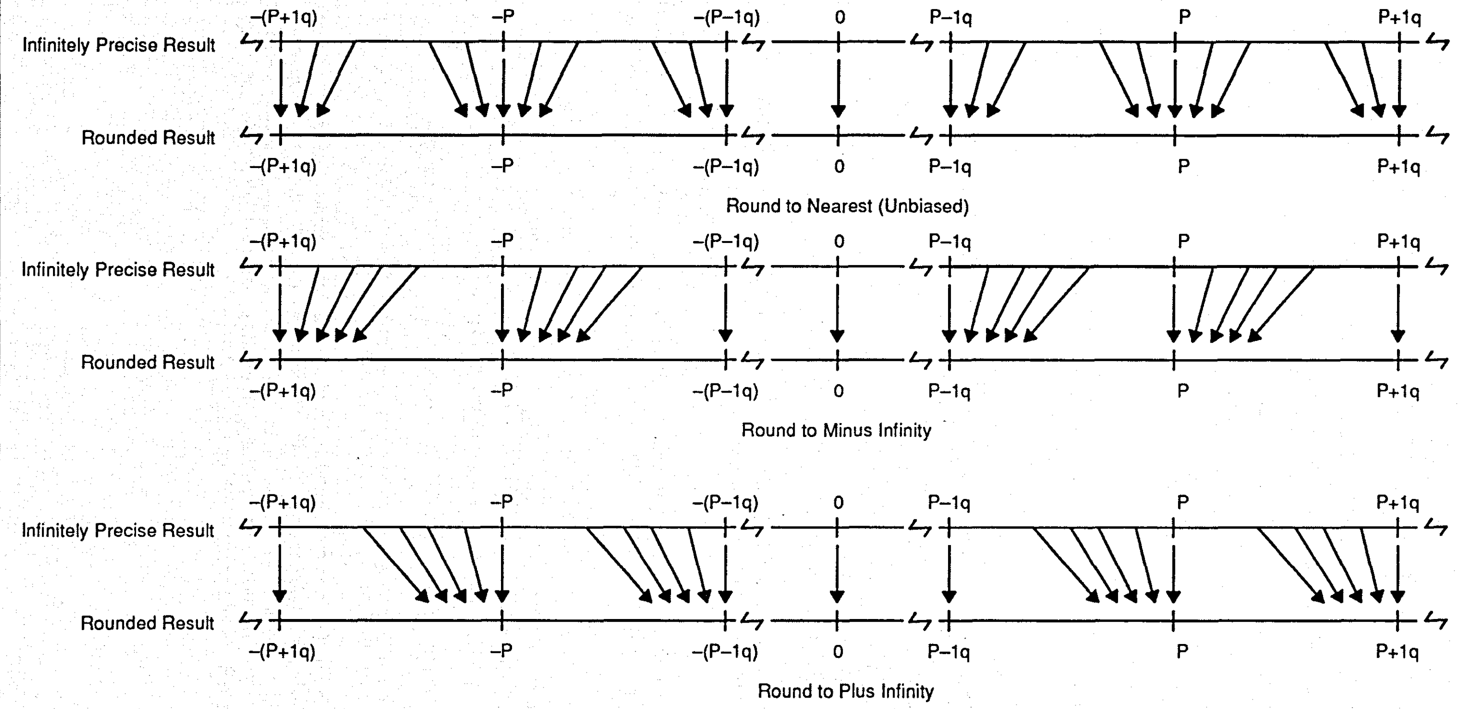


Figure B1. Graphical Interpretation of Round-to-Nearest (Unbiased), Round-to-Minus-Infinity, and Round-to-Plus-Infinity Rounding Modes

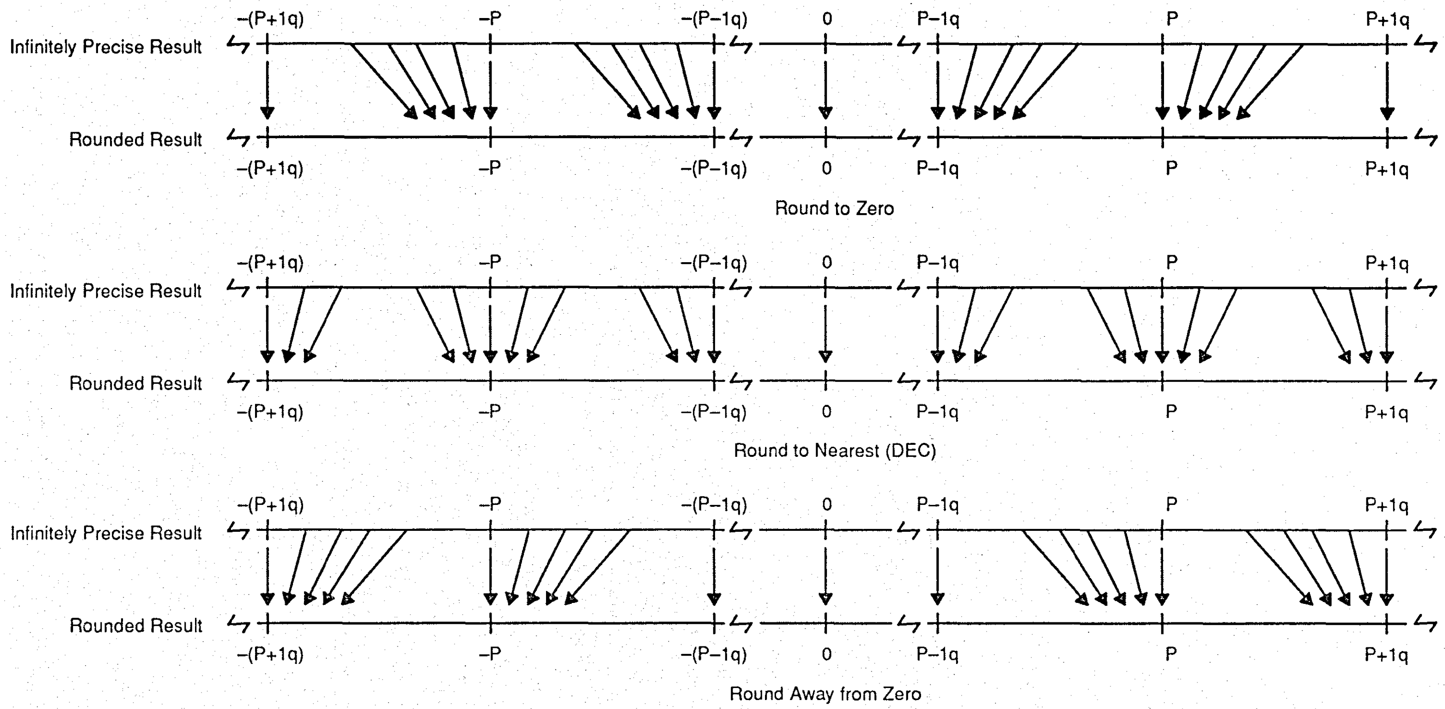


Figure B2. Graphical Interpretation of Round-to-Zero, Round-to-Nearest (DEC), and Round-Away-from-Zero Rounding Modes

APPENDIX C—ADDITIONAL OPERATION DETAILS

There are several cases in which the implementation of the IEEE, DEC, and IBM floating-point standards in the Am29C327 differs from the formal definitions of those standards. This appendix describes these differences.

Differences Between Floating-Point Arithmetic and Am29027 IEEE Operation

Section 7.3 of the IEEE-754 standard specifies that "Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that *or a wider format*, possibly with the exponent bias adjusted, but rounded *to the destination's precision*."

According to the IEEE standard, then, if a double-to-single IEEE operation overflows while traps are enabled, the result is a *double-precision* operand, rounded to single-precision width (23-bit fraction), together with a correctly adjusted (double-precision) exponent and the appropriate flags for a trapped overflow.

In the case of an overflow in *any* IEEE operation, the Am29027 returns a result in the destination format specified by the user, rounded to that destination format.

In the case of the double-to-single overflow described above, the result from the Am29027 is a *single-precision* operand, together with a correctly adjusted (single-precision) exponent and the appropriate flags for a trapped overflow.

A simple example serves to illustrate the discrepancy by describing the conversion of the double-precision IEEE number 52B123456789ABCD to single-precision, with traps enabled, and the round-to-nearest rounding mode selected. This number is too large to be represented in single-precision format.

According to the IEEE standard, the result of this operation is the double-precision number 52B1234560000000, comprising the double-precision exponent of the input and a fraction truncated to 23 bits, together with flags V and X.

When the operation is performed in the Am29027, however, using the F′ = P′ operation with appropriate precision controls, the result is the single-precision number 75891A2B, comprising the single-precision (overflowed) exponent reduced by 192 (decimal) and a single-precision fraction, together with flags V and X.

It should be noted that trapped operation is an optional part of the IEEE standard. Full adherence to the IEEE specification of trapped operation is therefore not necessary to ensure compliance with IEEE-754.

Differences Between DEC Floating-Point Arithmetic and Am29027 DEC Operation

The DEC F, DEC D, and DEC G standards, as implemented in the Am29027, differ from the implementations in a VAX only in the way in which the subfields of the floating-point word are arranged. The differences are listed in Table C1.

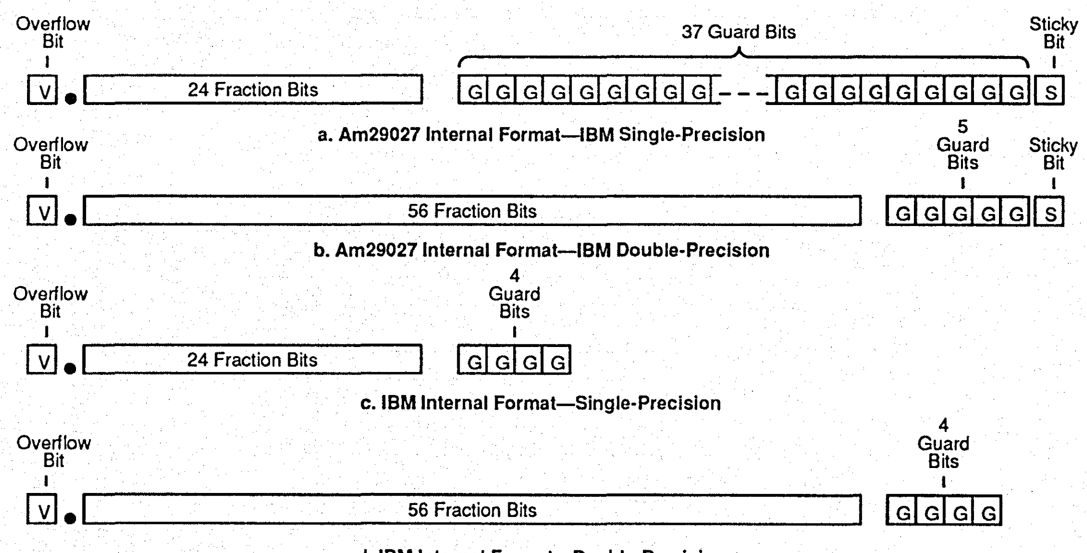
Table C1. Differences In Am29027 and DEC Floating-Point Formats

	Am29027 Arrangement	VAX Arrangement
DEC F	sign: bit 31 exponent: bits 30–23 fraction: bits 22–0	sign: bit 15 exponent: bits 14–7 fraction: bits 6–0, bits 31–16
DEC D	sign: bit 63 exponent: bits 62–55 fraction: bits 54–0	sign: bit 15 exponent: bits 14–7 fraction: bits 6–0, bits 31–16, bits 47–32, bits 63–48
DEC G	sign: bit 63 exponent: bits 62–52 fraction: bits 51–0	sign: bit 15 exponent: bits 14–4 fraction: bits 3–0, bits 31–16, bits 47–32, bits 63–48

Differences Between IBM 370 Floating-Point Arithmetic and Am29027 IBM Operation

The Am29027's deviations from the IBM standard may be summarized as follows, *assuming that the user has selected the round-to-nearest rounding mode*:

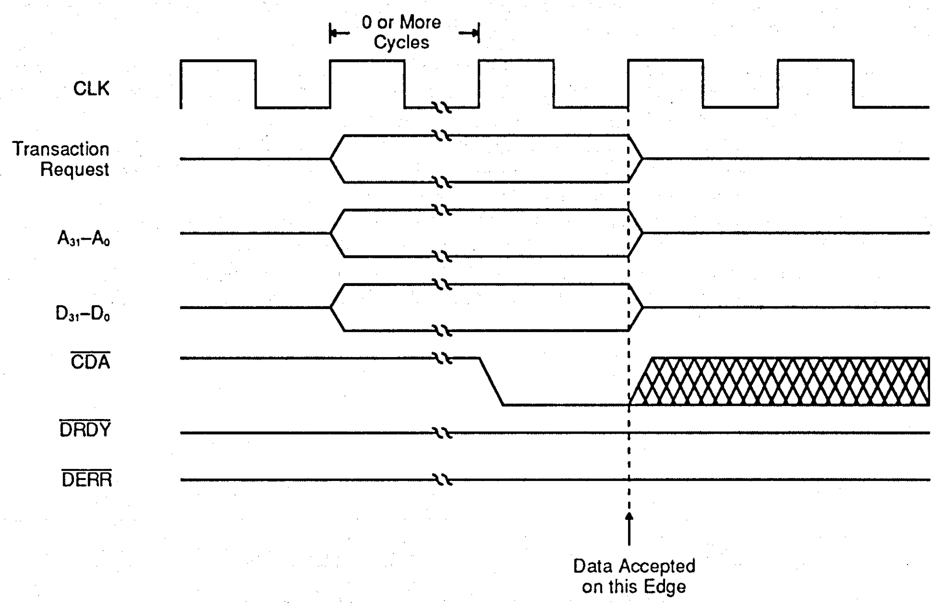
1. The Am29027 provides more guard bits in its internal format than specified by the IBM standard. With certain combinations of input operands, the Am29027 produces more accurate results than a standard IBM processor for instructions based on addition operations and comparisons.
2. The discrepancies are much larger for single-precision operations than double-precision operations, because the difference in the number of guard bits is much greater (33 more for single, one more for double).
3. There is no universal rule for determining whether a given set of input operands will result in a discrepancy. Pro vided the conditions in (1) above are met, the user must examine each operation on a case-by-case basis, taking into account the input operands and the internal formats discussed in this section.
4. The Am29027 does not produce unnormalized results from additions. The results of all addition operations are renormalized. Am29027 internal formats are compared with IBM internal formats in Figure C1.


Overflow Bit
37 Guard Bits
Sticky Bit
V
24 Fraction Bits
G
S
a. Am29027 Internal Format—IBM Single-Precision
5 Guard Bits
56 Fraction Bits
b. Am29027 Internal Format—IBM Double-Precision
4 Guard Bits
c. IBM Internal Format—Single-Precision
d. IBM Internal Format—Double-Precision

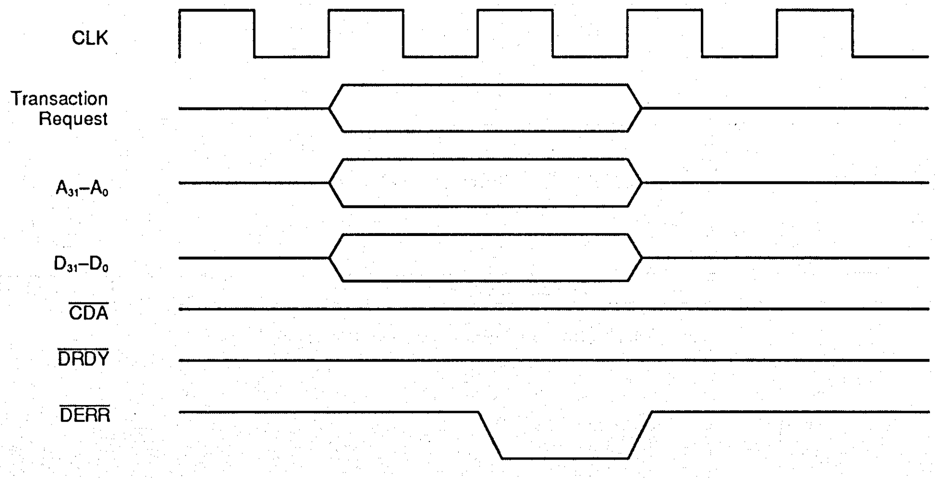
09114-016C

Figure C1. Differences in Internal Mantissa Formats of an IBM CPU and the Am29027

APPENDIX D—TRANSACTION REQUEST/OPERATION TIMING



a. Normal Operation, Data Accepted

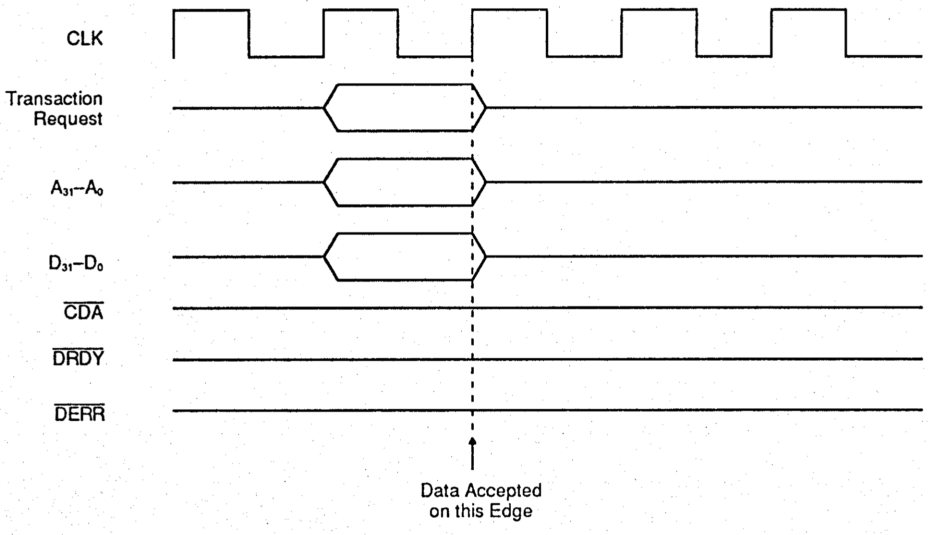


b. Halt On Error Mode, Unmasked Exception Present

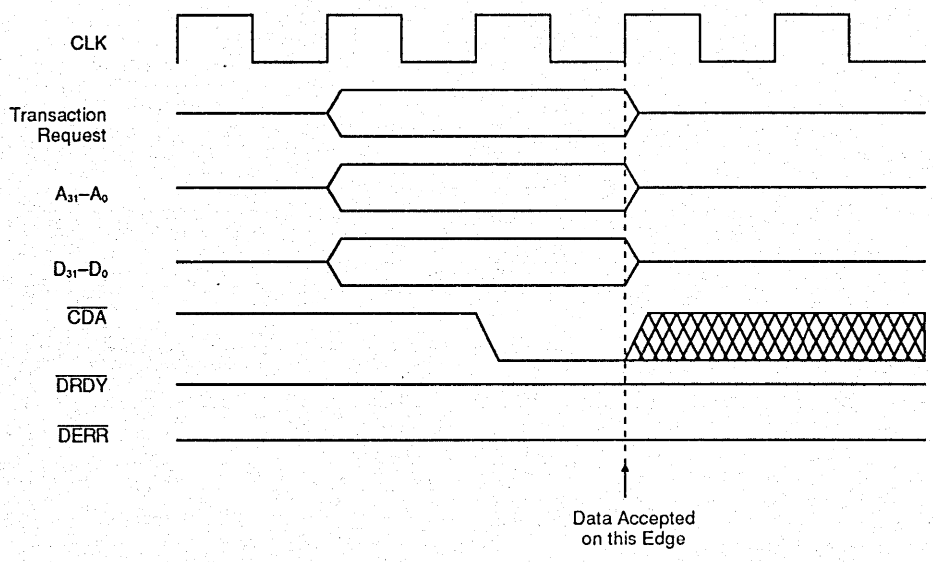
091148-017C

Note: Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D1. Timing for the Write Operand R, Write Operand S, Write Operands R, S, and Write Instruction Transaction Requests



a. $\overline{\text{CDA}}$ Low

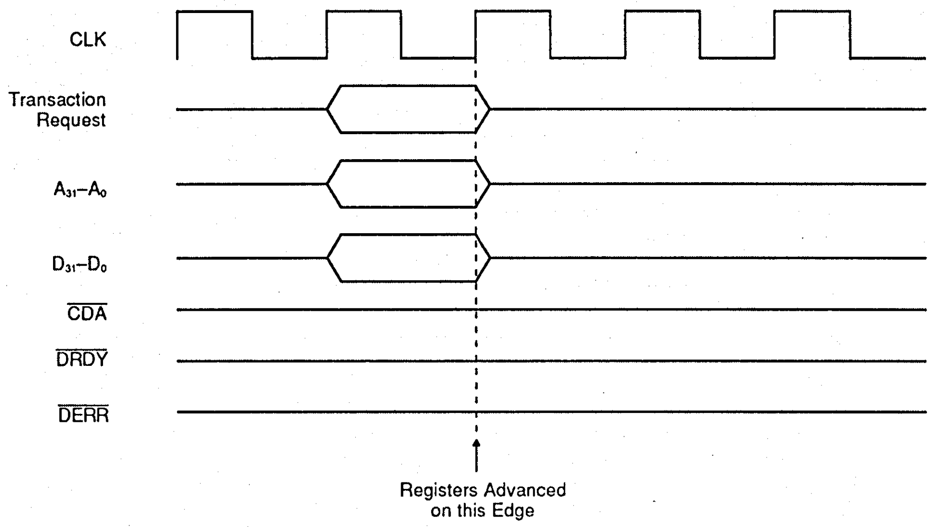


b. $\overline{\text{CDA}}$ High Initially

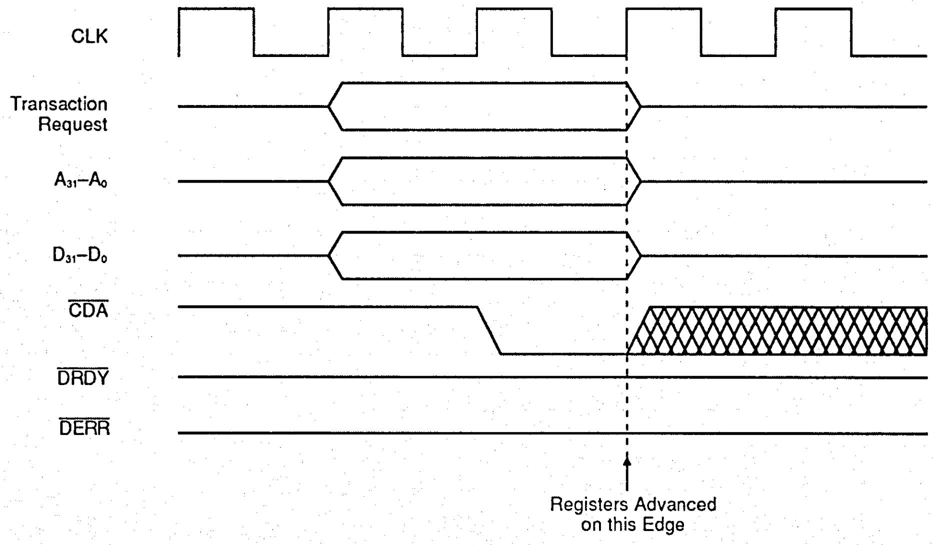
Note: Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

09114-018C

Figure D2. Timing for the Write Mode, Write Status, and Write Register File Precisions Transaction Requests



a. $\overline{\text{CDA}}$ Low

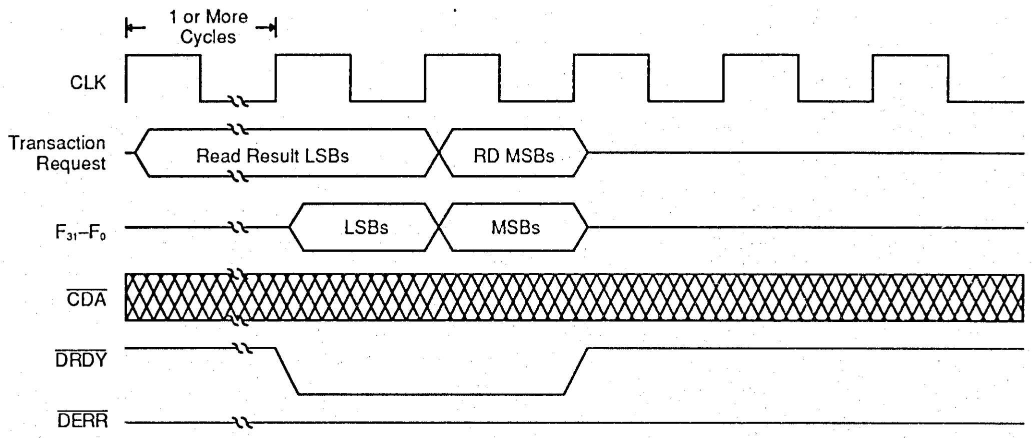


b. $\overline{\text{CDA}}$ High Initially

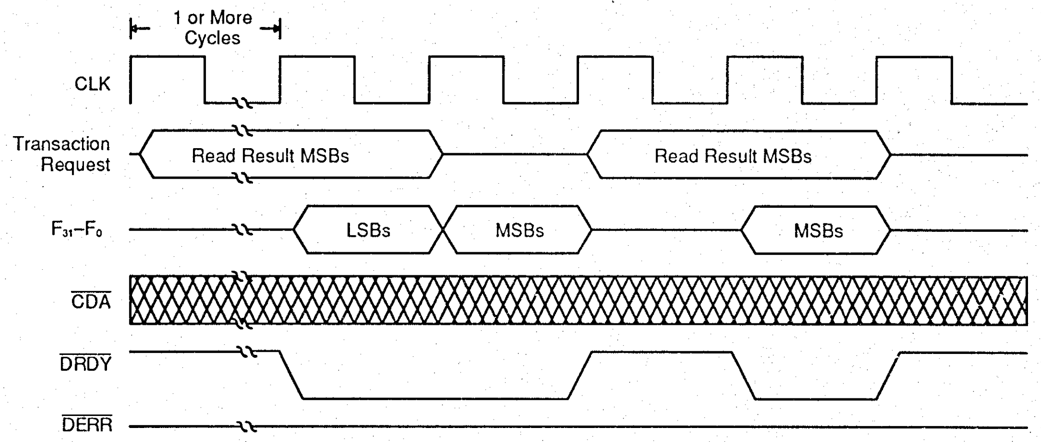
09114-019C

Note: Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D3. Timing for the Advance Temp. Registers Transaction Request



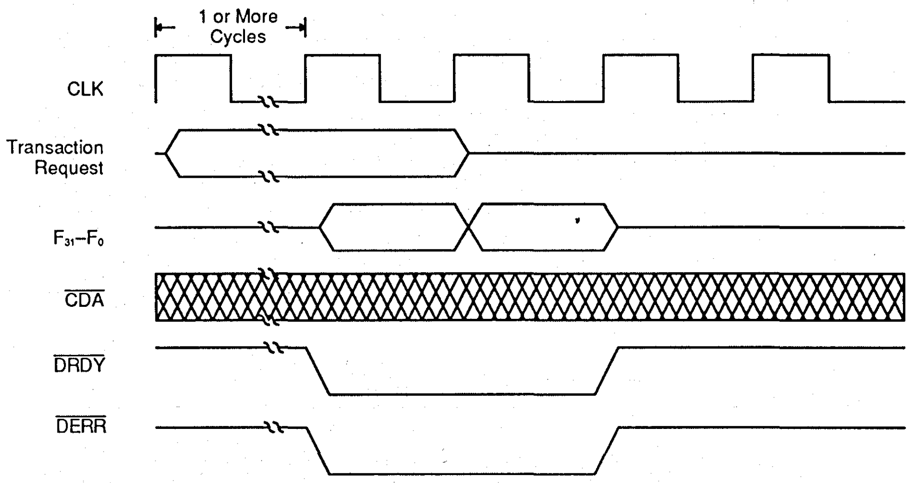
a. Read Result MSBs Request Issued in Cycle after Read Result LSBs Request



b. Read Result MSBs Request Issued Two or More Cycles after Read Result LSBs Request

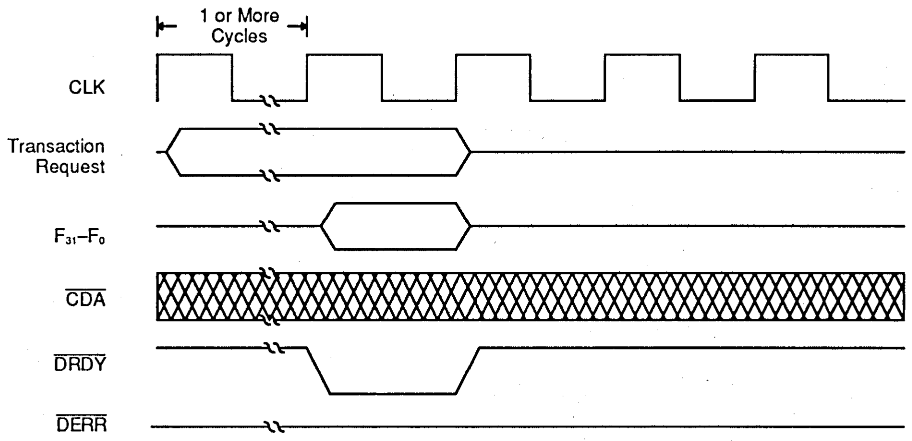
09114-020C

Figure D4. Timing for the Read Result LSBs Transaction Request, No Unmasked Exceptions

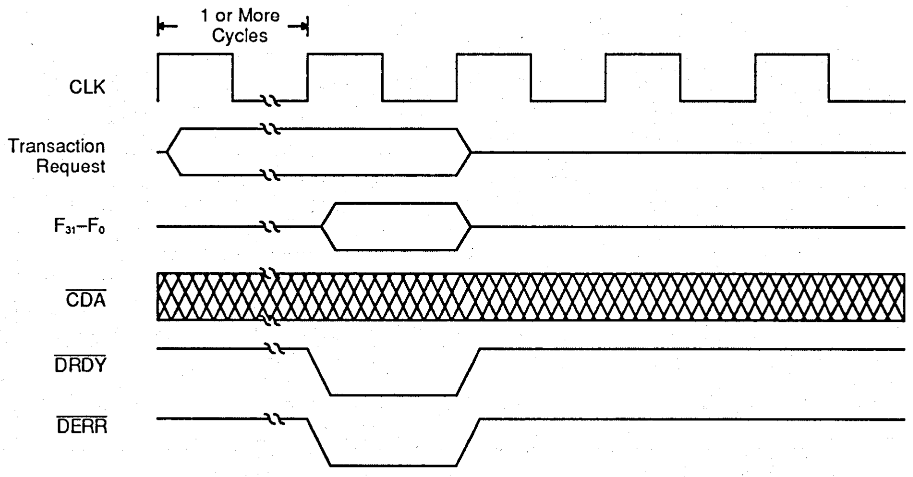


09114-021C

Figure D5. Timing for Read Result LSBs Transaction Request, Unmasked Exception Present



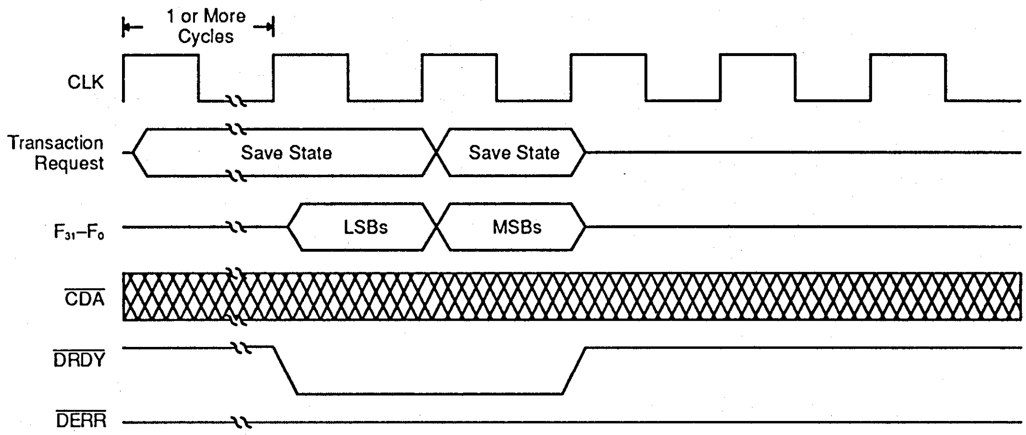
a. No Unmasked Exceptions Present



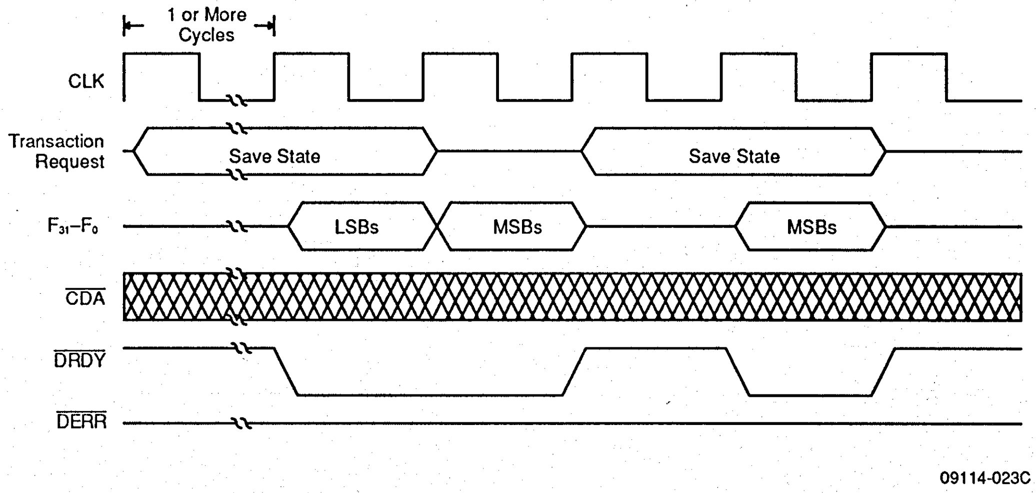
b. Unmasked Exceptions Present

09114-022C

Figure D6. Timing for Read Result MSBs, Read Flags, and Read Status Transaction Requests



a. Second Save State Request Issued in Cycle Following First Request



b. Second Save State Request Issued Two or More Cycles after First Request

Figure D7. Timing for the Save State Transaction Request, 64-Bit Resources (Registers R, R-Temp, S, S-Temp; Register File Locations RF_7–RF_0: Mode Register)

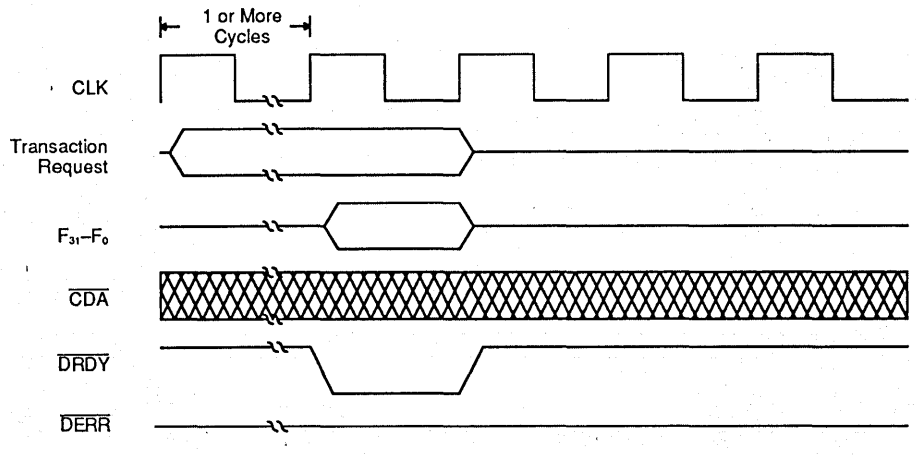
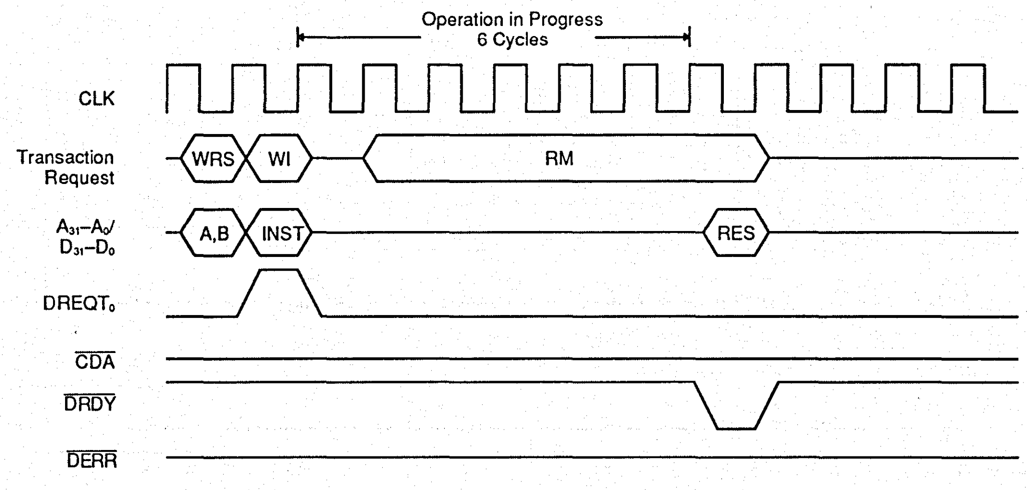


Figure D8. Timing for the Save State Transaction Request, 32-Bit Resources (Instruction Register, Register I-Temp, Status Register, Precision Register)

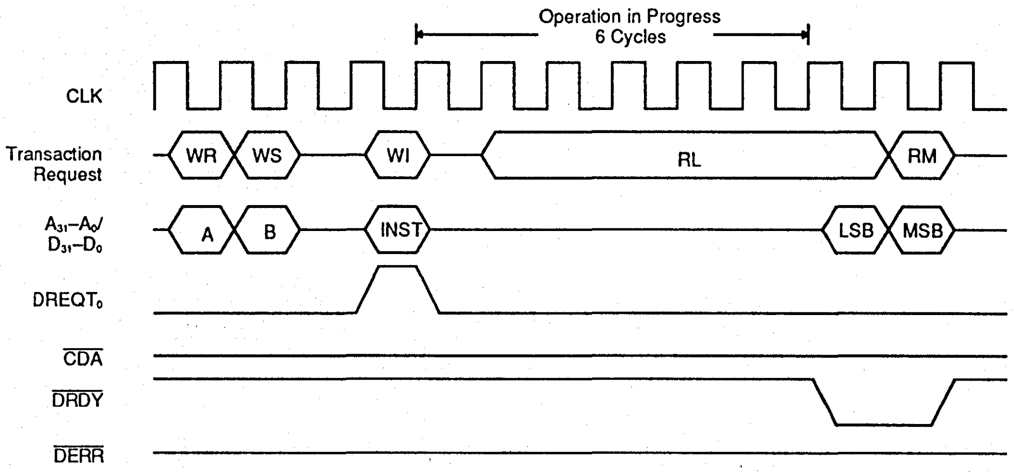


Notes:					
	WRS	= Write Operands R, S	WI	=	Write Instruction
	RM	= Read MSBs	A, B	=	Operands A, B
	INST	= Addition Instruction	RES	=	Result

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

09114-025C

Figure D9. Typical Timing for Single-Precision Operation in Flow-Through Mode—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC=6

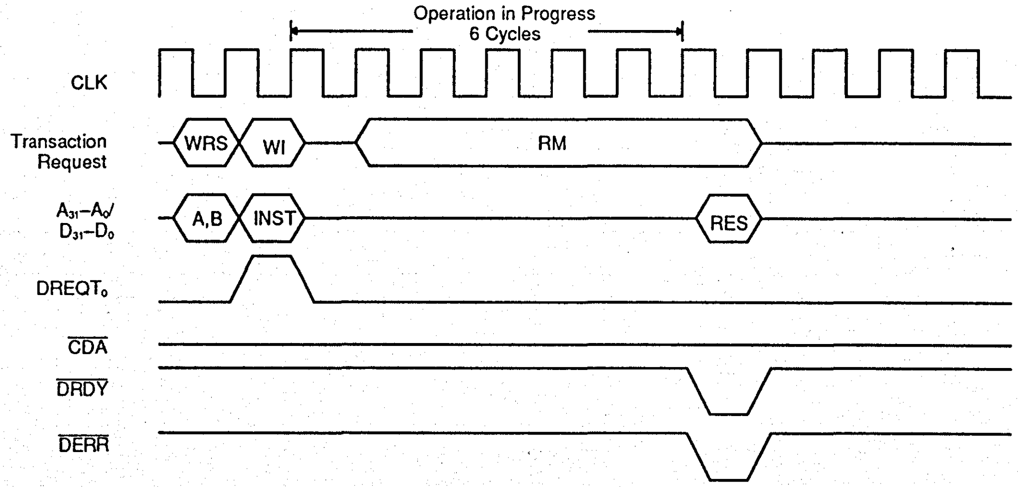


09114-026C

Notes:		
WR = Write Operand R	WS = Write Operand S
WI = Write Instruction	RL = Read LSBs
RM = Read MSBs	A = Operand A
B = Operand B	INST = Addition Instruction
LSB = Result LSBs	MSB = Result MSBs

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D10. Typical Timing for the Double-Precision Operation in Flow-Through Mode—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC = 6

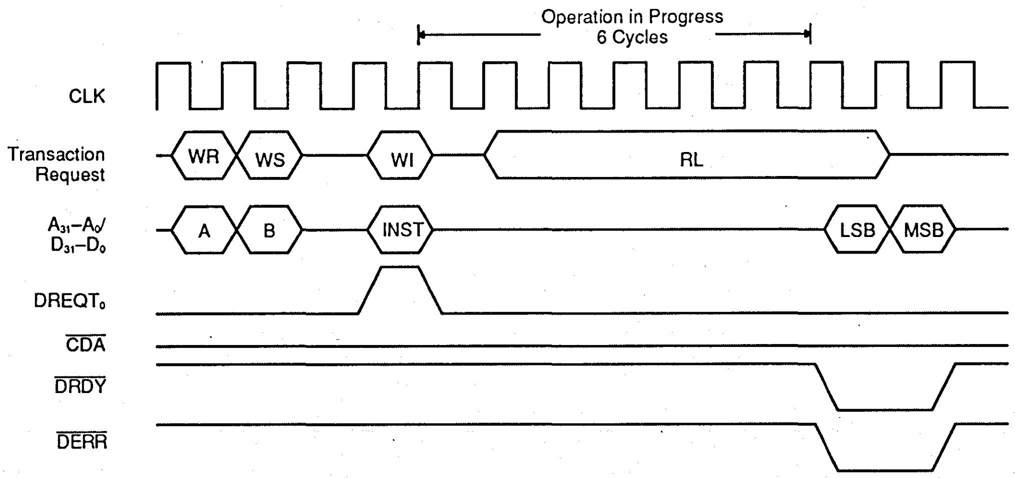


09114-027C

Notes:	
WRS = Write Operands R, S	WI = Write Instruction
RM = Read MSBs	A, B = Operands A, B
INST = Addition Instruction	RES = Result

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

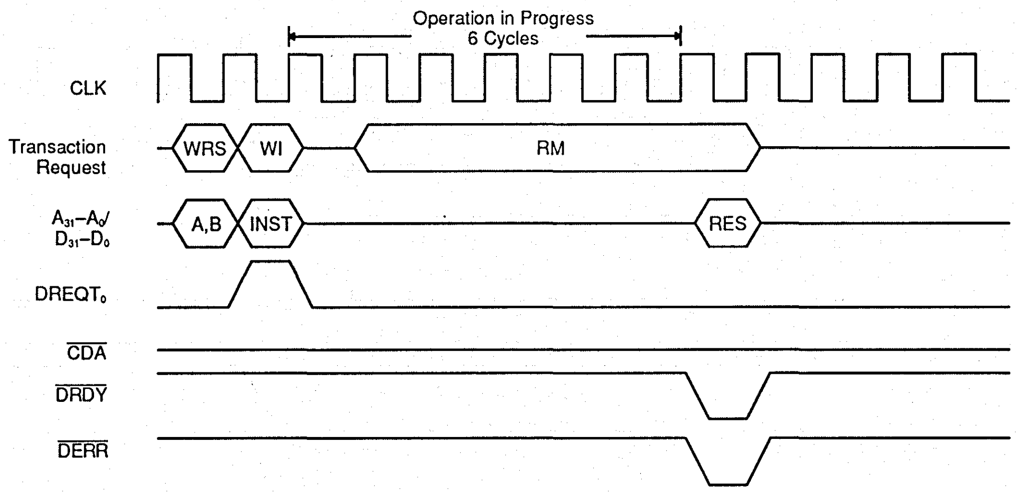
Figure D11. Typical Timing for Single-Precision Operation in Flow-Through Mode, with Unmasked Exception Present—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC = 6



Notes:	WR	= Write Operand R	WS	= Write Operand S	
	WI	= Write Instruction	RL	= Read LSBs	
	A	= Operand A	B	= Operand B	09114-028C
	INST	= Addition Instruction	LSB	= Result LSBs	
	MSB	= Result MSBs			

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

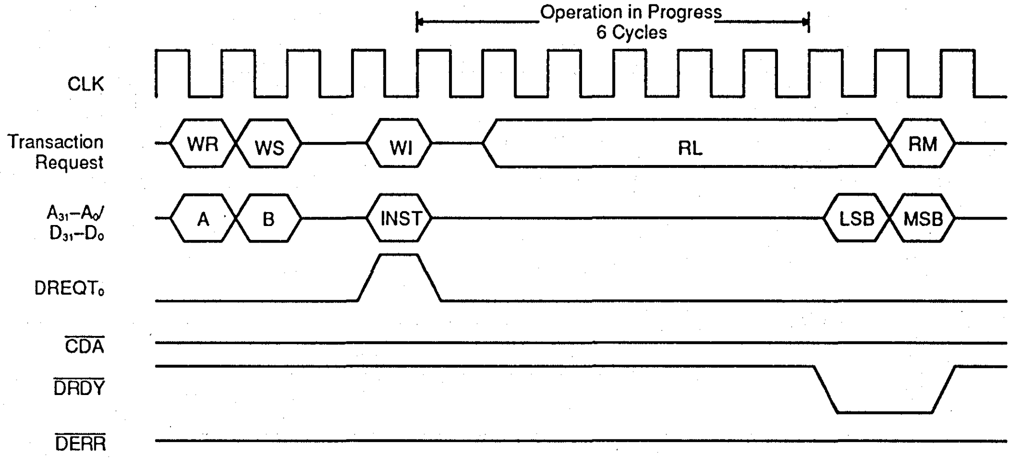
Figure D12. Typical Timing for Double-Precision Operation in Flow-Through Mode, with Unmasked Exception Present—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC = 6



Notes:	WRS	= Write Operands R, S	WI	= Write Instruction	
	RM	= Read MSBs	A, B	= Operands A, B	09114-029C
	INST	= Addition Instruction	RES	= Result	

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D13. Typical Timing for Single-Precision Operation in Flow-Through Mode, with $\overline{\text{DRDY}}$ Advanced—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC = 6

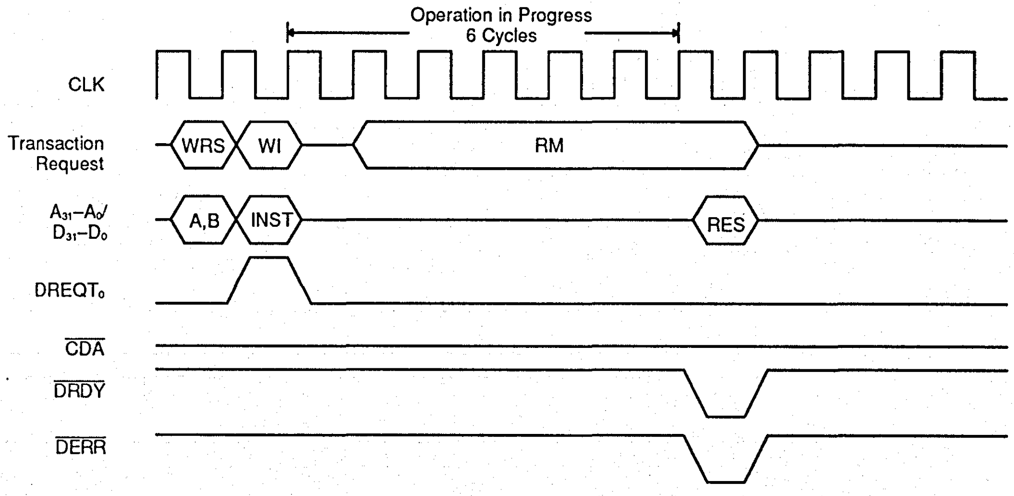


Notes: WR	= Write Operand R	WS	= Write Operand S
WI	= Write Instruction	RL	= Read LSBs
RM	= Read MSBs	A	= Operand A
B	= Operand B	INST	= Addition Instruction
LSB	= Result LSBs	MSB	= Result MSBs

09114-030C

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D14. Typical Timing for Double-Precision Operation in Flow-Through Mode, with $\overline{DRD}$ Advanced—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC=6

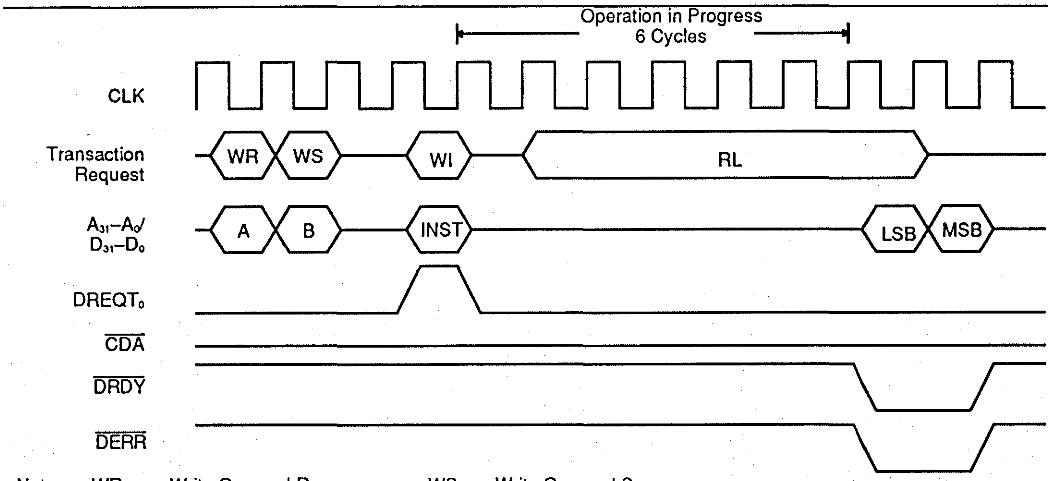


Notes: WRS	= Write Operands R, S	WI	= Write Instruction
RM	= Read MSBs	A, B	= Operands A, B
INST	= Addition Instruction	RES	= Result

09114-031C

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D15. Typical Timing for Single-Precision Operation in Flow-Through Mode, with $\overline{DRDY}$ Advanced and Unmasked Exception Present—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC=6



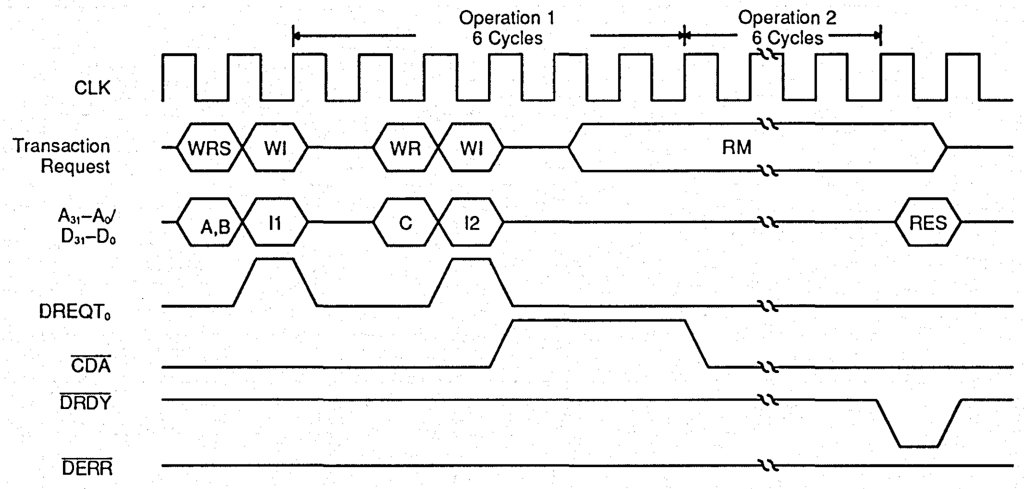
Notes:

WR = Write Operand R		WS = Write Operand S	
WI = Write Instruction		RL = Read LSBs	
A = Operand A		B = Operand B	
INST = Addition Instruction		LSB = Result LSBs	
MSB = Result MSBs			

09114-037C

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D16. Typical Timing for Double-Precision Operation in Flow-Through Mode, with $\overline{DRDY}$ Advanced and Unmasked Exception Present—Perform the Operation A PLUS B, Read the Result; Mode Register Field PLTC = 6



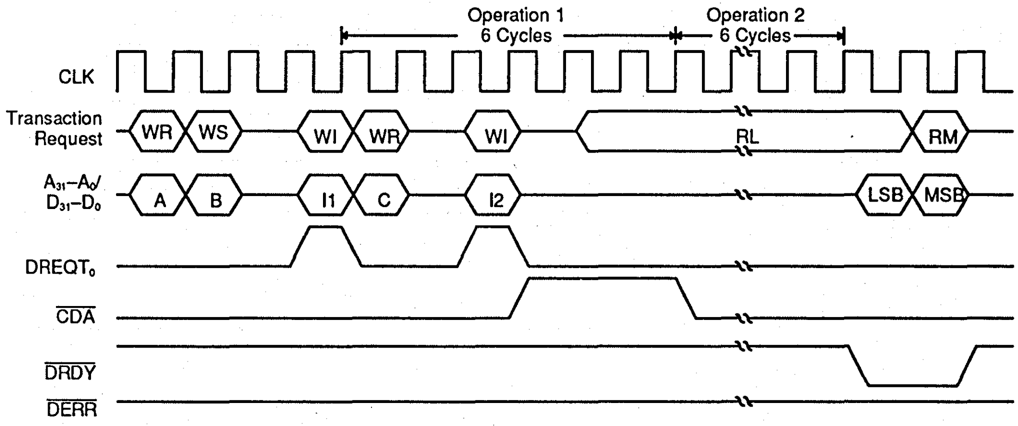
Notes:

WRS = Write Operands R, S	WI = Write Instruction
WR = Write Operand R	RM = Read MSBs
A, B = Operands A, B	I1 = Addition Instruction
C = Operand C	I2 = Multiplication Instruction
RES = Result	

09114-032C

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D17. Typical Timing for Overlapped Single-Precision Operations in Flow-Through Mode; Perform the Compound Operation (A PLUS B) × C by Performing Operations: (1) $RF_0 \leftarrow$ A PLUS B, (2) $RF_0 \times$ C Mode Register Field PLTC = 6

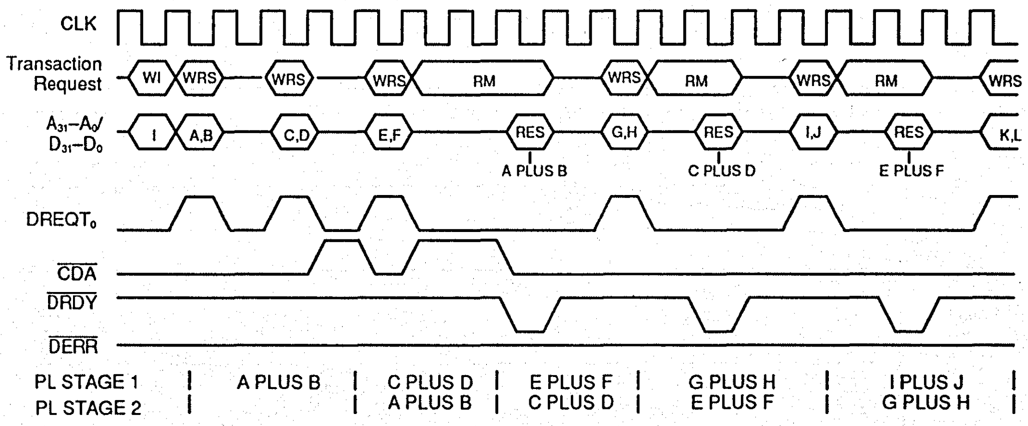


Notes:					
WR	=	Write Operand R	WS	=	Write Operand S
WI	=	Write Instruction	RL	=	Read LSBs
RM	=	Read MSBs	A	=	Operand A
B	=	Operand B	C	=	Operand C
I1	=	Addition Instruction	I2	=	Multiplication Instruction
LSB	=	Result LSBs	MSB	=	Result MSBs

Signals A_{31}–A_{01} and D_{3}–D_0 are the Am29000 address and data buses, respectively.

09114-033C

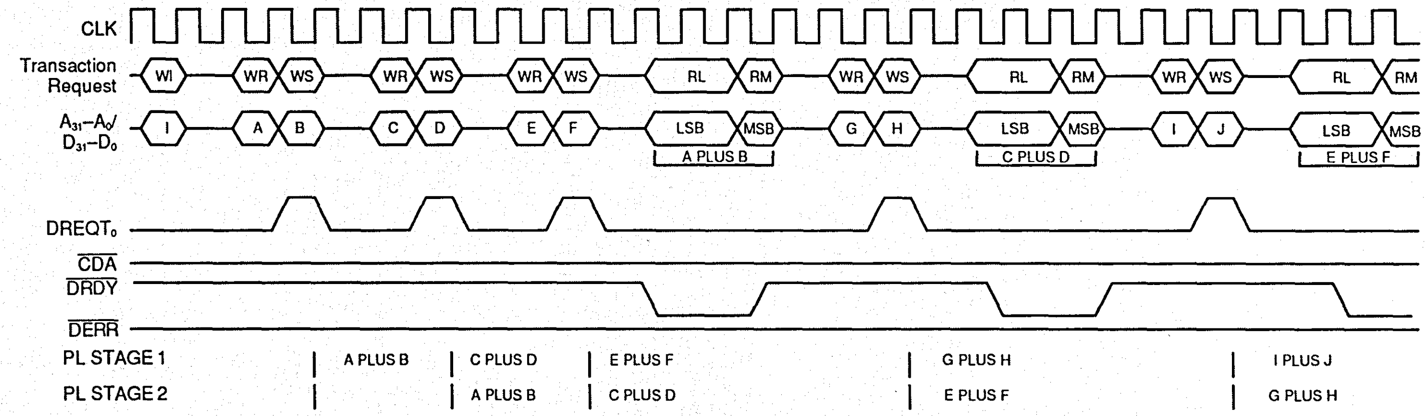
Figure D18. Typical Timing for Overlapped Double-Precision Operations in Flow-Through Mode; Perform the Compound Operation (A PLUS B) × C by Performing Operations: (1) $RF_0 \leftarrow$ A PLUS B, (2) $RF_0 \times$ C; Mode Register Field PLTC=6 Mode Register Field PLTC=6



Notes:					
WI	=	Write Instruction	WRS	=	Write Operands R, S
RM	=	Read MSBs	I	=	Addition Instruction
A, B, . . .	=	Operands	RES	=	Result

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

Figure D19. Typical Timing for Single-Precision Operations in Pipeline Mode; Perform a Series of Addition Operations A PLUS B, C PLUS D, E PLUS F, . . . Mode Register Field PLTC = 3



Notes:		
WI	= Write Instruction	WR = Write Operand R
WS	= Write Operand S	RL = Read LSBs
RM	= Read MSBs	I = Addition Instruction
A, B, . . .	= Operands	LSB = Result LSBs
MSB	= Result MSBs	

Signals A_{31}–A_0 and D_{31}–D_0 are the Am29000 address and data buses, respectively.

09114-035C

Figure D20. Typical Timing for Double-Precision Operations in Pipeline Mode; Perform a Series of Addition Operations A PLUS B, C PLUS D, E PLUS F, . . . Mode Register Field PLTC = 3

CHAPTER 2
29K Family Support Tools

Preliminary

ASM29K

Cross-Development Toolkit, Release 2

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- **Relocatable Macro Assembler supports complete Am29000™ microprocessor instruction set.**
- **Linker/Loader combines separately assembled modules by resolving external references and by searching libraries.**
- **Librarian provides management facility for organizing modules into logical collections of functions.**
- **IEEE Software Floating-Point Emulation routines.**
- **Available for the PC-AT™, and Sun-3™ development environments.**

GENERAL DESCRIPTION

Processor performance depends on the processor's hardware and software environment. The key to maximizing performance lies in the realization that the processor is part of a system that is a collection of components that must be integrated properly. To take advantage of the advanced RISC architecture of the Am29000 microprocessor, equally sophisticated software tools must be available.

The ASM29K™ cross-development toolkit offers such a development environment for creating efficient and portable Am29000 microprocessor software. The package consists of the assembler, the linker, the floating-point emulation routines, and the object module librarian. These tools allow users to design more efficient systems and applications than ever before.

Cross-development is the design of an application program on one computer (the *host* system) and the execution of that same application program on a different computer (the *target* system). The operating system on the host, such as UNIX™ or DOS, provides the tools needed to create the application program. These tools include editors for writing the source code, compilers and assemblers for translating the modules into executable code, and utilities for preparing the application for execution. The Am29000 microprocessor-based target computer generally does not provide the tools required to develop the application program. Figure 1 shows the path that an application follows from development on the host system to execution on the target system.

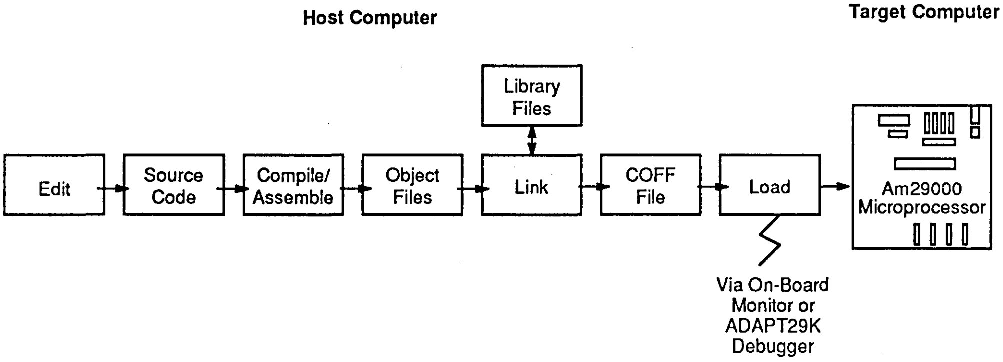


Figure 1. Cross Software Development

Publication #	Rev.	Amendment
10292	B	/0
Issue Date: September 1989		

The ASM29K cross-development toolkit transforms a PC or Sun-3 workstation host into a powerful software development environment. ASM29K software assembles user source and produces a relocatable object module. This module can be combined with other relocatable object modules (derived from the assembler or high-level language cross-compilers) using the ASM29K linker. Library modules prepared by the librarian can be linked in at this point as well. The resulting absolute object module then can be downloaded to a target system.

AMD has established and published the Am29000 microprocessor Common Object File Format (COFF) to which all Am29000 development tools conform. The AMD COFF format extends the already standard AT&T COFF format to support source-level debugging and other Am29000 microprocessor-specific features. Similarly, AMD has established a common calling convention that maximizes performance on the Am29000 microprocessor as well as defining another standard for software vendors. This has led to a variety of compilers, assemblers, debuggers, and associated tools that may be mixed freely by developers of Am29000 microprocessor software.

The contents of the ASM29K cross-development toolkit include:

- ASM29K macro assembler
- ASM29K linker
- ASM29K librarian
- Hex utilities
- IEEE floating-point emulation routines
- Documentation

ORDERING INFORMATION

Licensing

The ASM29K cross-development toolkit is licensed through AMD's Standard End-User Software License Agreement (Boxtop). This license does not require a signature; breaking the seal on the software envelope indicates acceptance of the license terms. If changes are required to the license agreement, they can be arranged through your AMD sales representative. Many software products require the customer to provide a CPU ID number when ordering the product. Contact your sales representative if this information is not available at the time of purchase. In addition, terms of the license require the customer to complete a Software Warranty card with the serial number and site of the host computer on which the software will reside. This card must be returned to AMD within 30 days of receipt for the warranty to be valid.

Order Numbers

The ASM29K cross-development toolkit is available for several different environments. Documentation can be ordered separately. The order number (valid combination) is formed as a combination of:

- Product Family
- Product Category
- Product Identifier
- License Type
- Host / OS Type
- Media Type

ORDER INFORMATION (continued)

AM29000 SW/ ASM B ## ##

Media Type
08 = 0.25″ Sun cartridge tape, TAR format
14 = 3.5″ DSHD floppies
21 = 9-track, 1600 BPI mag tape, TAR format
24 = 5.25″ DSHD floppies

Host / OS Type
07 = Sun-3
10 = PC-AT

License Type
B = Boxtop
S = Signed
"–" = Not Applicable

Product Identifier
ASM = ASM29K Cross-Development Toolkit

Product Category
SW/ = Software Product
DC/ = Documentation Product
MA/ = Maintenance Agreement

Product Family
Am29000 Microprocessor

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly released combinations.

Order Number	Product	Host	Media
AM29000SW/ASMB0708	ASM29K Toolkit	Sun-3	0.25″ cartridge tape, TAR format
AM29000SW/ASMS0708	ASM29K Toolkit	Sun-3	0.25″ cartridge tape, TAR format
AM29000SW/ASMB0721	ASM29K Toolkit	Sun-3	9-track, 1600 BPI tape, TAR format
AM29000SW/ASMS0721	ASM29K Toolkit	Sun-3	9-track, 1600 BPI tape, TAR format
AM29000SW/ASMB1014	ASM29K Toolkit	PC-AT	3.5″ DSHD floppies
AM29000SW/ASMS1014	ASM29K Toolkit	PC-AT	3.5″ DSHD floppies
AM29000SW/ASMB1024	ASM29K Toolkit	PC-AT	5.25″ DSHD floppies
AM29000SW/ASMS1024	ASM29K Toolkit	PC-AT	5.25″ DSHD floppies
AM29000DC/ASM-99	ASM29K Documentation	UNIX	Not Media Specific
AM29000MA/ASM-07	ASM29K Maintenance	Sun-3	Not Media Specific
AM29000MA/ASM-10	ASM29K Maintenance	PC-AT	Not Media Specific

FUNCTIONAL INFORMATION

Assembler

The ASM29K assembler converts user-written Am29000 assembly code into relocatable object modules. It produces standard COFF object modules that can be linked with other assembled or compiled modules. Its advanced features permit the design of well-structured modules that are easily maintained.

The assembler processes Am29000 microprocessor instructions as defined in Chapter 8 of the *Am29000 User's Manual.* Each instruction mnemonic and register identifier is recognized in both upper and lower case. Identifiers (that is, user-named variables) can have up to 63 characters, all of which are significant. Integer, character, string, and floating-point constants are supported as well as complex expression analysis.

In addition to the Am29000 microprocessor instructions, the assembler supports a powerful macro facility. Programmers can define macros with multiple parameters and direct macros to be repeated a specified number of times. Macro code is inserted into the source code at the position of the macro call. Macros may use local labels—labels that are visible only within the macro itself—to label an instruction that can be copied several times throughout the program. Local labels are distinguished from regular labels by using the format "$n," where n can be from one to six digits.

The assembler also provides a number of directives for organizing the code into efficient sections or modules. Use of the *include* directive merges separate files during assembly. The *section* directive assigns areas of code to named text, data, uninitialized memory, or initialized memory sections. Conditional assembly is also supported. This useful feature allows the programmer to assemble code conditionally for debugging. The assembler directives are listed in Table 1.

The ASM29K software also produces a cross-reference table for symbols. Flags allow the programmer to print listings that contain expanded macros, instructions not assembled due to conditional statements, and symbol tables; and to insert user-specified headers into the listing.

The assembler optionally emits debug information for use with the XRAY29K™ source-level debugger. This information allows the programmer to specify the symbolic names of variables and labels during debugging sessions.

The wide selection of features available in the ASM29K assembler gives the user the latest tools to produce well-structured and maintainable code.

Linker

The ASM29K linker integrates a group of separately compiled or assembled modules into a composite module in which all references between modules are resolved. It processes and produces COFF modules, including any module produced by a compiler in any language and any assembler that adheres to the AMD-defined COFF and calling-convention standards. Incremental linking is supported also. The ASM29K linker produces an extensive load map with an optional symbol cross-reference table.

Object module libraries are searched with required modules automatically included. All code and data sections are given absolute addresses as specified by the programmer. The linker provides options that create ROMable programs, generate warnings for possible undefined external references, produce a global cross-reference, and list defined symbols. Directives to the linker may be included in a file (batch mode), on the command line, or in combination. Programmers can use the ASM29K to:

- Resolve external references between separately compiled or assembled modules.
- Assign absolute addresses.
- Direct section ordering.
- Perform incremental linking.
- Load only those library modules referenced for efficient code space use.
- Generates optionally ROMable programs.

Librarian

The ASM29K librarian is a management facility for organizing independently developed pieces of software into logical units. It permits the addition, deletion, and replacement of object modules in one or more libraries. The ASM29K librarian:

- Organizes and initializes modules into a library file.
- Lists library contents and information.
- Lists a library directory.

Table 1. Assembler Directives

Group	Directives	Meaning
File Processing	.end	End of Assembly
	.err	Generate Assembly Error
	.ident	Specify Module Name
	.include	Include Text File
Conditional Assembly	.else	Alternate Condition
	.endif	End of Conditional Assembly Block
	.if	Assemble if Value is Not Zero
	.ifdef	Assemble if Identifer is Defined
	.ifeqs	Assemble if Strings are Equal
	.ifnes	Assemble if Strings are Not Equal
	.ifnotdef	Assemble if Identifier is Not Defined
Listing Control	.eject	Advance to Top of Page
	.lflags	Set Listing Flags
	.list	Enable Listing
	.nolist	Disable Listing
	.print	Print to Standard Output
	.sbttl	Set the Listing Subtitle
	.space	Space N Lines
	.title	Set the Listing Title
Symbol Declaration	.equ	Equate a Symbol to a Value (Unlimited Scope)
	.extern	Declare Symbols as External to This Module
	.global	Make Symbols Visible to Other Modules
	.reg	Declare a Symbol as a Synonym for a Register
	.set	Set a Symbol to a Value (Limited Scope)
Section Declaration	.comm	Declare a Common Symbol
	.data	Use the .data Section
	.dsect	Declare a Dummy Section
	.lcomm	Declare a Local bss Symbol
	.sect	Declare a New Section
	.text	Use the .text Section
	.use	Use a Declared Section
Data Storage Declaration	.align	Specify Byte Alignment
	.ascii	Store the String
	.block	Reserve Bytes
	.byte	Initialize Bytes
	.double	Initialize Double-Precision Values
	.extend	Initialize Extended-Precision Values
	.float	Initialize Single-Precision Values
	.hword	Initialize Half-Words
	.word	Initialize Words
Repeat Block	.endr	End of Repeat Block
	.irep	Repeat for Each Item in the List
	.irepc	Repeat for Each Character in the String
	.rep	Repeat N Times
Macro Definition	.endm	End Macro Definition
	.exitm	Terminate Macro Expansion
	.macro	Macro Heading
	.purgem	Purge All Macros Listed
High-Level Language (HLL) Debugging	.def	Define Symbol Table Entry Directive
	.dim	Dimensions of an Array Attribute
	.endef	End of Symbol Definition Block Directive
	.file	Source Filename Directive
	.line	Source-File Line-Number Directive
	.ln	HLL Source-File Line-Number Directive
	.scl	Storage Class of a Symbol Attribute
	.size	Size of a Symbol Attribute
	.tag	Structure, Union, or Enumeration Identifier Attribute
	.type	Basic and Derived Type of a Symbol Attribute
	.val	Value of a Symbol Attribute

Floating-Point Emulation

The Am29000 microprocessor instruction set includes floating-point and integer math operations. In the current processor implementation, these instructions cause traps to routines that perform the operations. The user is provided with source to two complete sets of routines that emulate IEEE Floating-Point Standard 754 for each of the instructions listed in Table 2.

The first set of routines is provided for users who have integrated an Am29027™ arithmetic accelerator into their systems. The Am29000 microprocessor math instructions are emulated using the Am29027 co-processor.

The second set of routines implements emulation of the floating-point operations entirely in software. No special hardware is required.

Documentation instructs users how to integrate the package into their target system. Both packages are designed to insure upward compatibility with next generation processors.

Table 2. Arithmetic Instructions

Type	Mnemonic	Operation
Integer Arithmetic	MULTIPLY	Signed Multiply
	MULTIPLYU	Unsigned Multiply
	DIVIDE	Signed Divide
	DIVIDEU	Unsigned Divide
Single-Precision Floating-Point Arithmetic	FADD	Single-Precision Add
	FSUB	Single-Precision Subtract
	FMUL	Single-Precision Multiply
	FDIV	Single-Precision Divide
Double-Precision Floating-Point Arithmetic	DADD	Double-Precision Add
	DSUB	Double-Precision Subtract
	DMUL	Double-Precision Multiply
	DDIV	Double-Precision Divide
Floating-Point Compare	FEQ	Single Compare Equal To
	DEQ	Double Compare Equal To
	FGT	Single Compare Greater Than
	DGT	Double Compare Greater Than
	FGE	Single Compare Greater Than Or Equal To
	DGE	Double Compare Greater Than Or Equal To
Data Format Conversion	CONVERT	Convert Data Format

Hex Utilities

A set of hex utilities are provided to create Hex files for downloading into target systems and for creating ROM images. These tools convert AMD standard COFF files into Motorola® S-Record or Tektronix® Extended Hex files. These hex utilities and a brief description of each are listed below.

■ btoa	Converts a binary file into an ASCII file.
■ coff2hex	Converts a COFF file into a hex file.
■ sim29	ASM29K software architectural simulator.
■ nm29	Prints name list of a COFF file,
■ romcoff	Generates COFF file for ROM.
■ cvcoff	Translates Am29000 microprocessor COFF files between big endian/little endian hosts.
■ strpcoff	Strips symbolic information from a COFF file.

WARRANTY and SUPPORT

Software Warranty

Software programs licensed by AMD are covered by the warranty and patent indemnity provisions appearing in AMD's standard software license forms. AMD makes no warranty, express, statutory, implied or by description, regarding the information set forth herein or regarding the freedom of the described software program from patent infringement. AMD reserves the right to modify, change or discontinue the availability of this software program at any time and without notice.

Customer Support

Maintenance

All orderable software products include one year of free Maintenance Support, which starts from the date of original purchase. Maintenance Support allows customers to receive technical assistance from highly trained field and factory personnel, to use a call-in on-line information system and to receive product and documentation updates at no additional charge. Customers may extend Maintenance Support in one-year increments. Customers can access support services by calling the 24-hour, toll-free 29K™ Family hotline at (800) 2929-AMD (292-9263).

On-Line Call-In Bulletin Board

In addition to the support engineering staff, AMD offers a 24-hour on-line technical support center. The customer can call (800) 2929-AMD at any time to query the system for the latest information on a particular product: bug fixes, work-arounds, information on upcoming releases, etc. Messages may be left for the support engineering staff during "after hours."

Training Classes

AMD offers training classes for the 29K Family products. These classes focus on 29K Family system design and implementation using the broad range of AMD software development tools. Customers can shorten the development process through extensive hands-on training covering a variety of topics. Contact your local AMD field office for more information on training classes.

Fusion29K Program

AMD encourages broad-based development and support for the Am29000 microprocessor with the Fusion29K™ program, a joint-effort program between AMD and third-party developers. Published twice a year, the Fusion29K program catalog reveals the breadth of development and system solutions for the 29K Family, including software generation and debug tools; hardware development tools; executive, kernel and multi-user operating systems; board-level products; silicon products; and more. For a copy of the Fusion29K program catalog, call your local AMD field sales office or the literature center at (800) 222-9323.

Preliminary

HighC29K

Cross-Development Toolkit, Release 2

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- **Efficient, globally optimizing C compiler technology developed by MetaWare™, Inc. ANSI Standard C support and conformance verification (ANSI document X3J11/88-159, December 7, 1988 and compile-time error checking.**
- **Compiler supports load scheduling and delayed branch optimizations to promote fast Am29000™ microprocessor code execution.**
- **Compiler supports AMD's Am29027™ Arithmetic Accelerator.**
- **Full ANSI standard run-time library of over 100 functions include all standard I/O routines (stdio).**
- **Available for the PC-AT™ and Sun-3™ development environments.**
- **Special library of high-performance transcendental functions.**
- **HighC29K™ toolkit includes the entire ASM29K™ Cross-Development Toolkit. The ASM29K package contains:**
  - Relocatable macro assembler supports complete Am29000 microprocessor instruction set.
  - Linker/loader combines separately compiled or assembled modules by resolving external references and by searching libraries.
  - Librarian provides management facility for organizing modules into logical collections of functions.
  - Full architectural simulator of the Am29000 microprocessor with user-defined memory access times. Allows designers to obtain price/performance statistics for their particular Am29000 microprocessor design.
  - IEEE software floating-point emulation functions accessible from C and assembly language modules.

GENERAL DESCRIPTION

Processor performance depends on the processor's hardware and software environment. The key to maximizing performance lies in the realization that the processor is part of a system which is a collection of components which must be properly integrated. To take advantage of the advanced RISC architecture of the Am29000 microprocessor, equally sophisticated software tools must be available to achieve this integration.

The HighC29K™ Cross-Development Toolkit offers such a development environment for creating efficient and portable software for the 29K™ Family. The package consists of the full ANSI standard, optimizing C compiler, run-time libraries, assembler, linking loader, floating-point emulation, and object module librarian. These tools allow users to design more efficient systems and applications.

Cross-development is the design of an application program on one computer (the host system) and the execution of that same application program on a different computer (the target system). The operating system on the host, such as UNIX or DOS, provides the tools needed to create the application program. These tools include editors for writing the source code, compilers and assemblers for translating the modules into executable code, and utilities for preparing the application for execution. The Am29000-based target computer generally does not provide the tools required to develop the application program. Figure 1 shows the path that an application follows from development on the host system to execution on the target system.

The HighC29K Cross-Development Toolkit transforms a PC or Sun workstation host into a powerful software development environment. The HighC29K cross-compiler generates 29K Family relocatable object modules which can be combined with other relocatable object modules derived from the assembler or HighC29K compiler using the 29K Family linker/loader. Library modules prepared by the librarian can be linked in at this point as well. The resulting absolute object module can then be downloaded to a target system.

AMD has established and published the 29K Family Common Object File Format (COFF) to which all 29K Family development tools conform. The AMD COFF format extends the already standard AT&T COFF format to support source-level debugging and other 29K Family-specific features. Similarly, AMD has estab-

Publication# 10957 Rev. B Amendment /0
Issue Date: September 1989

lished a common calling convention that maximizes performance on the 29K Family of microprocessors as well as defining standards for software vendors. This has led to a variety of compilers, assemblers, debuggers, and associated tools that may be mixed freely by developers of 29K Family software.

The contents of the HighC29K Cross-Development Toolkit include:

HighC29K:	**ASM29K (Included in HighC29K Development Package):**
Optimizing C Compiler	Relocatable Macro Assembler
Documentation	Documentation
Function Libraries	Architectural Simulator
	Linker/Loader
	Librarian
	IEEE Floating Point Emulation Routines
	Utilities

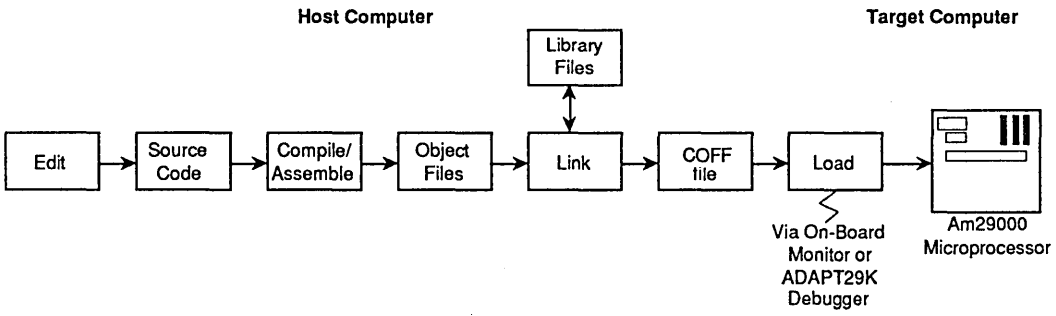


Figure 1. Cross Software Development

ORDERING INFORMATION

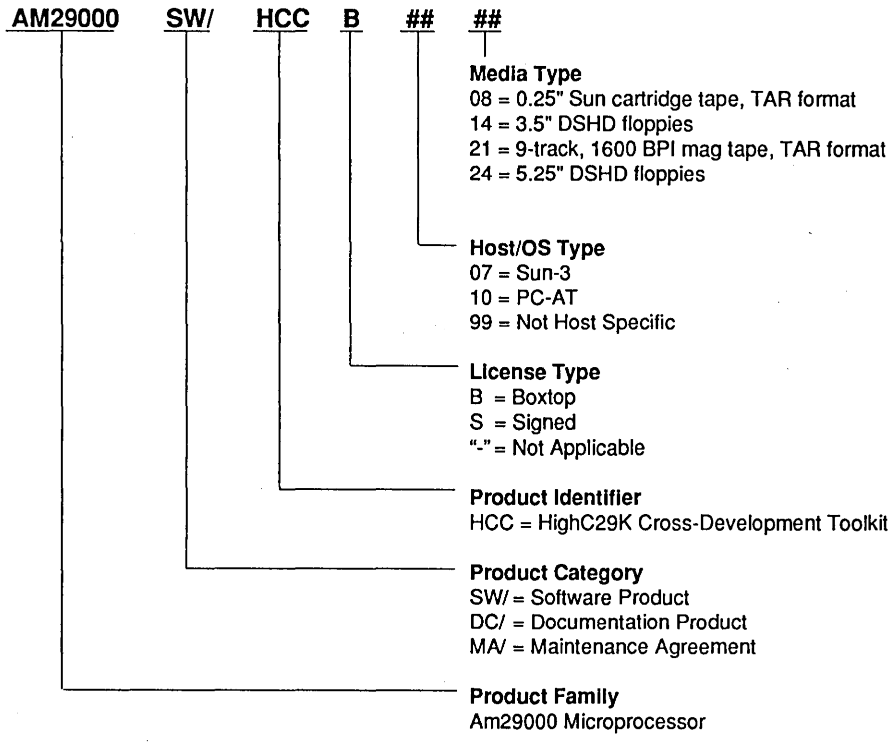
Licensing

The HighC29K Cross-Development Toolkit is licensed through AMD's Standard End-User Software License Agreement (Boxtop). This license does not require a signature; breaking the seal on the software package indicates acceptance of the license terms. If changes are required to the license agreement, they can be arranged through your AMD sales representative. Many software products require the customer to provide a CPU ID number when ordering the product. Contact your sales representative if this information is not available at time of purchase. In addition, terms of the license require the customer to complete a Software Warranty card with the serial number and site of the host computer on which the development package will reside. This card must be returned to AMD within 30 days of receipt for the warranty to be valid.

Order Numbers

The HighC29K Cross-Development Toolkit is available for several different environments. Documentation can be ordered separately. The order number (Valid Combination) is formed as a combination of:

- Product Family
- Product Category
- Product Identifier
- License Type
- Host/OS Type
- Media Type



AM29000 SW/ HCC B ## ##

Media Type
08 = 0.25" Sun cartridge tape, TAR format
14 = 3.5" DSHD floppies
21 = 9-track, 1600 BPI mag tape, TAR format
24 = 5.25" DSHD floppies

Host/OS Type
07 = Sun-3
10 = PC-AT
99 = Not Host Specific

License Type
B = Boxtop
S = Signed
"-" = Not Applicable

Product Identifier
HCC = HighC29K Cross-Development Toolkit

Product Category
SW/ = Software Product
DC/ = Documentation Product
MA/ = Maintenance Agreement

Product Family
Am29000 Microprocessor

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly released combinations.

Order Number	Product	Host	Media
AM29000SW/HCCB0708	HighC29K Toolkit	Sun-3	0.25" cartridge tape, TAR format
AM29000SW/HCCS0708	HighC29K Toolkit	Sun-3	0.25" cartridge tape, TAR format
AM29000SW/HCCB0721	HighC29K Toolkit	Sun-3	9-track, 1600 BPI tape, TAR format
AM29000SW/HCCS0721	HighC29K Toolkit	Sun-3	9-track, 1600 BPI tape, TAR format
AM29000SW/HCCB1014	HighC29K Toolkit	PC-AT	3.5" DSHD floppies
AM29000SW/HCCS1014	HighC29K Toolkit	PC-AT	3.5" DSHD floppies
AM29000SW/HCCB1024	HighC29K Toolkit	PC-AT	5.25" DSHD floppies
AM29000SW/HCCS1024	HighC29K Toolkit	PC-AT	5.25" DSHD floppies
AM29000DC/HCC-99	HighC29K Documentation	Not Host Specific	Not Media Specific
AM29000MA/HCC-07	HighC29K Maintenance	Sun-3	Not Media Specific
AM29000MA/HCC-10	HighC29K Maintenance	PC-AT	Not Media Specific

FUNCTIONAL INFORMATION

Compiler

The HighC29K cross-compiler supports an extended version of the C language designed for professional programmers. It includes a full ANSI implementation for portable applications, yet also allows user access to the best features of other languages such as nested functions from Pascal and named parameter association from Ada. Extensions to the C language also are supported, such as range notation in case statements and enumerated data types. The compiler allows users to create re-entrant procedures and to generate efficient code in terms of space and execution speed.

The HighC29K cross-compiler facilitates program development for dedicated or stand-alone Am29000 designs. The compiler generates optimized, sharable code that takes full advantage of the Am29000 instruction set. The language contains a variety of control statements, data types, and predeclared procedures and functions that promote the development of well-structured programs. For example, the user may specify the parameter types for external functions so that the compiler can check that arguments are passed correctly.

The HighC29K cross-compiler generates 29K Family object modules directly. The HighC29K compiler optionally generates information necessary for symbolic debugging at the C or assembly level with XRAY29K™, AMD's source-level debugger for the 29K Family. The compiler preprocessor allows the user to define macros, merge files into source and conditionally include or exclude code.

Optimization

As a highly optimizing cross-compiler, HighC29K software ensures the generation of fast, compact code by using advanced optimization techniques including common subexpression elimination, loop invariant analysis, global register allocation and automatic allocation of variables to registers. Many of the optimizations are particularly effective when using the unique features of the Am29000 microprocessor architecture. For example, its large register set means passing parameters in registers is more effective on the Am29000 microprocessor than on any other microprocesor. Optimizations specifically developed for the Am29000 RISC microprocessor architecture are also performed such as load scheduling for maximum instruction throughput. Additionally, the compiler makes extensive use of Am29000 microprocessor's large register file as a stack cache to store frequently accessed values. The list of optimizations performed include:

- Common subexpression elimination
- Retention/reuse of register contents
- Automatic allocation of variables to registers
- Dead code elimination and cascaded jumps
- Cross jumping (tail merging)
- Constant folding
- Switch statements optimally encoded using in-line branch table, binary search or linear search.
- Global flow analysis leading to removal of loop invariant values
- Load Scheduling
- Delayed Branch

Several of these optimizations are explained below:

Loop Invariant Analysis: Computations made inside of loops that do not change value in the loop can be moved outside the loop. The value is stored in a register for optimum access. Since an application may spend as much as 90% of its time executing loops, this optimization produces a significant gain in performance.

Fold Constants: Operands that are constant can often be folded into a single constant, or into a temporary value. If constants are defined at compile time, the compiler can reduce them to a single value.

Load Scheduling: The Am29000 microprocessor supports overlapped load and store capabilities to decrease delays incurred while waiting for data. The compiler recognizes when certain instructions can be advanced in the pipeline for efficient operation.

Delayed Branch: The Am29000 microprocessor branch instruction is delayed by one cycle to allow the processor pipeline to achieve maximum throughput. The instruction following the branch instruction, called the delayed instruction is executed whether the branch is successful or not. In most cases, the compiler can easily place a useful instruction, i.e. an instruction other than NO-OP, as the delay instruction by reorganizing the code.

Data Types

The single addressing mode of the Am29000 microprocessor combines with high-level language implementations to provide efficient access to all data types.

Data Type	Size (Bits)
int	32
long int	32
pointer	32
short int	16
char	8
float	32
double	64
unsigned	32
unsigned char	8
unsigned short	16
enum (default)	32
enum (option)	8,16,32

Am29027 Arithmetic Accelerator Support

Target systems that include the Am29027 Arithmetic Accelerator for high-speed computations are directly supported through the compiler. Users may direct the compiler to generate in-line code to access the control and instruction registers of the accelerator. Versions of the libraries that assume direct use of the Am29027 microprocessor are included.

Alternatively, the user can signal the compiler to generate Am29000 microprocessor floating-point instructions that are used in conjunction with the IEEE Floating-Point Emulation Routines to access the accelerator.

The HighC29K Cross-Development Toolkit includes AMD's entire ASM29K Cross-Development Toolkit. Details of this package are contained in the ASM29K Cross-Development Toolkit data sheet (order #10292).

Function Libraries

The HighC29K toolkit includes three different sets of function libraries that enhance the functionality of the compiler. The library sets are comprised of:

- the ANSI standard library which provides the full set of functions specified by the ANSI C language standard
- a library of routines implementing the floating-point environment functions specified in the IEEE-754 standard
- a library of hand-coded transcendental functions optimized for use with the Am29000/Am29027 microprocessor combination.

Each library set contains several versions of the library which reflect the different possible target environments. The compiler driver is able to select the proper version of the library to use based on the compile-time options specified.

ANSI Standard Library

This library contains the full functionality specified by the ANSI standard for the C language (X3J11/88-159, December, 1988). At the lowest level, the library functions interface with HIF (Host Interface), a small kernel system defined by AMD. HIF is supported in all AMD products, and is defined in the HighC29K toolkit manual for the customer who needs to adapt to a different environment.

The functions included in the ANSI Standard Library are:

Mathematical Routines

abs	atan2	exp	frexp	modf	sqrt	acos
ceil	fabs	ldexp	pow	tan	asin	cos
floor	log	sin	tanh	atan	cosh	fmod
log10	sinh					

Memory Allocation

calloc	free	malloc	realloc

Standard Formated I/O

fprintf	printf	sprintf	vfprintf	vsprint	fscanf	scanf
sscanf	vprintf	_setmode				

Standard File I/O

fclose	fopen	remove	setbuf	tmpfile
fflush	freopen	rename	setvbuf	tmpnam

Character Routines

isalnum	iscntrl	isgraph	isprint	isspace
isxdigit	toupper	isalpha	isdigit	islower
ispunct	isupper	tolower		

Character I/O Routines

fgetc	fputc	getc	gets	putchar
ungetc	fgets	fputs	getchar	putc puts

String Routines

memchr	strcat	strcspn	strncpy	strtok
_strncat	memcmp	strchr	strerror	strpbrk
strxfrm	memcpy	strcmp	strlen	strrchr
_rmemcpy	memove	strcoll	strncat	strspn
_rstrcpy	memset	strcpy	strncmp	strstr
_strcats				

Direct I/O Routines

fgetpos	fread	fseek	fsetpos	ftell	fwrite
rewind					

General Routines

abort	atol	getenv	mbstowcs	rand
strtoul	atexit	bsearch	labs	mbtowc
srand	system	atoi	div	ldiv on-
exit	strtod	wctombs	atof	exit
mblen	qsort	strtol	wctomb	

Date and Time Routines

asctime	ctime	gmtime	localtime	mktime
strftime	time	clock	difftime	

Miscellaneous Routines

assert	ferror	localeconv	perror	setjmp
signal	va_end	clearerr	kill	longjmp
raise	setlocale	va_arg	va_start	feof

Floating-Point Environment Library

The functions included in the Floating-Point Environment Library are:

class	rclass	copysign	rcopysign	finite
rfinite	isnan	risnan	logb	rlogb
nextafter	rnextafter	remainder	rremainder	scalb
rscalb	unordered	runordered		

Fast Transcendental Library

This library provides special hand-coded versions of the standard transcendental functions. These functions are optimized for performance with the Am29000/Am29027 microprocessor combination.

The functions included are:

atan	cos	exp	log	pow
sin	sqrt	tan		

Floating-Point Emulation

The Am29000 microprocessor's instruction set includes floating-point and integer math operations. In the simplest processor implementation, these instructions cause traps to routines that perform the operations. The user is provided with source to two complete sets of routines that emulate IEEE Floating-Point Standard 754 for each of the instructions listed below.

The first set of trap handlers is provided for users who have integrated the Am29027 arithmetic accelerator into their systems. The Am29000 microprocessor math instructions are performed using the Am29027 microprocessor.

The second set of trap handlers implements emulation of the floating-point operations entirely in software. No special hardware is required.

Documentation instructs users how to integrate the package into their target system. Both packages are designed to insure upward compatibility with future generation processors. The floating-point routines are accessible from both the assembler and compiler.

To eliminate the overhead incurred by using the trap handlers, direct code generation (in-line coding) of Am29027 microprocessor floating-point operations is an included option of the HighC29K Cross-Development Toolkit.

Am29000 Microprocessor Floating-Point Instructions

Mnemonic	Operation
CONVERT	Convert values between types Integer, Float, and Double
FEQ	Compare Floats Equal
DEQ	Compare Doubles Equal
FGT	Compare Floats Greater Than
DGT	Compare Double Greater Than
FGE	Compare Floats Less Than
DGE	Compare Double Less Than
FADD	Float Add
DADD	Double Add
FSUB	Float Subtract
DSUB	Double Subtract
FMUL	Float Multiply
DMUL	Double Multiply
FDIV	Float Divide
DDIV	Double Divide

Utilities

A set of utilities is provided to work with the output files produced by the development tools. They allow the user to prepare output files for downloading into target systems and to create ROM images. The utilities include:

- coff2hex: Converts Am29000 microprocessor COFF files to Motorola® S-record or Extended Tektronix® Hex Files.
- romcoff: Allows creation of ROM images from Am29000 microprocessor COFF files.
- cvcoff: Translates Am29000 microprocessor COFF files between big endian/little endian hosts.
- strpcoff: "Strips" symbolic information from an executable COFF file.

MAINTENANCE AND SUPPORT

Software Warranty

Software programs licensed by AMD are covered by the warranty and patent indemnity provisions appearing in AMD's standard Software License Forms. AMD makes no warranty, express, statutory, implied or by description regarding the information set forth herein or regarding the freedom of the described software program from patent infringement. AMD reserves the right to modify, change or discontinue the availability of this software program at any time and without notice.

Support

Customer Support

All orderable software products include one year of free maintenance support, which starts from the date of original purchase. Maintenance support allows customers to receive technical assistance from highly trained field and factory personnel, to use a call-in on-line information system and to receive product and documentation updates at no additional charge. Customers may extend maintenance support in one-year increments. Customers can access suppport services by calling the 24-hour, toll-free 29K Family hotline at (800) 2929-AMD (292-9263).

On-Line Call-In Bulletin Board

In addition to the support engineering staff, AMD offers a 24-hour on-line technical support center. The customer can call (800) 2929-AMD at any time to query the system for the latest information on a particular product: bug fixes, work-arounds and information on up-coming releases. Messages may be left for the support engineering staff during "after hours."

Training Classes

AMD offers training classes for the 29K Family products. These classes focus on 29K Family system design and implementation using the broad range of AMD software development tools. Customers can shorten the development process through extensive hands-on training covering a variety of topics. Contact your local AMD field sales office for more information on training classes.

Fusion29K Program

AMD encourages broad-based development and support for the Am29000 with the Fusion29K™ program, a joint-effort program between AMD and third-party developers. A bi-annual Fusion29K program catalog reveals the breadth of development and system solutions for the 29K Family, including software generation and debug tools; hardware development tools; executive, kernel and multi-user operating systems; board-level products; silicon products; and more. For a copy of the Fusion29K program catalog, call your local AMD field sales office or the literature center at (800) 222-9323.

Preliminary

MON29K

Target Resident Debug Monitor

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- Provides local control of an Am29000™ microprocessor-based system
- Interfaces to the XRAY29K™ Source-Level Debugger
- Allows modification and display of memory, registers and I/O ports
- Supports modification and display of special-purpose registers by group
- Allows access to both user- and system-level code
- Supports the AMD Am29027™ Arithmetic Accelerator
- Allows modification and display of Am29027 microprocessor registers
- Provides eight breakpoints plus single- and multiple-instruction stepping
- Allows selection of user-defined displays after each breakpoint or single step
- Provides in-line assembler and disassembler
- Supports downloading of COFF and hex files from remote systems
- Provided in source form (C and Am29000 microprocessor assembly) to simplify installation of I/O devices
- Offers familiar user interface, similar to DEBUG on IBM® PC

GENERAL DESCRIPTION

The Target Resident Debug Monitor (MON29K™) resides on Am29000 microprocessor-based hardware. It provides all the control a designer needs to load, execute and debug Am29000 microprocessor programs. MON29K software is provided in source form so its I/O drivers and service routines can be modified easily, which allows MON29K software to be customized for various hardware configurations.

MON29K software provides the ability to set breakpoints, to set and display memory and registers, to read and write I/O ports, to trace execution in single or multiple steps, and to download files from a remote host. MON29K software is controlled by either an ASCII terminal or a host computer connected to a serial port on the target system.

MON29K software supports high-level language debugging through XRAY29K, the Am29000 microprocessor source-level debugger. In addition to its own standard command set, the XRAY29K debugger supports all the MON29K software commands.

The MON29K product includes:

- MON29K source code
- Documentation

Publication #	Rev.	Amendment
10287	B	/0

Issue Date: September 1989

ORDERING INFORMATION

Licensing

The MON29K Resident Monitor is licensed through AMD's Standard End-User Software License Agreement (Boxtop). This license does not require a signature; breaking the seal on the product package indicates acceptance of the license terms. If changes are required to the license agreement, they can be arranged through your AMD sales representative. Many software products require the customer to provide a CPU ID number when ordering the product. Contact your sales representative if this information is not available at the time of purchase. In addition, terms of the license require the customer to complete a Software Warranty card with the serial number and site of the host computer on which the resident monitor source will reside. This card must be returned to AMD within 30 days of receipt for the warranty to be valid.

Order Numbers

MON29K software executes on Am29000 microprocessor-based systems but is distributed in machine readable source form for several hosts. Thus, media type is the only distinguishing characteristic when ordering MON29K software. Documentation can be ordered separately. The order number (Valid Combination) is formed as a combination of:

- Product Family
- Product Category
- Product Identifier
- License Type
- Host/OS Type
- Media Type

AM29000 SW/ MON B ## ##

Media Type
08 = 0.25" cartridge tape. TAR format
14 = 3.5" DSHD floppies
21 = 9-track, 1600 BPI mag tape, TAR format
24 = 5.25" DSHD floppies

Host/OS Type
99 = Not Host Specific

License Type
B = Boxtop
S = Signed
"-" = Not Applicable

Product Identifier
MON = MON29K Target Resident Debug Monitor

Product Category
SW/ = Software Product
DC/ = Documentation Product
MA/ = Maintenance Agreement

Product Family
Am29000 Microprocessor

Valid Combinations

Valid Combinations lists configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly released combinations.

Part Number	Product	Host	Media
AM29000SW/MONB9908	MON29K Resident Monitor	Not Host Specific	0.25" cartridge tape, TAR format
AM29000SW/MONS9908	MON29K Resident Monitor	Not Host Specific	0.25" cartridge tape, TAR format
AM29000SW/MONB9914	MON29K Resident Monitor	Not Host Specific	3.5" DSHD floppies
AM29000SW/MONS9914	MON29K Resident Monitor	Not Host Specific	3.5" DSHD floppies
AM29000SW/MONB9921	MON29K Resident Monitor	Not Host Specific	9-track, 1600 BPI tape, TAR format
AM29000SW/MONS9921	MON29K Resident Monitor	Not Host Specific	9-track, 1600 BPI tape, TAR format
AM29000SW/MONB9924	MON29K Resident Monitor	Not Host Specific	5.25" DSHD floppies
AM29000SW/MONS9924	MON29K Resident Monitor	Not Host Specific	5.25" DSHD floppies
AM29000DC/MON-99	MON29K Documentation	UNIX	Not Media Specific
AM29000MA/MON-99	MON29K Maintenance	Not Host Specific	Not Media Specific

FUNCTIONAL DESCRIPTION

MON29K software resides on the target system and interfaces to the user through an ASCII terminal connected to a serial port on the target system. All commands and formatted displays are communicated through this serial link. MON29K software supports simple display formats so that compatibility can be maintained with any CRT.

MON29K software provides program development support at the assembler source level. High-level source code development is provided by the XRAY29K debugger when it is connected to MON29K monitor. MON29K serves as the target resident monitor that interrogates memory and registers for the host-resident source-level debugger.

Memory, Register and I/O Addresses

MON29K software supports three address spaces: register, memory, and I/O. Data values are always represented in hex, as are memory and I/O addresses. Register addresses are represented by decimal numbers and grouped as general, local, global, special-purpose, and TLB. Special-purpose and TLB registers can be accessed by register number or by their abbreviated mnemonic. The Special-Purpose Registers section that follows discusses other commands for accessing these registers.

Memory and I/O addresses are assumed to be real because MON29K software has no mechanism for calculating or interpreting virtual addresses. MON29K software allows specification of user and supervisor modes and specification of OPT lines with all memory and I/O addresses.

Displaying Memory and Registers

The *Display* command shows data for a specified range of addresses, beginning at a specified address or from the currently active address. Each line in the display contains 16 bytes of data. The 16 bytes are displayed as either bytes, half-words, words, single-precision, or double-precision floating points, depending on the command entered.

Floating-point numbers are displayed in decimal format if the value can be represented accurately within the digits available. Otherwise, scientific notation, E format, is used.

Following the numeric data is a string of ASCII characters in which each character corresponds to one byte of data. When no ASCII equivalent exists for the byte of data, a period is displayed. Figure 1 shows examples of memory and register displays.

Altering Memory and Registers

Memory and register contents can be set, filled, or moved. The set command allows the contents of registers and memory to be examined and optionally changed. One or more values can be set without examining the previous contents. The fill command sets a range of register or memory addresses to a specific value. The move command copies blocks of data from one range of addresses to another. Blocks in the destination address range may overlap blocks in the source address range.

Special-Purpose Registers

The special-purpose register commands provide another method for accessing the Am29000 microprocessor special-purpose and TLB registers. These registers are organized into groups: Unprotected, Protected, TLB Entries, and Coprocessor. Specific commands are used for examining the contents of registers in each group. Within a group, each register's contents can be examined or changed explicitly.

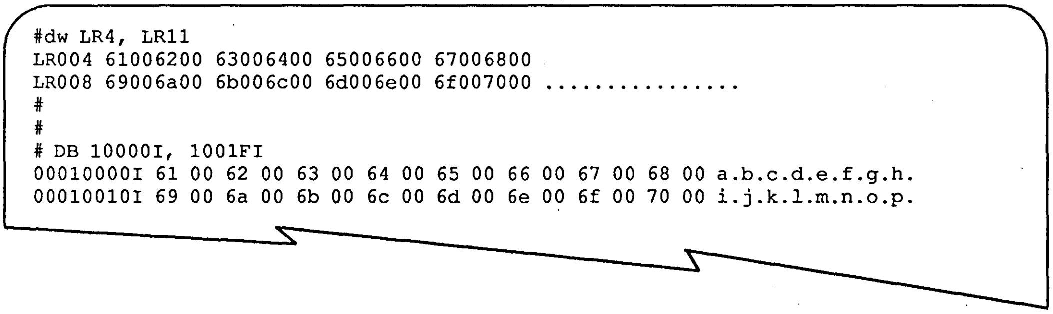
The large number of registers necessitates special register display screens that clearly present each group's registers. To enhance display efficiency, the single command X is available. It displays the registers most likely to be in use: all the global registers, half the local registers, and all the unprotected registers. Figures 2 and 3 show examples of special-purpose register display screens.

In-Line Assembler/Disassembler

An in-line assembler/disassembler allows the user to examine and change memory using instruction mnemonics rather than hex values. This improves readability and minimizes user efforts while entering changes to instruction memory. The lexical conventions and statement syntax used are identical to the standard AMD assembler, ASM29K™.

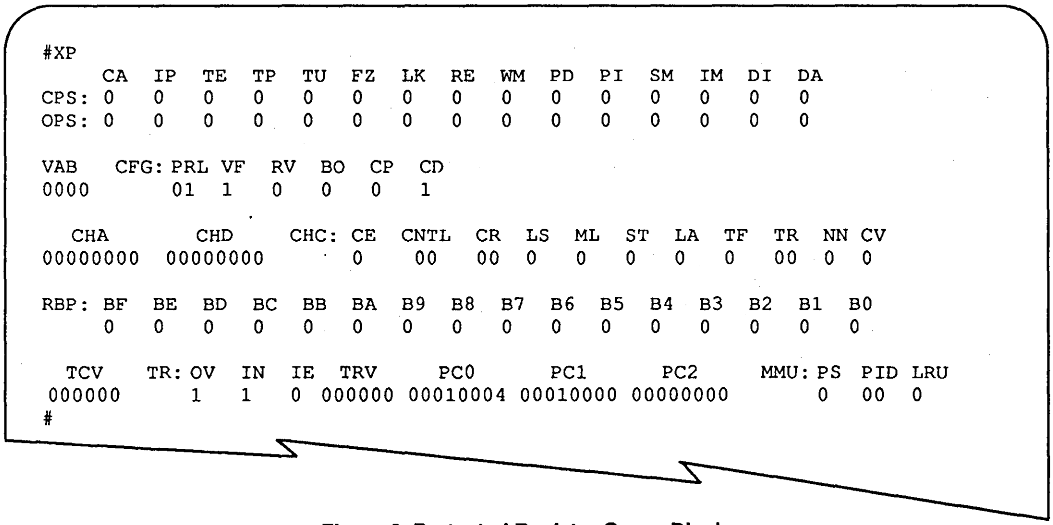
I/O Commands

I/O commands provide simple forms of input and output. They are intended to allow quick examination and simple control of devices. These commands read or write a full word of data to or from a real I/O address.



```
#dw LR4, LR11
LR004 61006200 63006400 65006600 67006800
LR008 69006a00 6b006c00 6d006e00 6f007000 ................
#
#
# DB 10000I, 1001FI
00010000I 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00 a.b.c.d.e.f.g.h.
00010010I 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00 i.j.k.l.m.n.o.p.
```

Figure 1. Register and Memory Display



```
#XP
      CA  IP  TE  TP  TU  FZ  LK  RE  WM  PD  PI  SM  IM  DI  DA
CPS:  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
OPS:  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

VAB   CFG: PRL VF  RV  BO  CP  CD
0000       01  1   0   0   0   1

  CHA       CHD       CHC: CE  CNTL  CR  LS  ML  ST  LA  TF  TR  NN CV
00000000  00000000          0   00   00  0   0   0   0   0   00  0  0

RBP:  BF  BE  BD  BC  BB  BA  B9  B8  B7  B6  B5  B4  B3  B2  B1  B0
      0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

  TCV     TR: OV  IN  IE  TRV     PC0      PC1      PC2      MMU: PS  PID LRU
 000000       1   1   0  000000 00010004 00010000 00000000        0   00  0
#
```

Figure 2. Protected Register Group Display

Downloading

Downloading controls the transmission of data from a remote system to the local memory on the target system. MON29K software can read COFF binary, Motorola S3 hex records, and TEK extended hex files. Each of these formats contains the address and byte count in-formation for loading memory, so no other parameters need to be specified.

An optional downloading parameter, *<host command>*, can be specified by the user. The *<host command>* is a character string that is uploaded by MON29K to the remote host system. This command can be used to initiate the host download procedure remotely from the MON29K monitor terminal.

Execution Control

Execution control commands allow the user to start program execution, setup through instruction singly or in groups, breakpoint execution, and specify monitor commands to be performed when termination occurs. Following each break in program execution, the MON29K monitor displays the address and disassembled contents of the next executable instruction. In addition, the user can identify registers and memory he wishes to view after the termination of each breakpoint or step command. This reduces the amount of information displayed to the data that is pertinent to the current debugging session.

MON29K software provides eight "sticky" and two "non-sticky" breakpoints. Sticky breakpoints remain set until expressly removed by the user. These are useful when debugging code within an instruction loop. Non-sticky breakpoints occur once and are removed automatically. Non-sticky breakpoints are optional parameters of the go command. Users can easily display, set, and reset breakpoint addresses.

Program execution can be stepped one instruction at a time or a group of instructions at a time. User-defined displays and the address and contents of the next executable instruction are displayed after each instruction step. When stepping by group, these displays can be delayed either until after the last instruction in the group is executed, or until after each instruction is executed. An option allows only register data that was changed to be displayed. This automatically informs the user of register changes, thus eliminating the need to visually monitor register contents.

Remote Mode

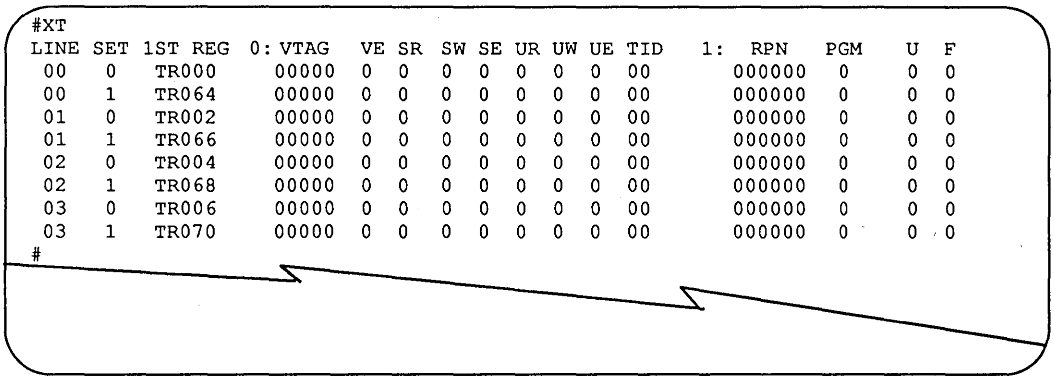
MON29K software supports two serial ports: one to a terminal and one to a host computer. In normal mode, either port can be used for initiating commands or for downloading programs. In remote mode, the two serial ports are linked together, allowing the terminal to communicate directly with the host computer.

Miscellaneous Commands

An on-screen help facility, as seen in Figure 4, lists all MON29K monitor commands. Information about a specific command is obtained by specifying the command name as a parameter to the help command.

Am29027 Arithmetic Accelerator Support

MON29K software is fully integrated with the AMD Am29027 Arithmetic Accelerator. In the same manner that the Am29000 microprocessor registers can be accessed, the Am29027 microprocessor registers can be both displayed and modified using MON29K software. An example of an Am29027 microprocessor register display is shown in Figure 5.



```
#XT
LINE SET 1ST REG  0: VTAG  VE SR  SW SE UR UW UE TID   1:  RPN    PGM    U  F
 00   0   TR000     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 00   1   TR064     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 01   0   TR002     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 01   1   TR066     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 02   0   TR004     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 02   1   TR068     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 03   0   TR006     00000  0  0   0  0  0  0  0  00      000000  0      0  0
 03   1   TR070     00000  0  0   0  0  0  0  0  00      000000  0      0  0
#
```

Figure 3. TLB Entries Group Display

Target System Requirements

The Am29000 microprocessor supports separate code and data spaces and provides no instructions for moving information between data and instruction spaces. Because of this, the target system must provide a mechanism for writing to code space in order for MON29K monitor to set breakpoints and load instruction memory.

MON29K software is designed to support a memory, mapped Z8530 SCC serial device. However, source code is provided so the user can change the MON29K monitor to support other devices on a particular target system.

Other Tools

MON29K is a stand-alone product that does not depend on other software to function. However, MON29K software is delivered in source form and will need to be compiled with the AMD HighC29K™ Cross-Development Toolkit; modification may be necessary if compiled with other Am29000 microprocessor C compilers.

```
# H
Help:                                  XP- Display/set protected reg
H or ?    to see this display          XT- Display/set TLB entries
H<name>   help with a named command    XC- Display/set Am29027 reg
?<name>   help with a named command    X - Display key registers
                                       Y - Load a file to memory
Target Resource Access:                V - Save memory to a file
D - Display registers/memory
S - Set registers/memory               Execution Control:
F - Fill registers/memory              E - End execution command list
M - Move registers/memory              B - Display/Set/Clear breaks
A - Assemble in memory                 G - Go (start execution)
L - List disassembly from mem          T - Trace (single/multiple step)
I - Input from port                    Miscellaneous:
O - Output to port                     R - Remote mode (talk to host)
XU- Display/set unprotected reg        N - Normal (change 'normal' char)
                                       Q - Re-initialize monitor
```

Figure 4. On-Screen Help Facility

```
#XC
         PR    MSW       LSW                      PR       MSW        LSW
RF0:     0   00000000  00000000       RF1:        0      00000000   00000000
RF2:     0   00000000  00000000       RF3:        0      00000000   00000000
RF4:     0   00000000  00000000       RF5:        0      00000000   00000000
RF6:     0   00000000  00000000       RF7:        0      00000000   00000000

R:           00000000  00000000       S:                 00000000   00000000
R TEMP:  00000000      00000000       S TEMP:            00000000   00000000

F:           00000000  00000000

         IP  RP  RF  RFS   PMS   QMS   TMS   SIP   SIQ   SIT   SIF  IF   CO
INSTR:   0   0   0   0     0     0     0     0     0     0     0    0    00
I TEMP:  0   0   0   0     0     0     0     0     0     0     0    0    00

STATUS:  OP IV SV RV ES ZE XE UE VE RE IE   FLAGS:FL6 FL5  FL4  FL3  FL2  FL1  FL0
         0  0  0  0  0  0  0  0  0  0  0         0    1    0    0    0    0    0

OP HE AD  MVTC MATC PLTC  ZM XM UM VM RM IM PL RMS MF MS BU BS SU TR AP SA AFF PFF
0  0  0    0    0    0    0  0  0  0  0  0  0   0  0  0  0  0  0  0  0  0   0   0
```

Figure 5. Am29027 Register Display

MAINTENANCE AND SUPPORT

Software Warranty

Software programs licensed by AMD are covered by the warranty and patent indemnity provisions appearing in AMD's standard software license forms. AMD makes no warranty, express, statutory, implied, or by description regarding, the information set forth herein or regarding the freedom of the described software program from patent infringement. AMD reserves the right to modify, change, or discontinue the availability of this software program at any time and without notice.

Customer Support Maintenance

All orderable software products include one year of free Maintenance Support, which starts from the date of original purchase. Maintenance Support allows customers to receive technical assistance from highly trained field and factory personnel, to use a call-in on-line information system, and to receive product and documentation updates at no additional charge. Customers may extend Maintenance Support in one-year increments. Customers can access support services by calling the 24-hour, toll-free 29K™ Family hotline at (800) 2929-AMD (292-9263).

On-Line Call-In Bulletin Board

In addition to the support engineering staff, AMD offers a 24-hour on-line technical support center. The customer can call (800) 2929-AMD at any time to query the system for the latest information on a particular product: bug fixes, work-arounds, information on up-coming releases, etc. Messages may be left for the support engineering staff during "after hours."

Training Classes

AMD offers training classes for the 29K Family products. These classes focus on 29K Family system design and implementation using the broad range of AMD software development tools. Customers can shorten the development process through extensive hands-on training covering a variety of topics. Contact your local AMD field office for more information on training classes.

Fusion29K Program

AMD encourages broad-based development and support for the Am29000 microprocessor with the Fusion29K™ program, a joint-effort program between AMD and third-party developers. Published twice a year, the Fusion29K program catalog reveals the breadth of development and system solutions for the 29K Family, including software generation and debug tools; hardware development tools; executive, kernel, and multi-user operating systems; board-level products; silicon products; and more. For a copy of the Fusion29K program catalog, call your local AMD field sales office or the literature center at (800) 222-9323.

Preliminary

XRAY29K

Source-Level Debugger

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- **Supports symbolic debugging with C expressions and statements for Am29000™ microprocessor development environments**
- **Controls and examines program execution in high-level and assembly-level modes**
- **Provides interface and start-up code for the Am29000 microprocessor, which allows use of the MON29K™ Target-Resident Monitor, ADAPT29K™ Advanced Development and Protoyping Tool and PCEB29K™ PC Execution Board**
- **Uses window-oriented display to segregate debug information in meaningful regions**
- **Allows single-step execution and placement of simple and complex breakpoints**
- **Supports custom screens and viewports, and one-key command functions**
- **Provides command, breakpoint, and viewport macros**
- **Supports automatic test sequences by processing command files and logging output to a file**
- **Includes on-line help, comprehensive documentation, and a sample debug session**

GENERAL DESCRIPTION

AMD's XRAY29K™ source-level debugger provides engineers with a multiwindow interactive environment for debugging high-level and assembly-level software programs for Am29000-based systems. XRAY29K software resides on IBM® ATs® and compatibles, and Sun Workstations®. Program execution is monitored and controlled in high-level source or assembly language, from the host system through the PCEB29K execution board, MON29K monitor or ADAPT29K debugger on the target system. Control is extensive, including debugger commands for setting breakpoints, single stepping through the program, and examining or altering register and memory contents.

XRAY29K software allows examination and modification of a variable's contents and computation of high-level and assembly language expression values. Symbols can be added, displayed, and deleted in the symbol table.

The XRAY29K product includes:

- XRAY29K Software
- Documentation
- Install testing program
- Start-up code for ADAPT29K or targets using MON29K

Publication # 10626 Rev. C Amendment /0
Issue Date: September 1989

ORDERING INFORMATION

Licensing

The XRAY29K Source-Level Debugger is licensed through AMD's Standard End-User Software License Agreement (Boxtop). This license does not require a signature; breaking the seal on the product package indicates acceptance of the license terms. If changes are required to the license agreement, they can be arranged through your AMD sales representative. Many software products require the customer to provide a CPU ID number when ordering the product. Contact your sales representative if this information is not available at the time of purchase.

Order Numbers

The XRAY29K Source-Level Debugger is available for several different environments. Documentation can be ordered separately. The order number (Valid Combination) is formed as a combination of:

- Product Family
- Product Category
- Product Identifier
- License Type
- Host / OS Type
- Media Type

AM29000 SW/ XRY B ## ##

Media Type
08 = 0.25" Sun cartridge tape, TAR format
14 = 3.5" DSHD floppies
21 = 9-track, 1600 BPI mag tape, TAR format
24 = 5.25" DSHD floppies

Host / OS Type
07 = Sun-3
10 = PC-AT

B = Boxtop
S = Signed
"-" = Not Applicable

Product Identifier
XRY= XRAY29K Source-Level Debugger

Product Category
SW/ = Software Product
DC/ = Documentation Product
MA/ = Maintenance Agreement

Product Family
Am29000 Microprocessor

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly released combinations.

Order Number	Product	Host	Media
AM29000SW/XRYB0708	XRAY29K Source-Level Debugger	Sun-3	0.25" cartridge tape, TAR format
AM29000SW/XRYS0708	XRAY29K Source-Level Debugger	Sun-3	0.25" cartridge tape, TAR format
AM29000SW/XRYB0721	XRAY29K Source-Level Debugger	Sun-3	9-track, 1600 BPI tape,TAR format
AM29000SW/XRYS0721	XRAY29K Source-Level Debugger	Sun-3	9-track, 1600 BPI tape,TAR format
AM29000SW/XRYB1014	XRAY29K Source-Level Debugger	PC-AT	3.5" DSHD floppies
AM29000SW/XRYS1014	XRAY29K Source-Level Debugger	PC-AT	3.5" DSHD floppies
AM29000SW/XRYB1024	XRAY29K Source-Level Debugger	PC-AT	5.25" DSHD floppies
AM29000SW/XRYS1024	XRAY29K Source-Level Debugger	PC-AT	5.25" DSHD floppies
AM29000DC/XRY-99	XRAY29K Documentation	UNIX	Not Media Specific
AM29000MA/XRY-07	XRAY29K Maintenance	Sun-3	Not Media Specific
AM29000MA/XRY-10	XRAY29K Maintenance	PC-AT	Not Media Specific

FUNCTIONAL DESCRIPTION

XRAY29K software aids the control and examination of program execution, and can set and examine memory and register contents, set and remove breakpoints in either high-level source or assembly language code, and display and alter the microprocessor state. In addition to symbolic debugging, the XRAY29K debugger's special features include help screens, macro capabilities, command files, conditional commands, and debugging through ports. For example, in batch mode, command files can issue directives to XRAY29K software to implement automated test sequences.

XRAY29K software functions in either high-level or assembly-level mode. In high-level mode, an application is debugged using C language source lines to control and monitor execution. C variables and expressions replace numeric addresses for memory access. Code can be viewed by line number or procedure name. In assembly-level mode, an application is debugged using assembly language statements. In addition to all the capabilities available in high-level mode, assembly-level mode includes machine-level register and status bit manipulation. For each mode, the monitor's screen is partitioned in areas called viewports, where information is displayed in meaningful regions and is easy to identify.

Viewport Commands

When the XRAY29K debugger executes, the screen is divided in areas called viewports. The number of viewports and the information shown in each depends on whether the object module was written in a high-level language (high-level mode) or assembly language (assembly-level mode).

The standard screen for high-level mode has four viewports: data, trace, code, and command. This screen is displayed when an object module generated by a high-level source program is executed. The standard screen for assembly-level mode has five viewports: data, stack, disassembled code, Am29000 microprocessor registers, and command. This screen is displayed when an object module generated by an assembly language program is executed. Figures 1 and 2 show examples of these screens.

Viewport commands control the way information is displayed on the screen. Changing a viewport's size, color, and cursor position as well as adding and deleting a custom viewport are viewport commands. In addition, viewports can be cleared of data, and macros can be associated with them. Frequently used viewport commands are associated with function keys for easy access.

vactive	Activate a viewport
vclear	Clear data from a viewport
vclose	Remove a user-defined viewport or screen
vmacro	Attach a macro to a viewport
vopen	Create a screen/create or resize a viewport
vscreen	Activate a screen
vsetc	Set a viewport's cursor position
zoom	Increase or decrease viewport size

Macro Commands

XRAY29K software supports macros to create and execute complex command procedures, such as testing program variables, and to conditionally execute other sets of commands. Macros can be defined and used any time during a debugging session and can include comments to explain its function. The macro definition may contain parameters that can be changed for each macro call.

Used as commands or in expressions, macros can be attached to a breakpoint to create complex breakpoint condition testing, or to a custom viewport to control data display. Complex initialization conditions can be represented as a sequence of macro commands in a command file. Statements to increment variables, perform loops and conditions, and control target program flow can be part of a macro.

XRAY29K software provides a set of macro flow control statements. These statements are similar to C conditional statements (e.g., IF, ELSE, WHILE, DO, FOR, RETURN and CONTINUE). To create a macro, the *define* command is used. After macro creation, the *show* command allows the macro's source to be viewed. Commands to attach a macro to a viewport are part of the viewport command set. Commands that attach a macro to a breakpoint are part of the execution and breakpoint command set.

define	Create a macro
show	Display a macro source

Debugger Commands

Commands, whether in high-level source or assembly language mode, can be entered interactively from the keyboard in the command viewport or placed in a command file and accessed as include or batch files. Some commands take qualifiers that provide additional information on how to execute the command and parameters that describe an object and communicate addresses or file specifications.

Breakpoints and Execution Commands

A breakpoint causes program execution to halt or causes the XRAY29K debugger to take some action, such as incrementing a counter each time the target program attempts to execute an instruction at a specified memory location. A macro can be associated with the breakpoint to control execution. A special breakpoint viewport shows breakpoint information during the debugging session, including the breakpoint identification number. Automatically assigned by XRAY29K software, the breakpoint number can reference or clear a breakpoint.

Execution commands start program execution or re-sume execution after explicit suspension. The program can be instructed to continue, single step, or set temporary instruction breakpoints. Single stepping is performed by C source line in high-level mode and microprocessor instruction in assembly-level mode. In addition, for each step, a macro can be invoked.

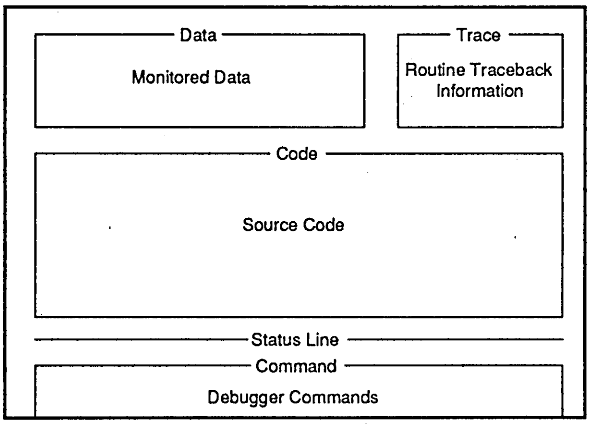


Figure 1. Standard High-Level Screen

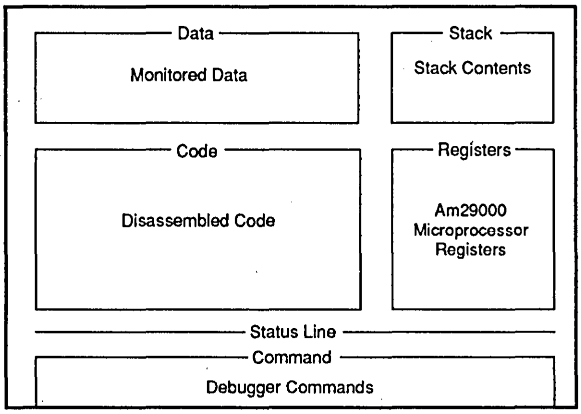


Figure 2. Standard Assembly-Level Screen

breakinstruction	Set an instruction breakpoint
clear	Clear a breakpoint
go	Start or continue program execution
gostep	Execute a macro after each instruction step
step	Execute a number of instructions or lines
stepover	Single step, but execute through procedures

Display Commands

Display commands write program information to a viewport or file about memory, expressions, or procedures. C source code, for example, can be listed starting at a particular line number or for a named procedure. Any active procedure—a procedure on the stack—can have its values displayed.

Memory contents can be dumped in both hexadecimal and ASCII text format, and, when in assembly-level mode, memory can be disassembled and displayed in the code viewport. Variables can be monitored and examined in the data viewport as the target program executes. An expression or expression range can be displayed in the command viewport according to type.

For type conversions, scaling, and output positioning, display commands can open a file or device and then write formatted output to it. Several format options are provided, similar in function to those provided to C in standard runtime libraries.

disassemble	Display disassembled memory (assembly mode)
dump	Display memory contents
expand	Display a procedure's local variables
find	Search for a string
fopen	Open a file or device for writing
fprintf	Print formatted output to a viewport
list	Display C source code
monitor	Monitor expressions
next	Find a string's next occurrence
nomonitor	Discontinue monitoring an expression
printf	Print formatted output to command viewport
printvalue	Print a variable's value

Memory and Register Commands

To help track down problems and test fixes, memory and registers can be examined and altered. Two blocks of memory, for example, can be compared for similarities or differences to check for a corrupt RAM image. Memory and registers can be modified temporarily to patch programs and continue testing during a debugging session. Expression evaluation is supported during searching and modification.

compare	Compare two blocks of memory
copy	Copy a memory block
fill	Fill a memory block with values
nomen	Prevent access to a memory location
search	Search a memory block for a value
setmem	Change a memory address
setreg	Change a register's contents
test	Examine memory area for invalid values

Symbol Commands

A symbol is a sequence of characters used to represent arithmetic values, memory addresses, and C variables. XRAY29K software knows about two types of symbols: program and debugger. Program symbols are symbolic data names or program labels that were defined during the source program's creation. Debugger symbols manipulate and direct the flow of the debugger and are specified by the user during a debugging session.

Symbol commands encompass both types of symbols. Debugger symbols can be added to the debugger symbol table, and then displayed or removed. Information about program symbols, such as name, data type, storage class, and memory location, can be displayed.

add	Create a symbol
context	Show the current context
delete	Delete a symbol from the symbol table
printsysbols	Display symbol information
scope	Specify current module and procedure scope

Utility Commands

Command files are commonly used to read macro definitions from a file or to change viewports. After a command file has been created, it may be included in a startup file and executed as if entered at the keyboard. When an include file error is encountered, XRAY29K software can be directed to quit, abort, or continue. A log of commands entered at the keyboard can be retained and then subsequently used as a command file. If XRAY29K software display and execution defaults are changed, they can be saved in a new startup file. All these operations are accessed through utility commands.

Other utility commands control the microprocessor's state. *Reset* simulates a microprocessor reset. *Restart* restores the microprocessor to its initial state without initializing memory or restarting the program, and it sets the program counter to the original starting address from the absolute file but maintains breakpoint declarations. In addition, the user can temporarily change the default values for debugger startup options, such as enabling procedure-level tracing in the trace viewport and intermixing C source code with assembly code in the code viewport.

XRAY29K software automatically selects the correct debugging mode-based on whether the object module was created by the high-level compiler or the assembler. When a program has both kinds of object modules, a utility command toggles between the two modes.

XRAY29K software includes a search facility that can find information in a source file and display the value of an expression in decimal, hexadecimal or ASCII format.

On-line help is provided for all debugger commands, command arguments, and function keys, and includes a selection menu.

alias	Replace the name of the command
cexpression	Calculate an expression's value
error	Set include file error handling
help	Display on-line help screen
history	Recall a specifc command
include	Read in and process a command file
journal	Save all viewport commands and data to a file
log	Record debugger commands and errors in a file
mode	Select debugging mode (high or assembly)
option	Set debugger options for this session
pause	Pause simulation
reset	Simulate microprocessor reset
restart	Reset the program starting address
startup	Save the default startup options

Session Control

The debugger session can be ended at any time or can be paused while the host operating system environment is used and then entered again. This area also controls which object modules are loaded for debugging.

host	Temporarily enter the host environment
load	Load an object module for debugging
quit	End a debugging session

System Requirements

The XRAY29K software resides on the host system and presents the user with a friendly, high-level interface to the Am29000 microprocessor-based system. The software communicates with the host system through a serial interface to the ADAPT29K unit or a target board running the MON29K target-resident debug monitor, or a bus interface to the PCEB29K personal computer execution board. The MON29K software and the ADAPT29K unit actually perform all the Am29000 microprocessor memory and register reads and writes requested by the user through XRAY29K debugger commands.

Before the XRAY29K debugger can be used, an absolute object module must be created and downloaded into the target system RAM memory. The object module is created using AMD's HighC29K compiler or ASM29K assembler. Once generated, the object module is loaded into target system RAM memory by invoking the XRAY29K software Load command. Figure 3 illustrates the AMD development tool chain.

Software Warranty

Software programs licensed by AMD are covered by the warranty and patent indemnity provisions appearing in AMD's standard software license forms. AMD makes no warranty, express, statutory, implied, or by description regarding the information set forth herein or regarding the freedom of the described software program from patent infringement. AMD reserves the right to modify, change or discontinue the availability of this software program at any time and without notice.

Customer Support

Maintenance

All orderable software products include one year of free Maintenance Support, which starts from the date of original purchase. Maintenance Support allows customers to receive technical assistance from highly trained field and factory personnel, to use a call-in on-line information system and to receive product and documentation updates at no additional charge. Customers may extend Maintenance Support in one-year increments. Customers can access support services by calling the 24-hour, toll-free 29K™ Family hotline at (800) 2929-AMD (292-9263).

On-Line Call-In Bulletin Board

In addition to the support engineering staff, AMD offers a 24-hour on-line technical support center. A customer can call (800) 2929-AMD at any time to query the system for the latest information on a particular product: bug fixes, work-arounds, information on upcoming releases, etc. Messages may be left for the support engineering staff during "after hours."

Training Classes

AMD offers training classes for the 29K Family products. These classes focus on 29K Family system design and implementation using the broad range of AMD software development tools. Customers can shorten the development process through extensive hands-on training covering a variety of topics. Contact your local AMD field sales office for more information on training classes.

Fusion29K Program

AMD encourages broad-based development and support for the Am29000 microprocessor with the Fusion29K™ program, a joint-effort program between AMD and third-party developers. Published twice a year, the Fusion29K program catalog reveals the breadth of development and system solutions for the 29K Family, including software generation and debug tools; hardware development tools; executive, kernel and multi-user operating systems; board-level products; silicon products; and more. For a copy of the Fusion29K program catalog, call your local AMD field sales office or the AMD literature center at (800) 222-9323.

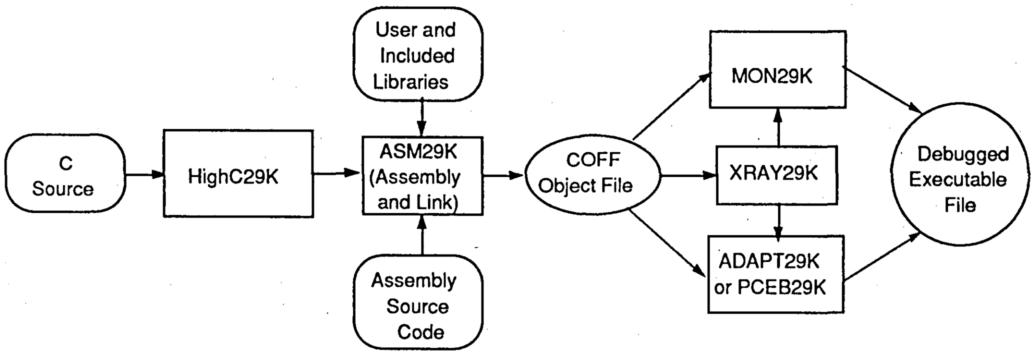


Figure 3. AMD Development Tool Chain

CHAPTER 3
29K Family Application Notes

Am29000 SYSCLK Driving Application Note

by Tom Crawford

INTRODUCTION

The purpose of this note is to describe the options of connecting the SYSCLK pin in an Am29000™ system.

GENERAL CONSIDERATIONS

SYSCLK in any Am29000 system is going to be a high-frequency, heavily loaded signal with strict duty factor requirements. The most important considerations are DC levels, capacitive loading, rise/fall times, high/low times, and transmission line effects.

There are basically two options. One may make SYSCLK a source or one may make SYSCLK a destination.

SYSCLK AS A SOURCE

The easiest (and the recommended) way to connect the clocks in the system is to have the Am29000 generate and drive SYSCLK. Figure 1 shows the connections.

In this configuration, PWRCLK (pin P3) is connected directly to V_{CC}. This is a power pin; it must not be just pulled up through a resistor.

Two times the desired operating frequency is injected into INCLK. This is a TTL signal and the duty factor is unimportant so long as it meets the minimum High time and Low time parameters (see the Am29000 data sheet, order# 09075).

SYSCLK is an output with CMOS levels (it swings from nearly ground to nearly V_{CC}). All the SYSCLK relative-timing parameters are measured with respect to SYSCLK at 1.5 volts, the normal TTL "trip point."

Since SYSCLK must have fairly fast rise and fall times and may be physically long, it may behave as a transmission line (i.e., exhibit reflections). These effects can be minimized using a few precautions.

If SYSCLK goes to more than one or, at most, two places on the board, separate traces to each destination should be used. This minimizes the length of each line and minimizes the capacitive loading on each line. Series resistors at the source (at the Am29000) for each line will reduce the edge rates. Using Schottky or Fast logic is often preferable to CMOS logic, which lacks input diodes to ground.

Before resorting to parallel termination, one should consider carefully the effects of relatively high DC loading on the buffer V_{OH} and V_{OL}.

The prudent engineer will analyze his SYSCLK signal with SPICE or a similar CAD package. This permits a prediction of the actual behavior of the circuit, which is essentially impossible to obtain without modeling.

At this time, there is no guaranteed relationship between the input on INCLK and the output on SYSCLK. Information on this relationship will be included in the Am29000 Data Sheet (order #09075).

SYSCLK AS A DESTINATION

SYSCLK can be driven externally. This is typically done to provide an external signal with a known phase relationship to SYSCLK, perhaps at twice the frequency. Figure 2 shows the connections.

PWRCLK and INCLK must both be connected directly to ground.

SYSCLK is an input and must be driven with a CMOS-level clock at the operating frequency. The fact that signals are generated from both edges of SYSCLK dictates that it be very nearly a perfect square wave (from 1.5 V to 1.5 V). Perhaps the best way to generate such a signal is to begin with one at 2X frequency and divide it by two with a flip-flop. The result is buffered with one or more pieces of a CMOS buffer. A typical clock generator is shown in Figure 3.

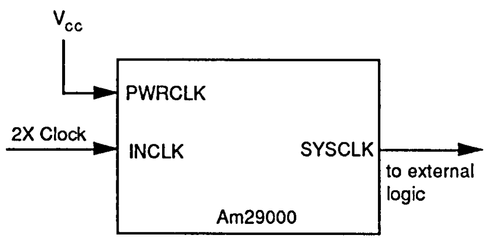


Figure 1. Source

Publication #	Rev.	Amendment	Issue Date
11024	A	/0	11/89

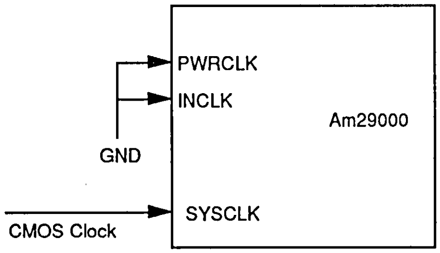


Figure 2. Destination

The TTL oscillator operates at twice the required frequency. Since the 74AC74 is edge triggered, it responds only to the Low-to-High transition of the oscillator. Its output is nominally a square wave (nominally because the tPHL may not be the same as tPLH).

The buffer is more interesting. Clearly, it has to be CMOS since SYSCLK is a CMOS input. It has to be characterized to drive substantial capacitance since the Am29000 has an input capacitance of 90 pF. One can put multiple elements in parallel as long as they are in the same package. In addition, one can drive different portions of the load with different sections of the device. As long as they are in the same package and are similarly loaded, they will exhibit similar delays. In some design groups, putting buffers in parallel is a prohibited activity, since it is sometimes difficult to determine when one of the buffers has failed. Local design rules should always prevail.

Take, for example, the IDT 74FCT240A. With light DC loading, the output swings within 0.2 V of the power supply. At 50-pF loading, the propagation delay is 1.5 ns minimum and 4.8 ns maximum. Putting two elements in parallel will solve the capacitive-loading situation, if it really needs to be solved. The actual waveforms should be examined before adding another buffer. The IDT data book does not distinguish between tPHL and tPLH. The device should be characterized at the actual expected loading, temperature, and voltage ranges to determine the actual switching characteristics.

Take, for a second example, the 74AC04. With light DC loading, the output swings within 0.1 V of the power supply. The guaranteed propagation delays for the 74AC00 are 1.0 ns to 7.0 ns; we expect an AC04 to be the same. In fact, a device actually driving an Am29000 has measured propagation delays of tPLH = 4 and tPHL = 5. Two elements in parallel appear to provide a somewhat cleaner waveform.

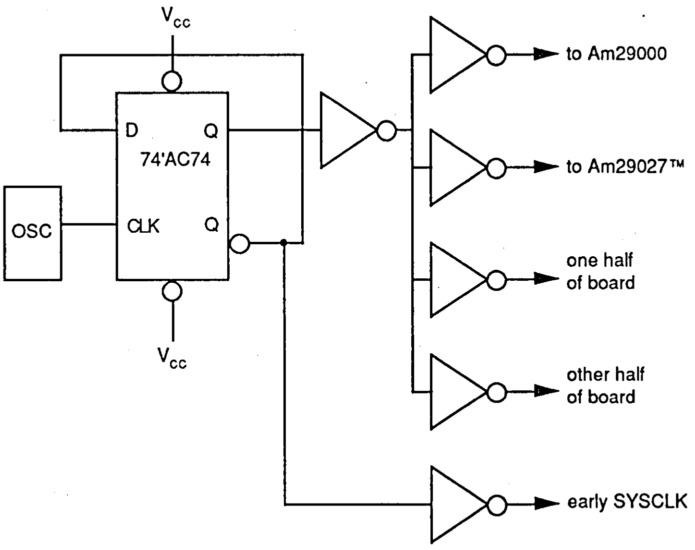


Figure 3. Clock Generator

Connected Am29000 Instruction/Data Buses Application Note

by Tom Crawford

The use of the Am29000™ has been proposed in a system where the instruction and data buses are connected directly to each other and to a single memory.

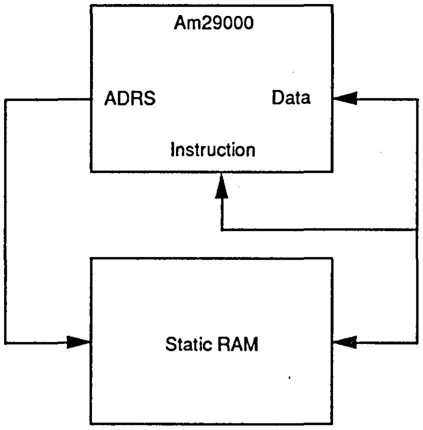


Figure 1. Block Diagram

If the memory is very fast (single cycle), then pipelined or burst accesses never need to take place. Every access is a simple one-cycle access. Data writes would have to be two cycles (because $\overline{\text{BINV}}$ is valid so late). Presumably this would be either a fairly high-end system with lots of very fast memory or a cache system with a modest amount of SRAM backed up by lots of DRAM.

This depends on the availability of *very* fast static RAMs. The equation below shows how to calculate the required access time of the RAMs.

tMAX = tCLK – (para6 + para9A)

For a 25-MHz device running at various clock rates:

FREQ	tCLK	para6	para9A	tMAX
25.00	40	14	6	20
22.22	45	14	6	25
20.00	50	14	6	30
18.18	55	14	6	35

An attempt to actually mechanize a system like this uncovered a problem. When the Am29000 follows an instruction read with a data write, there is a guaranteed "bus crash."

Parameter 10 requires that the data remain on the bus for 2 ns after the rising edge of SYSCLK; in fact, RAM disable times are typically 15 ns. This means there is no known method to get the instruction off the instruction bus until as long as 15 ns after the clock rises. Additionally, in the best possible case, a PAL® delay must be added to allow for the use of SYSCLK to turn off the

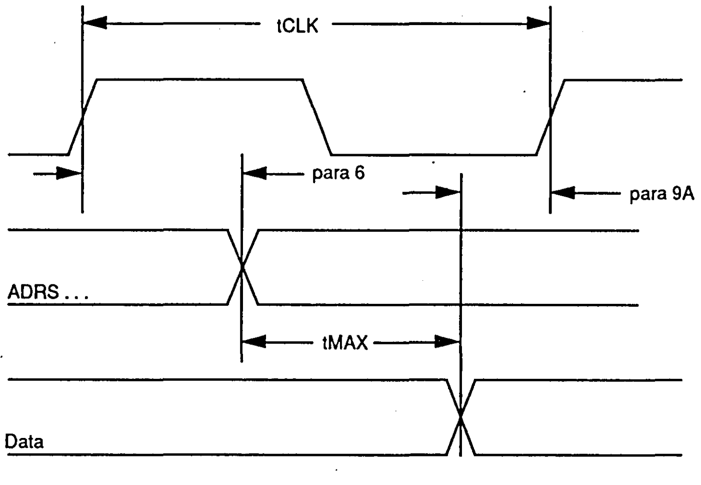


Figure 2. RAM Timing

Publication # 11637 Rev. A Amendment /0 Issue Date 11/89

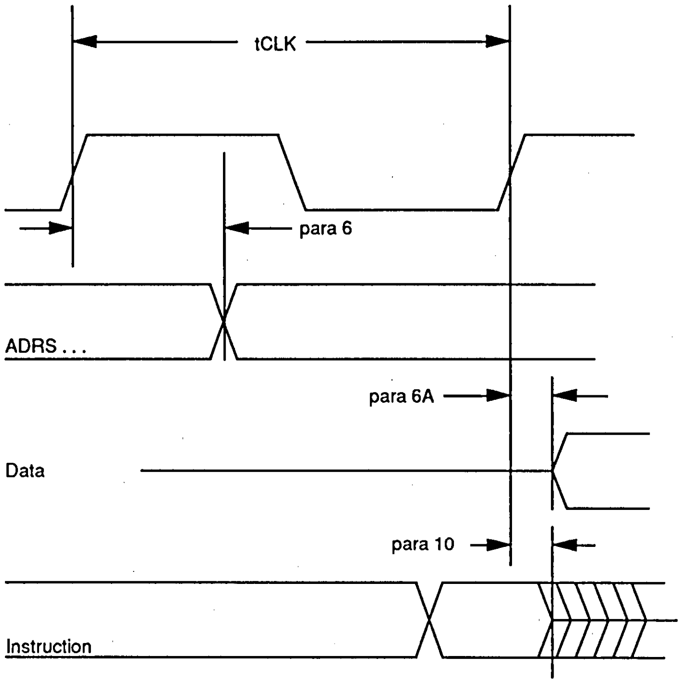


Figure 3. Bus Crash

buffers. Transceivers have a similar problem and, in addition, deduct from the allowable access time.

Parameter 6A specifies the maximum delay of SYSCLK to write data valid. There is no minimum and, in practice, the buffers come out of Hi-Z with the rising edge of SYSCLK. Since the instruction bus and data bus are tied together, there is an unavoidable collision. The memory continues to drive the common bus after the Am29000 begins to drive it.

This problem does not occur in the case of data read followed by a data write. The Am29000 is guaranteed to insert an unused cycle. This provides adequate time for the memory to get off the data bus.

A way to prevent this from occurring is to place a set of transceivers between the data bus and the instruction bus.

Now the block diagram looks like this:

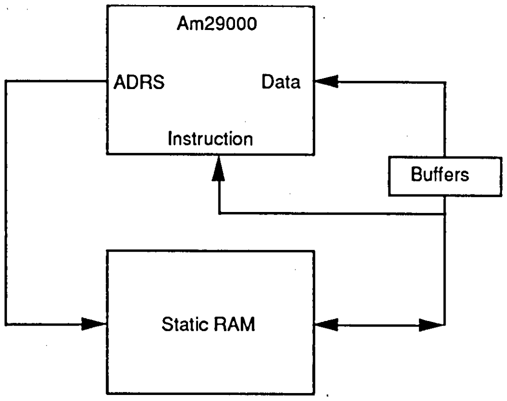


Figure 4. Buffers Added

The transceivers between the data bus and the instruction bus will isolate the Am29000 data drivers from the RAM drivers long enough to allow the RAM drivers to go into high impedance. The transceivers are then turned on, pointed in the correct direction, and the data can be driven into the array.

The instruction path has no additional delays (other than the added capacitance of the transceivers). It can still do single-cycle instruction fetches. The delay imposed in the the data path certainly dictates a two-cycle load, unless the memory is substantially faster than would otherwise be necessary for instruction fetches. Stores are not affected since the $\overline{\text{BINV}}$ comes out too late to allow single-cycle operations anyway.

The buses may also be required to be connected together when the memory must be common because of software requirements. With a slow memory, the access time added by the insertion of a buffer is a much smaller percentage of the total access time.

Byte-Writable Memories For The Am29000 Application Note

by Tom Crawford

OVERVIEW

This document describes how to implement a byte-write memory design for an Advanced Micro Devices Am29000™-based system. While this document will concentrate on the specific case of unsigned bytes, an analogous case exists with signed bytes and signed and unsigned halfwords.

There are three important benefits that accrue from incorporating a partial write capability.

Assembly code can run faster

The code to perform a byte write currently generated by the compilers and recommended for assembly-language programmers looks like Figure 1.

A substantially faster way to do the same thing (given that the memory can write single bytes) looks like Figure 2.

This is faster since the initial LOAD is avoided. In addition, the compiler is more likely to be able to "bury" an isolated STORE by scheduling than both a LOAD and a STORE.

Future compiler releases will support byte-write memories

AMD is enhancing our compiler to optionally generate the byte writable code shown above. To benefit from these enhancements, an application's memory must be able to support byte writes.

Future revisions of silicon will support byte writes

Future Am29000 CPU products will be designed to directly support byte writes. One approach would involve having the processor replicate the byte in question onto all byte positions. Analogous logic would have the processor pick the correct byte during a LOAD. The system design would have to be able to execute byte writes to take advantage of the saved cycle.

The AMD Binary Compatibility Standard (BCS)

AMD's BCS will assume a memory that has byte-write capability. Therefore, if binary compatibility is important to your application, your memory will need to support byte writes.

WHAT MEMORY DESIGNERS MUST DO

The bottom line to support byte-write capability is, "you have to be able to suppress writes to one or more bytes." This has two implications. The first is that some control signal or signals must be generated and distributed by byte. The second is that you must choose between suppressing the write by completely suppressing the memory cycle or by turning it into some kind of cycle other than a write.

```
load 0,17,temp,addr          ;load full word into register,
                                  ;set BP to correct address. 0x11
                                  ;in the CNTL field selects SB and
                                  ;OPT bits corresponding to byte
inbyte temp,temp,data        ;insert byte into proper position
store 0,17,temp,addr         ;store full word into memory
                                  ; (not byte writable)
```

Figure 1. Compiler-Generated Byte-Write Code

```
mtsrim bp,addr               ;put address into BP
inbyte temp,data,data        ;insert byte into proper data
                                       ;position and low order byte
store 0,17,temp,addr         ;write a single byte. The external
                                  ;memory looks at OPT, Ax bits
```

Figure 2. Streamlined Byte-Write Code

Publication #	Rev.	Amendment	Issue Date
11636	A	/0	11/89

There are four distinguishable memory configurations, each of which can be treated in its own way. Whether the devices have an explicit output enable really determines one's choices in selecting an alternative cycle type. If there is not explicit output enable and the I/O pins are common or tied together, one must not allow a "complete" read or there will be a bus crash.

Static RAMs with explicit output enables

The IDT 32K by 8 CMOS device is an example of a static RAM with an explicit output enable. In the case of these devices one can arrange to suppress either the / Chip Select (the device will not cycle at all) or the /Write Enable and the /Output Enable (the device will internally execute a read but will not come out of high-impedance).

Note that one cannot activate both /Chip Select and /Output Enable to these devices without having them drive their data pins.

Static RAMs without explicit output enables

The Toshiba 5561 64K by 1 CMOS device is an example of a static RAM without explicit output enables. If they get /Chip Enable, they will either drive their data-out pins or execute a write, depending on the state of Write Enable.

If their data inputs are connected to their data outputs (typical when connected to a bi-directional bus), /Chip Enable must be suppressed.

Video DRAMs (VDRAMs) with explicit output enable

VDRAMs allow more choice than any other technology. /RAS can be suppressed, preventing the cycle altogether. /CAS can be suppressed, turning the write into a RAS-only refresh cycle. /WE (and /DT-OE) can be suppressed, turning the cycle into an internal read. Of the three, I much prefer suppressing /CAS. First, I like the elegance of generating a RAS-only refresh, and second, /CAS is easier to suppress because it is generated later in the cycle than /RAS or /WE, as shown in the code below.

The equations in Figure 3 allow for a Byte-Order (little/ big endian)† input that effectively is XORed with the address bits. This signal is not a pin on the Am29000. It is a bit in the configuration register. If this bit always is programmed to the same value in a given system, one implements only the appropriate min-terms. If the signal is dynamic in a system, a copy must be kept up-to-date in an external register.

DRAMs without explicit output enable

256K or 1 Meg (by one) DRAMs do not have an explicit output enable. Rather, if /CAS falls with /RAS low and /WE high, the device will enable its output buffers. This means having the option of suppressing the cycle altogether by suppressing /RAS, or turning it into a RAS-only refresh by suppressing /CAS. 256K or 1 Meg by four DRAMs have an explicit output enable. This makes them similar to the VRAM case.

```
!CS31   = !OPT1 & !OPT0                         & CAS_TIME      /*Word*/
        # !OPT1 &  OPT0 & !BO & !A1 & !A0 & CAS_TIME      /*Byte, Big*/
        # !OPT1 &  OPT0 &  BO &  A1 &  A0 & CAS_TIME      /*Byte, Little*/
        #  OPT1 & !OPT0 & !BO & !A1 & !A0 & CAS_TIME      /*HW, Big*/
        #  OPT1 & !OPT0 &  BO &  A1 & !A0 & CAS_TIME      /*HW, Little*/
!CS23   = !OPT1 & !OPT0                         & CAS_TIME      /*Word*/
        # !OPT1 &  OPT0 & !BO & !A1 &  A0 & CAS_TIME      /*Byte, Big*/
        # !OPT1 &  OPT0 &  BO &  A1 & !A0 & CAS_TIME      /*Byte, Little*/
        #  OPT1 & !OPT0 & !BO & !A1 & !A0 & CAS_TIME      /*HW, Big*/
        #  OPT1 & !OPT0 &  BO &  A1 & !A0 & CAS_TIME      /*HW, Little*/
!CS15   = !OPT1 & !OPT0                         & CAS_TIME      /*Word*/
        # !OPT1 &  OPT0 & !BO &  A1 & !A0 & CAS_TIME      /*Byte, Big*/
        # !OPT1 &  OPT0 &  BO & !A1 &  A0 & CAS_TIME      /*Byte, Little*/
        #  OPT1 & !OPT0 & !BO &  A1 & !A0 & CAS_TIME      /*HW, Big*/
        #  OPT1 & !OPT0 &  BO & !A1 &  A0 & CAS_TIME      /*HW, Little*/
!CS07   = !OPT1 & !OPT0                         & CAS_TIME      /*Word*/
        # !OPT1 &  OPT0 & !BO &  A1 &  A0 & CAS_TIME      /*Byte, Big*/
        # !OPT1 &  OPT0 &  BO & !A1 & !A0 & CAS_TIME      /*Byte, Little*/
        #  OPT1 & !OPT0 & !BO &  A1 & !A0 & CAS_TIME      /*HW, Big*/
        #  OPT1 & !OPT0 &  BO & !A1 &  A0 & CAS_TIME      /*HW, Little*/
```

Figure 3. /CAS-Suppressing Code

† Note that all AMD 29K Family software uses big endian byte ordering only. The little endian min-terms are shown for completeness only. Always use big endian.

Am29027 Hardware Interface Application Note

by Bob Perlman

INTRODUCTION

The Am29027™ arithmetic accelerator interfaces simply and efficiently to the Am29000™ streamlined instruction processor. The interface is designed to run at speeds in excess of 25 MHz; so care must be taken when connecting the two parts on a circuit board.

This application note describes the rules to use (and the hazards to be aware of) when designing a 29K™ system containing the Am29027.

PROCESSOR/ACCELERATOR INTERCONNECT

A diagram of an Am29000/Am29027 interconnect is shown in Figure 1. The interconnect contains the following signals:

Control signals—Eleven signals control the transfer of data and instructions between the Am29000 and the Am29027. Eight of these signals, R/$\overline{\text{W}}$, $\overline{\text{DREQ}}$, DREQT$_0$, DREQT$_1$, OPT$_2$–OPT$_0$, and $\overline{\text{BINV}}$, are generated by the Am29000. These specify the accelerator transaction requested by the Am29000. The three remaining signals, $\overline{\text{CDA}}$, $\overline{\text{DRDY}}$, and $\overline{\text{DERR}}$, are generated by the Am29027. The $\overline{\text{CDA}}$ signal indicates whether the Am29027 is ready to accept new instructions or operands. The $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ signals indicate that data requested by the Am29000 is available on the Am29027 output port or that an error has occurred, respectively.

Data signals—The Am29027 R and S data input ports (R$_{31}$–R$_0$ and S$_{31}$–S$_0$), instruction port (I$_{31}$–I$_0$), and data output port (F$_{31}$–F$_0$) are connected to the Am29000 address (A$_0$–A$_{31}$) and data (D$_0$–D$_{31}$) buses. The Am29000 uses its address and data buses to transfer instructions and operands to the Am29027, and uses its data port to read results from the Am29027.

Clock—The Am29027 CLK input is connected to the Am29027 SYSCLK pin. The SYSCLK signal can be generated in two ways: internal to the Am29000, by applying a 2X clock signal to the Am29000 INCLK input (as shown in Figure 1); or externally, by applying a clock signal to the Am29000 SYSCLK pin.

System reset—The system reset signal is applied to the Am29000 and Am29027 $\overline{\text{RESET}}$ inputs.

Most interconnect signals are direct connections. The only exceptions are signals $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$, which must be passed through negative-logic OR gates (i.e., through conventional AND gates). These gates form the logical OR of the $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ signals of all resources on the Am29000 processor channel. The 33kΩ resistors shown connected to the $\overline{\text{CDA}}$, $\overline{\text{DRDY}}$, and $\overline{\text{DERR}}$ signals leaving the Am29027 need be present only if the system sometimes is operated without the Am29027.

One interconnection is optional. The Am29027 $\overline{\text{EXCP}}$ signal, which indicates the presence of an unmasked arithmetic exception created by an accelerator operation, can be connected to an Am29000 trap or interrupt input. This connection is necessary only if the system designer desires an imprecise processor interrupt in the presence of an accelerator exception. The Am29027 contains internal mechanisms for recovering from errors; these mechanisms make the use of $\overline{\text{EXCP}}$ unnecessary in most systems.

Publication #	Rev.	Amendment	Issue Date:
12215	A	/0	11/89

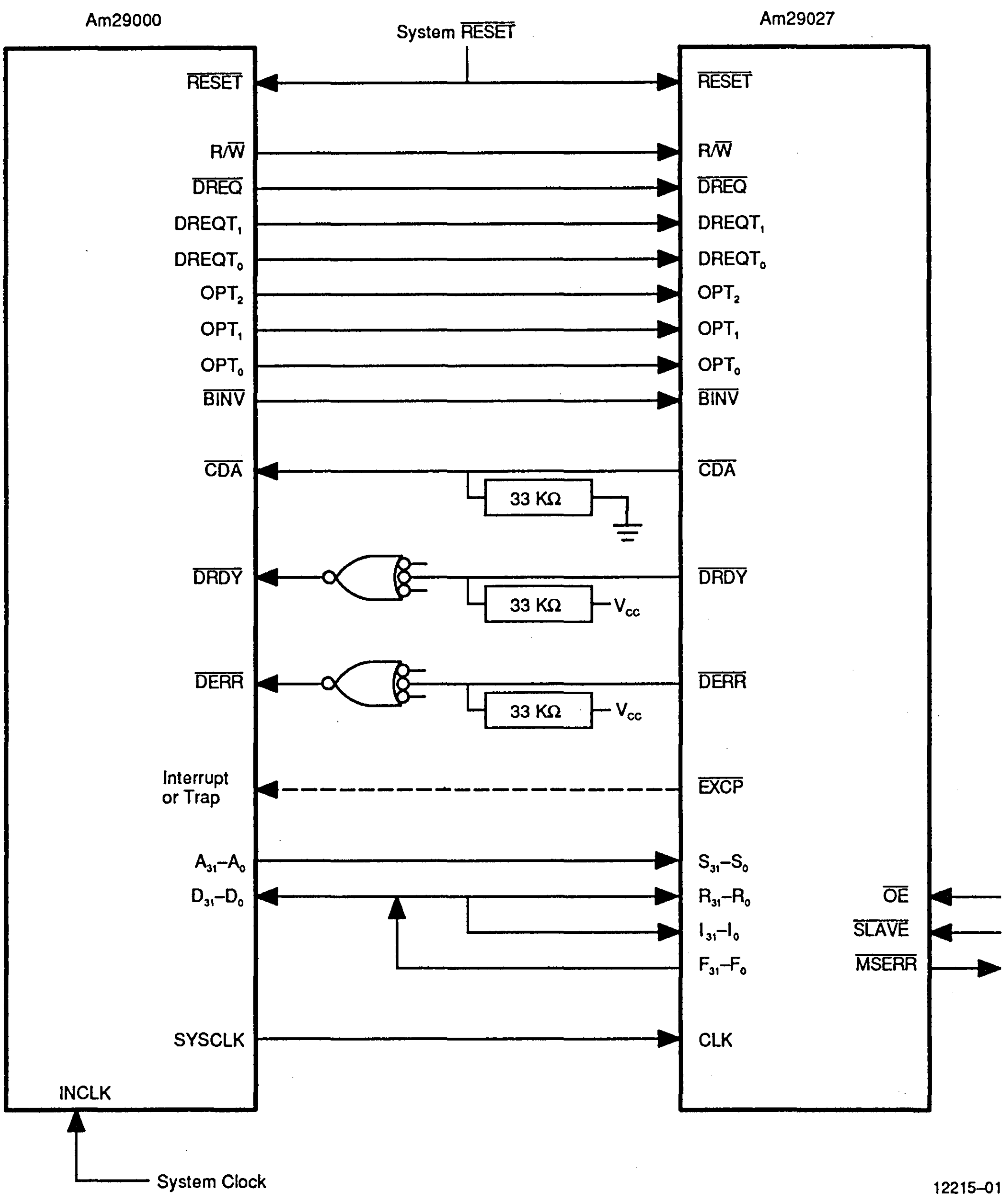


Figure 1. Am29000/Am29027 Hardware Interconnect

AN ALTERNATE INTERCONNECT

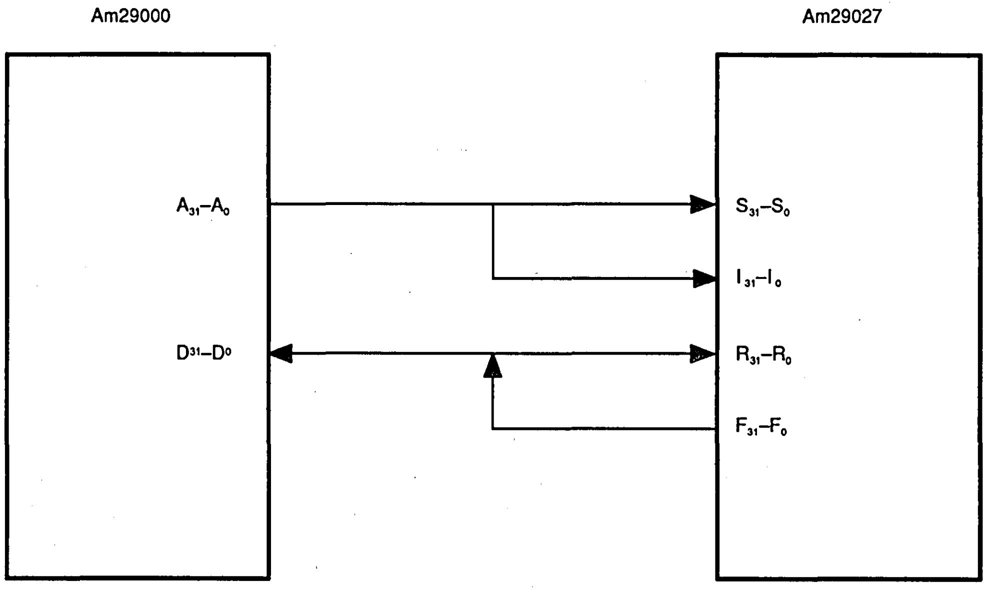
In the interconnect shown in Figure 1, three Am29027 ports are connected to the Am29000 data bus: input data port R, output data port F, and the instruction port. This places considerable capacitive loading on the Am29000 data bus: 12 pF each for input data port R and the instruction port, and 20 pF for output port F, for a total of 44 pF.

The Am29000 data bus can drive an 80-pF load without derating. In systems where the 44-pF load presented to the data bus by an Am29027 is excessive, an alternate interconnect can be used, as shown in Figure 2. In this configuration, the Am29027 instruction bus is connected to the Am29000 address bus, rather than to the data bus. This interconnect more evenly distributes the Am29027 capacitive load between the Am29000 address and data buses. In this configuration the address bus has a load of 24 pF, the data bus 32 pF.

The alternate interconnect, shown in Figure 2, is software compatible with the interconnect of Figure 1. The only requirement for this compatibility is that, when transferring an accelerator instruction from the Am29000 to the Am29027, the instruction must appear on both the Am29000 address and data buses. For example, an Am29000 co-processor store that transfers an accelerator instruction from general-purpose register *gr96* to the accelerator instruction register must have the form:

```
store 1,CP_WRITE_INST,gr96,gr96
```

Note that *gr96* is specified for both the RA and RB instruction fields, thus ensuring that the accelerator instruction to be transferred is placed on both the address bus and the data bus. All 29K accelerator code, including that produced by the 29K compilers, follows this convention.



12215–02

Figure 2. Alternate Am29000/Am29027 Bus Connections

RULES TO FOLLOW

Even though interconnecting the Am29000 and Am29027 is straightforward, a few precautions must be taken to ensure correct accelerator operation:

- All signals except $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ must be direct connects; the signals should not pass through other devices. For example, if the Am29000 address bus is buffered before being fed to a memory array, the Am29027 address bus connections must be made on the processor side of the buffers.
- Signals $\overline{\text{DRDY}}$ and $\overline{\text{DERR}}$ should pass through one (and only one) fast AND gate. The system designer should take care to choose high-speed AND gates; a 74AS08, 74AS11, 74AS20, or 7.5 ns PAL® device will suffice at 25 MHz.
- Keep signal interconnects short. Heavily loaded traces may have propagation speeds on the order of 3–4 ns/foot. All signal traces, and in particular those with the heaviest loading, should be kept as short as possible.
- Minimize loading on the Am29000 data and address buses. These buses are designed to drive 80-pF loads without AC timing derating, and higher capacitances with derating. If Am29000 bus capacitances exceed 80 pF, be sure to derate the AC parameters per the information provided in the Am29000 Streamlined Instruction Processor Data Sheet, order #09075.

While the alternate bus connections shown in Figure 2 will lower the capacitive loading presented to the Am29000 data bus, they do present a greater routing challenge than the connections of Figure 1.

WARNING: With the alternate connections of Figure 2, many signal lines must cross one another either under or near the Am29027. Before using the alternate connections, be sure to examine layout and routing requirements.

When is Interleaved Memory with the Am29000 Unnecessary? Application Note

by Tom Crawford

INTRODUCTION

ABSTRACT

This application note presents a graphic method of finding the maximum acceptable access time of an Am29000™ memory system that avoids the use of an interleaved memory.

GENERAL

The advantage of an interleaved memory is that slower and less expensive memory chips can be used. However, the use of interleaved memory in systems that need only a limited amount of memory should be avoided, since interleaving doubles the minimum memory size. The need to support two memory banks may waste a substantial amount of memory space and result in a higher system cost.

Advanced Micro Devices is developing a complete line of Am29000 simulators, hardware target execution vehicles, and high-level language development tools for the Am29000 32-bit Streamlined Instruction Processor. These products are designed to support end-users who are building embedded system applications based on the Am29000 processor. For these users, often there is no existing operating system or kernel for their hardware design.

The design trade-off is component count versus the required device access speed and density of memory. By analyzing the required access speed of memory devices for both interleaved and non-interleaved memory, it is possible to determine the relative cost and performance for each approach. The analysis also identifies the situations in which the system clock rate dictates the use of interleaved memory because sufficiently fast memory devices, needed to support a single-bank architecture, are unavailable.

WHEN IS INTERLEAVING NECESSARY?

Figure 1 shows a routine method of obtaining data for an instruction burst-mode access. (The *instruction burst-mode access* considerations discussed in this application note also apply to the *data burst-mode access* considerations.)

A counter is loaded with the beginning address of the burst, then incremented to fetch successive words. The output of the counter goes through an address multiplexer and then to the address inputs of the memory chips. The data output pins of the memory chips are connected directly to the Am29000 bus.

Assuming the counter increments on the positive edge of SYSCLK, it is possible to calculate available time before the data must be valid. Figure 2 shows the available time for a Static Column DRAM (tMAX). Any data buffers between the memory and the Am29000 would cause additional delays.

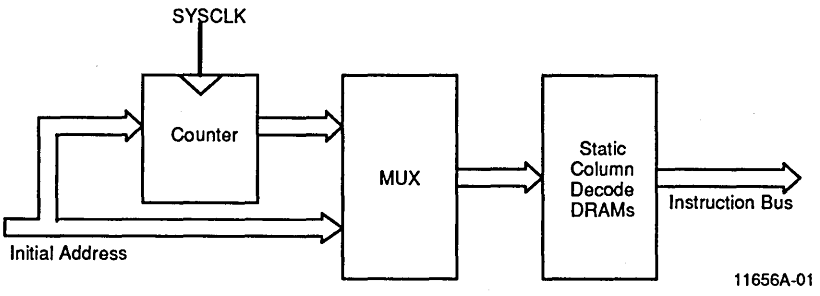


Figure 1. Typical Memory

Publication #	Rev.	Amendment	Issue Date:
11656	A	/0	11/89

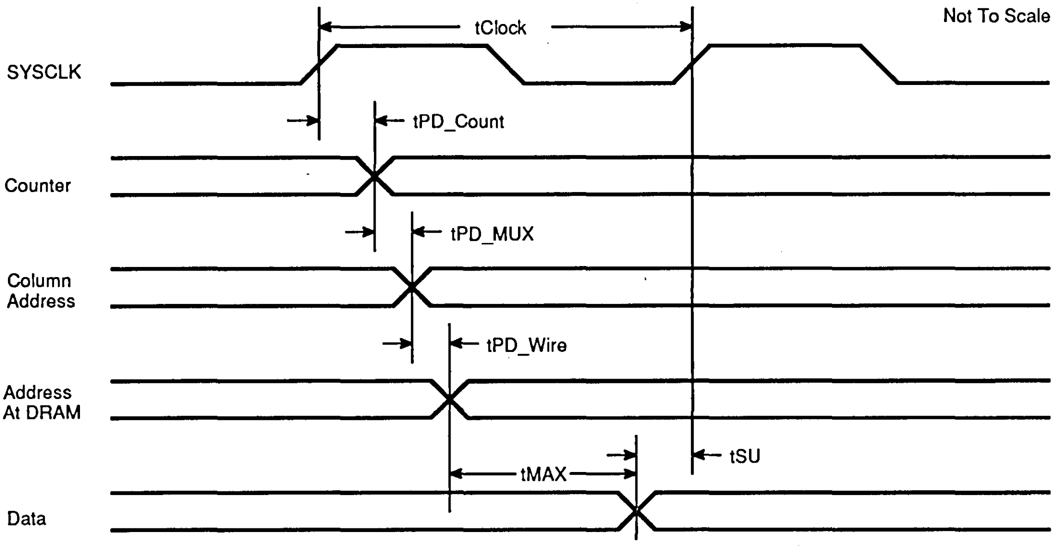


Figure 2. Single-Cycle Burst

In order to guarantee positive margins, the following inequality must be satisfied:

Equation 1: $n * tCLOCK - (tMAX + tPD\_COUNT + tPD\_MUX + tSU + tPD\_WIRE) > 0$

The value *n* is the number of clock cycles available for memory. If there is no interleaving or wait states, n = 1. For two-way interleaving, n = 2, and so on.

The maximum column address delay (static column decode DRAM) that can be allowed is tMAX. The clock-to-output delay of the counter is tPD_COUNT. The value of tPD_MUX is the input-to-output delay of the multiplexer. The value of tSU is the setup time for Am29000 instructions or data.

The value of tPD_WIRE is the propagation delay from the multiplexer output to the furthest memory chip input. This is the propagation delay per unit length of wire times the length of the wire. The propagation delay per unit length can be estimated from the equation:

$$tpd' = tpd \sqrt{(1 + (Cd / Co)} \quad (1)$$

The unloaded propagation delay *(tpd)* is determined only by the board material dielectric constant. It is equal to approximately 1.77 ns/ft. The trace capacitance *(Co)* is a function of the trace impedance and propagation delay and is usually taken to be approximately 18.5 pF/ft. The distributed capacitance *(Cd)* resulting from the memory chips is calculated from the per-device input capacitance and the device spacing; assuming 5 pF per device and two devices per inch gives: 120 pF/ft.

Using these numbers in the above equation yields:

$$tpd' = 1.77 \sqrt{(1 + (120 / 18.5)} = 4.84 \text{ ns/ft}$$

Finally, assuming that 32 devices at 24 devices per foot equals 1.33 ft, then the value for tPD_WIRE is 6.45 ns. These numbers are summarized in Table 1.

Table 1. Initial Numbers

Name	Value	Obtained From
tPD_COUNT	6.5 ns	PAL16R8-7
tPD_MUX	8.0 ns	74F253 In to Zn
tPD_WIRE	6.5 ns	See discussion above
tS_25 MHz	6.0 ns	Am29000 25MHz tSU
tSU 20/16 MHz	8.0 ns	Am29000 20/16 MHz tSU

Figure 3 shows the results of these values in equation 1. The x-axis is tCLOCK and the y-axis is the allowable access time. The solid line shows the allowable access time for n = 1 (single-cycle operation [no interleaving]). The dotted line shows the allowable access time for n = 2.

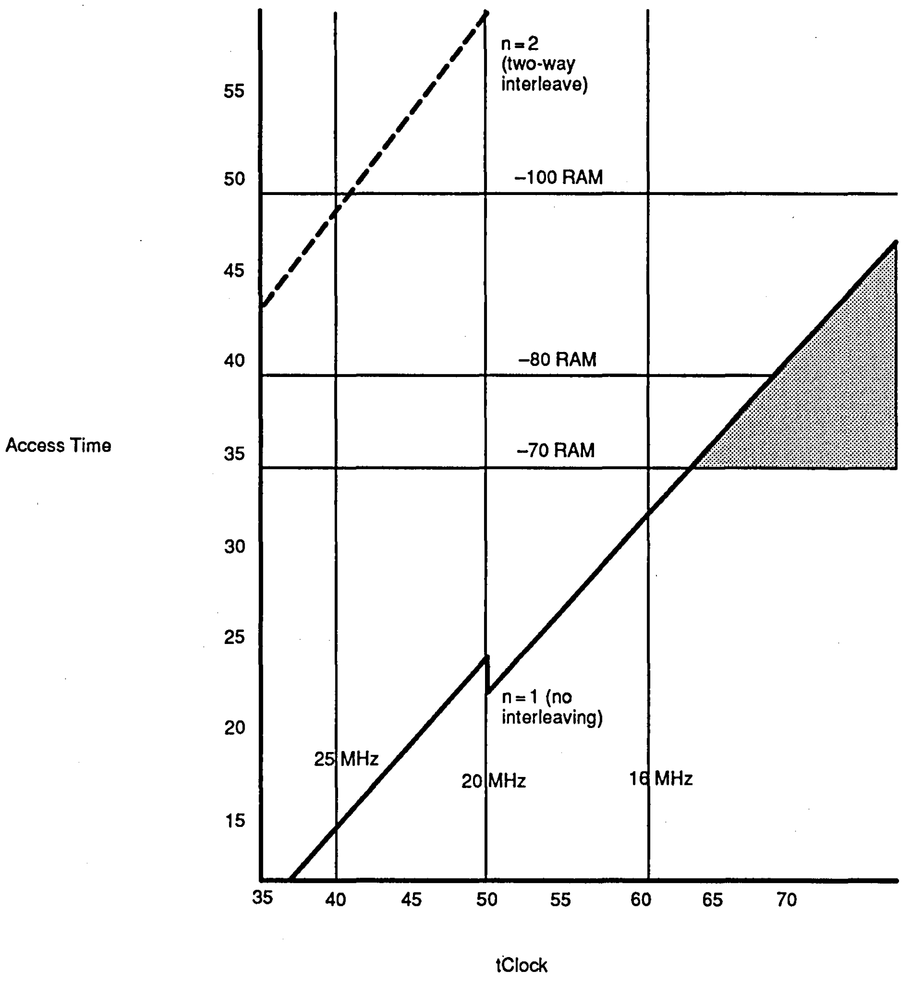
1 See Appendix A of the Am29000 Memory Design Handbook (order #10623) for additional information on this equation.

The discontinuity in the n = 1 line reflects the difference in tSU between 25 MHz and 20/16 MHz. The horizontal lines show the access times for –70, –80, and –100 Toshiba 1M-by-1 DRAMs. The vertical lines show the minimum tCLOCK times for 25-, 20-, and 16-MHz Am29000s. The hatched area indicates where operation is possible without interleaving.

INITIAL RESULTS

From inspection of Figure 3, it might be concluded that it is almost possible to build a single-cycle burst memory for a 16-MHz Am29000 from "fast" DRAMs with no interleaving. However, one cannot build a single-cycle burst memory for a 20- or 25-MHz system without interleaving with any available DRAM.

Finally, using two-way interleaving, it is possible to build a memory that supports single-cycle bursts at a clock rate of 25 MHz or below, from memories with a column address access time of less than 50 ns.



11656A-03

Figure 3. Initial Results

ARE IMPROVEMENTS POSSIBLE?

Could a system be built with single-cycle bursts without interleaving to run at 20 MHz? To answer this question graphically, move the heavy line in Figure 3 upwards (extending the hatched area to the left). This is done by reducing or eliminating the numbers, other than tMAX, in the inequality. These are examined below, one at a time.

tPD_COUNT

The 6.5 ns value is based on using a –7 PAL®. This is already faster than any 74F, 74AS, or 74ACT counter (or flip-flop, for that matter) in any data book this author has examined.

It is certainly possible to "play games" with the clock scheme. SYSCLK on the Am29000 could be driven a little later than the clock to the counter. Data hold time is unlikely to ever be a problem. But the uncertainties in propagation delay through a CMOS clock driver are likely to cancel a lot of what could be gained. Furthermore, delaying the clock to the Am29000 delays the address on the initial cycle.

tPD_MUX

The 8.0 ns value is based on using a 74F253. A 1/2 ns reduction could be realized by building a multiplexer with a 16L8-7 (7.5 ns). A better way is to completely eliminate the multiplexer delay by building a three-state bus. Figure 4 shows one way to do this.

The counter is implemented with a 16R8-7 (actually, more than one is probably required). An 8-bit counter is required and 2 additional bits of address must be maintained. Since the clock is not gated, some additional inputs are required to indicate whether the counter should load, hold, or count.

Just before RAS falls, the three-state buffer is enabled. When the Column Address is required, the three-state buffers of the PAL device are enabled and the counter is driven into the array.

In this configuration, a worst-case design requires that the extraordinary loading on the PAL device be considered. The total capacitance connected to the outputs of the PAL devices is greater than the standard load. However, the capacitances are distributed rather than lumped. The driver never sees the entire load, so the wire delay allowance is sufficient.

tPD WIRE

The wire delay can be reduced only by reducing the wiring length. Instead of connecting all the memory chips in serial, the board can be designed so that there are two sets of chips connected in parallel. This halves the 1.33-foot length previously calculated and reduces the wire delay to 3.22 ns.

To reduce tSU, a fast Am29000 at a reduced clock rate can be used. For example, a 30-MHz Am29000 has a tSU of only 5 ns; this is 3 ns better than a 16-MHz part, but it is expensive.

Another approach is to insert a pipeline register with a very low setup time. For example, the data setup time of a 74F374 is only 2 ns. Of course, including a pipeline register has adverse consequences. The first access of a burst-mode access will then be one SYSCLK cycle longer than would otherwise be required. In addition, the control logic is made slightly more complicated. A positive side effect is that three-state buffers are included in the register packages. Figure 5 shows registers in the instruction path.

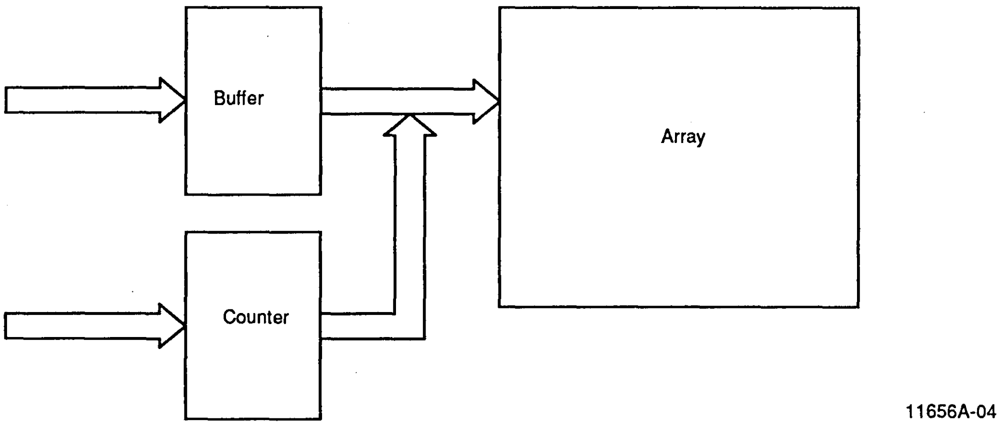


Figure 4. Multiplexer Avoidance

Now, assuming the implementation of all the changes described above, the fixed numbers become the values shown in Table 2.

Table 2. The Improved Numbers

Name	Value	Obtained From
tPD COUNT	6.5 ns	PAL16R8-7
tPD MUX	0.0 ns	Three-state multiplexor
tPD_WIRE	3.2 ns	Length
tSU	2.0 ns	74F374 data sheet

If this is plotted as a function of cycle time, the line has moved up a considerable amount as compared to Figure 3. This indicates that it is possible to build a 20-MHz system with the fastest available DRAMs. It also indicates that it is possible to build a 16-MHz system with 100-ns DRAMs.

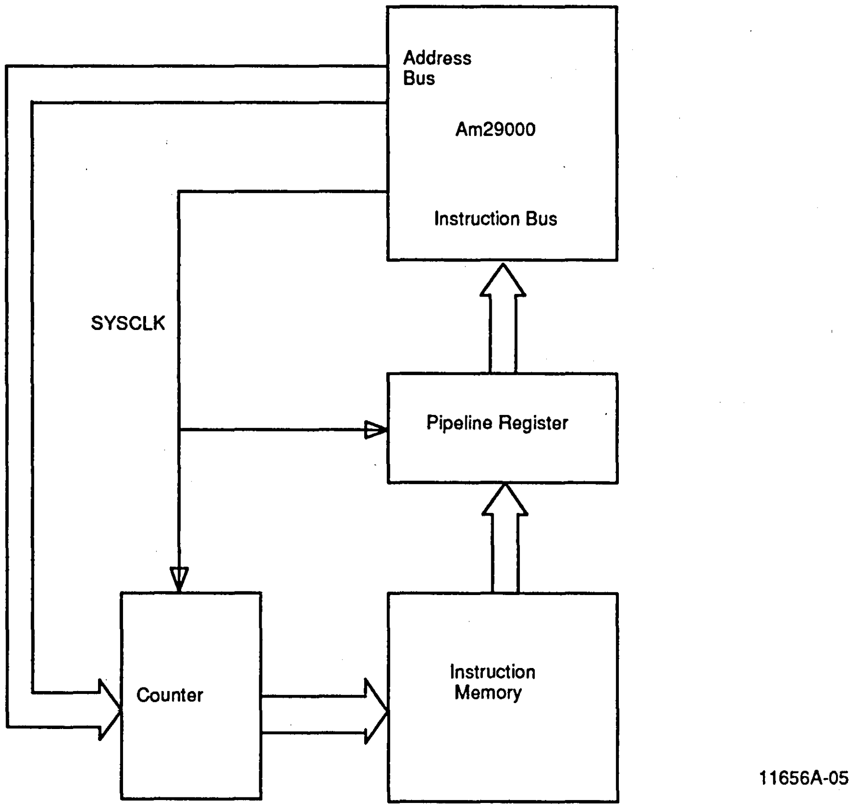


Figure 5. Pipeline Access

CONCLUSION

By using the values for proposed memory architectures into Equation 1, two to four specific values of tMAX can be determined for appropriate values of tCLOCK. With this information it is easy to draw graphs like those of Figures 3 and 6. Such graphs provide a simple display of the available trade-offs between system clock rate, memory architecture, and the memory device access speed. Multiplying the memory device count for each configuration by the access-speed driven memory device costs of the configuration yields an approximate cost for each memory system approach.

Such an analysis may point out significant cost reductions by quickly identifying those situations in which a non-interleaved memory architecture and reduced clock rate can support the required system performance.

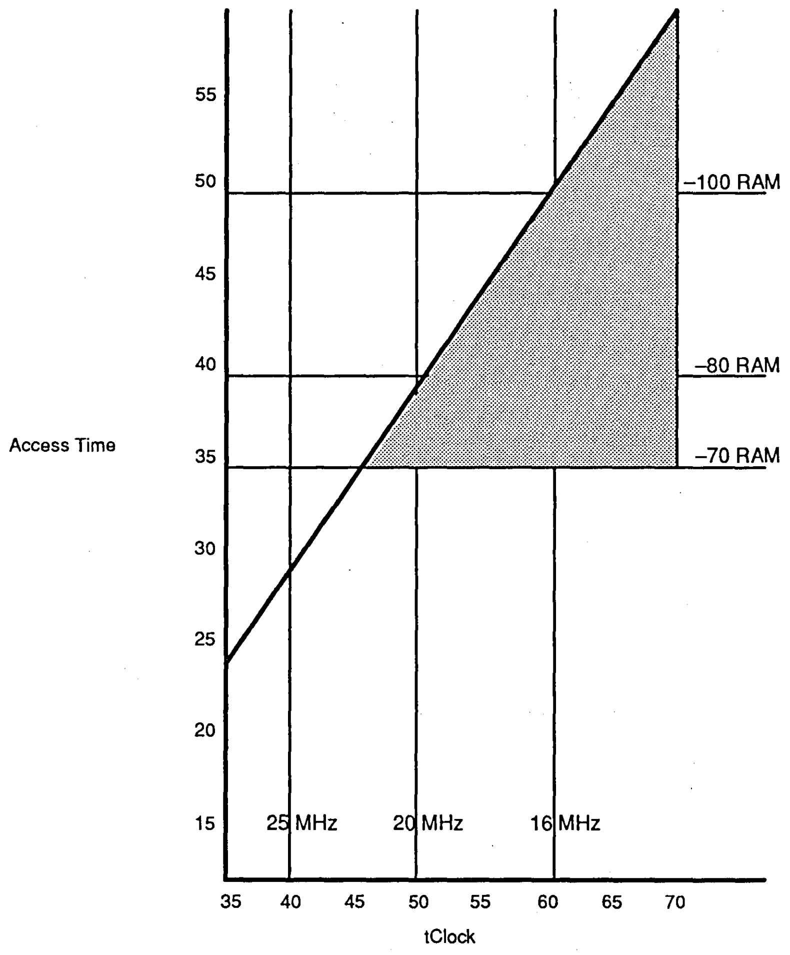


Figure 6. Final Results

Implementation of an Am29000 Stack Cache Application Note

by Phil Bunce and Erin Farquhar

INTRODUCTION

This application note will describe the basic mechanisms of the AMD Am29000's cache of the run-time stack. The stack cache is an important performance feature, because it permits a procedure's entire context to be resident in on-chip registers, thus eliminating, or at least reducing, the need for memory accesses.

Our discussion is centered around a single example program, which is shown in its entirety in Appendix B. Before discussing this example, we provide a brief overview of the basic operation of the stack cache.

OVERVIEW

Procedures executing on the Am29000 make use of a run-time stack, which consists of consecutive, overlapping structures called activation records. An activation record contains the dynamically allocated information specific to a particular activation of a procedure. Each time a procedure is called, a new activation record is allocated on the stack; when the procedure has finished executing, its activation record is deallocated from the stack.

Compilers and assemblers for the Am29000 use two run-time stacks for activation records: the register stack and the memory stack. A procedure's activation record may be divided between these stacks. Both stacks grow toward lower addresses in memory, and items on the stacks are referenced as positive offsets from RSP (Register Stack Pointer) and MSP (Memory Register Stack Pointer). Both pointers are realized using internal Am29000 global registers. The global and local registers are both subsets of the general-purpose registers.

The register stack contains parameters passed to the procedure, the local scalar variables used by the procedure, return linkage information, and the arguments that the procedure will pass to procedures that it in turn calls. The register stack is cached in the local registers, *lr0–lr127*, as explained below.

The memory stack is used for local structured data, for example, arrays and records. It also is used for additional scalar data when needed. When the scalar portion of the activation record for a particular procedure requires more than 128 words of local-register storage, the excess may be kept in the procedure's activation record in the memory stack.

Both stacks are aligned on a double-word (64-bit) boundary. Procedures are required to maintain this alignment by adjusting the size of the register stack frame allocated at procedure entry to be a multiple of eight bytes.

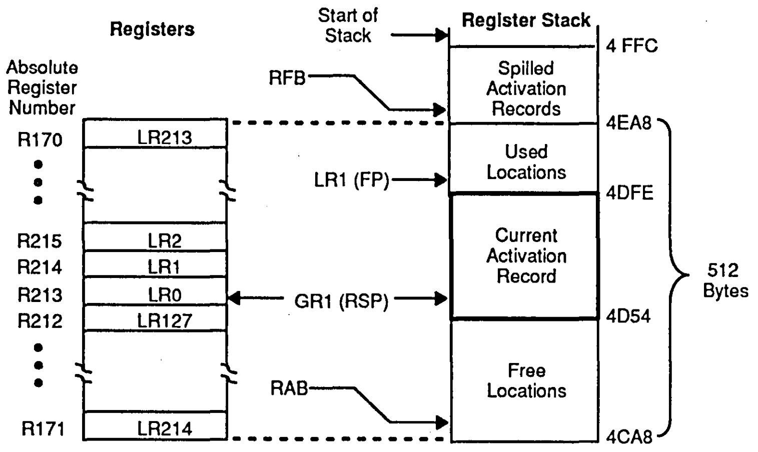
STACK CACHE

The 128 local registers are used to cache locations in the register stack, such that when a procedure is active, its entire register-stack activation record is mapped to the local registers.

Each word location in the register stack is mapped to a single local register. The register number corresponding to a location in the register stack is given by bits 8–2 of the 32-bit memory address of that location in the register stack. Because there are 128 local registers, quantities whose addresses differ by 512 (all addresses are byte addresses) are mapped to the same local register and cannot be in the cache at the same time.

Figure 1 shows a snapshot of the register stack in memory after some calls have been made, and the mapping of the register stack to the local registers. As shown in the figure, Global Register 1, called the Register Stack Pointer (RSP), contains the 32-bit virtual address of the top of the register stack in memory. This virtual address on the Am29000 is the lowest-addressed valid stack location in the current activation record.

Publication #	Rev.	Amendment	Issue Date:
13053	A	/0	11/89



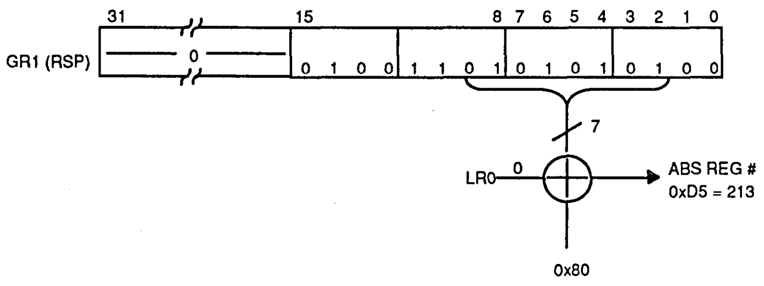
11031A-01

Figure 1. Mapping of Register Stack to Stack Cache

Local registers are addressed as positive word offsets from RSP, as in Figure 2. Specifically, when a local register operand is specified in an instruction (that is, the most significant bit of the register number is set), the seven least significant bits are added to bits 8–2 of RSP and the result is truncated to seven bits. For example, if RSP has the value 0, as shown in Figure 2, then *lr0* is absolute register 128 (the first local register), and *lr1* is absolute register 129 (the second local register); if RSP has the value four, then *lr0* is absolute register 129 and *lr1* is absolute register 130.

Referring again to Figure 1, the current activation record is delimited by the Frame Pointer (FP), which by software convention uses Local Register 1, and RSP. FP points to the "top" of the previous activation record, that is, to the lowest-addressed word location above the current activation record. When a procedure is active, this entire area must be cached in local registers.

The register stack between FP and RFB (Register Free Bound) contains the saved activation records of previously called procedures, which are also currently mapped to the local-register cache. RFB, by convention Global Register 127, is set to point to the lowest-addressed word in the register stack that is not mapped to the local registers.



11031A-02

Figure 2. Local Register Addressing

The register stack between RSP and RAB (Register Allocate Bound) represents stack locations (and corresponding local registers) that are currently "unused" and thus available for allocation when another procedure is called. RAB (by convention Global Register 126) is set to point to the lowest-addressed word in the register stack that is currently mapped to a local register.

When a procedure is called, RSP is decremented by the number of words required to accommodate the called procedure's activation record. When RSP is decremented beyond the location pointed to by RAB and thus beyond the available local registers, more local registers will be required for the activation record, and some locations in the stack cache must be written to memory (or "spilled") before the new activation record is created. This condition is called overflow. Note that in Figure 1, locations between RFB and the Start of Stack are saved activation records that have been previously spilled to memory.

On return from a procedure, the activation record is de-allocated by incrementing RSP by the same amount it was decremented when the procedure was called. If the caller's FP (which points to highest location in the caller's activation record) is greater than RFB (which points to the first unmapped register stack location above the activation record), the contents of that portion of the register stack will have to be loaded into the local registers to accommodate the caller's activation record. This condition is called underflow.

Overflow and underflow conditions are detected by instruction sequences in the prologue and epilogue, which are the instruction sequences that execute as a result of a procedure call and procedure return, respectively, and cause a transfer of control to the appropriate trap handler routine. In the case of an overflow, the trap handler moves the contents of the required number of local registers to the register stack in memory and adjusts the value in RAB and RFB. In the case of an underflow, the trap handler loads the required number of register stack locations into the local registers and adjusts the value in RAB and RFB.

OVERVIEW OF EXAMPLE PROGRAM

Our example program consists of the four text files listed below.

regdcl.h:	Register name declarations
macros.h:	Macro definitions for prologue and epilogue
start.s:	CPU Initialization Overflow and Underflow trap handler routines
example.s:	Two procedures main and recurse

Appendix A contains partial listings from the example program that are described individually in the sub-sections below.

Appendix B contains the source for the entire example program which includes all of the above files.

INCLUDE FILES

There are two include files, **regdcl.h** and **macros.h**. Note that **regdcl.h** must be included before **macros.h**, because **macros.h** uses definitions from **regdcl.h**.

In **regdcl.h** (see Appendix A–1, Register Declarations), we assign the value 80 as the base of registers to be used as temporaries by system software. Additional temporaries will be addressed as offsets from it. These registers will be used for work space in the start code and the two trap handler routines.

```
.equ SYS_TMP, 80 ;system temp registers
```

We also assign symbolic names to global and local registers, in accordance with the software calling conventions of the Am29000.

```
.reg rsp,gr1     ;local reg stack pointer
.reg msp,gr125   ;memory stack pointer
.reg rab,gr126   ;register allocate bound
.reg rfb,gr127   ;register free bound
.reg fp,lr1      ;frame pointer
.reg raddr,lr0   ;return address
```

The overflow and underflow trap vectors, V_SPILL and V_FILL, are set to the constant values 64 and 65. These are the vector numbers for the trap handlers chosen for this example.

```
.equ     V_SPILL,64
.equ     V_FILL,65
```

The second include file in our example program, **macro.h**, contains the macro definitions for PROLOGUE and EPILOGUE. These macros are discussed in the Prologue and Epilogue sections.

START CODE

The module start.s contains code that sets up the execution environment for our example program. The initial portion of the start code is shown in Appendix A–2, Start Code. The overflow and underflow trap handlers, also in **start.s**, will be discussed later.

We set the beginning of the stack (its highest address in memory) at 0x5000. The "& ~7" in the expression ensures that the value is a multiple of eight, with rounding downward if necessary.

```
.equ TOP_STK,(0x5000 & ~7) ;create
                           ;double word
                           ;alignment
```

The two temporary registers, *tmp1* and *tmp2*, are assigned values that are offsets of SYS_TMP, which means that *tmp1* is Global Register 80, and *tmp2* is Global Register 81.

```
.reg     tmp1,      %%(SYS_TMP + 0)
.reg     tmp2,      %%(SYS_TMP + 1)
```

Then we initialize the four pointers that define the stack environment.

```
const  rsp,(TOP_STK-8)    ;set stack
                          ;pointer
add    rsp,rsp,0          ;update rsp
const  rab,(TOP_STK-512)  ;set register
                          ;alloc bound
const  fp,TOP_STK         ;set frame ptr
const  rfb,TOP_STK        ;set reg free
                          ;bound
```

Figure 3 shows the initialized stack. Because there has been no spilling of local registers to the stack in memory, RFB points to the top of the stack. RAB is, by definition, 512 bytes less than RFB. In the initial activation record, defined by FP and RSP, FP points to the top of the stack (because there has been no prior context) and RSP is set to a value eight bytes less than FP to allow for the current FP and *raddr* when a new activation record is created. Note that the setting of RSP must precede the setting of FP by at least two instructions because of the delayed effect of modifying RSP, and that an explicit arithmetic or logical instruction must be used to update RSP.

The CPS (Current Processor Status Register) is initialized with the value 0x0072. Assuming the prior state of this register was Reset mode (shown in Figure 4), we have in effect cleared FZ, DA, and RE, and left the other bits unchanged. The FZ (Freeze) bit is cleared because the processor is unfrozen for normal operation. (For a description of the Freeze bit, refer to the section called "Special-Purpose Registers," in the Am29000 User's Manual). We clear the DA (Disable All Interrupts and Traps) bit to enable all traps. The RE (ROM Enable) bit is cleared because this example assumes we are executing from RAM.

```
mtsrim  cps,0x72    ; PD, PI, SM, DI
```

PD, PI, SM, and DI remain set, meaning that address translation is disabled (PD and PI), supervisor mode is selected (SM), and external interrupts are disabled (DI). Supervisor mode is selected because some of the instructions in our example program are privileged. Address translation is disabled because this example is designed for systems not using the TLB. External interrupts are disabled because we have no interrupting devices and want to eliminate any spurious interrupt requests.

We set the Vector Fetch bit in the Configuration Register to select a vector table configuration for the Vector Area.

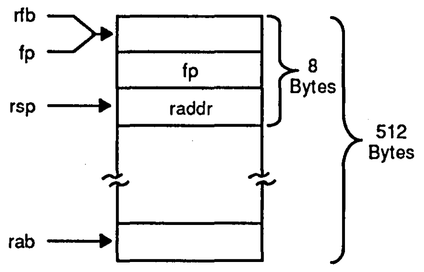
```
mtsrim  cfg,0x10        ; VF
```

The VAB (Vector Area Base Address) register, which specifies the beginning address of the vector table in memory, is set to zero.

```
mtsrim  vab,0
```

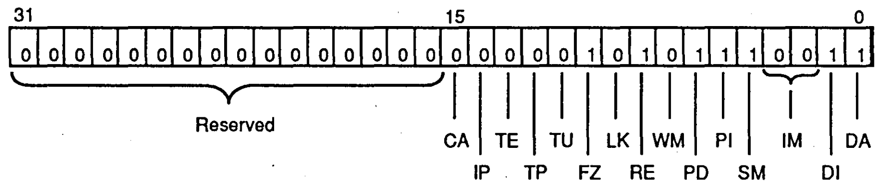
Next we initialize the vector table with the address of the Overflow trap handler routine, called SpillHandler. First we load the address of the SpillHandler into a temporary register, using two CONST instructions for the case when SpillHandler is not in the first 64K-bytes of memory.

```
const    tmp1,SpillHandler
consth   tmp1,SpillHandler
```



11031A-03

Figure 3. Initialized Stack



11031A-04

Figure 4. Current Processor Status Register in Reset Mode

Because each entry in the vector table is four bytes, we compute the address in the vector table by multiplying the vector number V_SPILL (64) by four (a shift left by two).

```
const tmp2,V_SPILL
sll   tmp2,tmp2,2  ;compute vector
                   ;address
```

Then we store the address of SpillHandler (in *tmp1*) into the vector table address we just computed.

```
store  0,0,tmp1,tmp2  ; write spill
                      ; vector
```

Initializing the vector table with the address of the underflow trap handler routine (vector number V_FILL) is done the same way:

```
const   tmp1,FillHandler
consth  tmp1,FillHandler
const   tmp2,V_FILL
sll     tmp2,tmp2,2    ; compute vect
                       ; addr
store   0,0,tmp1,tmp2 ; write fill
                       ; vector
```

The procedure start then calls main, passing it the return address (*lr0*). A NOP follows the call because the Am29000 always executes one instruction beyond a call instruction before the call is taken.

```
call  raddr,main
nop
halt     ;halt after successful
         ;completion
```

EXAMPLE FUNCTIONS MAIN() AND RECURSE()

After the start code has executed, control is passed to the procedure main(). The purpose of main() is to call the procedure recurse(), providing it with an initial set of values. Recurse() calls itself a total of 86 times, then returns to itself 86 times before returning to main(). An overflow condition occurs with the 21st call, and each subsequent call causes an additional spill of local registers to memory. When the program returns, the 22nd return causes an underflow condition, and each subsequent return causes an additional fill from memory to the local registers.

The basic operation of main() and recurse() is summarized by the following C program:

```
main()
{
    recurse(1,42);
}
recurse(n,m)
int n,m;
{
    int i,j;
    if (n > 85) return;
    i = n + 1;
    recurse(i,m);
}
```

The code for main() and recurse() is shown in Appendix A–3 and A–4, Code for Main() and Code for Recurse(), respectively.

PROLOGUE

As with all Am29000 procedures, main() begins with a prologue. The macro definition of PROLOGUE and the expansion of PROLOGUE for main() are shown in Appendix A–5 and A–6, Prologue Macro and Prologue Expansion for Main(), respectively.

The purpose of PROLOGUE is to allocate an activation record and check for overflow before the body of the procedure is executed. It is invoked with three parameters: the number of arguments passed (INCNT), the number of registers required for the procedure's local variables (LOCCNT), and the maximum number of arguments that the procedure may pass to any one function it in turn calls (OUTCNT).

```
.macro  PROLOGUE,INCNT,LOCCNT,OUTCNT
```

The values of ALLOC_CNT and SIZE_CNT are computed from the parameters.

```
.set  ALLOC_CNT,((2+OUTCNT+LOCCNT+1)&~1)
```

```
.set  SIZE_CNT,(ALLOC_CNT+2+INCNT)
```

ALLOC_CNT is the amount of space on the stack that must be newly allocated by the Prologue for the procedure's activation record. SIZE_A is the amount of space that must be accessible by the procedure, that is, the size of its activation record.

The expression for ALLOC_CNT does not use INCNT, because incoming parameters were already allocated space on the stack as the outgoing parameters (OUTCNT) of the calling procedure. "2" is the number of words needed for the called procedure's FP and return address when it calls another procedure. ANDing the expression with the complement of 1 (& ~1) maintains double-word alignment on the stack by setting the least significant bit to zero. The "+1" ensures that the amount is rounded up, not down.

The expression for SIZE_CNT includes INCNT and two additional words for *lr0* (return address) and FP of the caller.

The three macro variables, IN_PRM, LOC_REG, and OUT_PRM are used to establish offsets into the stack for input, local, and output arguments. These macro variables are set only if the corresponding value of the parameter is not equal to zero.

```
.if (INCNT)
    .set IN_PRM,  (2 +  ALLOC_CNT + 0x80)
.endif
.if (LOCCNT)
    .set LOC_REG,  (2 + OUTCNT + 0x80)
.endif
.if (OUTCNT)
    .set OUT_PRM,  (2 + 0x80)
.endif
```

In the above, a macro variable is set equal to an expression that is evaluated to a local register number when the program is assembled. The macro variable can then be used as the base register for offset addressing of parameters of that type (as shown in Figure 5). The "0x80" provides the 125-word offset required for a local register access.

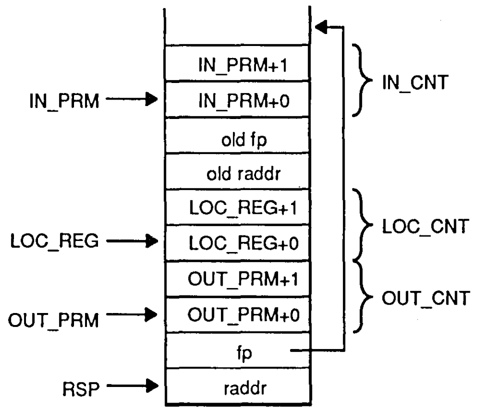


Figure 5. Prologue Parameters

The body of the PROLOGUE macro has three instructions:

```
sub      rsp, rsp, (ALLOC_CNT << 2)
asgeu    V_SPILL, rsp, rab
add      fp, rsp, (SIZE_CNT << 2)
```

In the above instructions, ALLOC_CNT and SIZE_CNT are shifted left by two to convert them from word quantities to the required byte quantities (the stack registers, whose contents will be modified, contain byte addresses).

The first instruction allocates an activation record by decrementing RSP by the amount ALLOC_CNT.

The second instruction asserts that RSP of the new activation record is greater than or equal to RAB. If this is not the case, (that is, RSP has been decremented beyond RAB), an overflow trap occurs, and there is a transfer of control to the trap handler routine, SpillHandler, pointed to by the vector V_SPILL. The trap handler will move (spill) the contents of the required number of local registers to the register stack in memory and adjust RFBand RAB, as described in the Overflow Trap Handler section.

The third instruction sets FP to point to the location just above the new activation record, so it can be used for underflow checking in the EPILOGUE macro of a procedure that is called by this procedure (see Epilogue section).

After the prologue, main() calls recurse(). The expansion of PROLOGUE for recurse() is shown in Appendix A–7, Prologue Expansion for Recurse().

OVERFLOW TRAP HANDLER

On the 21st call to itself, recurse() causes an overflow trap. The code that services this trap is shown in Appendix A–8, Overflow Trap Handler, and is described below.

In the following discussion of SpillHandler, we assume the reader is familiar with the processor's response to traps. If not, refer to the section called Interrupt and Trap Handling in the Am29000 User's Manual.

The first three *.reg* directives assign symbolic names to the three temporary system registers used by SpillHandler.

```
.reg     R_Cnt,%%(SYS_TMP+0)  ;temp for
                              ;count
.reg     R_TmpPC0,%%(SYS_TMP+1);temp for
                              ;PC0
.reg     R_TmpPC1,%%(SYS_TMP+2);temp for
                         ;PC1
```

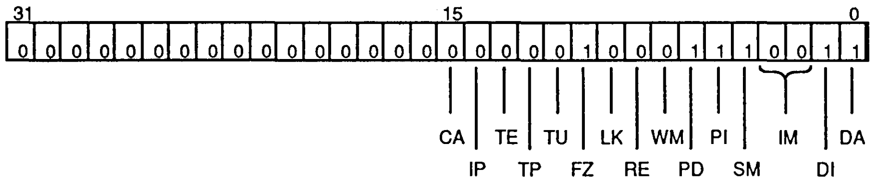
The old PCs are saved in two of the temporary registers just declared.

```
mfsr  R_TmpPC0, pc0        ;save the PCs
mfsr  R_TmpPC1, pc1
```

The CPS (Current Processor Status Register) is set to the value 0x73. This clears the FZ (Freeze) bit, which was set by hardware when the trap was taken (see Figure 6), so that the trap handler can execute a Store Multiple instruction. (Note that the PCs must be saved before the FZ bit is cleared.) The DA (Disable All Interrupts and Traps) bit remains set, which prevents the processor from taking any traps except the *WARN, Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps. PD, PI, SM, and DI also remain set.

```
mtsrim    cps,0x73     ; PD, PI, SM, DI, DA
```

Now we can use the Store Multiple instruction to store the required number of local registers into the register stack in memory. This instruction requires a source, a destination, and a count.



11031A-06

Figure 6. Current Processor Status Register After an Interrupt or Trap

As explained earlier (and shown in Figure 1), the area between RSP and RAB represents the local registers available for allocation when a procedure is called. Because there has been an overflow and RSP has been decremented beyond RAB, we can compute the size of the required spill (the count for the Store Multiple) by subtracting RSP from RAB.

```
sub  R_Cnt,rab,rsp    ;R_Cnt = number of
                      ;bytes to spill
```

Then we use R_Cnt to adjust RFB, so that it correctly reflects the area in the register stack that will be mapped to the local registers.

```
sub  rfb,rfb,R_Cnt    ;move down the
                      ;frame bound
```

Before using the Load Multiple instruction, R_Cnt must be written as a word amount into the CR field of the Channel Control register, which is used by the processor to determine the number of loads to memory. So we convert R_Cnt from a byte to a word amount using the Shift Right Logical instruction.

```
srl  R_Cnt,R_Cnt,2    ;R_Cnt = count of
                      ;words to spill
```

Because the CR field is zero-based, we subtract one from R_Cnt

```
sub  R_Cnt,R_Cnt, 1  ;correct for storem
```

and then use the Move to Special Register instruction to write it to the CR field.

```
mtsr  cr,R_Cnt        ;set up count for
                      ;storem
```

The local registers that have to be spilled are those corresponding to register-stack locations between RSP and RAB, because they are the local registers that must be occupied by the new activation record. So the instruction source will be *lr0*, which corresponds to RSP. The instruction's destination will be the register-stack location pointed to by the previously modified RFB, because that is the register-stack location at the correct 512-byte offset from RSP.

```
storem 0, 0, lr0, rfb ;spill from the
                      ;allocated area
```

Then we set RAB to point to the top of stack, because that is now the lowest stack address currently cached in local registers.

```
add     rab, rsp, 0   ;move down the
                      ;allocate bound
```

We set CPS to the value 0x473. This sets the FZ bit, which must be set before we restore PC0 and PC1. PD, PI, SM, DI, and DA remain set.

```
mtsrim  cps,0x473     ;FZ, PD, PI, SM,
                      ;DI, DA
```

Then the two PCs are restored and the IRET (Interrupt Return) instruction restores the previous contents of CPS from the Old Processor Status Register, unfreezes the processor, and begins fetching from PC0 and PC1.

```
mtsr   pc0, R_TmpPC0    ;restore the PCs
mtsr   pc1, R_TmpPC1
iret
```

EPILOGUE

When recurse has called itself 86 times, it returns and executes an Epilogue. The EPILOGUE macro is shown in Appendix A–9, EPILOGUE Macro.

EPILOGUE's first instruction de-allocates the procedure's activation record by adding ALLOC_CNT to RSP. This is followed by a NOP, because a change in the value of RSP must be separated by at least one cycle from an instruction that references a local register (in this case, the instruction JMPI, whose operand *raddr* is *lr0*).

```
add     rsp, rsp, (ALLOC_CNT << 2)
nop
jmpi    raddr
```

Before the Jump Indirect instruction finishes executing, the next instruction, ASLEU, is executed. This instruction asserts that the caller's FP, now restored because the caller's RSP has been restored, is less than or equal to RFB. If the assertion is false (which means that FP is pointing to an unmapped, previously spilled register-stack location), an underflow trap occurs, and control is transferred to the trap handler routine, FillHandler, pointed to by the vector V_FILL. The trap handler will move the contents of locations in the register stack to the local registers and adjust RAB and RFB, as described in the Underflow Trap Handler section.

```
asleu    V_FILL, fp, rfb
```

At the end of the Epilogue, the parameters are set to an illegal value. This ensures that if they are used again before they are explicitly set, an assembly-time error will be reported.

```
.set  IN_PRM, (1024)     ;illegal, to
                         ;cause
                         ;err on ref
.set  LOC_REG, (1024)    ;illegal, to
                         ;cause
                         ;err on ref
.set  OUT_PRM, (1024)    ;illegal, to
                         ;cause
                         ;err on ref
.set  ALLOC_CNT, (1024)  ;illegal, to
                         ;cause

                         ;err on ref
```

The expansion of EPILOGUE for recurse() is shown in Appendix A–10, Epilogue Expansion for Recurse().

UNDERFLOW TRAP HANDLER

On the 22nd return of recurse() to itself, an underflow trap occurs. The code that services this trap is shown in Appendix A–11, Underflow Trap Handler, and is discussed below.

The two old PCs are saved in temporary registers declared in the SpillHandler routine.

```
mfsr   R_TmpPC0, pc0    ;save the PCs
mfsr   R_TmpPC1, pc1
```

The CPS (Current Processor Status Register) is set to the value 0x73. This clears the FZ bit, so that the trap handler can execute a Load Multiple instruction. The DA bit remains set, which prevents the processor from taking any traps except the *WARN, Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps. PD, PI, SM, and DI also remain set.

```
mtsrim    cps, 0x73    ;PD, PI, SM, DI, DA
```

We will use the Load Multiple instruction to load locations in the register stack into the local registers. The Load Multiple instruction requires a source, a destination, and a count.

Clearly, the source for the Load Multiple instruction is the location pointed to by RFB, since RFB points to the first location in the register stack that was previously spilled from the local registers.

The destination of the Load Multiple instruction will, of course, be the local register corresponding to RFB. Local registers may be specified as instruction operands in one of two ways: using a local register number (in the range from 0 to 127), or using the absolute register number (in the range 126 to 255) in an Indirect Pointer Register. With the first method, the local register number is computed as a positive word offset of RSP. This option is not available to us because the trap handler has no way of knowing the offset from RSP (that is, the local register number) corresponding to RFB.

So we will convert the address in RFB to an absolute local register number, put this number in Indirect Pointer A (because the destination operand uses Indirect Pointer A), and then specify Global Register 0 (which indicates an indirect pointer access) as the destination register in the Load Multiple instruction.

To convert the address in RFB to an absolute local register number, we OR it with 512. This sets bit 9, which selects a local register; bits 2–8 give the absolute local register number.

```
const   R_Cnt,512          ;make local reg
                           ;ip
or      R_Cnt,R_Cnt,rfb    ;from rfb
```

Then we use the Move To Special Register instruction to put this value in the Indirect Pointer A Register.

```
mtsr    ipa, R_Cnt      ;set up indirect
                        ;ptr
                        ;for loadm
```

Recalling that the underflow trap was signaled because FP is pointing to an unmapped and previously spilled register stack location at a higher memory address than RFB, we can compute the number of local registers to fill by subtracting RFB from FP.

```
sub     R_Cnt, fp, rfb     ;R_Cnt = # of
                           ;bytes to fill
```

We use the just-computed value to adjust RAB, so that it correctly points to the new lower bound of the register stack mapped to local registers. We perform this operation now because it requires a byte amount, and R_Cnt will be converted to a word amount in the next instruction.

```
add     rab, rab, R_Cnt    ;move up the
                           ;allocate bound
```

Before use of the Load Multiple instruction, the count must be written as a word amount into the CR field of the Channel Control Register. Hence, we convert R_Cnt from a byte to a word amount using the Shift Right instruction.

```
srl   R_Cnt,R_Cnt,2   ;R_Cnt = number of
                      ;words to fill
```

Because the CR field is zero-based, we subtract one from R_Cnt

```
sub    R_Cnt, R_Cnt,1  ;correct for loadm
```

and then use the Move to Special Register instruction to write it to the CR field.

```
mtsr    cr, R_Cnt       ;set up count for
                        ;loadm
```

Now we use the Load Multiple instruction to transfer the contents of the register stack in memory to the local registers, specifying RFB as the address in the register stack from which to load, and *gr0* (Indirect Pointer A) as the local register number at which to begin the fill.

```
loadm    0,0,gr0,rfb    ;fill area freed
```

After the registers have been filled, we update RFB so that it correctly points to the upper bound of the register stack that is currently cached.

```
add    rfb,fp,0       ;move up frame bound
```

We set CPS to the value 0x473. This sets the FZ bit, which must be set before we restore PC0 and PC1. PD, PI, SM, DI, and DA remain set.

```
mtsrim  cps,0x473      ;FZ, PD, PI, SM,
                       ;DI, DA
```

Then the two PCs are restored and the IRET (Interrupt Return) instruction restores the previous contents of CPS, unfreezes the processor, and begins fetching from PC0 and PC1.

```
mtsr   pc0,R_TmpPC0     ;restore the PCs
mtsr   pc1, R_TmpPC1
iret
```

APPENDIX A: PARTIAL LISTINGS EXTRACTED FROM EXAMPLE PROGRAM

A–1. REGISTER DECLARATIONS

```
;------------------------------------------------------------------------
; Global registers
;------------------------------------------------------------------------
        .equ       SYS_TMP, 80                  ; system temp registers
        .reg       rsp, gr1                     ; local register stack pointer
        .reg       msp, gr125                   ; memory stack pointer
        .reg       rab, gr126                   ; register allocate bound
        .reg       rfb, gr127                   ; register free bound


;------------------------------------------------------------------------
; Local compiler registers
;  (only valid if frame has been established)
;------------------------------------------------------------------------
        .reg       fp, lr1                      ; frame pointer
        .reg       raddr, lr0                   ; return address


;------------------------------------------------------------------------
; Vectors
;------------------------------------------------------------------------
        .equ       V_SPILL, 64
        .equ       V_FILL, 65
```

A–2. START CODE

```
        .include   "regdcl.h"
        .equu      TOP_STK, (0x5000 & ~7)        ; create double word aligned value
        .text      .global start

start:
        .reg       tmp1, %%(SYS_TMP + 0)
        .reg       tmp2, %%(SYS_TMP + 1)
        const      rsp,(TOP_STK-8)               ; set stack ptr
        add        rsp,rsp,0                     ; set shadow rsp
        const      rab,(TOP_STK-512)             ; set reg alloc bound
        const      fp,TOP_STK                    ; set frame ptr
        const      rfb,TOP_STK                   ; set reg free bound

; set correct mode
        mtsrim     cps, 0x72                     ; PD, PI, SM, DI
        mtsrim     cfg, 0x10                     ; VF
        mtsrim     vab,0

; connect up spill handler
        const      tmp1,SpillHandler
        consth     tmp1,SpillHandler
        const      tmp2,V_SPILL
        sll        tmp2,tmp2,2                   ; compute vect addr
        store      0,0,tmp1,tmp2                 ; write spill vector

; connect up fill handler
        const      tmp1,FillHandler
        consth     tmp1,FillHandler
        const      tmp2,V_FILL
        sll        tmp2,tmp2,2                   ; compute vect addr
        store      0,0,tmp1,tmp2                 ; write fill vector

; call main program
        call       raddr,main
        nop
        halt                                     ; halt after successful completion
```

A–3. CODE FOR MAIN()

```
        .include    "regdcl.h"
        .include    "macros.h"
        .global     main

; main()
; {
; recurse(1,42);
; }

main:
        PROLOGUE    0,0,2                          ; invoke macro 0 ic, 0 loc, 2 og

; name outgoing args
        .reg        M_out_n, %%(OUT_PRM + 0)
        .reg        M_out_m, %%(OUT_PRM + 1)

; recurse(1,42)
        const       M_out_m, 42
        call        raddr,recurse
        const       M_out_n, 1

        EPILOGUE
```

A–4. CODE FOR RECURSE()

```
        .global    recurse

; recurse(n,m)
; {
; int i,j;
;
; if (n > 85) return;
; i = n + 1;
; recurse(i,m);
; }

recurse:

        PROLOGUE   2,2,2                          ; invoke macro 2 ic, 2 loc, 2 og

; name ic args
        .reg       R_in_n, %%(IN_PRM + 0)
        .reg       R_in_m, %%(IN_PRM + 1)

; name locals
        .reg       R_i, %%(LOC_REG + 0)
        .reg       R_j, %%(LOC_REG + 1)

; name outgoing args
        .reg       R_out_n, %%(OUT_PRM + 0)
        .reg       R_out_m, %%(OUT_PRM + 1)

; name temporary register
        .reg       R_tmp, lr0

; if (n > 85) return
        cpgt       R_tmp, R_in_n, 85
        jmpt       R_tmp,rec_01

; i = n + 1
        add        R_i, R_in_n, 1

; recurse(i,m)
        add        R_out_m, R_in_m, 0
        call       raddr, recurse
        add        R_out_n, R_i, 0

rec_01:
        EPILOGUE
```

A-5. PROLOGUE MACRO

```
; macro PROLOGUE
; Parameters: INCNT   input parameter count
;             LOCCNT  local register count
;             OUTCNT  output parameter count

         .set      ALLOC_CNT, ((2 + OUTCNT + LOCCNT + 1) & ~1)
         .set      SIZE_CNT, (ALLOC_CNT + 2 + INCNT)
         .set      IN_PRM, (2 + ALLOC_CNT + 0x80)
    .endif

    .if  (LOCCNT)
         .set      LOC_REG, (2 + OUTCNT + 0x80)
    .endif

    .if  (OUTCNT)
         .set      OUT_PRM, (2 + 0x80)
    .endif

         sub       rsp, rsp, (ALLOC_CNT << 2)
         asgeu     V_SPILL, rsp, rab
         add       fp, rsp, (SIZE_CNT << 2)
    .endm
```

A-6. PROLOGUE EXPANSION FOR MAIN()

```
main:
         PROLOGUE  0,0,2                        ; invoke macro
         .set      ALLOC_CNT, ((2 + 2 + 0 + 1) & ~1)
         .set      SIZE_CNT, (ALLOC_CNT + 2 + 0)
         .set      OUT_PRM, (2 + 0x80)
         sub       rsp, rsp, (ALLOC_CNT << 2)
         asgeu     V_SPILL, rsp, rab
         add       fp, rsp, (SIZE_CNT << 2)
```

A–7. PROLOGUE EXPANSION FOR RECURSE()

```
recurse:
        PROLOGUE   2,2,2                      ; invoke macro

        .set       ALLOC_CNT, ((2 + 2 + 2 + 1) & ~1)
        .set       SIZE_CNT, (ALLOC_CNT + 2 + 2)
        .set       IN_PRM, (2 + ALLOC_CNT + 0x80)
        .set       LOC_REG, (2 + 2 + 0x80)
        .set       OUT_PRM, (2 + 0x80)

        sub        rsp, rsp, (ALLOC_CNT << 2)
        asgeu      V_SPILL, rsp, rab
        add        fp, rsp, (SIZE_CNT << 2)
```

A–8. OVERFLOW TRAP HANDLER

```
        .reg       R_Cnt, %%(SYS_TMP + 0)     ; temp for count (shared)
        .reg       R_TmpPC0,%%(SYS_TMP + 1)   ; temp for PC0
        .reg       R_TmpPC1,%%(SYS_TMP + 2)   ; temp for PC1

        .global    SpillHandler

SpillHandler:

; This routine handles a false assertion in the standard prologue.
;
; In:     rab > rsp               (requiring an allocation)
;         lr1 <= rfb
;         rfb == rab + 512
;
; Out:    rab == rsp              (just enough allocated)
;         lr1 <= rfb
;         rfb = rab + 512

        mfsr       R_TmpPC0, pc0              ; save the PCs
        mfsr       R_TmpPC1, pc1
        mtsrim     cps, 0x73                  ; PD, PI, SM, DI, DA
        sub        R_Cnt, rab, rsp            ; R_Cnt = # of bytes to spill
        sub        rfb, rfb, R_Cnt            ; move down the frame bound
        srl        R_Cnt, R_Cnt, 2            ; R_Cnt = count of words to spill
        sub        R_Cnt, R_Cnt, 1            ; correct for storem
        mtsr       cr, R_Cnt                  ; set up count for storem
        storem     0, 0, lr0, rfb             ; spill from the allocated area
        add        rab, rsp, 0                ; move down the allocate bound
        mtsrim     cps, 0x473                 ; FZ, PD, PI, SM, DI, DA
        mtsr       pc0, R_TmpPC0              ; restore the PCs
        mtsr       pc1, R_TmpPC1
        iret
```

A–9. EPILOGUE MACRO

```
; macro EPILOGUE
.macro EPILOGUE
       add        rsp, rsp, (ALLOC_CNT << 2)
       nop
       jmpi       raddr
       asleu      V_FILL, fp, rfb
   .else
       jmpi       raddr
       nop
   .endif
       .set       IN_PRM, (1024)             ; illegal, to cause err on ref
       .set       LOC_REG, (1024)            ; illegal, to cause err on ref
       .set       OUT_PRM, (1024)            ; illegal, to cause err on ref
       .set       ALLOC_CNT, (1024)          ; illegal, to cause err on ref
   .endm
```

A–10. EPILOGUE EXPANSION FOR RECURSE()

```
EPILOGUE
add   rsp, rsp, (ALLOC_CNT << 2)
nop
jmpi  raddr
asleu V_FILL, fp, rfb
```

A-11. UNDERFLOW TRAP HANDLER

```
        .global    FillHandler

FillHandler:

;This routine handles a false assertion in the standard epilogue.
;
;In:    lrl > rfb              (requiring de-allocation)
;       rsp >= rab
;       rfb == rab + 512
;
;Out:   lrl == rfb             (just enough freed)
;       rsp >= rab
;       rfb = rab + 512

        mfsr       R_TmpPC0, pc0             ; save the PCs
        mfsr       R_TmpPC1, pc1
        mtsrim     cps, 0x73                 ; PD, PI, SM, DI, DA
        const      R_Cnt, 512                ; make local reg ip
        or         R_Cnt, R_Cnt, rfb         ;      from rfb
        mtsr       ipa, R_Cnt                ; set up indirect ptr for loadm
        sub        R_Cnt, lrl, rfb           ; R_Cnt = # of bytes to fill
        add        rab, rab, R_Cnt           ; move up the allocate bound
        srl        R_Cnt, R_Cnt, 2           ; R_Cnt = number of words to
        sub        R_Cnt, R_Cnt, 1           ; correct for loadm
        mtsr       cr, R_Cnt                 ; set up count for loadm
        loadm      0, 0, gr0, rfb            ; fill area freed
        add        rfb, lrl, 0               ; move up frame bound
        mtsrim     cps, 0x473                ; FZ, PD, PI, SM, DI, DA
        mtsr       pc0, R_TmpPC0             ; restore the PCs
        mtsr       pc1, R_TmpPC1
        iret
```

APPENDIX B: COMPLETE LISTING OF EXAMPLE PROGRAM

```
        .include    "regdcl.h"

        .equ        TOP_STK, (0x5000 & ~7)          ;create double word
                                                    ;aligned value

        .text
        .global start
start:
        .reg        tmp1,(SYS_TMP + 0)
        .reg        tmp2,(SYS_TMP + 1)

        const       rsp,(TOP_STK-8)                 ;set stack ptr
        const       rab,(TOP_STK-512)               ;set reg alloc bound
        const       fp,TOP_STK                      ;set frame ptr
        const       rfb,TOP_STK                     ;set reg free bound

        ;set correct mode
        mtsrim      cps, 0x72                       ;PD, PI, SM, DI
        mtsrim      cfg, 0x10                       ;VF
        mtsrim      vab,0

        ;connect up spill handler
        const       tmp1,SpillHandler
        consth      tmp1,SpillHandler
        const       tmp2,V_SPILL
        sll         tmp2,tmp2,2                     ;compute vect addr
        store       0,0,tmp1,tmp2                   ;write spill vector

        ;connect up fill handler
        const       tmp1,FillHandler
        consth      tmp1,FillHandler
        const       tmp2,V_FILL
        sll         tmp2,tmp2,2                     ;compute vect addr
        store       0,0,tmp1,tmp2                   ;write fill vector

        ;call main program
        call        raddr,main
        nop

        halt                                        ;halt after successful completion


;----------------------------------------------------------

;The routines below handle overflow and underflow conditions.
;The temps which they use are given below.

        .reg     R_Cnt,(SYS_TMP + 0)                ;temp for count (shared)
        .reg     R_TmpPC0,(SYS_TMP + 1)             ;temp for PC0
        .reg     R_TmpPC1,(SYS_TMP + 2)             ;temp for PC1
        .global SpillHandler
SpillHandler:
;This routine handles a failed assertion in the standard prologue
;
;In:rab > rsp(requiring an allocation)
;fp <= rfb
;rfb == rab + 512
;
;Out:rab == rsp(just enough allocated)
;fp <= rfb
;rfb == rab + 512

        mfsr      R_TmpPC0, pc0               ;save the PCs
        mfsr      R_TmpPC1, pc1

        mtsrim    cps, 0x73                   ;PD, PI, SM, DI, DA

        sub       R_Cnt, rab, rsp             ;R_Cnt = # of bytes to spill
        sub       rfb, rfb, R_Cnt             ;move down the frame bound
        srl       R_Cnt, R_Cnt, 2             ;R_Cnt = count of words to spill
        sub       R_Cnt, R_Cnt, 1             ;correct for storem
        mtsr      cr, R_Cnt                   ;set up count for storem
        storem    0, 0, lr0, rfb              ;spill from the allocated area
        add       rab, rsp, 0                 ;move down the allocate bound

        mtsrim    cps, 0x473                  ;FZ, PD, PI, SM, DI, DA

        mtsr      pc0, R_TmpPC0               ;restore the PCs
        mtsr      pc1, R_TmpPC1

        iret
;-----------------------------------------------------------------------

        .global FillHandler
FillHandler:

;This routine handles a failed assertion in the standard epilogue
;
;In:fp > rfb(requiring de-allocation)
;rsp >= rab
;rfb == rab + 512
;
;Out:fp == rfb(just enough freed)
;rsp >= rab
;rfb == rab + 512

        mfsr      R_TmpPC0, pc0               ;save the PCs
        mfsr      R_TmpPC1, pc1

        mtsrim    cps, 0x73                   ;PD, PI, SM, DI, DA

        const     R_Cnt, 512                  ;make local reg ip
        or        R_Cnt, R_Cnt, rfb           ;from rfb
        mtsr      ipa, R_Cnt                  ;set up indirect ptr for loadm
        sub         R_Cnt, fp, rfb              ;R_Cnt = # of bytes to fill
        add         rab, rab, R_Cnt             ;move up the allocate bound
        srl         R_Cnt, R_Cnt, 2             ;R_Cnt = number of words to fill
        sub         R_Cnt, R_Cnt, 1             ;correct for loadm
        mtsr        cr, R_Cnt                   ;set up count for loadm
        load        m0, 0, gr0, rfb             ;fill area freed
        add         rfb, fp, 0                  ;move up frame bound

        mtsrim      cps, 0x473                  ;FZ, PD, PI, SM, DI, DA

        mtsr        pc0, R_TmpPC0               ;restore the PCs
        mtsr        pc1, R_TmpPC1

        iret
;----------------------------------------------------------------------
```

Introduction to the Am29000 Development Tools Application Note

by Doug Kern and Douglas Walton

INTRODUCTION

The development of a microprocessor-based system is a complicated and detailed undertaking that requires skilled personnel and efficient test equipment. Because of the sophistication of modern microprocessing systems, they usually cannot be flawlessly designed on the first iteration, and nearly always require extensive debugging and testing time. Experienced developers know that few designs function perfectly at power-up. Faults occur due to erroneous logic, poor assembly, or defective parts, so some debugging is virtually always necessary. Therefore, every effort should be made to plan the debugging and testing process before the first prototype is built. Without advance planning, the designer may find that the circuit either cannot be successfully debugged, or that the necessary debug time is prohibitive.

Planners should keep in mind that testing and debugging continues throughout the life of the product. Because different phases in the product life cycle have different characteristics, the requirements for each must be considered. The major phases of the product life cycle are development, production (pilot, limited, and large-scale), and field service.

Apart from the skill of the personnel, the efficiency of test equipment is a critical area that affects the testing time in every phase. Outdated or ineffective equipment will slow down even the most highly trained personnel. More importantly, expensive, state-of-the-art test equipment will be wasted if its use is not preplanned. Careful consideration must be given to the type of equipment needed to service the product, as well as its cost and how it will be disbursed to the field.

AMD offers a comprehensive array of development tools that allow development teams to effectively test and debug Am29000™-based systems throughout the life cycle of the product. This document discusses those Am29000 development tools, and provides information for gauging their usefulness in specific applications with respect to cost, capabilities, and target design requirements.

Am29000 DEVELOPMENT TOOLS

The Am29000 development tools covered in this document are those used for debugging and testing actual system hardware. They normally are used with a prototype or production system to determine the cause of failure, and are distinguished from the 29K™ tools used to prepare programs for execution on a target system (see the 29K Tool Chain section).

Figure 1 shows the relationship of these development tools to the application and each other. The components are described below:

ADAPT29K—Advanced Development and Prototyping Tool. ADAPT29K™ is a standalone system that interfaces to the application like an in-circuit emulator. It provides a wide range of debugging functions without intruding on the application's execution.

MON29K—Target Resident Monitor. MON29K™ is a monitor program that executes on the target Am29000. It provides many of the same debugging functions as the ADAPT29K, even though it is a software product.

XRAY29K—Source-Level Debugger. XRAY29K™ is a source-level debugging program. It supplies an interactive, windowed environment for debugging Am29000 applications using MON29K or ADAPT29K.

Probe Interface. The Hewlett-Packard® probe interface provides an interface between the Am29000 and an HP 1650 or 16500 logic analyzer. When using a suitable logic analyzer, the probe interface allows the tracing of Am29000 signals with a 10-ns sample time and disassembly of Am29000 instructions.

Publication #	Rev.	Amendment	Issue Date:
12908	A	/0	11/89

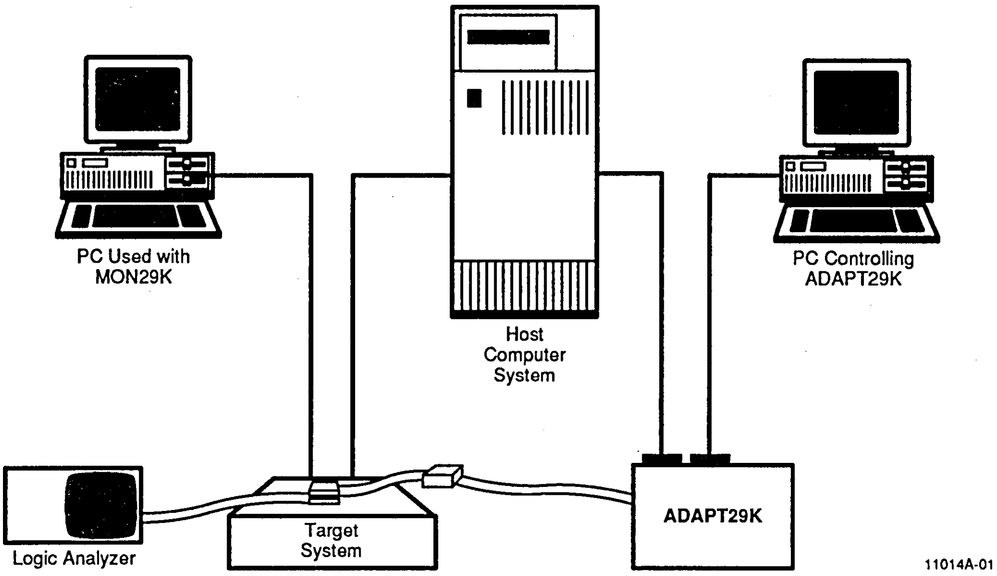


Figure 1. The Am29000 Development Tools

THE 29K TOOL CHAIN

The Am29000 development tools discussed in this document are a subset of the 29K tool chain, which are compatible resources provided by AMD for developing Am29000-based systems. Only the tools used for debugging are described in this document; other components of the 29K tool chain are needed to create the executable object modules that run on an Am29000-based system.

An object module can be obtained from the set of programs shown in Figure 2. Detailed information on using the tools to create an executable object module is contained in the following documents:

ASM29K Documentation Set. It provides complete information on the installation and use of the ASM29K™ assembler, linker, and librarian manager. It also includes documentation on the Am29000 utilities.

HighC29K Documentation Set. It covers how the HighC29K™ C compiler for the Am29000 is used.

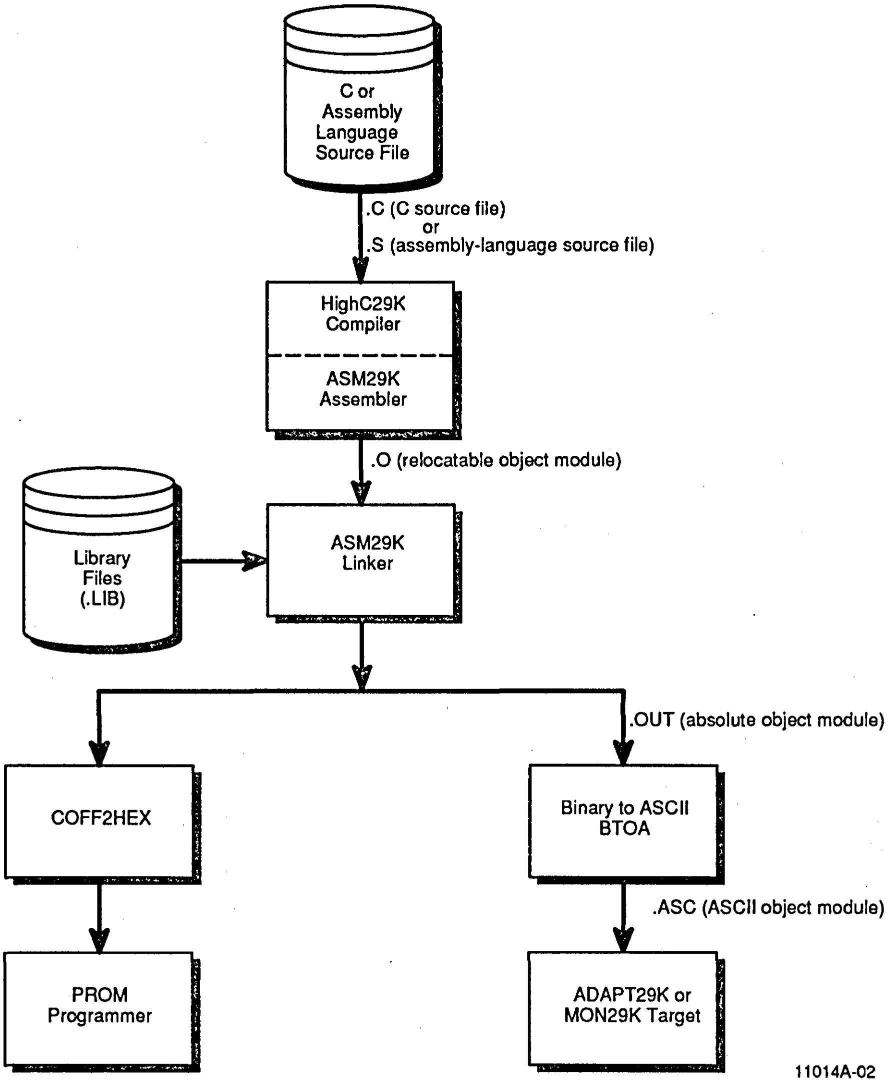

C or Assembly Language Source File
.C (C source file) or .S (assembly-language source file)
HighC29K Compiler
ASM29K Assembler
.O (relocatable object module)
Library Files (.LIB)
ASM29K Linker
.OUT (absolute object module)
COFF2HEX
Binary to ASCII BTOA
.ASC (ASCII object module)
PROM Programmer
ADAPT29K or MON29K Target
11014A-02

Figure 2. The 29K Tool Chain

REFERENCE MATERIALS

This document covers only information concerning critical requirements to consider during development planning. Detailed usage of each tool is not covered. Additional information can be found in the following documents:

ADAPT29K User's Manual. It provides detailed information on the ADAPT29K including installation, commands, theory of operation, and target design requirements.

MON29K Documentation Set. It provides detailed information on the MON29K including installation, commands, theory of operation, and target design requirements.

XRAY29K Documentation Set. This set of documents includes an installation guide, user's manual, and reference guide for XRAY29K, the high-level/assembly-language debugger.

Hewlett-Packard Probe Interface Data Sheet. It gives a description and electrical specifications for the probe interface.

These materials can be obtained by writing to:

Advanced Micro Devices, Inc.
901 Thompson Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

or by calling 1-800-222-9323.

For questions that cannot be resolved with the current literature, further technical support can be obtained by writing or calling:

29K Support Products Engineering
Mail Stop 561
5900 E. Ben White Blvd.
Austin, TX 78741
(800) 2929-AMD (US)
0-800-89-1131 (UK)
0-031-11-1129 (Japan)

HOW TO USE THIS DOCUMENT

This document discusses the Am29000 development environment. However, different readers have different requirements and initial levels of knowledge. The layout of this document should help readers locate the desired information while avoiding redundant or known material. In this document, special emphasis is placed on answering the questions:

1. What is the development tool?
2. Where does it fit in the 29K tool chain?
3. What capabilities does the development tool have?
4. What requirements must be met to effectively use the development tool with the target system?

The "Summary of the Tools" section summarizes the advantages and disadvantages of each development tool. Their compatibility requirements also are summarized.

The "Standalone Execution Board" section details the Standalone Execution Board (STEB) manufactured by STEP Engineering. The STEB is not actually a development tool, but an example of an Am29000 system that is compatible with all the development tools. The section highlights important areas of the development environment, demonstrating how the STEB was designed to comply with the compatibility requirements of the development tools.

Appendix A contains logic diagrams for the Standalone Execution Board. These should be used in conjunction with the discussion in the "Standalone Execution Board" section to show how the STEB was designed to comply with the compatibility requirements of the development tools.

ADAPT29K ADVANCED DEVELOPMENT AND PROTOTYPING TOOL

The ADAPT29K is a standalone unit used for non-intrusive supervision and monitoring of the target circuit, much like an in-circuit emulator. Completely self-contained, it has its own processor, memory, I/O, and power supply. It is connected to the target by a cable inserted between the Am29000 and its socket. When the target is running, the ADAPT29K monitors bus activity. When the target is halted, the ADAPT29K can use the target Am29000 to modify memory, provide processor status, or perform other debugging functions. Figure 3 shows the ADAPT29K.

Either an ASCII terminal or a host computer can be used to control the ADAPT29K. The commands have a format similar to the DEBUG program on the IBM® PC. When using an engineering workstation (running a terminal emulator program), screen logging facilities, file storage with uploading and downloading, and batch file support are available. Also, XRAY29K (see the "XRAY29K Source-Level Debugger" section) can be run on a mainframe or workstation, providing source-level debugging support. See Figure 4.

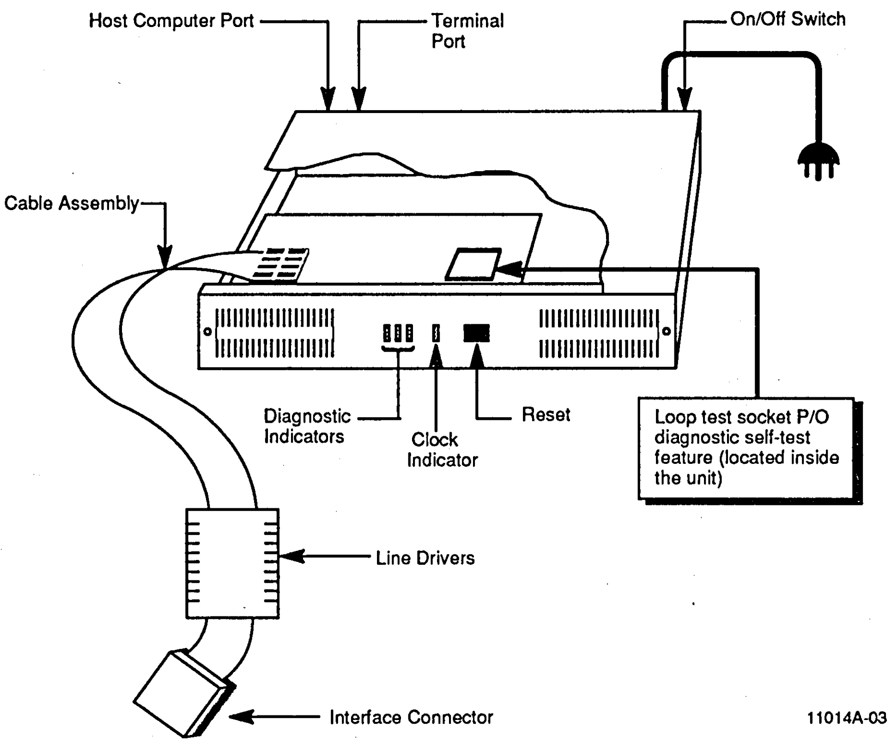


Figure 3. The ADAPT29K

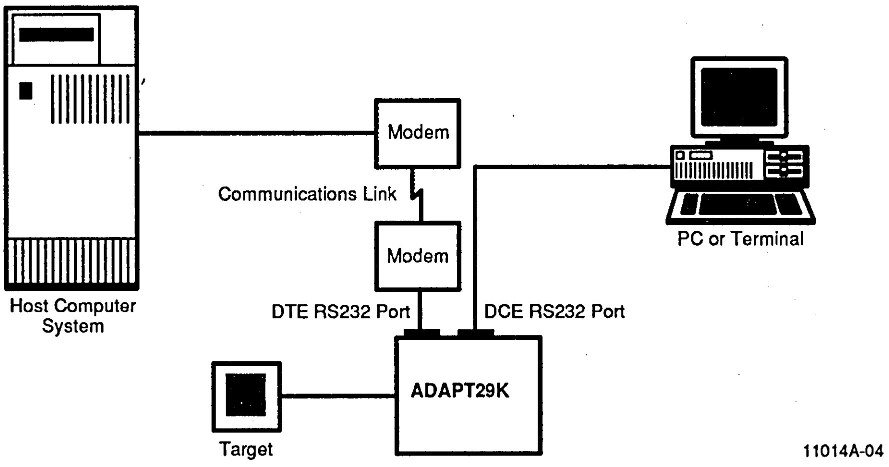


Figure 4. Connections to the ADAPT29K

One major advantage of the ADAPT29K is that, as a separate unit running on a separate processor from the target, hardware control signals can be asserted to gain control over the processor, regardless of the state of the program executing on the target. This allows the ADAPT29K to be used for debugging a system that cannot yet run its program. This type of debugging support is often useful when testing a prototype for the first time.

FEATURES

The ADAPT29K has powerful debugging capabilities that are important when bringing up a new design. For example, it is often necessary to inspect or alter memory contents, force test conditions, and patch in code sections. By using the ADAPT29K, the developer gains these capabilities for supervising the processor execution, thus greatly facilitating the initial debugging and development of Am29000-based applications.

Display and Modification of Memory

Using the ADAPT29K, all Am29000 memory spaces can be accessed. This includes instruction ROM, instruction/data RAM, Am29000 internal registers (global, local, and special), and coprocessor registers. Target data can be displayed or modified. The contents of a register or ranges of memory locations can be moved or filled; individual bits of special registers may be set separately. Table 1 shows the ADAPT29K commands available for managing memory.

Table 1. ADAPT29K Memory Display and Modification Commands

Command	Description
D	Display registers/memory
F	Fill registers/memory
I	Input from a port
M	Move memory
O	Output to a port
S	Set registers/memory
X	Display key registers
XC	Display/set coprocessor registers
XP	Display/set protected registers
XT	Display/set TLB registers
XU	Display/set unprotected registers

Memory operations can be performed in byte, half-word, word, floating-point, or double-precision format. For example, to display *lr4* through *lr11* as words, enter:

```
dw LR4,LR11
```

Or, to display addresses F0 to FF in instruction/data RAM, enter:

```
db 10000i, 1001fi
```

Figure 5 shows the results of these operations.

```
# DW LR4,LR11
LR004   61006200 63006400 65006600 67006800   a.b.c.d.e.f.g.h.
LR008   69006a00 6b006c00 6d006e00 6f007000   i.j.k.l.m.n.o.p.
#

# DB 10000I,1001FI
00010000I   61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00   a.b.c.d.e.f.g.h.
00010010I   69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00   i.j.k.l.m.n.o.p.
#
```

11014A-05

Figure 5. ADAPT29K Memory Displays

Several ADAPT29K commands make displaying of common memory groups easier. Frequently, when debugging specific areas of an application, the same data areas will need to be displayed repeatedly. For example, when testing a TLB-miss trap handler, it may be necessary to stop program execution after reloading the TLB to determine if the proper entry has been updated. The TLB entries can be displayed easily using the XT command, as shown in Figure 6.

Likewise, the processor status information contained in the special protected registers can be displayed using the XP command, as shown in Figure 7.

Often, the best time to examine memory locations is immediately after program execution has halted. A substantial amount of repetitive key entry can be eliminated by using the E command, which defines a command list that executes whenever the Am29000 halts. For example, to automatically perform the same operations shown in Figure 5 every time the Am29000 halts, the execution list could be defined as:

```
E DW LR4,LR11;DB 10000I,1001FI
```

The next time execution halts, the local registers *lr4* through *lr8* will be displayed, followed by a display of memory locations 000000F0 through 000000FF, just as it would have occurred if the commands had been entered individually.

```
# XT
LINE SET  1ST REG 0: VTAG   VE SR SW SE UR UW UE  TID    1:  RPN    PGM  U  F
 00   0    TR000       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 00   1    TR064       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 01   0    TR002       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 01   1    TR066       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 02   0    TR004       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 02   1    TR068       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 03   0    TR006       00000 0  0  0  0  0  0  0   00        000000  0   0  0
 03   1    TR070       00000 0  0  0  0  0  0  0   00        000000  0   0  0
#
```

11014A-06

Figure 6. TLB Entries Display

```
# xp
     CA IP TE TP TU FZ LK RE WM PD PI SM IM DI DA
CPS: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
OPS: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

VAB     CFG: PRL VF RV BO CP CD
0000          00  0  0  0  0  0

  CHA       CHD      CHC: CE CNTL CR LS ML ST LA TF TR NN CV
00000000 00000000        0   00  00 0  0  0  0  0  00 0  0

RBP: BF BE BD BC BB BA B9 B7 B6 B5 B4 B3 B2 B1 B0
     0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

  TCV TR: OV IN IE  TRV    PC0      PC1       PC2      MMU: PS PID LRU
000000    0  0  0  000000 00000000 00000000  00000000       0   00  0
#
```

11014A-07

Figure 7. Protected Register Display

Execution Control

The ADAPT29K can completely control target execution. Processing may be at full speed, or the target may be single-stepped, or it can be run until a breakpoint is encountered. Table 2 shows the ADAPT29K commands that control program execution.

Table 2. ADAPT29K Execution Control Commands

Command	Description
B	Breakpoint display, set, and reset
C	Check execution state
E	End execution command list
G	Go (start program execution)
K	Kill program execution
T	Trace (single step) instructions

Two types of breakpoints are available: "non-sticky" and "sticky." Non-sticky breakpoints are temporary breakpoints set as optional parameters of the G (Start Program Execution) command. They are reset when program execution stops. Fixed, or "sticky," breakpoints are set by using the B command. They remain in effect until they are expressly removed.

Debugging Support

Because the ADAPT29K was designed to aid debugging, it has several unique features that aid in testing the target. The testing aids include running memory diagnostics, assertion of repetitive signals, pulsing interface lines, and forced execution of Am29000 instructions. The commands are shown in Table 3.

Table 3. ADAPT29K Debugging Commands

Command	Description
A	Assemble in memory
J	Jam an instruction
L	List memory
P	Pulse the reset line
W	Run interface diagnostics
Z	Display trace buffer

The ADAPT29K's J command forces the processor to execute a user-specified Am29000 instruction. Issuing the P command pulses the processor reset line, initiating a hardware restart. Options of the W command specify various diagnostics to be executed, including a target memory test over a specified range of addresses; it also can be used to generate repetitive read and write signals for easy triggering of an oscilloscope.

Bus Tracing

A real-time bus trace facility is supported. Whenever the target Am29000 is executing, the ADAPT29K traces most CPU pins and stores their states in a 4096 entry ring buffer. All Am29000 signals are traced, except INCLK, SYSCLK, CNTL0, CNTL1, *TEST, and *RESET.

The state condition of the traced signals at each bus cycle is numbered sequentially and stored as an entry in the trace buffer. It may later be displayed to the terminal or host using the Z command shown in Table 3. A range of entries may be displayed in any of three formats. One (Figure 8) shows the disassembled instructions. Another (Figure 9) shows the states of the traced control signals. The remaining display is a combination of both figures.

```
# z 13
 Line        Address          Data      Instruction
 ----   ----------------    --------    -----------
-  13   ROM  RD  00009ef0    (none)     STORE       0,0x0,GR116,LR6
-  12   ROM  RD  00009ef4    (none)     LOAD        0,0x0,LR2,LR10
-  11   ROM  RD  00009ef8    (none)     ADD         LR3,LR11,0x0
-  10   ROM  RD  00009efc    (none)     CALL        LR0,.-0xCE0
-   9   DATA WR  00103fbc   00000000    (none)
-   8   DATA RD  00104528   0010442c    (none)
#
```

11014A-08

Figure 8. Bus Trace Display

```
# zc 13
               |        |         |   |    *       *|    *        |
          * * *|        |    *    |   |* I I   * * I|* D D   * * D|* * *
          B B B|        |    L M  |* *|I R B * I I B|D R B * D D B|W I T S
          R G I|        |R S O P O|C P|R E R P R E A|R E R P R E A|A N R T
          E R N|        |/ / C G P|D E|E Q E I D R C|E Q E D D R C|R T A A
 Line     Q T V|ADDRESS |W U K M T|A N|Q T Q A Y R K|Q T Q A Y R K|N R P T
-   13    1 1 1|00009ef0|1 1 1 0 0|1 1|0 1 0 1 0 1 1|1 0 1 1 1 1 1|1 f 3 7
-   12    1 1 1|00009ef4|1 1 1 0 0|1 1|0 1 0 1 0 1 1|1 0 1 1 1 1 1|1 f 3 7
-   11    1 1 1|00009ef8|1 1 1 0 0|1 1|0 1 0 1 0 1 1|1 0 1 1 1 1 1|1 f 3 7
-   10    1 1 1|00009efc|1 1 1 0 0|1 1|0 1 0 1 0 1 1|1 0 1 1 1 1 1|1 f 3 7
-    9    1 1 1|00103fbc|0 1 1 0 0|1 1|1 1 1 1 1 1 1|0 0 1 1 0 1 1|1 f 3 7
-    8    1 1 1|00104528|1 1 1 0 0|1 1|1 1 1 1 1 1 1|0 0 1 1 0 1 1|1 f 3 1
#
```

11014A-09

Figure 9. Control Signal Trace Display

Assembly/Disassembly

The ADAPT29K has a built-in, in-line assembler/disassembler that allows instruction memory examination and alteration using Am29000 mnemonics rather than hex values. The syntax corresponds to the ASM29K macro-assembler.

Serial Ports

The ADAPT29K has two serial ports. One is a data communications equipment (DCE) port; the other is a data terminal equipment port (DTE). Both ports conform to EIA convention RS232. Generally, a user gives commands to the ADAPT29K from an ASCII terminal or engineering workstation connected to the DCE port. (See Table 4 for a list of the commands.) A source-level debugger, such as XRAY29K (see the "XRAY29K Source-Level Debugger" section) running on a remote host, would use the DTE port.

Either port may be used to upload or download programs to the target. In this way, a user can control the ADAPT29K from an ASCII terminal while downloading programs from a remote host connected to the DTE port. Both Tektronix® Hex and Motorola® S3 formats are accepted. The ports can be connected together, enabling the terminal device to communicate with the remote host.

Table 4. ADAPT29K Serial Port Commands

Command	Description
N	Change the "normal character" (used to connect DCE and DTE ports)
R	Enter remote mode
V	Save memory to a file
Y	Load a file to memory

On-Line Help

On-line help is available for all commands. A command summary can be obtained by entering:

```
H <CR>
```

Specific help on an individual command can be obtained by entering H followed by the letter of the command. All command explanations show the complete command syntax and give a short description of how the command functions.

HOW THE ADAPT29K WORKS

The ADAPT29K runs on a different processor than the target. It performs all operations on the target by controlling the target Am29000. A buffered cable connects the ADAPT29K to the target's Am29000 socket. Figure 10 shows the signals carried on the cable. Note that although the ADAPT29K traces the address bus, it cannot drive it, and, consequently, cannot provide an overlay memory. It uses the target Am29000 to set up all memory addresses before it can access them.

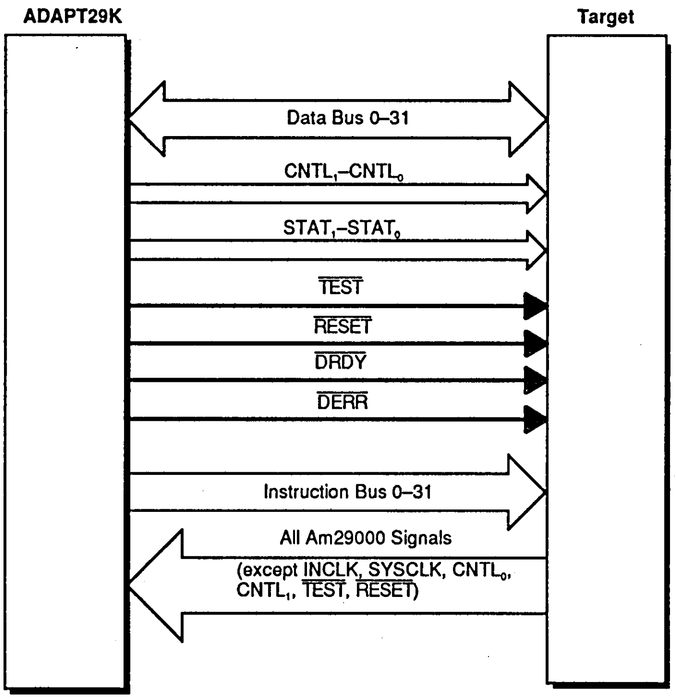
Execution Control

The execution state of the target Am29000 is controlled by using the CNTL0 and CNTL1 signals. By asserting different combinations of the two signals, the Am29000 can be placed in one of four states: RUN, HALT, STEP, and LOAD TEST INSTRUCTION. How these states affect the processor is explained in detail in the Am29000 User's Manual, order #10620.

The LOAD TEST INSTRUCTION state should be noted due to its importance to the ADAPT29K. Because the LOAD TEST INSTRUCTION state interrupts normal sequential processing and permits a sequence of instructions to be loaded into the processor's instruction stream, the ADAPT29K, using the LOAD TEST INSTRUCTION STATE, can force the processor to perform operations on the target.

Memory Access

Due to the high speed of the Am29000, the ADAPT29K, unlike some in-circuit emulators, does not provide any overlay memory. To maintain real access times, the processor must be kept as physically close to its memory as possible. There is no time available for the propagation delay that would be experienced in accessing memory across the interface cable to the ADAPT29K.



11014A-10

Figure 10. The ADAPT29K-to-Target Interface

All target code and data is stored on the target. When the ADAPT29K is commanded to display a data object, it places the target Am29000 in the LOAD TEST INSTRUCTION state. Then a sequence of instructions is inserted to store the present Am29000 state, set up a new memory address, load the data into an Am29000 register, store the data to the ADAPT29K, and restore the Am29000 state.

This method imposes certain requirements. Because data is transferred between the ADAPT29K and the target over the data bus, the target memory must be protected from corruption. To prevent inadvertent changes to the target memory, it must be disabled from responding when the ADAPT29K and the target processor are transferring data. There are two ways of doing this: (1) the memory can be disabled by a low state on the PIN169 alignment pin (pin D4), or (2) the target memory can be disabled when an 06 hex is decoded on the OPT_2–OPT_0 pins.

When the contents of instruction ROM must be displayed, the ADAPT29K must instruct the processor to read instruction ROM as data. Hence, a hardware path must exist for data stored in the instruction ROM space (on the instruction bus) to be loaded into an Am29000 register from the data bus.

Similarly, when the ADAPT29K is used to download a program, the code will be written word-by-word to the target Am29000, which then writes the instructions into proper memory space. Suppose, for example, code is to be written into the instruction/data RAM. Because the ADAPT29K has no means for virtual translation of addresses, it will use Store instructions to write the code into the absolute address in the instruction/data space. When the Am29000 goes to execute the code, it will expect to fetch its instructions over the instruction bus.

This requires that there be a hardware path from the data bus to the instruction bus and a one-to-one correspondence between addresses on the data bus and the addresses on the instruction bus. This occurs because the instruction is stored at an address on the data bus, but is fetched via the instruction bus. In other words, instructions fetched from an address in the instruction RAM space via the instruction bus must produce the exact information as would be retrieved from the same address in the data RAM space via the data bus.

Breakpoints

Because the Am29000 is one of the fastest commercial processors available, there is no practical way to read each address on the address bus and compare it against a breakpoint table to determine if a break should occur, as is done in an in-circuit emulator. The method used by the ADAPT29K is to swap a halt instruction into memory at the location of the breakpoint. When the executing processor encounters the breakpoint, it halts. Then, the ADAPT29K, upon detecting the halt, compares the halt address with the breakpoint table and determines if there is a match. If there is, it swaps the original instruction back into memory and informs the operator that a breakpoint has occurred.

This method of setting breakpoints also contributes to the requirement for a one-to-one translation of addresses between the data bus and the instruction bus. For example, when the ADAPT29K sets a breakpoint in the instruction ROM space, it does so by using the target Am29000 to read the original instruction, then writes the halt into the address location. This is performed as a data movement operation, using the bi-directional path to the instruction bus discussed in the Memory Access section. For the breakpoint to be effective, the executing program must encounter the breakpoint at the same address at which it was stored.

TARGET DESIGN REQUIREMENTS

Throughout the preceding discussion, it should be clear that the ADAPT29K only interfaces to the target via the target Am29000, and uses only the target memory for storage of the application program. This places certain hardware requirements on the application. These are listed below. For a specific example, see the Standalone Execution Board section.

1. The physical device in the instruction ROM space must be a RAM device if code is to be downloaded to the instruction ROM space, or if breakpoints will be set in the instruction ROM space.
2. A bi-directional path must exist between the instruction and data buses.
3. There must be a one-to-one translation between instruction bus addresses and data bus addresses.
4. The ADAPT29K must be able to disable the target memory using a low signal on the PIN169 alignment pin (D4), or when OPT_0–OPT_2 are 06 hex.
5. Physical clearance must be provided for the connection of the interface cable at its proper orientation.
6. Signals driven by the ADAPT29K (see Table 5) must be open-collector or tri-state.

Table 5. Am29000 Signals Driven by the ADAPT29K

Pin	Configuration
Alignment pin	(Input with pull-up resistor)[1,2]
D_{31}–D_0	(Tri-state)
I_{31}–I_0	(Tri-state)
$\overline{DERR}$	(Input with pull-up resistor)[1]
$\overline{RESET}$	(Open coll. pull-up with 1 K ohm resistor)
$\overline{DRDY}$	(pull-up resistor)[1]
$STAT_1$–$STAT_0$	(Input)
$\overline{TEST}$	(Open collector)[3]

1. Pull-up resistors should be 330 to 1000 ohms.
2. This is an optional configuration. It is used if memory will be disabled by the alignment pin (PIN169).
3. Note that $\overline{TEST}$ is active longer than $\overline{RESET}$. Since all outputs will be in a high-impedance state, it may be prudent to pull up all Am29000 outputs to avoid ambiguous inputs (to other devices).

MON29K TARGET RESIDENT MONITOR

MON29K is a target-resident monitor that has functionality similar to the ADAPT29K monitor. MON29K provides many important debugging capabilities, including memory display and alteration, code uploading and downloading, and assembly and disassembly. However, unlike the ADAPT29K, MON29K is an entirely software product. It resides completely in the target memory and executes on the target Am29000 (see Figure 11).

MON29K has I/O driver routines to handle two serial ports. Either port can be used to receive commands, although the hardware must be supplied by the target. With the proper hardware, MON29K can receive commands from an ASCII terminal or a remote host. It also can act as the interface between XRAY29K and the target. MON29K is supplied in C source code form so the I/O drivers and service routines can be modified to fit the particular hardware environment.

Since it is entirely software, MON29K can be permanently embedded in the product. It takes only 256K of address space in instruction ROM; thus, it can remain with the application and be used to diagnose problems at all stages of the product life cycle, from development to field support.

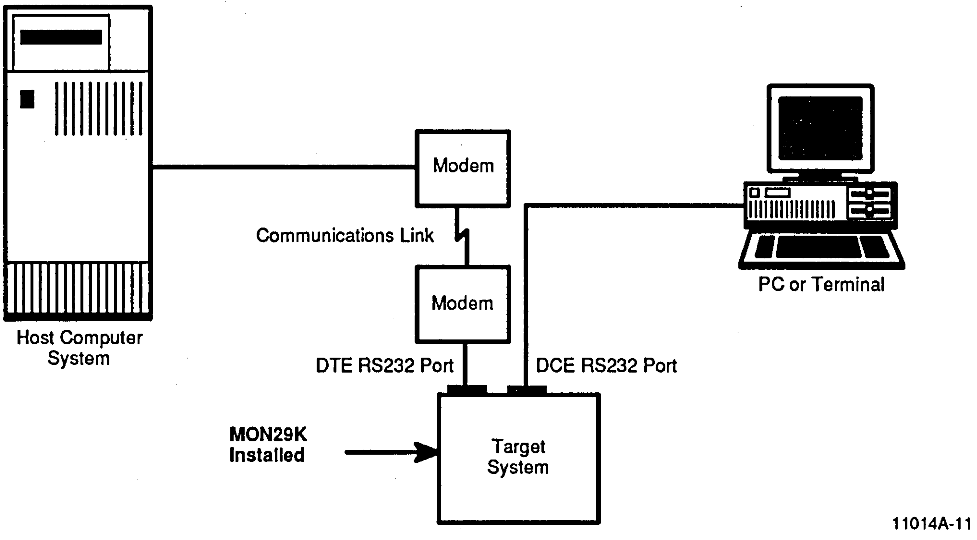


Figure 11. MON29K System Connections

FEATURES

MON29K provides powerful testing capabilities. Many of MON29K's features are, in fact, the same as the ADAPT29K. These include:

- *Display and alteration of memory, I/O ports, and registers.* Using MON29K, target data can be displayed, set, or altered. All Am29000 memory spaces may be accessed, including: Am29000 internal registers (global, local, and special), coprocessor registers, instruction/data RAM, or instruction ROM.
- *In-line assembly and disassembly.* MON29K comes with a built-in, in-line assembler/disassembler. Am29000 instruction mnemonics can be converted to machine codes and stored at a specified location, or ranges of addresses may be disassembled and displayed in mnemonic form.
- *Uploading and downloading of programs.* MON29K can use two serial ports, assuming they are provided by the target hardware. One port is a data communications equipment (DCE) port; the other is a data terminal equipment port (DTE). Files may be uploaded or downloaded in Motorola or Tektronix formats. Also, XRAY29K can communicate with MON29K through one of the ports.
- *Execution Control.* MON29K can control target execution. It can initiate full-speed execution, or single-step the processor.
- *Set/Reset Breakpoints.* Both permanent and temporary breakpoints are supported.
- *On-line help.* On-line help that shows the complete syntax is available for all commands.

MON29K Commands

Many of the MON29K commands (and consequently the features) are identical to those of the ADAPT29K. The MON29K commands, all of which are implemented in ADAPT29K, are listed in Table 6.

Table 6. MON29K Commands

Command	Description
A	Assemble in memory
B	Breakpoint display, set, and reset
C	Check execution state
D	Display registers/memory
E	End execution command list
F	Fill registers/memory
G	Go (start program execution)
I	Input from a port
L	List memory
M	Move memory
N	Change the "normal character"
O	Output to a port
R	Enter remote mode
S	Set registers/memory
T	Trace (single-step) instructions
V	Save memory to a file
X	Display key registers
XC	Display/set co-processor registers
XP	Display/set protected registers
XT	Display/set TLB registers
XU	Display/set unprotected registers
Y	Load a file to memory

Differences Between MON29K and ADAPT29K

Because MON29K runs on the target processor, not as a separate unit, it has limitations that the ADAPT29K does not have. In particular, MON29K has no K (Kill), S (Jam), Z (Trace), or W (interface diagnostics) commands.

MON29K is not able to assert a kill command because when the application is running, the application controls the processor. Clearly, when MON29K is not in control of the processor, it has no means of evaluating serial input and taking 29K polled the serial I/O device, but such continuous polling would hinder real-time execution. Instead, to allow programs to be forcefully terminated, MON29K can be configured to respond to interrupt-driven serial I/O. When MON29K is initialized to respond to interrupt-driven serial I/O, it intercepts a CTRL-C and passes control to a handler that recovers the processor to MON29K. This technique is effective in most cases, except if the application program has reached a HALT instruction. Then, the system must be reset. Usage of interrupt-driven serial I/O is determined as an option of the Q command (not present on the ADAPT29K).

TARGET DESIGN REQUIREMENTS

MON29K does place some requirements on the target design. They are listed below. For a sample implementation of the compatibility requirements, see the Standalone Execution Board section.

1. The physical part in the instruction ROM space must be a RAM device if the code will be downloaded to the instruction ROM space, or if breakpoints will be set in the instruction ROM space.
2. The Am29000 cannot write on the instruction bus, so a bi-directional path must exist between instruction and data buses.
3. Instruction bus addresses must produce the same data as data bus addresses.
4. As a target-resident monitor, MON29K does take up some of the target memory; thus, sufficient memory must be provided for MON29K. An application using MON29K must have 256 Kbytes of memory in the instruction ROM space for the program, and a 64-Kbyte workspace in instruction/data RAM. Both spaces must begin at address 0 (0r and 0d).
5. If program control must be recovered from the application before it ends or returns control normally, accommodations must be made to use interrupt-driven serial I/O. When interrupt-driven serial I/O is used, a MON29K interrupt routine will handle a CTRL-C by terminating the application program and returning control to MON29K.
6. MON29K expects the serial I/O driver to be an 8530 serial communications controller. Using a different I/O driver will require modifications to be made to MON29K.
7. AMD cannot anticipate every possible scenario in which the Am29000 will be introduced, and it is possible that MON29K will require some modifications to the I/O drivers and service routines before it can run on the target. Although binary code is available from AMD, MON29K is supplied in source code form. Of course, any changes will have to be compiled using a C compiler that produces object modules for the Am29000.

XRAY29K SOURCE-LEVEL DEBUGGER

XRAY29K, the high-level/assembly-level debugger, is a program that provides an interactive, windowed environment for debugging Am29000-based systems. Using XRAY29K, program statements may be read in source language, and data objects may be modified and changed by referencing symbol names. Thus, target operations can be performed using source-level constructs, rather than machine codes and numeric addresses. To further clarify the target environment, XRAY29K's multi-window interface simultaneously displays user-selected program information.

Commands are issued to XRAY29K using a comprehensive debugger command language. The language supports a wide range of functions, including setting breakpoints, single-stepping, and examining or altering any C- or assembly-language variables. The language syntax is very similar to C, and also supports debugging commands, creation of symbols during a debugging session, and convenient specification of address ranges.

XRAY29K resides on a host system and communicates with the target system through either the ADAPT29K or MON29K. Frequently, the host system is an engineering workstation attached to the ADAPT29K, as shown in Figure 12. In that system, XRAY29K provides a comfortable user-interface, while operations are asserted on the target by the ADAPT29K. Alternately, XRAY29K could reside on a mainframe and communicate with a target running MON29K. The user interface could then be done via an ASCII terminal.

FEATURES

XRAY29K supports source-level debugging in either of two modes: high-level or assembly-level. In high-level mode, an application can be debugged using C-language expressions and statements. In this way, C variables and expressions replace numeric addresses for memory access, and the code can be viewed by line number or procedure name.

In assembly-level mode, an application can be debugged using assembly-language statements. The assembly-level mode additionally allows machine-level register and status bit manipulation.

Commands are given to XRAY29K using its powerful debugger language, thus gaining access to XRAY29K's full range of debugging services. The services include:

- Setting and examination of memory and register contents using the declared format and the variable name.
- Simple and complex breakpoints that can be set and removed in either C-language or assembly-language source code.
- Single-step and full-speed program execution.
- Assembly and disassembly of object code.
- Simulated I/O and interrupts.
- Execution time measurement.

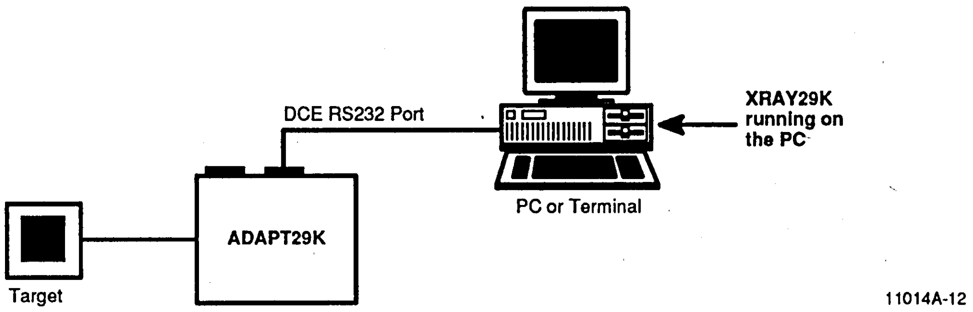


Figure 12. XRAY29K System Connections

The commands for manipulating memory and registers are shown in Table 7.

Table 7. XRAY29K Memory and Register Commands

Command	Description
compare	Compare two blocks of memory
copy	Copy a memory block
fill	Fill a memory block with values
search	Search a memory block for a value
setmem	Change the values of memory locations
setreg	Change a register's contents
test	Examine memory area for invalid values

Commands for controlling program execution are listed in Table 8; other display commands are listed in Table 9.

Table 8. XRAY29K Breakpoint and Execution Commands

Command	Description
breakinstruction	Set an instruction breakpoint
clear	Clear a breakpoint
go	Start or continue program execution
gostep	Execute macro after each instruction step
step	Execute a number of instructions or lines
stepnocall	Step, but execute through procedures

Table 9. XRAY29K Display Commands

Command	Description
disassemble	Display disassembled memory
dump	Display memory contents
expand	Display a procedure's local variables
find	Search for a string
fopen	Open a file or device for writing
fprintf	Print formatted output to a viewport
list	Display C source code
monitor	Monitor variables
next	Find string's next occurrence
nomonitor	Discontinue monitoring variables
printf	Print formatted output to command viewport
printvalue	Print a variable's value

Windowed Information Display

XRAY29K shows all critical program information at once in multi-windowed displays. The contents of the run-time stack, the selected general-purpose registers, the current source lines being executed, or virtually any other program information, can be checked at a glance, without the need to constantly request each piece of information individually.

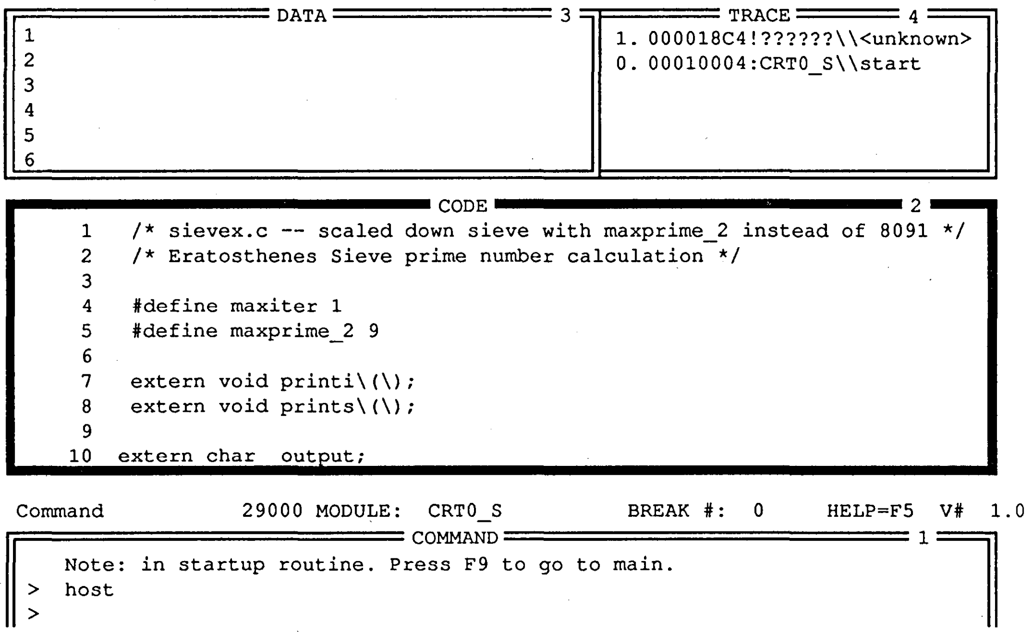
Information is grouped into screens, which are composed of one or more windows of specific data called viewports. There are three predefined screens: high-level, assembly-level, and standard I/O. Distributed among these screens are the 17 pre-defined viewports listed in Table 10.

Figure 13 shows the high-level mode screen display. It has four viewports: data, trace, code, and command. This screen is displayed when an object module generated by a C source program is executed.

Figure 14 shows the assembly-level mode screen display. It has five viewports: data, stack, disassembled code, registers (Am29000), and command. This screen is displayed when an object module generated by an assembly-language program is executed.

Table 10. XRAY29K Predefined Viewports

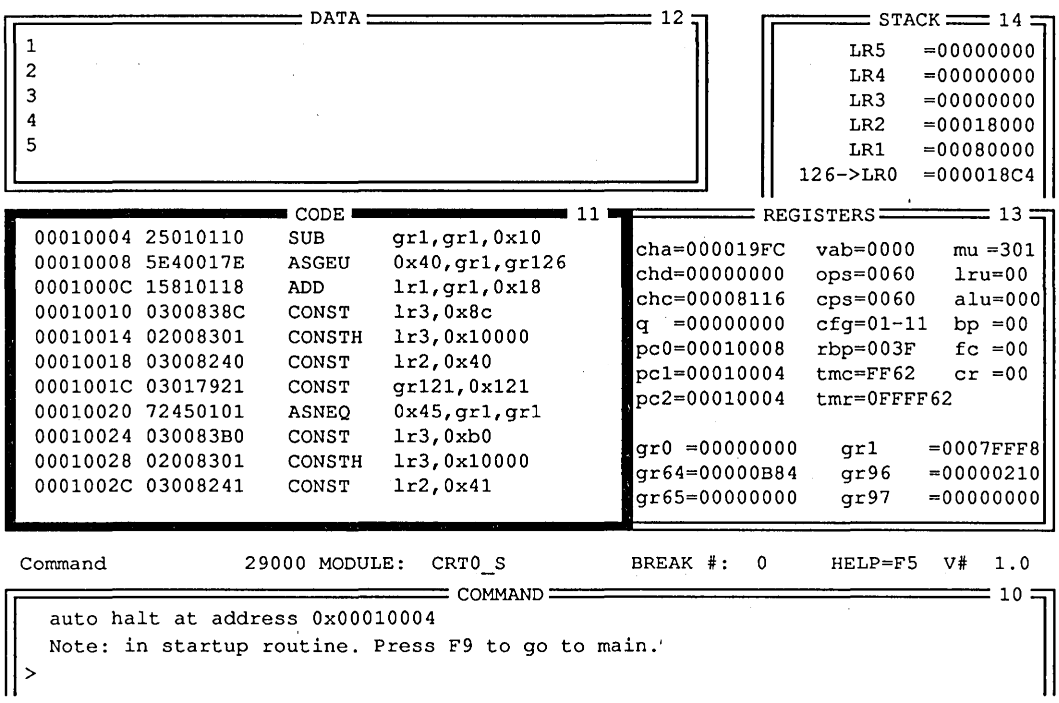
Viewport	Description
Command(2)	Debugger commands are submitted to XRAY29K from this viewport. There is a command viewport for both high-level and assembly-level modes.
Code(2)	Displays source code in high-level mode or disassembled instructions in assembly-level mode.
Data(2)	Displays monitored variable expressions in high-level and assembly-level mode.
Trace	Shows the procedure calling chain (high-level mode only).
Stack	Shows stack contents beginning from the stack pointer (assembly-level mode only).
Register	Displays current values of Am29000 registers (assembly-level mode only).
Status Line(2)	Used for debugger command information such as CPU type, current module name, and current operation. This viewport is present in both high-level and assembly-level modes.
Standard I/O	Shows interactive information being received from the **std.in** or sent to the **std.out**.
Break	Shows breakpoint information such as number, address, module name. Temporarily overlays top of screen when breakpoint is encountered.
Error	Appears when an error occurs to indicate type and source of error.
Help	Shows on-line help information when requested.
Log	Displays logged keystrokes.
Journal	Shows all previous commands and their results.



```
═══════════════ DATA ═════════════ 3 ══╗╔══════ TRACE ═══════ 4 ═══
1                                       1. 000018C4!??????\\<unknown>
2                                       0. 00010004:CRT0_S\\start
3
4
5
6

══════════════════════ CODE ══════════════════════════ 2 ══
   /* sievex.c -- scaled down sieve with maxprime_2 instead of 8091 */
   /* Eratosthenes Sieve prime number calculation */

   #define maxiter 1
   #define maxprime_2 9

   extern void printi\(\);
   extern void prints\(\);

  extern char  output;

Command         29000 MODULE:  CRT0_S        BREAK #:  0     HELP=F5  V#  1.0
═══════════════════════ COMMAND ══════════════════════ 1 ══
  Note: in startup routine. Press F9 to go to main.
> host
>
```

11014A-13

Figure 13. The Standard High-Level-Mode Screen



```
═══════════════════ DATA ══════════════════ 12      ═══════ STACK ═══ 14
1                                                         LR5    =00000000
2                                                         LR4    =00000000
3                                                         LR3    =00000000
4                                                         LR2    =00018000
5                                                         LR1    =00080000
                                                      126->LR0   =000018C4

═════════════ CODE ═════════════ 11   ══════════ REGISTERS ═══════════ 13
00010004 25010110  SUB     gr1,gr1,0x10       cha=000019FC  vab=0000   mu =301
00010008 5E40017E  ASGEU   0x40,gr1,gr126     chd=00000000  ops=0060   lru=00
0001000C 15810118  ADD     lr1,gr1,0x18       chc=00008116  cps=0060   alu=000
00010010 0300838C  CONST   lr3,0x8c           q  =00000000  cfg=01-11  bp =00
00010014 02008301  CONSTH  lr3,0x10000        pc0=00010008  rbp=003F   fc =00
00010018 03008240  CONST   lr2,0x40           pc1=00010004  tmc=FF62   cr =00
0001001C 03017921  CONST   gr121,0x121        pc2=00010004  tmr=0FFFF62
00010020 72450101  ASNEQ   0x45,gr1,gr1
00010024 030083B0  CONST   lr3,0xb0           gr0 =00000000   gr1    =0007FFF8
00010028 02008301  CONSTH  lr3,0x10000        gr64=00000B84   gr96   =00000210
0001002C 03008241  CONST   lr2,0x41           gr65=00000000   gr97   =00000000

Command            29000 MODULE:  CRT0_S          BREAK #:  0      HELP=F5   V#  1.0
═══════════════════════════════ COMMAND ═══════════════════════════════════ 10
  auto halt at address 0x00010004
  Note: in startup routine. Press F9 to go to main.'
>
```

11014A-14

Figure 14. The Standard Assembly-Level Mode Screen

The standard I/O screen has one regular viewport: the standard I/O viewport, although the breakpoint, error, and help viewports also will appear. The standard I/O screen is used when interactive input is requested from the standard input device, or when output is directed to the standard output device.

The viewport commands, shown in Table 11, control the way information is displayed on the screen. By using the viewport commands, a viewport's size, color, and cursor position can be changed. Viewports can be added or deleted, and custom screens and viewports can be defined.

Table 11. XRAY29K Viewport Commands

Command	Description
vactive	Activate a viewport
vclear	Clear data from a viewport
vclose	Remove user-defined viewport or screen
vcolor	Select viewport colors
vmacro	Attach a macro to a viewport
vopen	Create a screen or viewport or change size
vsetc	Set a viewport's cursor position
zoom	Increase or decrease a viewport's size

Utility Functions

In addition to its powerful features for execution control and display of system information, XRAY29K provides several utility features. These features ease debugging by streamlining the routine operations. The services include command keys, macros, and command files.

Command Keys

The most frequently used XRAY29K functions have been assigned to a key combination referred to as a "command key." By using command keys, common debugger commands can be entered with the minimum number of keystrokes, often only one key or a CTRL-key combination.

Macros

XRAY29K has a powerful, multifaceted macro facility. Because a macro may contain complex user command procedures, which are executed by entering the macro name on the command line, the facility can be used for several purposes. Table 12 shows the debugging language's macro-related commands.

Table 12. Macro Commands

Command	Description
define	Create a macro
show	Display the macro source

Macros can be invoked when a breakpoint is encountered. Powerful conditional and looping statements in the command language allow the macro to evaluate program or register variables, and alter program flow depending on their condition. Hence, macros can be used to establish very complex breakpoints that take specific action, depending on their environment.

Macros also can be attached to user-defined viewports. When the associated window is opened, the macro will execute. This type of macro can write specific data into the window, which is useful for monitoring environmental information.

Command/Batch Files

XRAY29K can process command files. A command file contains one or more debugger commands that can be processed by XRAY29K automatically, without the need for user interaction. This is also called batch-mode operation. Command files can be used to recreate a debugging session, easily implement automated test procedures, and eliminate reentering of frequently used command sequences.

Other XRAY29K Utility Functions

XRAY29K possesses several other utility functions. These include services for manipulating symbols, evaluating expressions, setting display and recording modes, and controlling the session. Table 13 lists the symbol commands, Table 14 lists the miscellaneous utility commands, and Table 15 lists the session commands.

Table 13. Symbol Commands

Command	Description
add	Create a symbol
delete	Delete a symbol from the symbol table
printsymbols	Display symbol, type, and address
scope	Specify current module and procedure scope

Table 14. Miscellaneous Utility Commands

Command	Description
cexpression	Calculate an expression's value
erro	Set include file error handling
help	Display on-line help screen
include	Read in and process a command file
log	Record debugger commands and errors in a file
mode	Select debugger mode (high-level or assembly-level)
option	Set debugger options for this session
pause	Pause simulation
reset	Simulate processor reset
restart	Reset the program starting address
startup	Save the default start-up options

Table 15. Session Command

Command	Description
host	Enter the host operating system environment
load	Load an object module for debugging
quit	End a debugging session

TARGET DESIGN REQUIREMENTS

XRAY29K itself places no restrictions on the target hardware design. However, being strictly a software product, XRAY29K needs a hardware connection to the target. For debugging Am29000-based systems, XRAY29K must be used in conjunction with either ADAPT29K or MON29K; the target design requirements for those tools apply.

XRAY29K requires a host system. Versions of XRAY29K currently exist for UNIX and DOS environments.

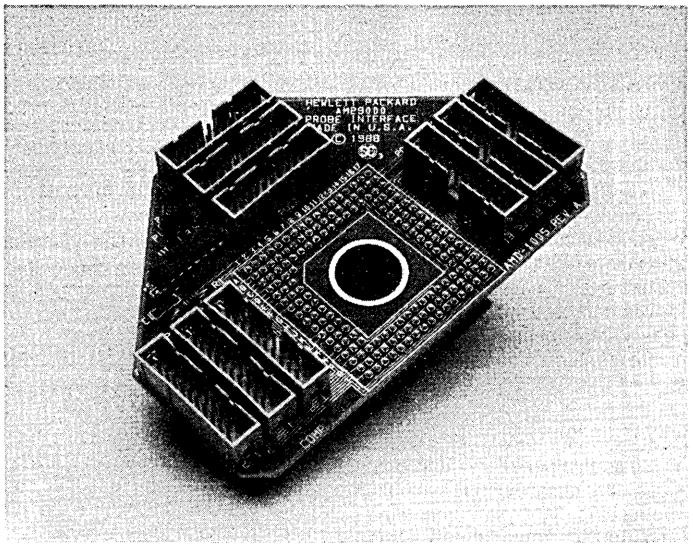
XRAY29K works only with object files that have been compiled in such a way that they contain debugger information regarding line numbers, etc. Thus, to use XRAY29K, either the ASM29K macro-assembler or HighC29K cross-compiler must be used, as well as the ASM29K linker. These are explained in the "29K Tool Chain" section.

Am29000 PROBE INTERFACE

The Am29000 probe interface provides a non-intrusive, low-capacitance connection to an Am29000. Inserted between the processor and its socket, the probe interface makes the Am29000 pins available for convenient attachment to a logic analyzer or other test equipment. Figure 15 shows the probe interface.

The software available with the probe interface supplies configuration information about the Am29000 pins and instruction mnemonics to either an HP 1650 or 16500 logic analyzer for display formatting. When the display is formatted, the logic analyzer will disassemble instructions into mnemonics and display processor, bus, and error status, as well as data bus activity. Figure 16 shows how the probe interface is connected between the logic analyzer and the target.

Although the probe interface was designed for the HP 1650 or 16500 logic analyzer, any type of test equipment can be attached to it. The following discussion assumes a connection to an HP 1650 or 16500 logic analyzer, unless otherwise stated.



11014A-15

Figure 15. The Probe Interface

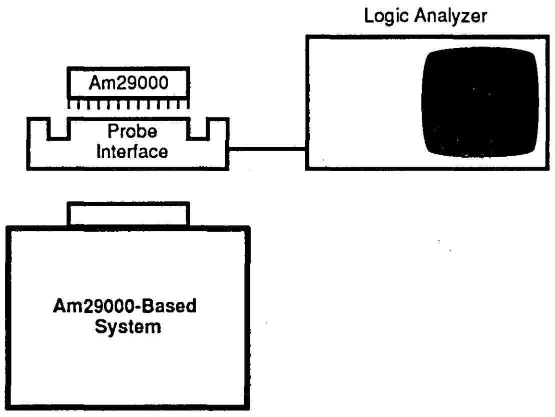


Figure 16. Connection of the Probe Interface

FEATURES

The probe interface can add important event-triggering and high-speed (10 ns) resolution capabilities, including:

- Convenient connection to the target.
- Low-capacitance probing.
- Completed status information, including identification of burst, pipeline, and simple accesses.
- Status reporting of bus conditions, such as slave accesses, wait states, and co-processor transfers.
- User-configurable setup and hold parameters allow triggering on a specific target condition.
- Monitoring of all Am29000 signals except INCLK.

The probe interface comes with the disassembler, configuration files, and a user's manual.

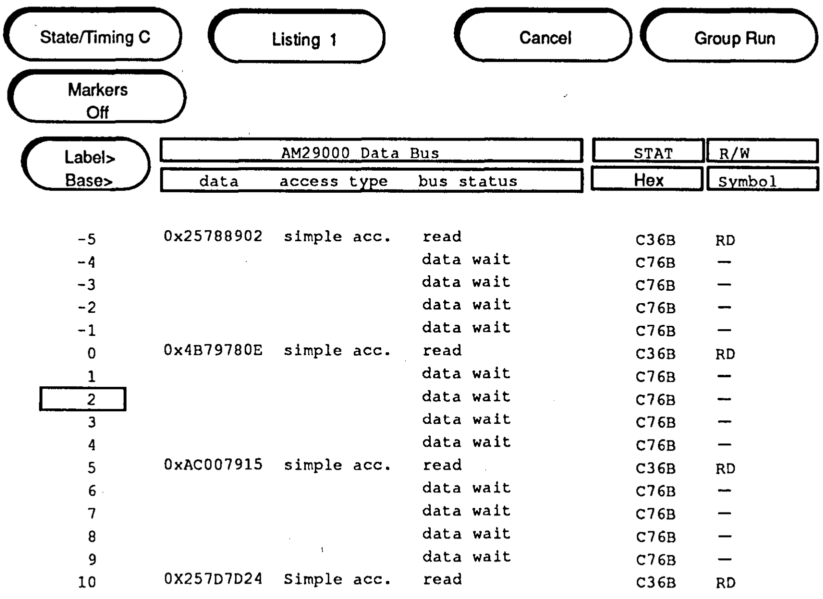
DISPLAYS

Figure 17 shows data bus information, as would be shown on an HP 16500 logic analyzer. Figures 18 and 19 show signal state and timing screens and the disassembly screen for the 16500 analyzer.

TARGET DESIGN REQUIREMENTS

Because the probe interface only monitors Am29000 signals, there are no particular target compatibility requirements except for sufficient clearance to install the probe interface. Most applications will not be affected by low-capacitance, high-impedance connection; however, see the probe interface data sheet for electrical and physical specifications.

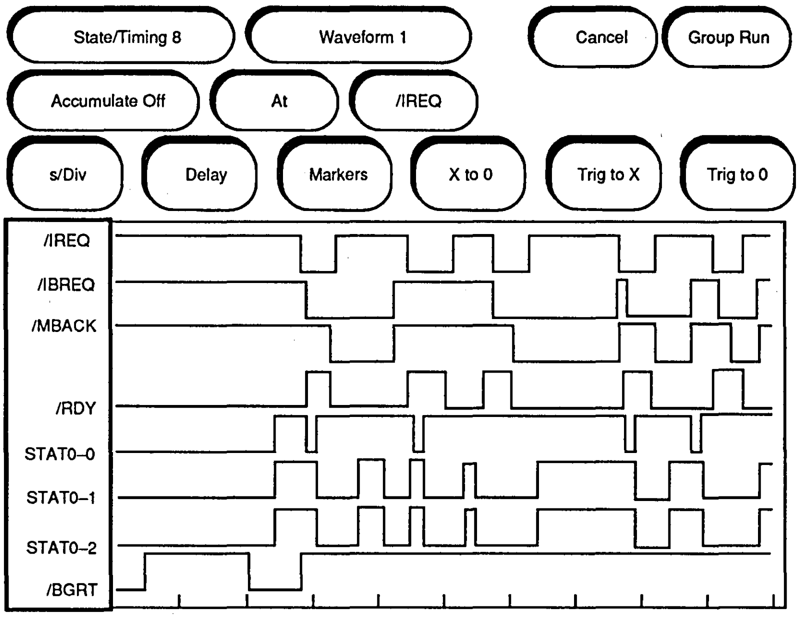
Apart from supporting the physical size and electrical specifications of the connection, a logic analyzer is needed. The logic analyzer should have 80 to 160 state channels. Some termination adapters also are needed, depending on the number of state channels on the logic analyzer.



```
State/Timing C    Listing 1                 Cancel    Group Run

Markers
Off

Label>   AM29000 Data Bus                         STAT   R/W
Base>    data     access type   bus status        Hex    Symbol

 -5   0x25788902  simple acc.   read               C36B   RD
 -4                             data wait          C76B   —
 -3                             data wait          C76B   —
 -2                             data wait          C76B   —
 -1                             data wait          C76B   —
  0   0x4B79780E  simple acc.   read               C36B   RD
  1                             data wait          C76B   —
  2                             data wait          C76B   —
  3                             data wait          C76B   —
  4                             data wait          C76B   —
  5   0xAC007915  simple acc.   read               C36B   RD
  6                             data wait          C76B   —
  7                             data wait          C76B   —
  8                             data wait          C76B   —
  9                             data wait          C76B   —
 10   0X257D7D24  Simple acc.   read               C36B   RD
```

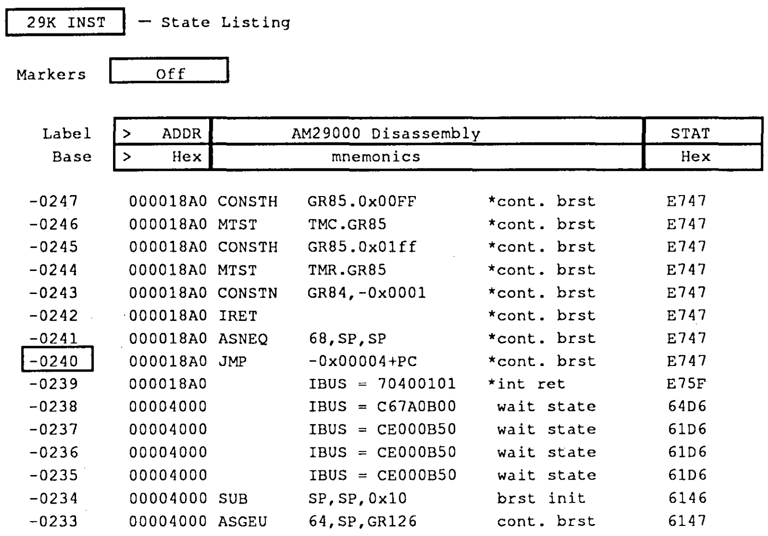
11014A-17

Figure 17. HP 16500 Data Bus Information Display



11014A-18

Figure 18. HP 16500 Signal and Timing Display



```
 29K INST  — State Listing

Markers   Off

 Label  > ADDR  AM29000 Disassembly                              STAT
 Base   > Hex   mnemonics                                         Hex

 -0247   000018A0 CONSTH  GR85.0x00FF        *cont. brst        E747
 -0246   000018A0 MTST    TMC.GR85           *cont. brst        E747
 -0245   000018A0 CONSTH  GR85.0x01ff        *cont. brst        E747
 -0244   000018A0 MTST    TMR.GR85           *cont. brst        E747
 -0243   000018A0 CONSTN  GR84,-0x0001       *cont. brst        E747
 -0242   000018A0 IRET                       *cont. brst        E747
 -0241   000018A0 ASNEQ   68,SP,SP           *cont. brst        E747
 -0240   000018A0 JMP     -0x00004+PC        *cont. brst        E747
 -0239   000018A0         IBUS = 70400101    *int ret           E75F
 -0238   00004000         IBUS = C67A0B00     wait state        64D6
 -0237   00004000         IBUS = CE000B50     wait state        61D6
 -0236   00004000         IBUS = CE000B50     wait state        61D6
 -0235   00004000         IBUS = CE000B50     wait state        61D6
 -0234   00004000 SUB     SP,SP,0x10          brst init         6146
 -0233   00004000 ASGEU   64,SP,GR126         cont. brst        6147
```

11014A-19

Figure 19. HP 16500 Disassembly Listing

SUMMARY OF THE TOOLS

From the sections on ADAPT29K, MON29K, XRAY29K, and the probe interface, it should be clear that a comprehensive range of tools exists for developing Am29000-based systems. Each of the available tools has unique characteristics that make it more advantageous in particular situations. Depending on the characteristics of the application, one or all of the tools may be needed. This section summarizes the information presented in the previous sections with emphasis on highlighting what conditions are most appropriate for a particular tool or tool combination, and what compatibility requirements are placed on the target as a result of the tool selection.

SELECTION GUIDE

In the development phase of virtually any Am29000-based system, either the ADAPT29K or MON29K will be needed. It is possible to debug a microprocessor system with only a logic analyzer and a PROM programmer, but this method is not very practical when compared against the following ADAPT29K and MON29K features:

- Memory display and modification, including special registers.
- Uploading and downloading of programs.
- Execution control, including setting breakpoints and single-stepping.

Apart from the advantages gained from MON29K and the ADAPT29K, their performance can be augmented in certain situations if they are combined with XRAY29K and/or the probe interface with a logic analyzer. The following questions highlight the critical target characteristics that suggest the optimum tool selection.

How much memory does the target have?

Perhaps the most crucial factor in deciding whether the ADAPT29K or MON29K is most appropriate depends on the size of the available target memory. This determines whether or not MON29K can be used. Because MON29K is target resident, it is necessary that the target have at least 256 Kbytes of space in instruction ROM, and 64 Kbytes of instruction/data RAM for MON29K's workspace. An application without this memory space will not be able to use MON29K, and will have to use the ADAPT29K.

For systems with sufficient memory, MON29K, ADAPT29K, or both may be used. While both have excellent debugging features, the ADAPT29K has some features MON29K does not, including:

- Can halt a failing program
- Provides a bus trace facility
- Can force execution of an Am29000 instruction
- Provides memory diagnostics
- Can be used with a target that cannot run its program

It should be noted that in most cases (see the Differences Between MON29K and ADAPT29K section), MON29K can halt a crashed program if an interrupt-driven serial I/O is provided on the target, and the target still is responding to interrupts.

How many units will be produced?

The number of units to be produced determines the volume over which the development and servicing costs can be defrayed. The ADAPT29K, while more powerful than MON29K, costs more and will raise the amount of nonrecurring charges that must be recovered. Of course, the difference will be insignificant for the advantages gained in large volumes. In fact, it may be advisable to use the ADAPT29K when the product is in development and final test, using MON29K for field service.

How and where will servicing be performed?

Servicing can be performed on-site or at service centers. Often this depends on the size, function, and value of the application system. If the system is moved to a service center for repair, the ADAPT29K will provide the most capabilities, particularly when coupled with the probe interface and XRAY29K.

However, the ADAPT29K may be too bulky to perform maintenance on-site. MON29K can be embedded in the application and used to diagnose faults via a portable ASCII terminal or PC (which could run XRAY29K).

How complex is the program?

If the program is complex, XRAY29K should be considered. Debugging complex programs using hex values and physical addresses can be very time consuming and error prone, especially programs containing many modules. Often, XRAY29K's windowed interface and source-level debugging language will greatly reduce time spent tracking down errors encountered in address calculations, decimal to hex conversions, or just looking up values in a listing.

SUMMARY OF COMPATIBILITY REQUIREMENTS

Once a combination of tools has been selected, it is important to ensure that they will be compatible with the target system. The following lists summarize the compatibility requirements for each tool. More detailed explanations can be found in the specific sections related to the particular tool.

ADAPT29K

1. The target must support RAM in instruction ROM.
2. A bi-directional path must exist between the instruction and data buses.
3. There must be a one-to-one translation of addresses between buses.
4. Target memory must be disabled either by a low signal on the alignment pin (D4), or when OPT_2–OPT_1 are 06 hex.
5. There must be physical clearance for the connection of the interface cable at the proper orientation.
6. The signals driven by the ADAPT29K must be open-collector or three-state.

MON29K

1. The target must support RAM in instruction ROM.
2. A bi-directional path must exist between the instruction and data buses.
3. There must be a one-to-one translation of addresses between buses.
4. The system memory must include 256 Kbytes in instruction ROM beginning at Address 0 to store the MON29K program, and 64 Kbytes of instruction/data RAM at Address 0 for MON29K's workspace.
5. If program control must be recovered from the application without it ending or returning control normally, accommodations must be made to use interrupt-driven serial I/O.
6. The I/O drivers may have to be modified.

XRAY29K

1. Requires a host system, such as an engineering workstation.
2. Requires MON29K or ADAPT29K.

Probe Interface

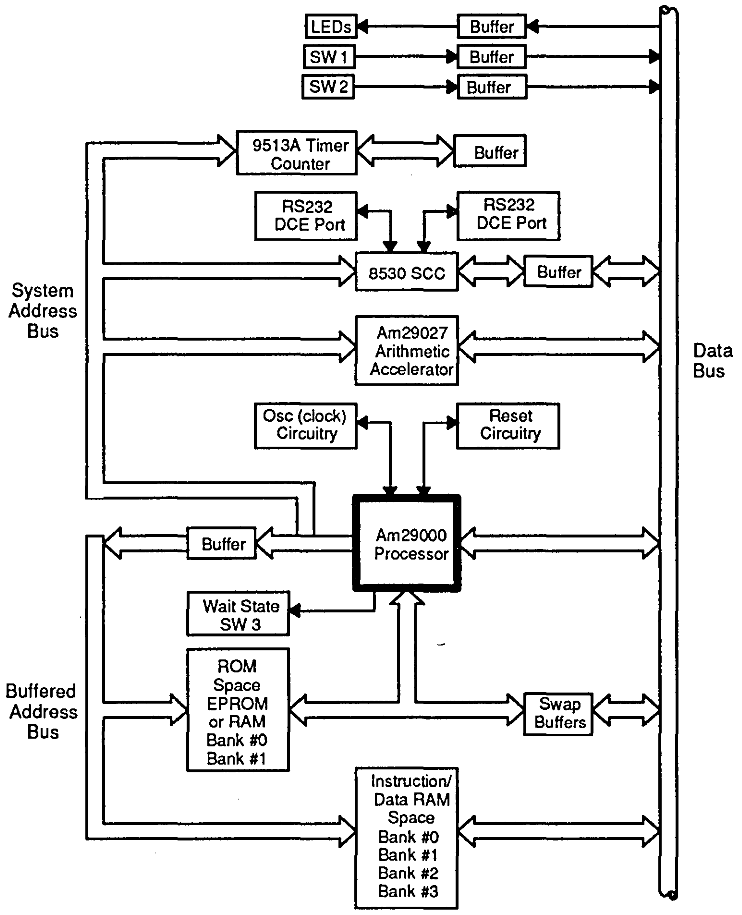
1. Requires a logic analyzer (an HP 1650 or 16500 is recommended).
2. Requires termination adapters.
3. There must be sufficient physical clearance to allow the probe to be attached to the target.

A COMPATIBILITY EXAMPLE: STANDALONE EXECUTION BOARD

The Standalone Execution Board (STEB) is an excellent example of compatibility with all the development tools. It is a complete Am29000-based system that can run many types of programs, including the software packages MON29K and VRTX32/29000®.

The STEB can also be used with the ADAPT29K and/or the HP probe interface. STEB also can be used as an execution vehicle for application software or a comparison system for isolating hardware faults.

This section focuses on how the STEB's design achieves compatibility with the development tools. The major areas of the STEB are discussed, with emphasis on how each area contributes to compatibility. See Figure 20 for a block diagram of the STEB.



11014A-20

Figure 20. Block Diagram of the STEB

FUNCTIONAL DESCRIPTION

Mounted on a single card, the STEB contains an Am29000 with memory, I/O, and system timing resources. See Appendix A for schematic diagrams, Sheets 1 through 12. In addition to the Am29000 (U51 on Sheet 2), the STEB supports the Am29027 arithmetic accelerator (U10 on Sheet 3). The Am29027 is capable of high-speed, single-precision and double-precision arithmetic using fixed and floating-point numbers. It can be operated in pipelined or non-pipelined (flow-through) mode, depending on system capability and requirements. The pipelined mode maximizes the overall execution time for scalar operations.

System timing can be provided by one of two methods. The Am29000 itself can generate the system clock, which is output on the SYSCLK pin; or circuitry on the board (U8, U9 on Sheet 4) can generate an external clock signal that can be applied to the SYSCLK pin of the processor. Clock selection is done by jumpers.

Memory is supported in both the instruction ROM and instruction/data RAM spaces. By using dip switch (SW3 on Sheet 7), between 0–7 wait states may be selected. Each space has its own wait-state generator, and may be configured separately, depending on the access speed of the installed memory devices.

A 9513A timing controller is installed at U55–58, and U64 on Sheet 10. The 9513A supports up to five 16-bit counters. Address decoding for various timer functions is provided by a PAL (U56 on Sheet 10). The clock source can be from the Am29000, a hardware oscillator, or a crystal oscillator.

Power to the STEB is provided by a series-regulated power supply that provides a regulated +12 VDC and +5 VDC to the board. Connectors are furnished for attachment to the type of power supply used with PCs.

CIRCUIT AREAS CONTRIBUTING TO COMPATIBILITY

In the following section, circuit sections related to compatibility issues are described. The circuit sections are referenced by their locations on the STEB, as indicated in Figure 21.

ADAPT29K and MON29K Compatibility

Because the ADAPT29K and MON29K are very similar to each other, several STEB design aspects simultaneously address their compatibility requirements. These include the type of memory supported, and the bus architecture for accessing memory.

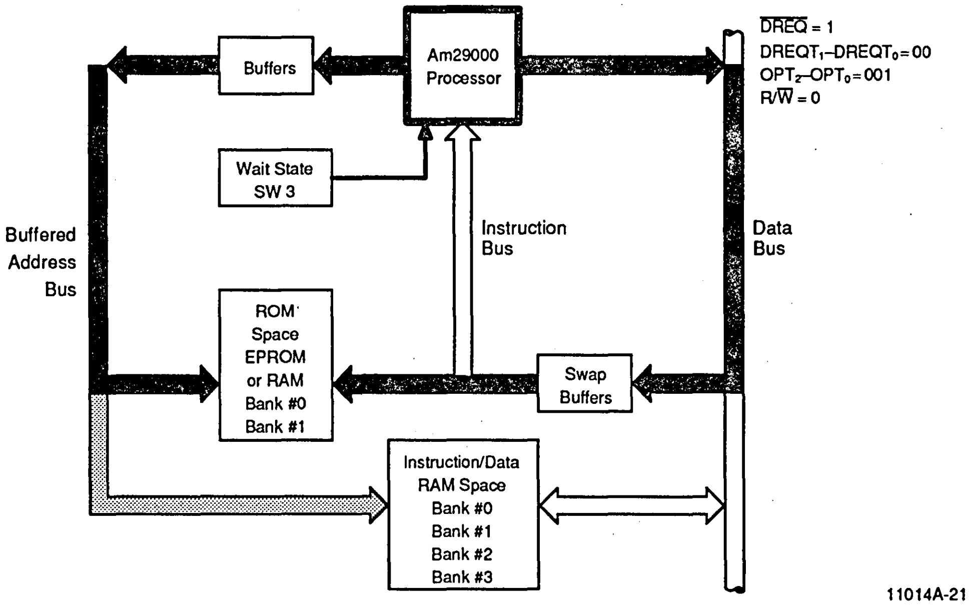


Figure 21. Data Read from Instruction/Data RAM

Support for RAM Devices in the Instruction ROM Space

The STEB supports RAM in the instruction ROM (U25, U32 on Sheet 5) space and the instruction/data RAM (U33–U43 on Sheets 6 and 7) space. The instruction ROM space has a maximum capacity of 1024 Kbytes and uses 27010 EPROMs. The instruction/data RAM space has a maximum capacity of 512 Kbytes and uses 32-Kbyte × 8 static RAMs.

Instructions may be executed from either space. So that programs can be downloaded via the ADAPT29K or MON29K, the instruction ROM area can be constructed from 32-Kbyte × 8 static RAMs. However, the maximum memory size using RAM is limited to 256 Kbytes.

Swap Buffer

On the STEB, a swap buffer provides the necessary bi-directional path between the data bus and the instruction bus (U11–U14 on Sheet 2). The swap buffer is created from four 74ALS245 octal bus transceivers. Transfer direction and timing are controlled by the transceiver's ENA and A→B inputs. By decoding the DREQT_1–DREQT_0, IREQT, OPT_2–OPT_0, $\overline{\text{DREQ}}$, and $\overline{\text{IREQ}}$ signals (U17, U18, U49 on Sheet 4) and applying the result to the transceiver, the STEB channels data between the buses at the appropriate time.

The swap buffer is not required in many straightforward operations. For example, when assembling/disassembling instructions or reading/writing other data into the instruction/data RAM space, data is written directly to the instruction/data RAM space over the data bus. Likewise, a standard instruction fetch from the instruction ROM space does not require the swap buffer, as instructions may be loaded directly into the processor's instruction pre-fetch buffer from the instruction bus.

However, when disassembling instructions in the instruction ROM space, the instructions must be read as data, which makes the swap buffers necessary. The configuration of the IREQT bits causes an instruction to be accessed from the instruction ROM, gated onto the data bus, and read into the processor. Note the combination of control signals indicated on the side of the figure. They are used to select the path for data movement.

Similarly, when instructions are fetched from the instruction/data RAM, they must be transferred to the instruction bus from the data bus. The direction of data movement is shown by the darkened path in Figure 22.

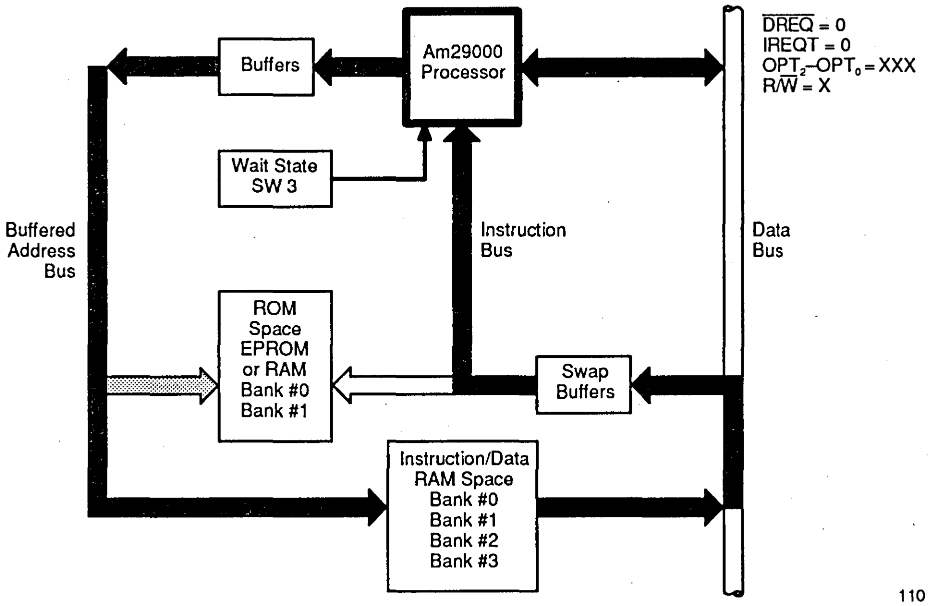


Figure 22. Instruction Fetch from Instruction/Data RAM

One-To-One Address Translation

Note that addresses in both memory spaces have a one-to-one translation. This means that when a data object is stored at a given address in the instruction/data RAM space, the exact same data object will be retrieved when the same address is asserted by an instruction fetch to the instruction/data RAM space. This is an important requirement for assuring compatibility with the ADAPT29K and MON29K because when they are downloading programs, they store instructions as data over the data bus. Neither tool has the capability to translate a virtual address, so when the program is executed it must find its instructions at their absolute addresses.

ADAPT29K Compatibility

In addition to the elements discussed in the ADAPT29K and MON29K Compatibility section, certain considerations were added to the STEB's design strictly for the ADAPT29K. These include tri-stating the control lines driven by the ADAPT29K and disabling memory during data transfers to and from the ADAPT29K.

Tri-Stated Control Lines

The STEB must relinquish some control lines to the ADAPT29K when it is operating. Therefore, these lines are tri-stated or open-collector, as was described in Table 7, thus preventing contention that they may cause unpredictable results.

When the ADAPT29K is not connected to the target, the $CNTL_0$ and $CNTL_1$ lines are pulled high to ensure that the processor is in a normal mode of operation. When the ADAPT29K is connected to the target, it isolates the $CNTL_1$–$CNTL_0$ signals from the board. Any use of those signals by the application will be inhibited.

Memory Disable

The STEB supports both methods of disabling memory for ADAPT29K accesses. Via a jumper selection, the STEB can be configured to either decode an 06 hex on the OPT bits or disable memory when the alignment pin is low.

When Jumper JP7 (on Sheet 7) has pins 1 and 2 connected together it causes the SEL_OP signal to PAL U20 (on Sheet 7) to be high. The ROM/RAM decode circuit (composed of U15, U20, U21, and U24 on Sheets 6 and 7) then decodes the OPT_2–OPT_0 pins to determine whether or not memory should be enabled.

Memory is disabled by a low state on the alignment pin (D4) when jumper JP7 is used to connect pins 2 and 3 together. The low condition is decoded by the ROM/RAM decode circuit, which then disables memory. When the ADAPT29K is not installed, the alignment pin is pulled high to prevent inadvertent and/or intermittent memory disables.

MON29K Compatibility

Apart from the requirements mentioned in the "ADAPT29K and MON29K Compatibility" section, MON29K needs at least one, and preferably two, serial port(s) to communicate with the host/operator. It also needs sufficient memory to contain the software.

Serial Ports

The serial ports are provided by the 8530 serial communications controller (SCC) and support circuits located at U1, U2, and U5–U7 (on Sheet 8). The SCC is a dual-channel, multi-protocol data communications peripheral designed for use with 8-bit and 16-bit microprocessors. The interrupt request line $\overline{INT}$ can be wired to provide a trap or interrupt to the processor for MON29K. Dip switches on the board are used to select port characteristics.

Because the 8530 is a dual-port device, it supports both the DTE and DCE RS232 ports on the STEB. The ports are standard RS232 ASCII ports. The DCE can be used to communicate with an ASCII terminal or PC running a terminal emulator; the DTE port can communicate with a remote host such as a UNIX machine.

Because the C language does not differentiate between address spaces, the serial ports must be memory-mapped into the Am29000 data space. This requirement allows C code to be used in place of assembly language.

Sufficient Memory Space

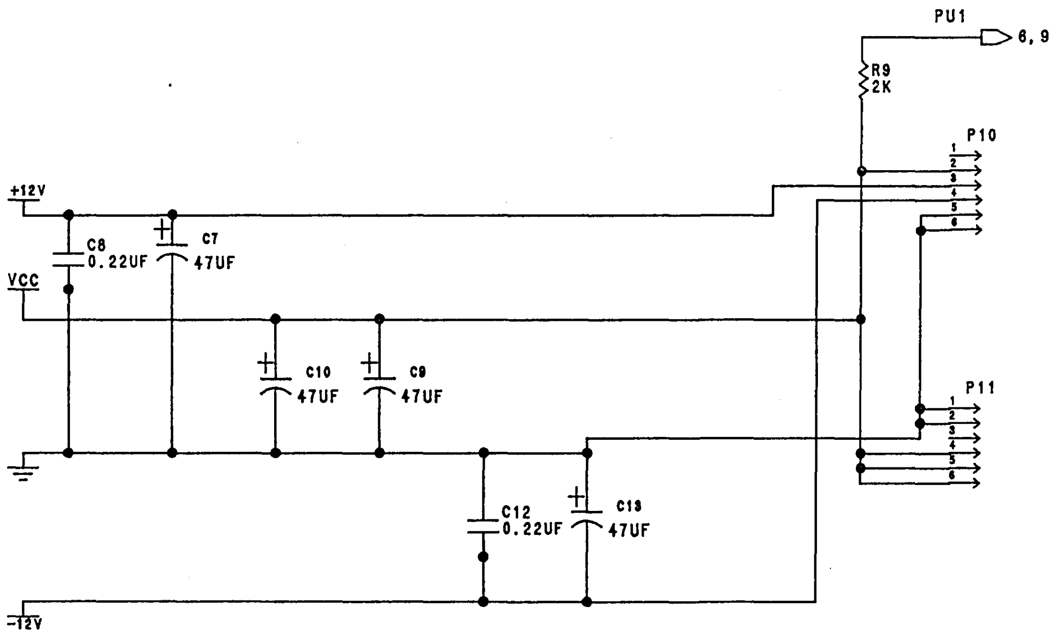
Sufficient memory is provided on the STEB for MON29K. There is also room for additional application programs in the ROM space. The space normally is configured with MON29K in EPROMs (Bank 0), and RAM in the remaining banks. MON29K then could be used to download an application into the RAM in the instruction ROM space.

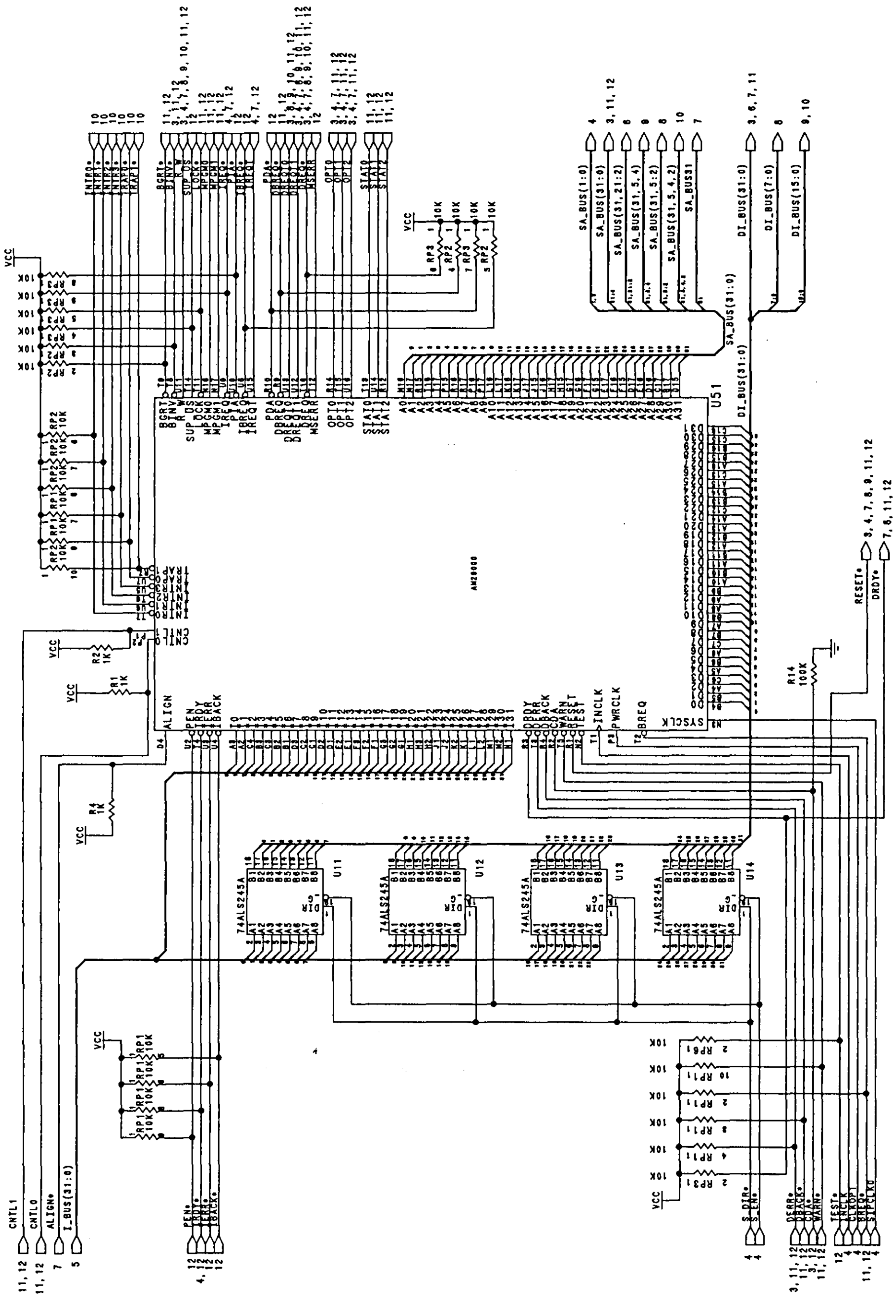
MON29K also uses 64 Kbytes of workspace in RAM. This is provided for, with additional space available for use by the application program.

Built-In Probe Interface

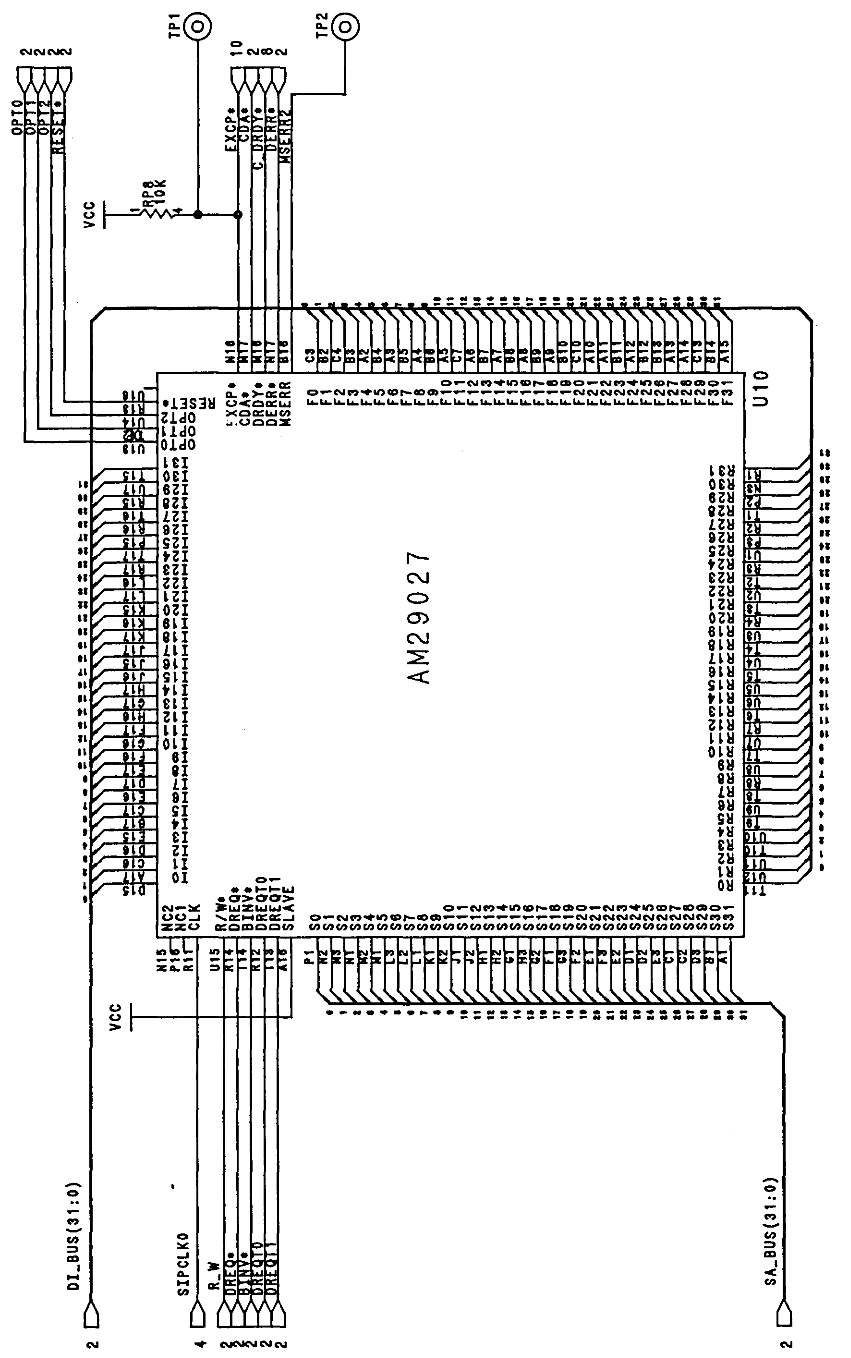
The STEB includes built-in probe interface connectors. Thus, test equipment like the HP1650 or 16500 logic analyzer can be connected directly to the STEB, eliminating the requirement for a separate probe interface.

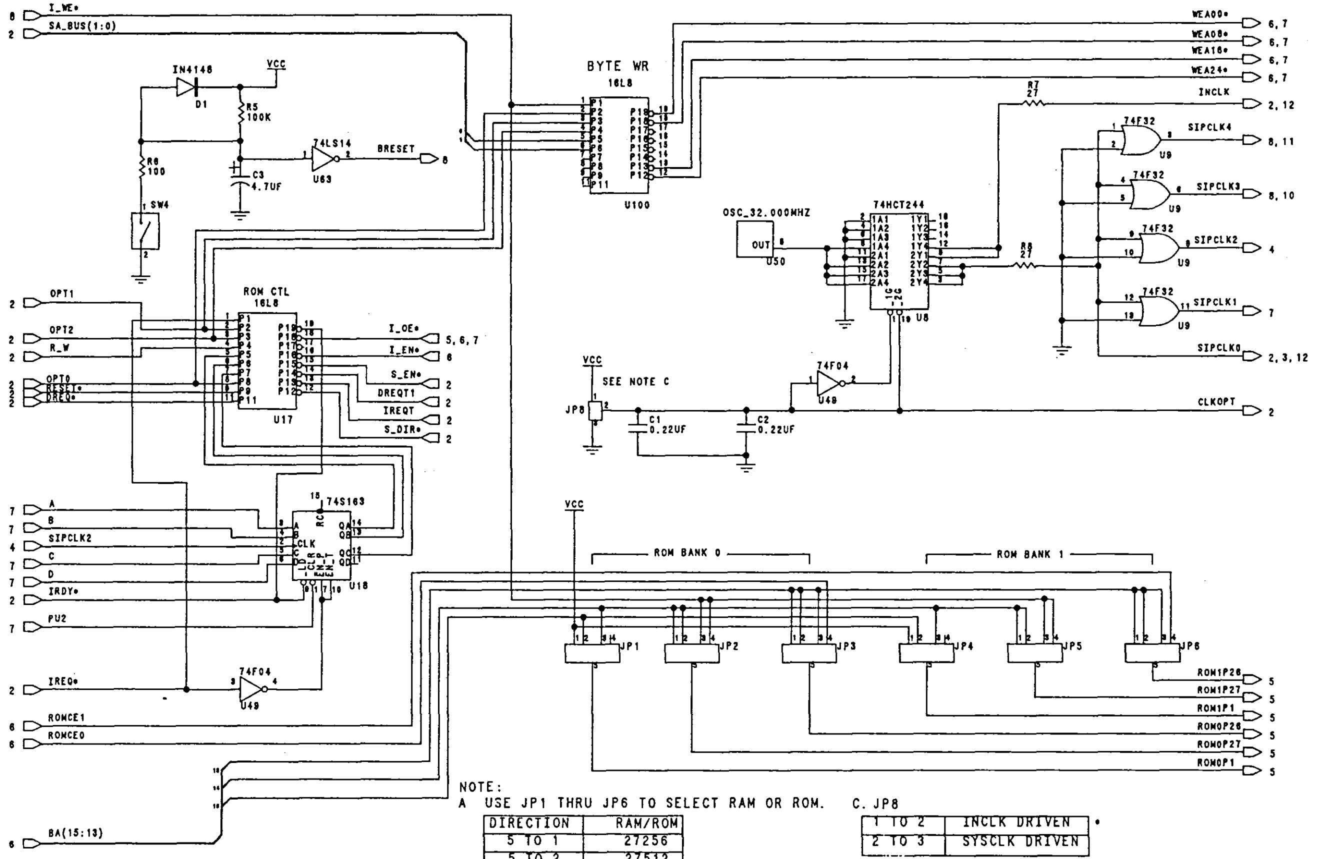
Appendix A: STEB Schematic Diagrams





U51
Am29000
U11
U12
U13
U14
74ALS245A
VCC
SA_BUS(31:0)
DI_BUS(31:0)
I_BUS(31:0)
RESET*
DRDY*
CNTL1
CNTL0
ALIGN*


AM29027
U10
TP1
TP2
VCC
RP8
10K
OPT0
OPT1
OPT2
RESET*
EXCP*
CDA*
C_DRDY*
DERR*
MSERR2
DI_BUS(31:0)
SIPCLKO
R_W
DREQ*
BINV*
DREQT0
DREQT1
SA_BUS(31:0)



NOTE:

A USE JP1 THRU JP6 TO SELECT RAM OR ROM.

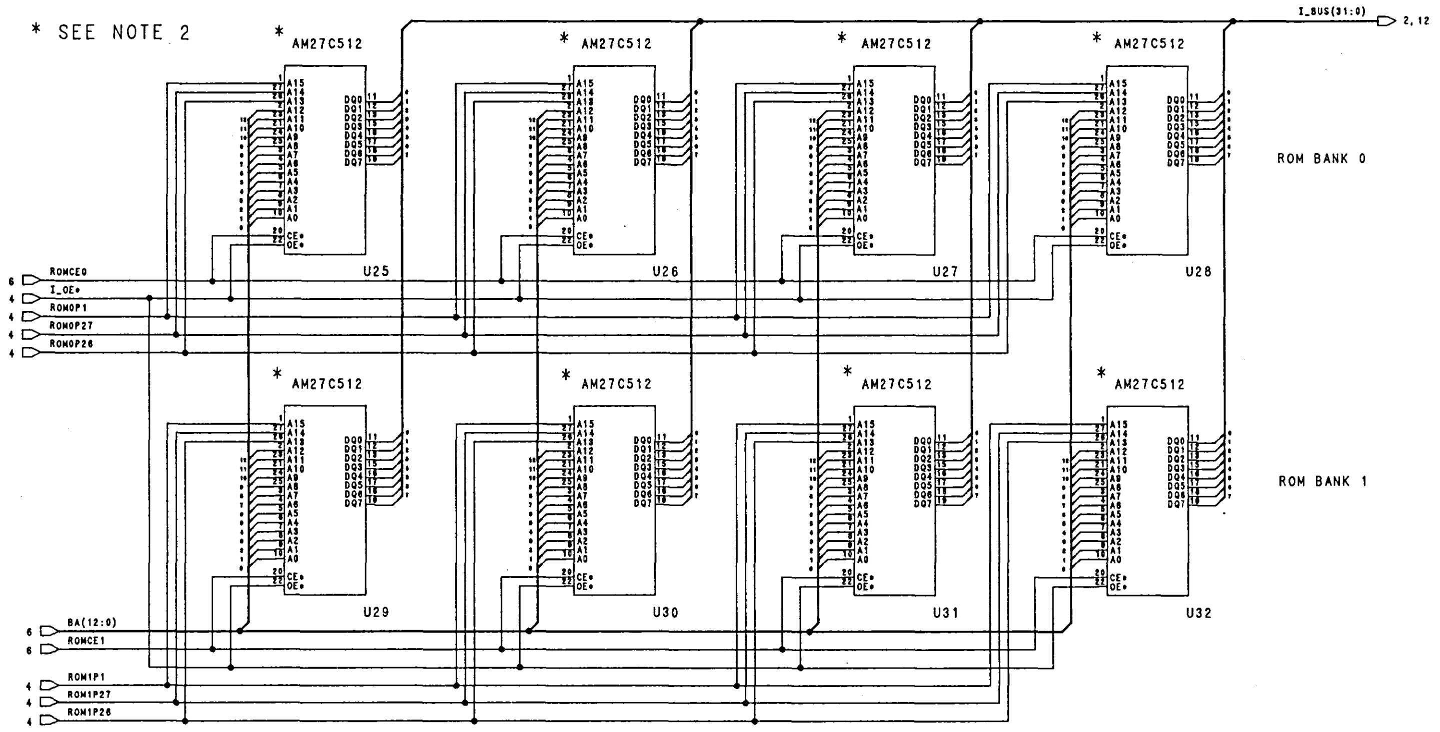
DIRECTION	RAM/ROM
5 TO 1	27256
5 TO 2	27512
5 TO 3	32K X 8
5 TO 4	8K X 8

C. JP8

1 TO 2	INCLK DRIVEN	•
2 TO 3	SYSCLK DRIVEN	

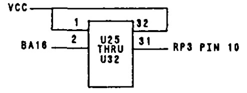
• = DEFAULT

B. ON THE SAME BLOCK ALL JUMPERS SHOULD POINT TO THE SAME DIRECTION.

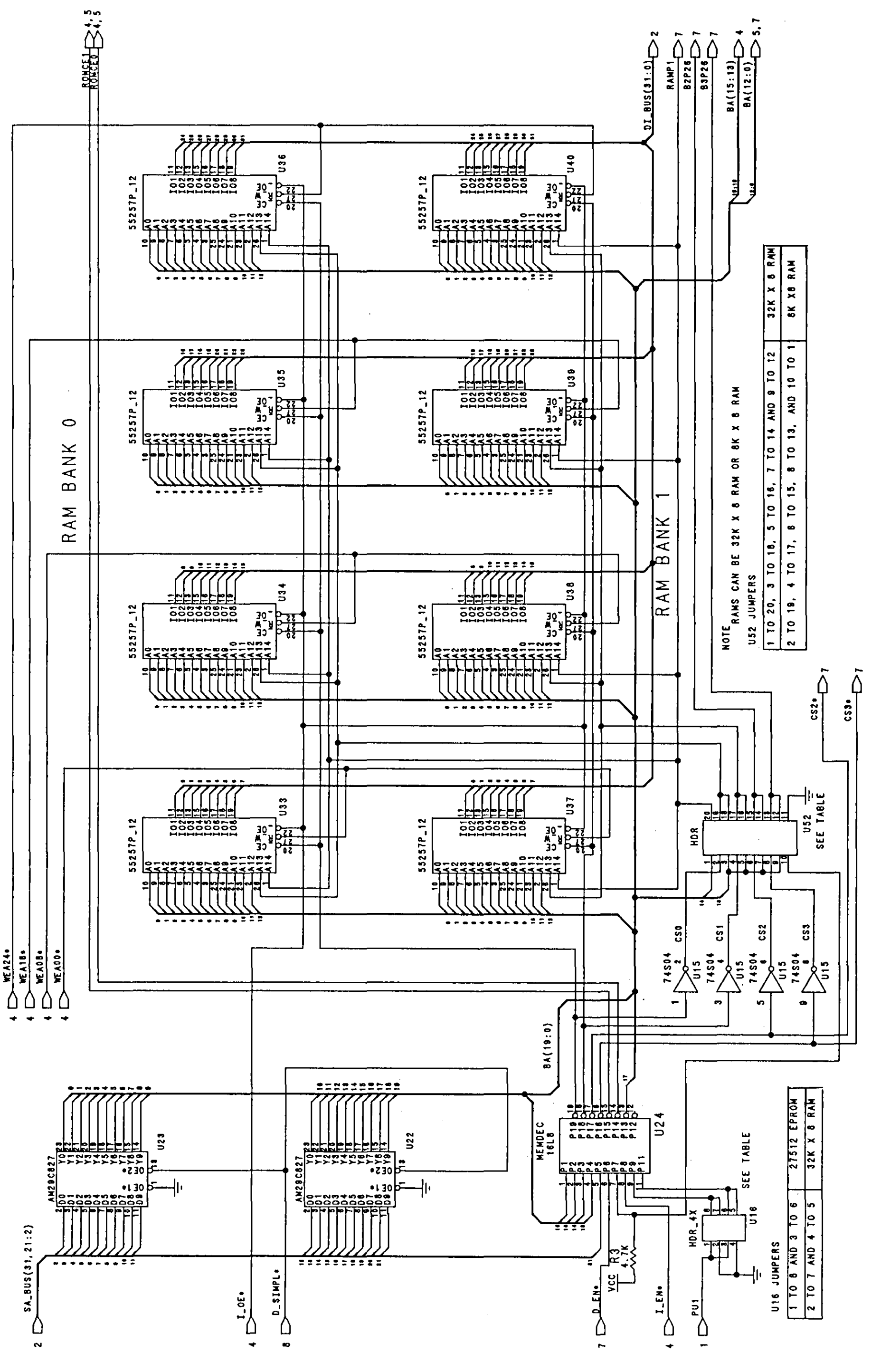


NOTES

1. ADDITIONAL PIN CONNECTIONS FOR 27010 (128K X 8 EPROM):

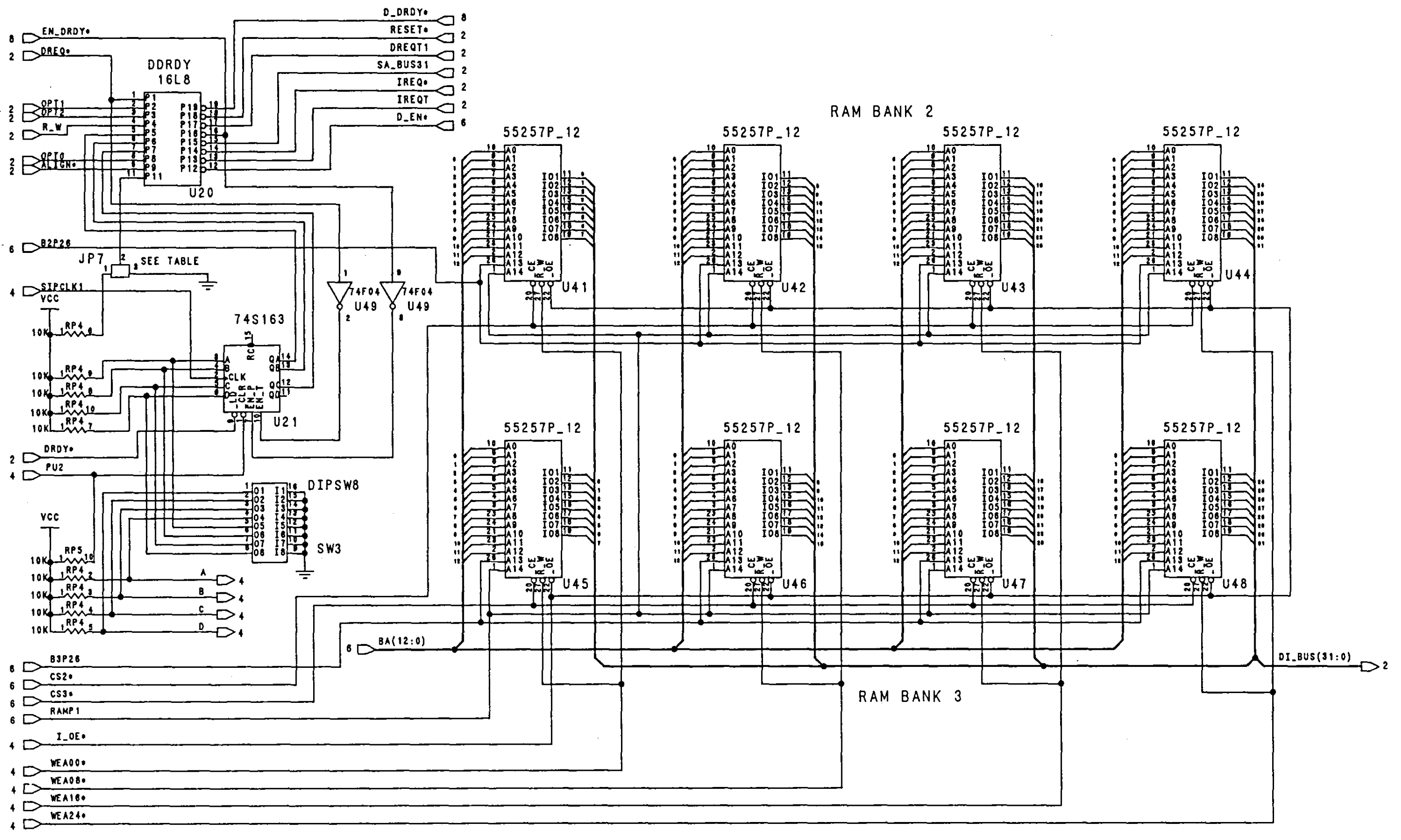


2. ROMS CAN BE 27010, 27512, 27256 EPROM OR 32K X 8 RAM, 8K X 8 RAM



1 TO 20, 3 TO 18, 5 TO 16, 7 TO 14 AND 9 TO 12	32K X 8 RAM
2 TO 19, 4 TO 17, 6 TO 15, 8 TO 13, AND 10 TO 11	8K X8 RAM

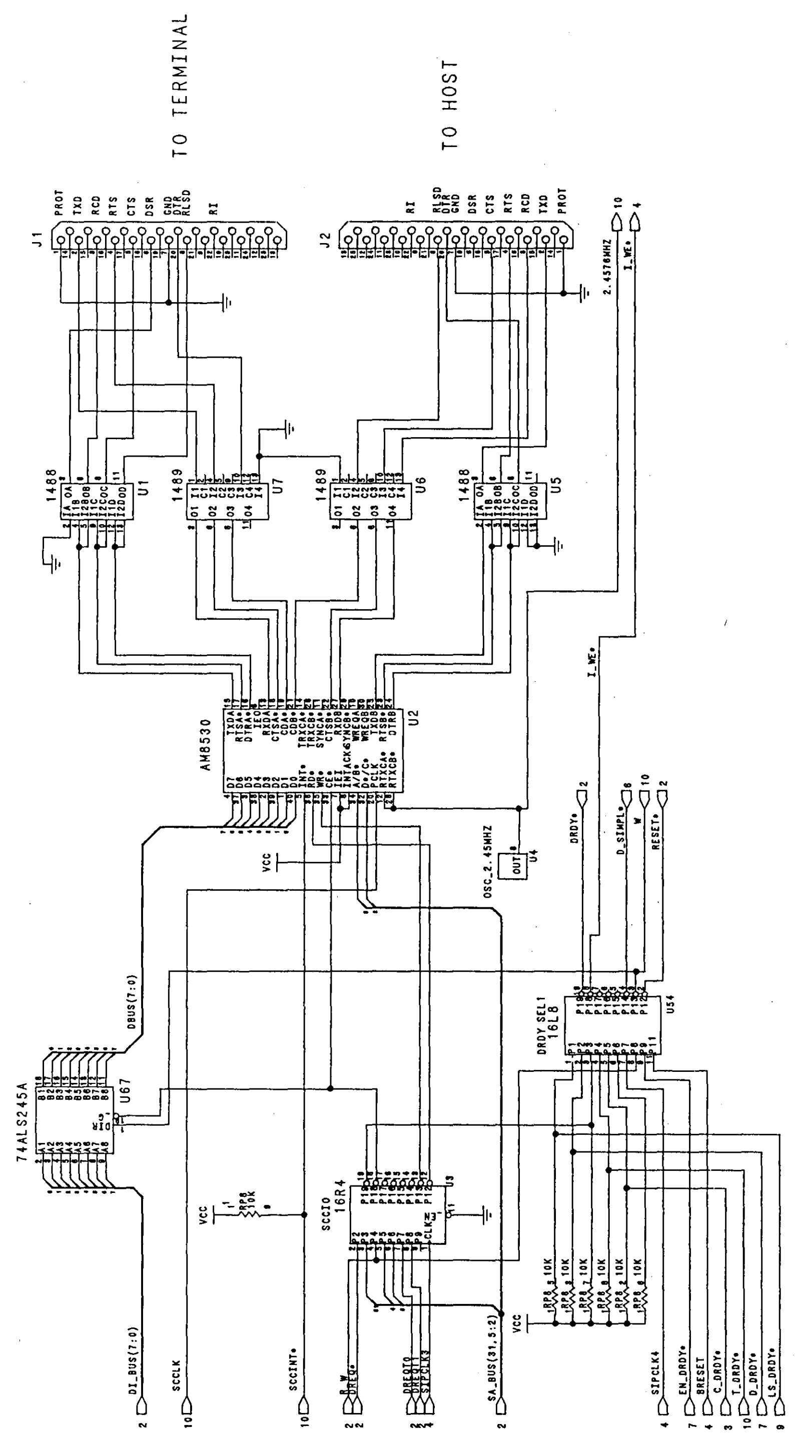
1 TO 8 AND 3 TO 6	27512 EPROM
2 TO 7 AND 4 TO 5	32K X 8 RAM

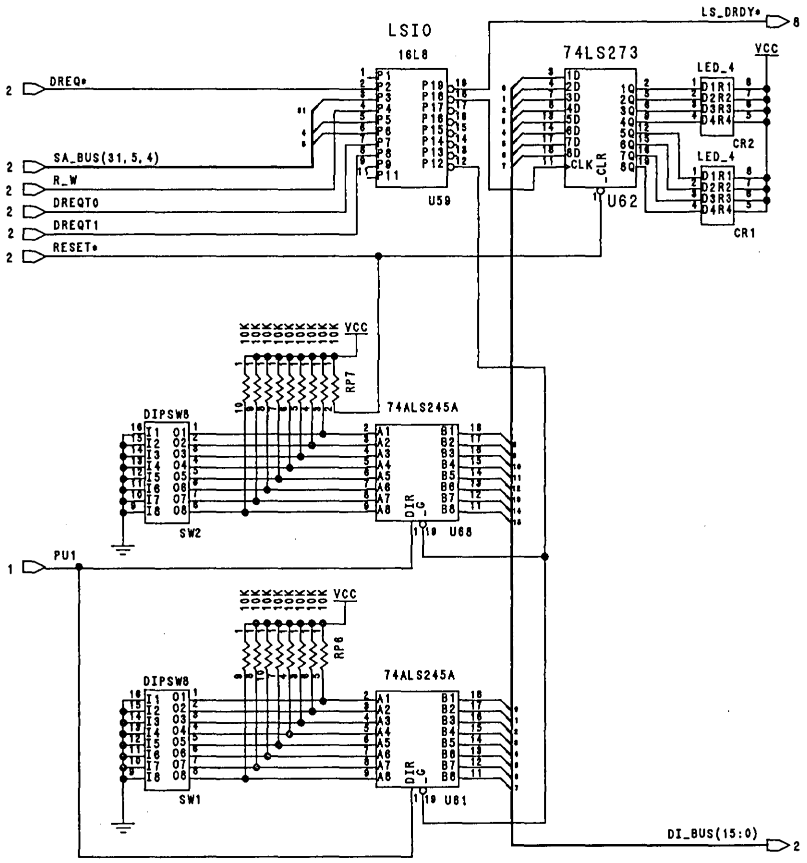


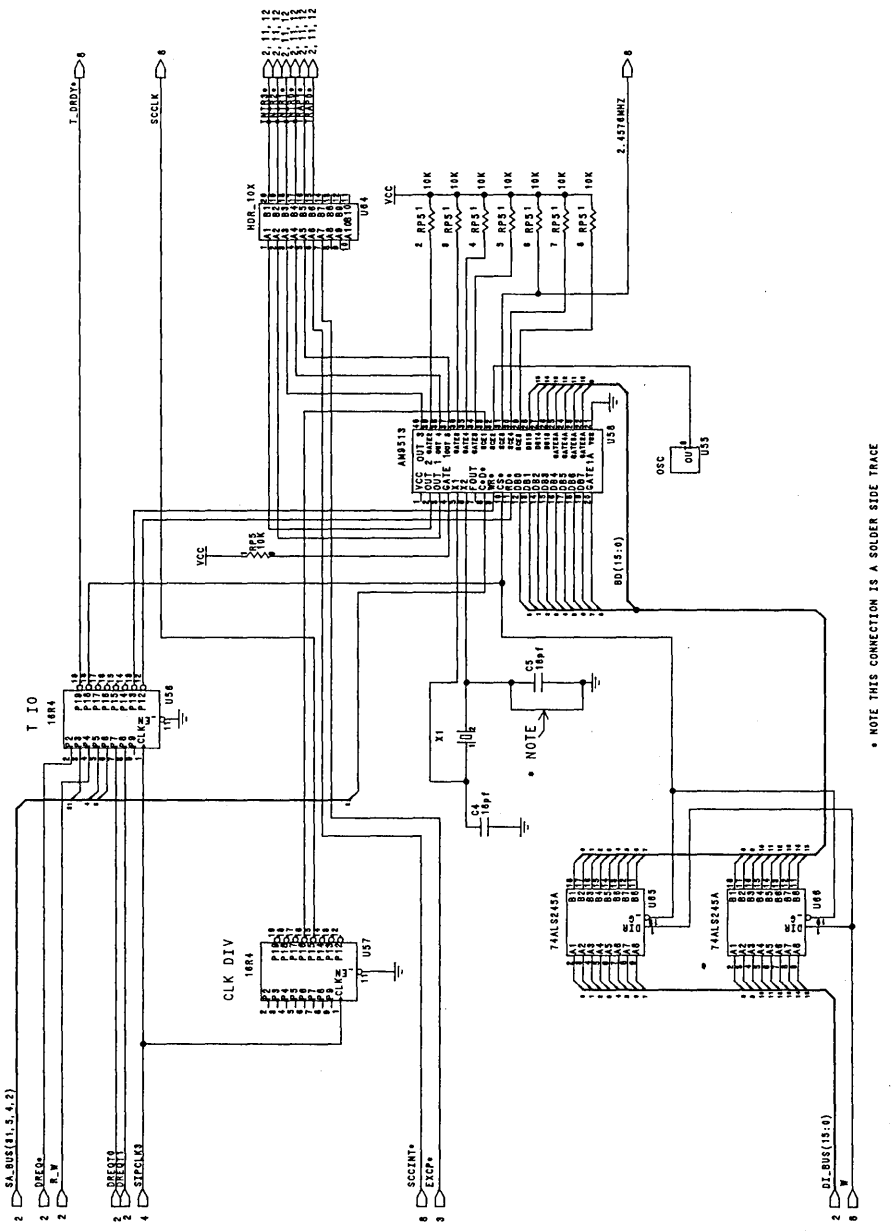
JP7

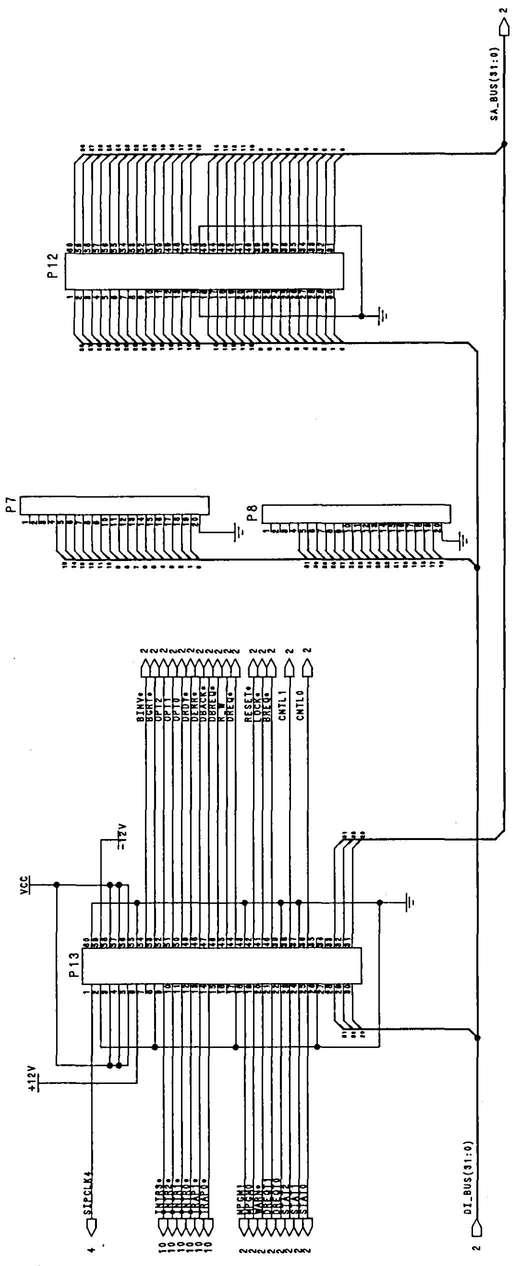
1 TO 2	OPT6
2 TO 3	ALIGN PIN

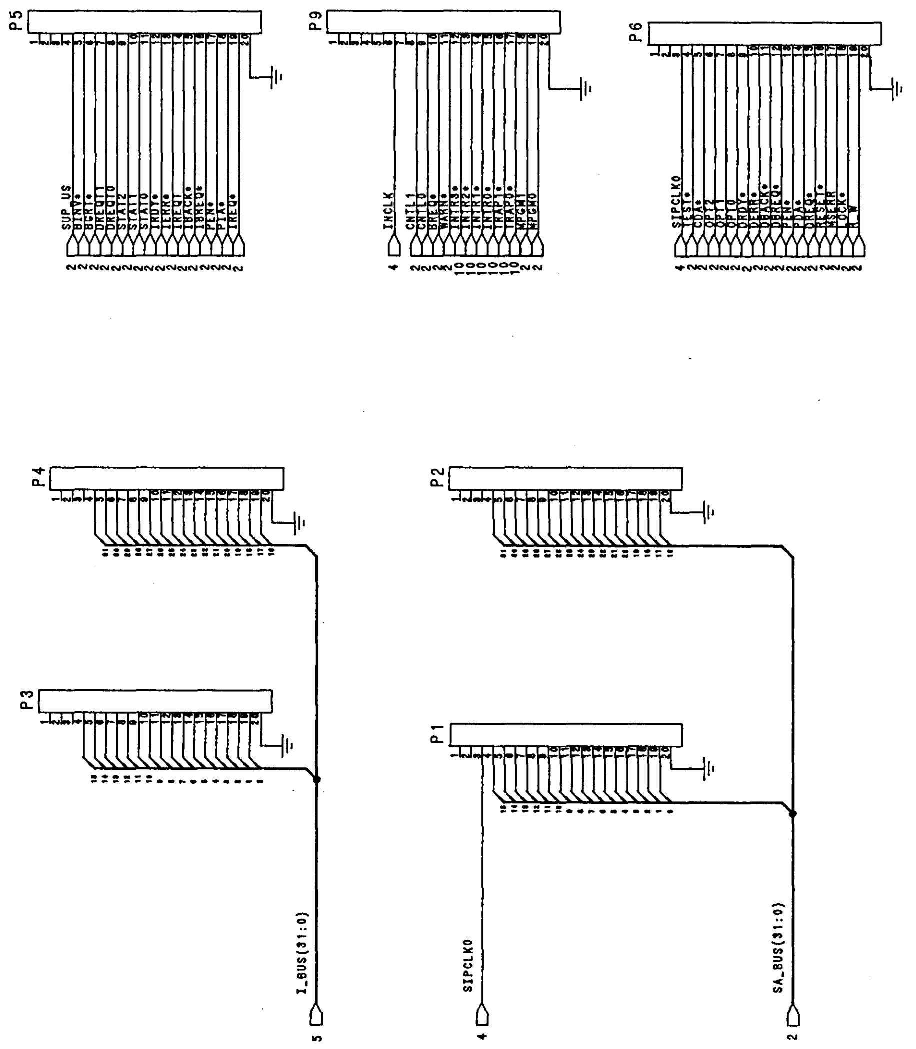
NOTE.
RAMS CAN BE 32K X 8 RAM OR 8K X 8 RAM


TO TERMINAL
TO HOST
J1
J2
PROT
TXD
RCD
RTS
CTS
DSR
GND
DTR
RLSD
RI
1488
U1
1489
U7
1489
U6
1488
U5
2.4576MHZ
I_WE*
AM8530
U2
OSC_2.45MHZ
U4
DRDY*
D_SIMPL*
W
RESET*
DBUS(7:0)
74ALS245A
U67
DRDY SEL1
16L8
U54
SCCIO
16R4
U3
VCC
DI_BUS(7:0)
SCCLK
SCCINT*
R_W
DREQ*
DREQ0
DREQ1
SIPCLK3
SA_BUS(31,5:2)
SIPCLK4
EN_DRDY*
BRESET
C_DRDY*
T_DRDY*
D_DRDY*
LS_DRDY*


LSIO
16L8
U59
DREQ*
SA_BUS(31,5,4)
R_W
DREQT0
DREQT1
RESET*
74LS273
U62
LS_DRDY*
VCC
LED_4
CR2
CR1
RP7
74ALS245A
U68
DIPSW8
SW2
PU1
RP6
U61
SW1
DI_BUS(15:0)
10K


T_DRDY*
SCCLK
INTR3*
2.4576MHZ
HDR_10X
U64
VCC
RP51
10K
AM9513
U58
OSC
U55
BD(15:0)
RP5
10K
T IO
16R4
U56
X1
C5
18pf
C4
16pf
NOTE
* NOTE THIS CONNECTION IS A SOLDER SIDE TRACE
74ALS245A
U63
U66
CLK DIV
16R4
U57
SA_BUS(01,5,4,2)
DREQ*
R_W
DREQT0
DREQT1
SIPCLK3
SCCINT*
EXCP*
DI_BUS(15:0)
W


P12
P7
P8
P13
SA_BUS(31:0)
DI_BUS(31:0)
VCC
+12V
-12V
BINV*
BGRT*
OPT2
OPT1
OPT0
DRDY*
DERR*
DBACK*
DBREQ*
R_W
DREQ*
RESET*
LOCK*
BREQ*
CNTL1
CNTL0
SIPCLK4
MPGM1
MPGM0
WARN*
DREQT1
DREQT0
STAT2
STAT1
STAT0


P5
SUP_US
BINV*
BCRT*
DREQT1
DREQT0
STAT2
STAT1
STAT0
IRDY*
IERR*
IREQT
IBACK*
IBREQ*
PEN*
PIA*
IREQ*
P9
INCLK
CNTL1
CNTL0
BREQ*
WARN*
INTR3*
INTR2*
INTR1*
INTR0*
TRAP1*
TRAP0*
MPGM1
MPGM0
P6
SIPCLKO
TEST*
CDA*
OPT2
OPT1
OPT0
DRDY*
DERR*
DBACK*
DBREQ*
PEN*
PDA*
DREQ*
RESET*
MSERR
LOCK*
R_W
P4
P2
P3
P1
I_BUS(31:0)
SIPCLKO
SA_BUS(31:0)

Preparing PROMs Using the Am29000 Development Tools Application Note

by Manoj Desai and Doug Walton

INTRODUCTION

Source code for a given application must be converted to executable Am29000™ object code and transferred to the appropriate storage media before it can be executed in a real system. Usually several utilities are involved; these include:

- Assemblers
- Compilers
- Linkers
- Format translators (optional, depending on the destination media)

This application note shows how an example program in source code form is made into object code and downloaded to a target board with the ADAPT29K™ Advanced Development and Prototyping Tool, or programmed into PROMs.

THE 29K TOOL CHAIN

The 29K™ tool chain is used to produce the executable object module. The tool chain is an integrated set of programs that includes compilers, assemblers, linkers, and format translators. These programs perform the operations necessary to translate the source code into a machine-readable format. The components of the 29K tool chain are:

- HighC29K™ Compiler
- ASM29K™ Assembler
- ASM29K Linker
- COFF2HEX (COFF to hexadecimal translator)
- ROMCOFF
- BTOA (binary to ASCII translator)

Figure 1 shows the relationship of the 29K tool chain elements to each other. In the following discussion, familiarity with these tools is assumed. Consult the appropriate reference manuals for more details.

The 29K tool chain can be run under UNIX®, SunOS®, or DOS, but it must be installed properly on the host system before the following example can be performed. The host in the following discussion is assumed to be an IBM® AT® or compatible.

Publication #	Rev.	Amendment	Issue Date:
11966	A	/0	11/89

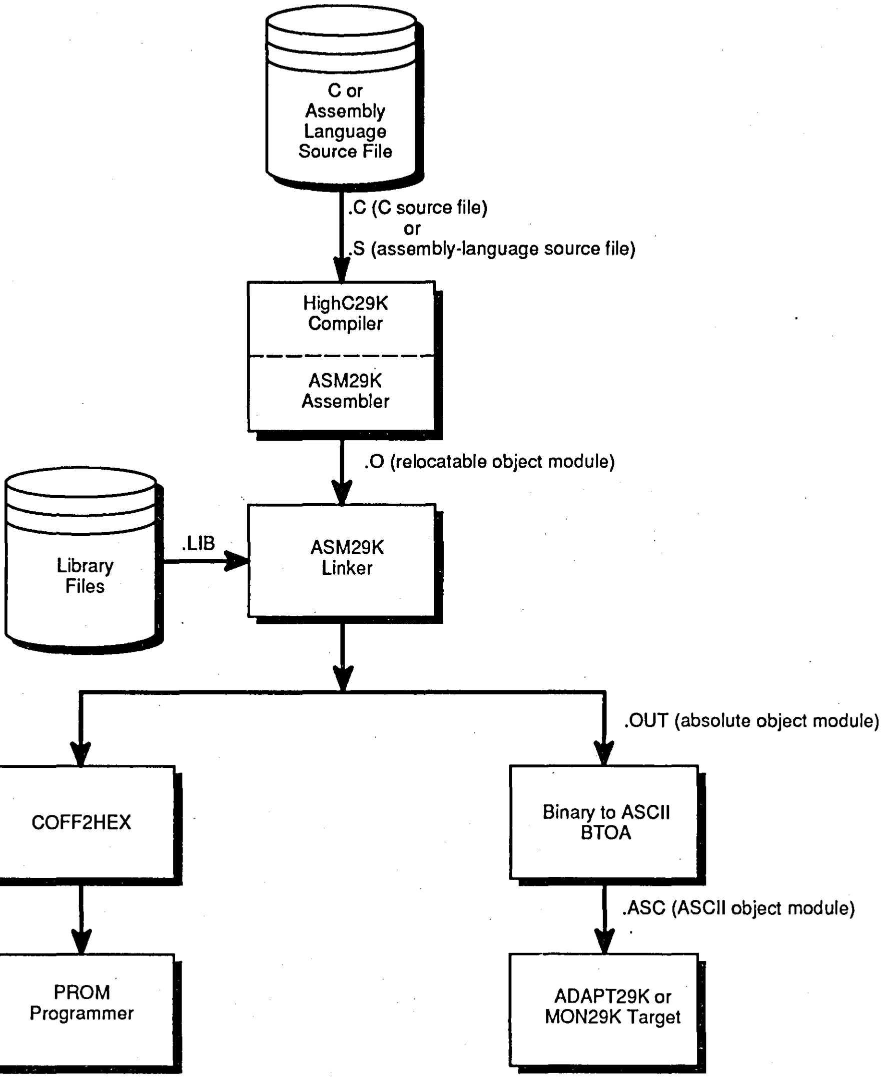


Figure 1. The 29K Tool Chain

SUGGESTED REFERENCE MATERIALS

Consult the following reference materials for more information on the topics covered in this application note.

- *Am29000 Streamlined Instruction Processor User's Manual*, order #10620. It contains details regarding the instruction set and register organization of the Am29000.
- *Am29000 Streamlined Instruction Processor Data Sheet*, order #09075. It embodies a great deal of information about the Am29000, including: distinctive characteristics, general description, simplified system diagram, connection diagram, pin designations and descriptions, functional description, absolute maximum ratings, operational ranges, DC characteristics, switching characteristics and wave-forms, and physical dimensions.
- *ADAPT29K User's Manual.* It provides detailed information on the ADAPT29K, including installation, commands, theory of operation, and target design requirements.
- *ASM29K Documentation Set.* It provides complete information on the installation and use of the ASM29K assembler, linker, and librarian manager. This includes information on using the ROMCOFF and COFF2HEX utilities.
- *HighC29K Documentation Set.* It covers how the Am29000 C compiler is used.

These materials can be obtained by writing to:

Advanced Micro Devices, Inc.
901 Thompson Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

or by calling (800) 222-9323.

For questions that cannot be resolved with the current literature, further technical support can be obtained by writing or calling:

29K Support Products Engineering
Mail Stop 561
5900 E. Ben White Blvd.
Austin, TX 78741
(800) 2929-AMD (US)
0-800-89-1131 (UK)
0-031-11-1129 (Japan)

THE EXAMPLE SYSTEM

The example system used for illustration in this document consists of a generic hardware environment and a small software program. The only function of this self-contained standalone system is to test a block of memory. This section describes how the example system works.

SOFTWARE

The software is a small program that initializes its operating environment and then continuously tests memory. It is comprised of a boot module and a C-language module. A flow chart for the complete application is shown in Figure 2.

The main portions of the program are contained in two source files: **smplboot.s** and **cprog.c**. The **smplboot.s** module is an assembly-language boot program that receives control on power up. The C-language program **cprog.c** performs the memory test.

The tasks performed by **smplboot.s** are: (1) establish the execution environment, (2) set up a block of initialized data in instruction/data RAM (using a routine generated by the ROMCOFF utility), (3) call the main program **cprog.c**, and (4) evaluate the results of the memory test. If the test fails, **smplboot.s** halts the processor.

The **cprog.c** program tests a 32K byte block of RAM, using a simple binary write and read test. Then, **cprog.c** checks the validity of the initialized data section in instruction/data RAM. After each successful completion, a flag is returned to **smplboot.s**, which increments a counter. If a test fails, **cprog.c** returns the address of the failing memory location. A memory map of the application is shown in Figure 3.

Three additional files (**traps.s**, **r29k.s**, and **scregs.def**) contain the supporting procedures and declarations. All of the files in the application are listed in Appendices A through E. To actually perform the example, the files must be entered onto the host system.

HARDWARE ENVIRONMENT

The application runs on the Standalone Execution Board (STEB), manufactured by STEP Engineering. Figure 4 shows a block diagram of the STEB, which contains an Am29000, some RAM and ROM, and two serial ports (provided by an 8530 serial communications controller).

A few important features of the STEB should be noted. First, data can be passed between the instruction and data buses via a bi-directional swap buffer. The swap buffer permits code to be downloaded into the instruction RAM area via the ADAPT29K. It also allows data objects in the instruction ROM space to be read as data.

Second, the instruction ROM space can contain RAM devices or ROM devices. RAM devices should be installed when working with the ADAPT29K (see Appendix F), so that code can be downloaded into the instruction ROM space.

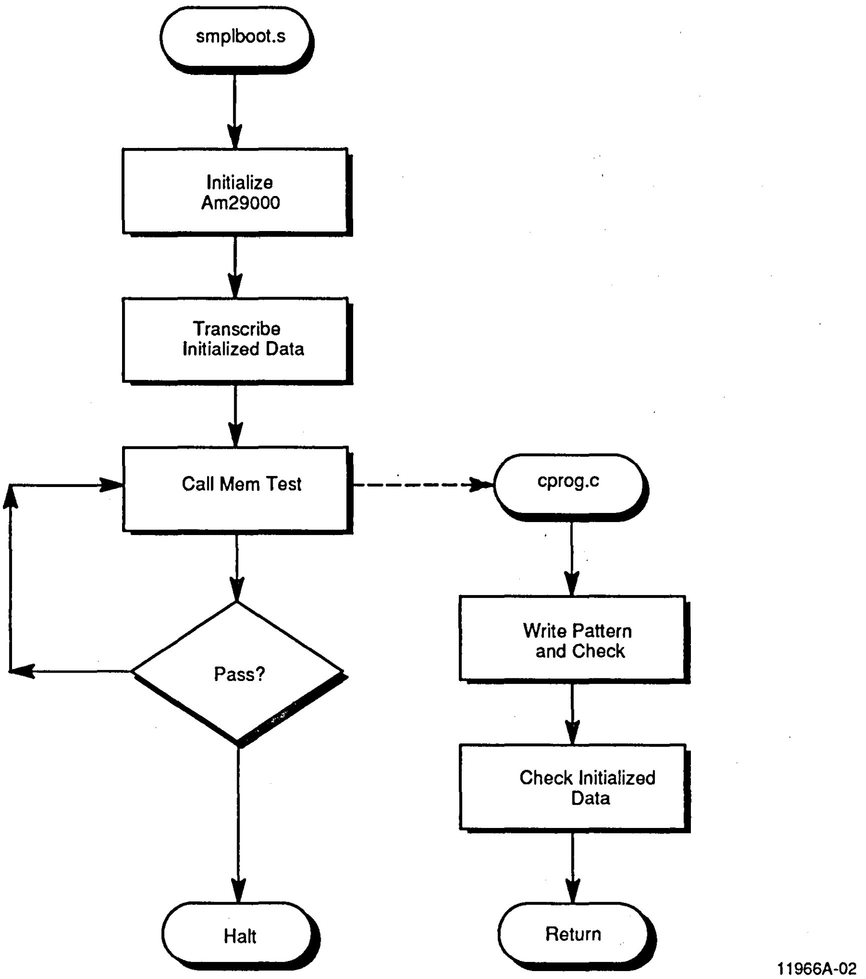
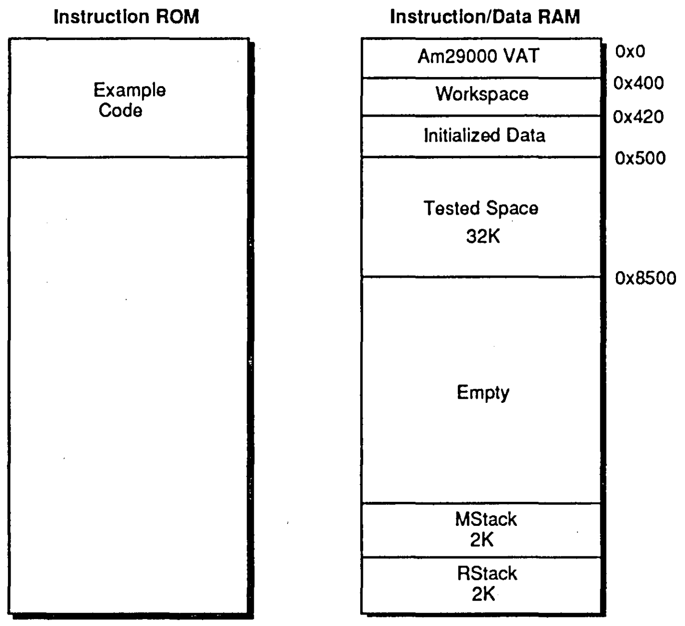


Figure 2. Flow Chart of the Example Application



11966A-03

Figure 3. Memory Map of the Example Application

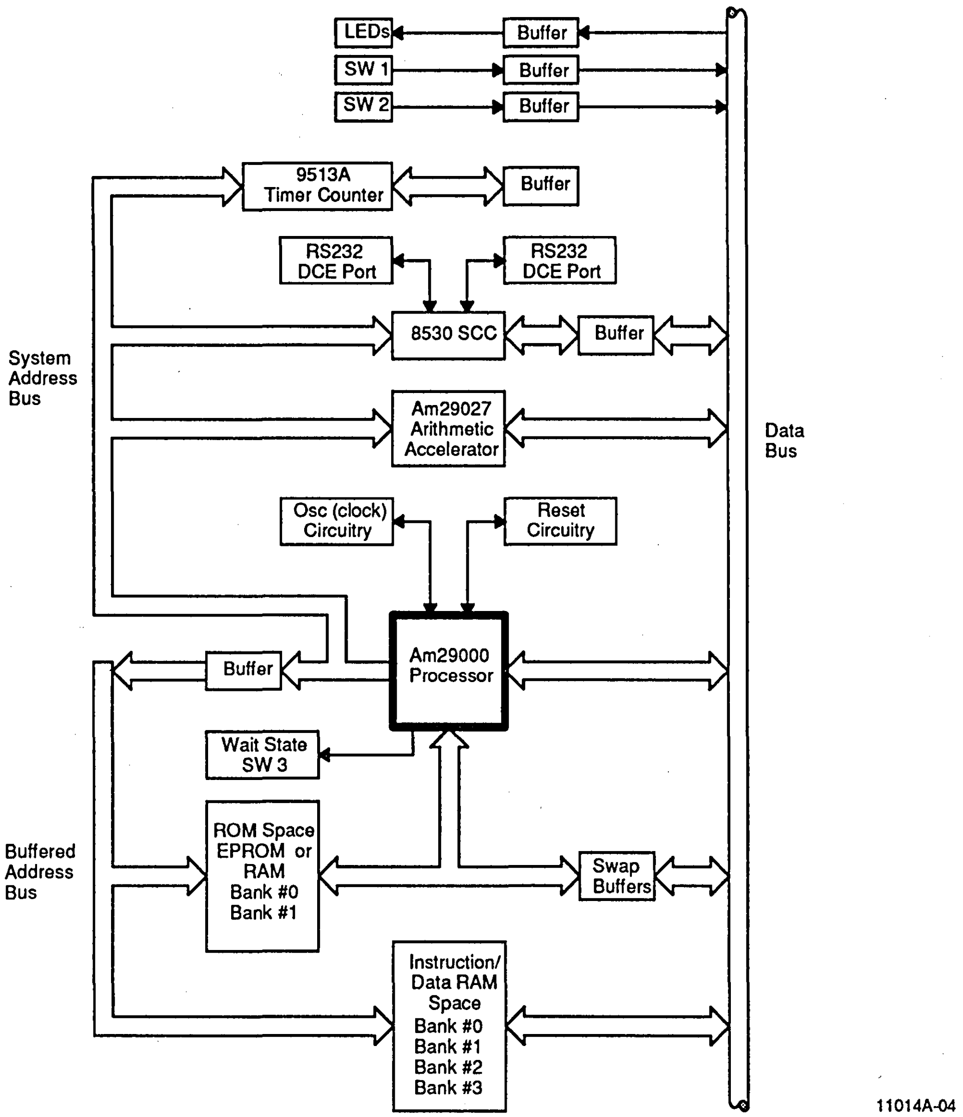


Figure 4. Block Diagram of the STEB

PREPARING AN EXECUTABLE OBJECT MODULE

Preparing the executable object module involves several steps. Typically, the steps are repeated frequently because errors must be corrected and revisions must be made. The process can be automated by placing the commands in a DOS batch file. Listing 1 shows the batch file **sc.bat**, which is used in the example application. Following the listing, each step is explained.

Listing 1. The Batch File sc.bat

```
@echo off
echo ******************************************************
echo "Compiling cprog.c and Assembling the .s files"
echo ******************************************************
hc29 -c -w cprog.c > cprog.e
hc29 -S -Hasm cprog.c > cprog.e
as29 -l > smplboot.lst -o smplboot.o smplboot.s
as29 -l > traps.lst -o traps.o traps.s
as29 -l > r29K.lst -o r29k.o r29k.s

echo ******************************************************
echo "Linking object files with libraries and generating"
echo "executable object module for ROMCOFF"
echo ******************************************************
ld29 -c step1.cmd -o step1.out -f tx -m > outlink.map

echo ******************************************************
echo "Using ROMCOFF"
echo ******************************************************
c:\29k\bin\romcoff -tlb step1.out rom.o

echo ******************************************************
echo "Linking object files with libraries and generating"
echo "final executable object module"
echo ******************************************************
as29 -l > smplboot.lst -DRAMINIT -o smplboot.o smplboot.s
ld29 -c step2.cmd -o step2.out -f tx -m > step2.map

echo ******************************************************
echo "Converting executable object code to downloadable format"
echo ******************************************************
c:\29k\bin\btoa step2s.out sc.a

echo ******************************************************
echo "Converting executable into PROM-programmable format"
echo ******************************************************
coff2hex -c t -m -p 27512 step2e.out > step2.e
echo on
```

COMPILING CPROG.C AND ASSEMBLING THE .S FILES

The first group of operations in the batch file obtains relocatable object modules from the source files. The C-language source file **cprog.c** is compiled by invoking the HighC29K compiler with the command line:

```
hc29 -c -w cprog.c
```

HighC29K replaces the symbolic instructions in the source file with equivalent machine-code routines. Then a relocatable object file (**cprog.o**) is produced, as shown in Figure 5.

The parameter -w suppresses warning messages, limiting the output to containing only errors; the -c parameter instructs the assembler to produce the object file. Note that a second compilation is performed with the -Hasm flag on. This produces an assembly listing (**.s** file) only.

Next, the ASM29K assembler is used to assemble the modules **smplboot.s**, **traps.s**, and **r29k.s**. This involves replacing assembly-language symbolic instructions in the source file with the corresponding machine instruction code. To assemble **smplboot.s** and obtain a relocatable object file, the following command line can be entered:

```
as29 -l > smplboot.lst -o    } Same line.
smplboot.o smplboot.s
```

A relocatable object file (**smplboot.o**) and a listing file (**smplboot.lst**) are produced from the assembly. All assembly-time errors are directed to the **std.out.** The operation is shown in Figure 6. The same operation is done on **traps.s** and **r29k.s.**

Linking

Once the relocatable object files have been made, they must be linked (i.e., assigned physical addresses). This is done using the ASM29K linker, which allows one or more object files from either the assembler or the compiler to be linked together into a single executable object file.

The object modules are linked by entering the command line:

```
ld29 -c step1.cmd -o step1.out} Same line.
-f tx -m > outlink.map
```

Using the command file **step1.cmd** (see Listing 2), the files **smplboot.o**, **r29K.o**, and **traps.o** are linked with **cprog.o** into a single, non-relocatable object file called **sc.out**. A reference to where each module was placed is put in the map file **step1.map**. Any error messages are sent to the **std.out**. The linking process produces a map file that lists the local symbol table, external symbols, and the cross-reference. This type of output is a good reference to the entire application program.

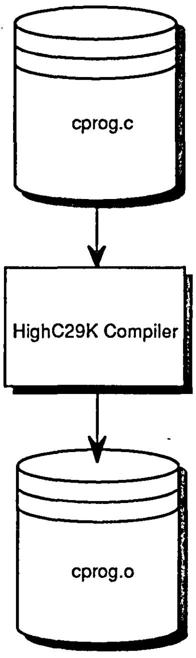
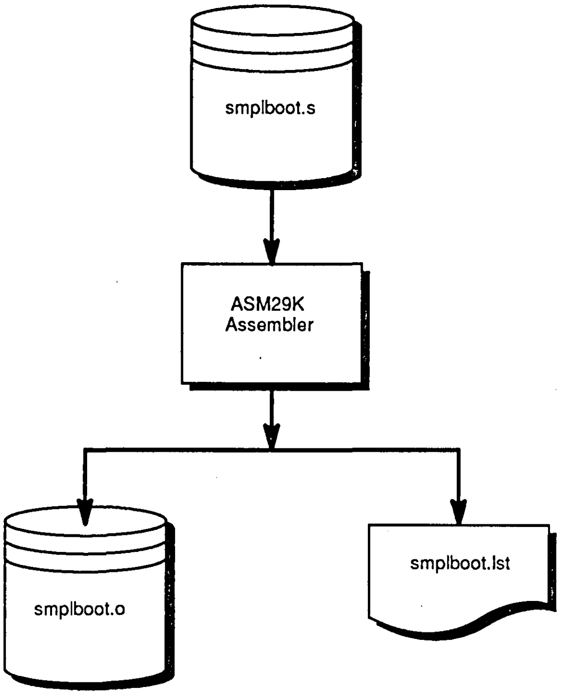


Figure 5. Compiling cprog.c



11966A-06

Figure 6. Assembling smplboot.s

Listing 2. The Linker Command File step1.cmd

```
ORDER .text=0x0
ORDER .bss=0x100400
ORDER .data=0x100420
PUBLIC _MSTACK=0x1f7fc
PUBLIC _RSTACK=0x1fffc
load smplboot.o,r29k.o,traps.o
load cprog.o
load c:\29k\lib\libmw.lib
```

TRANSFERRING CODE FROM ROM TO RAM: ROMCOFF

The **smplboot.s** file contains a section of initialized data that must be loaded into instruction/data RAM and tested by the application program. This could be accomplished by writing many lines of const, consth, and storem instructions into the **smplboot.s** file. Another method is to use the ROMCOFF utility.

The ROMCOFF utility transforms user-specified sections of an Am29000 program into a stream of instructions that will perform the transcription. From a fully linked, executable Am29000 program, the ROMCOFF utility generates a COFF output file containing initializers that will establish the image of an executable COFF input file in instruction/data RAM. The output file contains one section, RI_text, within which is one routine, RAMInit. The output file can then be linked with other relocatable modules that will remain in Instruction ROM, to produce a single non-relocatable module for programming PROMs.

ROMCOFF can be used to transcribe entire sections of code into instruction/data RAM. Then, once the application's boot program has finished preparing the environment, it transfers control to the transcribed program in instruction/data RAM. This allows the code to be executed out of high-speed RAM devices, which are frequently more cost effective than high-speed PROMs. See Figure 7.

In the example program, only a section of initialized data in **smplboot.s** is transferred to RAM. ROMCOFF creates a relocatable object module that transcribes the data sections to RAM when the following command line is entered:

```
romcoff -tlb step1.out rom.o
```

The linked output file **step1.out** is made into the file **rom.o**. Only the data section is output, because of the ROMCOFF options –tlb, which specify that the text, literal, and bss sections should be ignored.

The output from ROMCOFF (**rom.o**) contains only code to transcribe data sections. It must be re-linked with the object files to produce a final absolute object module. First, the code in **smplboot.s**, which contains a call to the RI_text section, must be assembled to include the conditional assembly statements.

To assemble **smplboot.s** so that it will contain the call, enter:

```
as29 -l > smplboot.lst -DRAMINIT  } Same
 -o smplboot.o smplboot.s         } line.
```

The –D option defines RAMInit so that conditional assembly statements in the source file will be assembled. The statements include a definition of RAMInit, and a call to it. Then, all of the object modules can be linked with **rom.o** as follows:

```
ld29 -c step2.cmd -o step2.out   } Same
-f tx -m > step2.map             } line.
```

A second linker command file is used because **rom.o** must identified to the linker (see Listing 3).

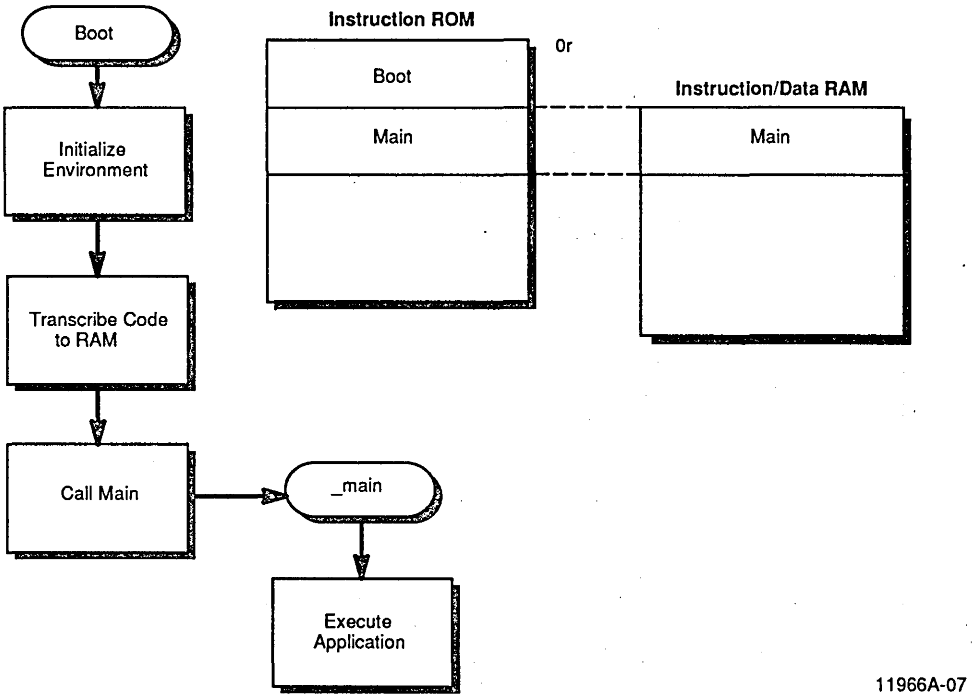


Figure 7. Using ROMCOFF

Listing 3. The Linker Command File step2.cmd

```
ORDER .text=0x0,RI_text
ORDER .bss=0x100400
ORDER .data=0x100420
PUBLIC _MSTACK=0x1f7fc
PUBLIC _RSTACK=0x1fffc
load smplboot.o
load rom.o
load r29k.o,traps.o
load cprog.o
load c:\29k\lib\libmw.lib
```

DOWNLOADING TO THE ADAPT29K

Once the final executable object module is created, the example program can be downloaded to the target system and tested using the ADAPT29K.

USING BTOA

The BTOA utility creates an ASCII COFF output from the input file. Although the ADAPT29K can handle Tektronics® or Motorola® hex files, using the BTOA utility to make the ASCII hex file has several advantages. Most importantly, BTOA encodes the input file into (7-bit) ASCII using a compact base-5 scheme that limits file expansion to only 25 percent, as opposed to 150 percent for standard hex formats. Hence, the resulting output file is smaller, and consequently quicker to transfer. Also, BTOA maintains the ASCII COFF format, rather than converting it to absolute addresses.

As shown in the **sc.bat** batch file, BTOA produces the output file **sc.a** and is invoked by:

```
btoa step2s.out sc.a
```

```
00000000R    c6400200    MFSR     GR64,CPS
00000004R    03fb41ff    CONST    GR65,0xFBFF
00000008R    90404041    AND      GR64,GR64,GR65
0000000cR    ce000240    MTSR     CPS,GR64
00000010R    03004000    CONST    GR64,0x0
00000014R    ce000040    MTSR     VAB,GR64
00000018R    0300403f    CONST    GR64,0x3F
0000001cR    ce000740    MTSR     RBP,GR64
```

11966A-08

Figure 8. List Memory Display

Listing 4. Results of "End Execution" Command List

```
> d 400,420
00000400     00000000 00000000 00000000 00000000    ................
00000410     00000000 00000000 00000000 00000000    ................
00000420     00000000                               ....
```

TESTING THE EXAMPLE PROGRAM WITH THE ADAPT29K

Once the object module has been translated using the BTOA utility, it can be downloaded to the target using ADAPT29K. For use with ADAPT29K, the STEB should be configured as indicated in Appendix F.

To download the file, communication must be established with the ADAPT29K. On a PC, this is done by invoking the terminal emulator program (for example, CrossTalk®), establishing communication with the ADAPT29K, and entering (note that # is the ADAPT29K monitor prompt):

```
# ya c,0r
```

The Y (load a file to memory) command prepares the ADAPT29K to receive an ASCII-encoded file from the DCE port. Then, the emulator must be instructed to transmit the file (for example, **se sc.a** when using CrossTalk). After the code has been downloaded, and the next prompt has appeared, the contents of the instruction ROM can be verified by entering:

```
# l 0r
```

The ADAPT29K should respond to the L (list memory) command with the display shown in Figure 8. The locations starting at 0x400 in instruction/data RAM contain the status of the test and number of successful loops, respectively. Which location actually contains which variable is a decision made by the linker, and must be determined by inspection.

To check these locations automatically when the execution stops, set up an "end execution" command list by entering:

```
# e d 400,420;
```

The list is executed on entry. It should appear as shown in Listing 4.

```
GR080  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GR088  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GR096  00104a18 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GR104  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GR112  00000000 00000000 00000000 00000000 80000020 000095d9 00100400 00000095
GR120  ffffffff 80000000 00000000 00000000 00000000 0001f7fc 00000fff 06050101

LR000  00000928 0001fffc 00100414 00108414 000000f0 0001fffc 00000000 00000000
LR008  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR016  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR024  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR032  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR040  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR048  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
LR056  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

   GR001     IPC      IPA      IPB       Q      ALU: DF V N Z C  BP   FC   CR
0001ffe4     00       00       00    00000000        0  0 0 0 0  0    00   00
 (R249)   (GR000)  (GR000)  (GR000)
```

Figure 9. Key Registers Display 11966A-09

Prior to starting the test, it is a good practice to reset the system by using the P reset command:

```
# p reset
```

To verify the condition of the system before execution, the X (Display Key Registers) command is entered as:

```
# x
```

This will result in a display as shown in Figure 9. The special-purpose protected registers can be checked using the XP (display protected registers) command. The display appears as shown in Figure 10.

To execute the program starting from address 0 in instruction ROM, the G (go—start execution) command is used:

```
# g 0r
```

During execution, the status of the program can be checked by invoking the previously defined "end execution" command list.

Enter:

```
# e
```

The display will be similar to that shown in Figure 11. The precise display in any given situation, particularly the loop count stored in location 40CD is dependent on the exact time elapsed between the start execution and the entry of the E command. At another time, it may appear as shown in Figure 12.

The state of the processor can be checked using the C (check execution state command):

```
# c
```

When the processor is running, ADAPT29K displays:

```
Am29000 is Running.
```

```
# xp
      CA  IP  TE  TP  TU  FZ  LK  RE  WM  PD  PI  SM  IM  DI  DA
CPS:  0   0   0   0   0   1   0   1   0   1   1   1   0   1   1
OPS:  0   0   0   0   0   1   0   1   0   1   1   1   0   1   1

VAB         CFG: PRL  VF  RV  BO  CP  CD
0000              01   1   0   0   1   1

  CHA       CHD        CHC:  CE  CNTL  CR  LS ML ST LA TF  TR  NN CV
00104a14  00000000            0   00    00  1  0  0  0  0  79   1  0

RBP: BF BE BD BC BB BA B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
      0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1

  TCV   TR: OV IN IE   TRV     PC0       PC1       PC2      MMU: PS  PID   LRU
000000      1  1  0   000000  00000a34  00000a30  00000a2c        0   00    0
#
```

11966A-10

Figure 10. Protected Registers Display

```
> d 400,420
00000400     009595d9 00000000 00108414 0000002d    ................
00000410     00100414 00000000 00000000 00000000    ................
00000420     00000000                               ....
```

11966A-11

Figure 11. Check Status Display

```
> d 400,420
00000400     009595d9 00000000 00108414 000000e1    ................
00000410     00100414 ffffffff ffffffff ffffffff    ................
00000420     ffffffff                               ....
#
```

11966A-12

Figure 12. Second Check Status Display

PREPARING PROMs

Once the absolute object file has been prepared, it must be transferred to the media from which the code will be executed. Often, this medium is a PROM set. Most PROM programmers require their input to be in an ASCII hex format, so a translation normally is performed before sending the program to the PROM programmer.

MAKING HEX FILES: COFF2HEX

The COFF2HEX utility produces a 32-bit ASCII hex file in either the Motorola S3 or Tektronics Extended format. Both of these formats are accepted by most PROM programmers, as well as the ADAPT29K. Note that the ADAPT29K requires the file to be one module, rather than being divided into separate modules by part size (see the options of the COFF2HEX utility).

In **sc.bat**, COFF2HEX is invoked by entering:

```
coff2hex -c t -m -p 27512     }  Same line.
step2e.out > sccoff.e
```

This produces 8-bit wide modules that will fit into a 27512 EPROM (–p option). The format is Motorola S3 (–m option), and will include only the text sections (–c t option).

The resulting file(s) will be named **a.a00**, **a.a08**, **a.a16**, and **a.a24**, indicating which bytes of the word they represent. If the file is larger than the capacity of the part size specified, additional sets of four will be generated with filenames **a.b00**, **a.b08**, **a.b16**, **a.b24**, and so on, with further sets having a corresponding nomenclature. Once generated, the files can then be transmitted to a PROM programmer.

PROGRAMMING THE PROMS

A PROM programmer is used to "burn" the binary object file into PROM devices. Many types of PROM programmers are available. The Data I/O Unisite® PROM programmer is used in the following example.

Assuming an object module had been created as described in the first part of this document (and a set of Motorola S3 modules were obtained using COFF2HEX), the following procedure could be used to create a PROM set.

1. Turn on the PROM programmer. Make sure the algorithm disk is properly inserted in the lower front slot.
2. Once the power-up sequence and diagnostics have completed, a screen should appear on the attached terminal. If there is no terminal, or the screen does not appear, refer to the set-up section of the user's manual for the PROM programmer.
3. Make sure a host system is attached. In this example, the use of a PC is assumed. At the PC, set the COM1 serial port of the PC to 9600 baud, no parity, 8-bit bytes, and one stop-bit by entering: **mode com1:96,n,8,1**. On the PROM programmer, select "Configure System," followed by "Edit," and then "Serial I/O." Make sure the remote port parameters are set properly.
4. The program will be placed in AMD 27512 PROMs. To inform the PROM programmer, choose "Select Device," "3" (AMD), and "25" (27512).

5. It is a good idea to clear the PROM programmer's memory before downloading data. This ensures that the PROMs do not become programmed with leftover data from a previous operation, which may cause troublesome errors. To clear the memory, select "Fill Memory." Enter 00 to 7FFFF as the address range, and FF as the data.
6. The PROM programmer must know the format of the incoming data. Select "Transfer Data," followed by "Format Select." Enter "95" for Motorola S3 Record.
7. Select "Load Device" on the programmer. On the PC, enter:

   ```
   copy a.a00 coml:
   ```

   This causes the lowest 8 bits of the application to be transmitted to the PROM programmer, which will load the data into its memory.
8. Properly insert a PROM into the ZIF socket on the PROM programmer and engage the locking mechanism. Select "Program Device" option on the PROM programmer.
9. Once the PROM has been burned, remove it and label it with the program name, range of bits, version, and date. Then, repeat steps 7–9 using the files **a.a08** through **a.a24**. If a larger program is used, it may be necessary to repeat steps 7–9 using modules **a.b00, a.b08, a.b16, a.b24,** and so on.

APPENDIX A: smplboot.s

```
         .extern    r29k_init            ; assembly module
         .extern    _main                ; C module
         .extern    V_SPILL,V_FILL       ; Linker definable V_SPILL and V_FILL vector numbers
         .extern    spill,fill           ; spill and fill procedure
         .extern    _RSTACK,_MSTACK      ; Link time definable stack pointer assignments
         .equ       ROM_TH,0x2           ; Spill and fill trap interface do truly reside in ROM space
         .equ       RSC_SIZE,0x200       ; Default reg_stack_cache usage=512
         .equ       TBM_SIZE,0x20000     ; 32K*4=128kb of Inst/RAM size
         .include   "scregs.def"
         .data
         .word      [20]170
         .comm      mtp_count,4
         .text
         .ifdef     RAMINIT              ; if RAMINIT Flag on
         .extern    RAMInit              ; make RAMInit available
         .endif
         .global    start
start:
         mfsr       tmp0,CPS             ; Read CPS
         const      tmp1,0xFBFF          ; Clear FZ bit
         and        tmp0,tmp0,tmp1
         mtsr       CPS,tmp0             ; Update CPS
         const      tmp0,0               ; Set VAB pointing to LOW memory
         mtsr       VAB,tmp0
         const      tmp0,0x11            ; Set VF=1, i.e., Vector table scheme and CD=1,
                                         ; i.e., Branch Target Cache is disabled
         mtsr       CFG,tmp0
         const      tmp2,0               ; Write Data pattern = 0x00000000
         const      tmp0,0               ; Low memory address
         consth     tmp1,TBM_SIZE        ; High memory address
         sub        tmp1,tmp1,tmp0       ; Get address difference
         srl        tmp1,tmp1,2          ; Get word count from diff value
         sub        tmp1,tmp1,2          ; adjustments for jmpfdec instr
mem_00:                                  ; fill TB_memory with all zeros
         store                           0,0,tmp2,tmp0
         jmpfdec    tmp1,mem_00
         add        tmp0,tmp0,4
         const      tmp0,256-2           ; Total of 256 vector table entries
         const      tmp1,illtrap+0x2     ; ROM based illegal trap handlers
         consth     tmp1,illtrap         ; address, by default
         const      tmp2,0
vtd_init:                                ; fill vector table with default
         store      0,0,tmp1,tmp2        ; trap handlers
         jmpfdec    tmp0,vtd_init
         add        tmp2,tmp2,4
         const      tmp0,spilltrap+ROM_TH ; get spill trap entry point
         consth     tmp0,spilltrap
         const      tmp1,V_SPILL         ; get spill trap vector number
         sll        tmp1,tmp1,2          ; generate vect number location
         store      0,0,tmp0,tmp1        ; store address of trap handler into vector table
         const      tmp0,filltrap+ROM_TH ; get fill trap entry point
         consth     tmp0,filltrap
         const      tmp1,V_FILL          ; get fill trap vector number
         sll        tmp1,tmp1,2          ; generate vect number location
         store      0,0,tmp0,tmp1        ; store address of trap handler into vector table
         const      rfb,_RSTACK          ; Set RFB
         consth     rfb,_RSTACK
         const      tmp0,RSC_SIZE        ; 0x200=512 bytes ie 128*4
         sub        rab,rfb,tmp0         ; Set RAB=RFB-512
         sub        rsp,rfb,0x8          ; Set RSP=RFB-8
         const      msp,_MSTACK          ; Set MSP
         consth     msp,_MSTACK
         add        lr1,rfb,0            ; Set lr1 to RFB
         const      tmp0,r29k_init
         consth     tmp0,r29k_init
         calli      lr0,tmp0             ; call procedure to init 29K registers
         nop
         .ifdef     RAMINIT              ; if RAMINIT on,
        const     tmp0,RAMInit          ; set up RAMInit call
        consth    tmp0,RAMInit
        calli     gr96,tmp0             ; and do the call
        .else
        nop                             ; make sure code takes same
        nop                             ; number of locations
        nop                             ; regardless of RAMINIT condition
        .endif
        nop                             ; in case we did calli
        const     tmp0,exec             ; get target application task address
        consth    tmp0,exec
        mtsrim    OPS,0x172             ; RE=1, PI=1, PD=1, SM=1 and DI=1
        mtsrim    CPS,0x573             ; Set Target application Task address
        mtsr      PC1,tmp0
        add       tmp0,tmp0,4
        mtsr      PC0,tmp0
        xor       tmp0,tmp0,tmp0        ; Any additional regs clean up
        iret                            ; Give control to application via IRET
exec:
        const     lr0,_main             ; get C-callable routine entry point
        consth    lr0,_main
        calli     lr0,lr0               ; make the call
        nop
        sll       gr97,gr64,0           ; Save user global registers gr64
        sll       gr98,gr65,0           ; through gr66
        sll       gr99,gr66,0
        const     gr64,mtp_count        ; get address of memory test pass
        consth    gr64,mtp_count        ; count recorder
        load      0,0,gr65,gr64         ; get current count so far
        cpeq      gr67,gr96,0           ; check for memory test pass?
        jmpt      gr67,again            ; true then run test again
        nop
        halt                            ; false halt further memory testing
again:
        add       gr65,gr65,1           ; bump mtp_count by 1
        store     0,0,gr65,gr64         ; update in memory also
        sll       gr64,gr97,0           ; Restore user global registers gr64
        sll       gr65,gr98,0           ; through gr66
        sll       gr66,gr99,0
        jmp       exec                  ; run the memory test once again
        nop
spilltrap:
        mfsr      tpc,PC1               ; save return address in tpc
        const     tmp0,spill            ; get spill procedure entry point
        consth    tmp0,spill
        mtsr      PC1,tmp0              ; fill Am29000 pipeline target address
        add       tav,tmp0,tmp0+4
        mtsr      PC0,tmp0              ; fill Am29000 pipeline with target address+4
        iret
filltrap:
        mfsr      tpc,PC1               ; save return address in tpc
        const     tmp0,fill             ; get fill procedure entry point
        consth    tmp0,fill
        mtsr      PC1,tmp0              ; fill Am29000 pipeline target address
        add       tav,tmp0,tmp0+4
        mtsr      PC0,tmp0              ; fill Am29000 pipeline with target address+4
        iret
illtrap:
        halt
        .end
```

APPENDIX B: cprog.c

```
#define MT_PASSED      0
#define SOLID_ONES     -1
#define SOLID_ZEROS    0
#define MT_BLK_SIZE    32768
#define WORD_SIZE      4
#define INIT_DATA      170
#define MEM_BLOCK      1056
#define NIT_DATA_BASE  1280
#define INIT_DATA_SIZE 15

int *mt_sts;
int lm_addr,hm_addr;
int initdata;
int *mem_test();

main()
{
    lm_addr =  INIT_DATA_BASE;
    hm_addr =  INIT_DATA_BASE+MT_BLK_SIZE/WORD_SIZE;
    initdata = MEM_BLOCK;
    mt_sts   = mem_test(lm_addr,hm_addr,initdata);
}

int *mem_test(low,high,initd)
int *low,*high,*initd;
{
    int *addr;

                                /* Solid Ones test */
    for(addr=low; addr<=high; addr++)
        *addr = SOLID_ONES;
    for(addr=low; addr<=high; addr++)
        if(*addr != SOLID_ONES)
            return(addr);

                                /* Solid Zeros test */
    for(addr=low; addr<=high; addr++)
        *addr = SOLID_ZEROS;
    for(addr=low; addr<=high; addr++)
        if(*addr != SOLID_ZEROS)
            return(addr);

    for(addr=initd;addr<in itd+INIT_DATA_SIZE;addr+)
        if(*addr != INIT_DATA)
            return(addr);
    return(MT_PASSED);
```

APPENDIX C: r29k.s

```
        .macro I29kGPR,gpr_nu
        xor       gpr_nu,gpr_nu,gpr_nu
        .endm
        .macro    I29kSPR,spr_nu
        mtsrim    spr_nu,0
        .endm
        .macro    I29kMPR,tlbr_nu
        const     gr65,tlbr_nu
        mttlb     gr65,gr66
        .endm
        .text
        .global   r29k_init
r29k_init:
        I29kGPR   gr67                 ; Set GR67-GR127 to known state = 0
        I29kGPR   gr68
        I29kGPR   gr69
        I29kGPR   gr70
        I29kGPR   gr71
        I29kGPR   gr72
        I29kGPR   gr73
        I29kGPR   gr74
        I29kGPR   gr75
        I29kGPR   gr76
        I29kGPR   gr77
        I29kGPR   gr78
        I29kGPR   gr79
        I29kGPR   gr80
        I29kGPR   gr81
        I29kGPR   gr82
        I29kGPR   gr83
        I29kGPR   gr84
        I29kGPR   gr85
        I29kGPR   gr86
        I29kGPR   gr87
        I29kGPR   gr88
        I29kGPR   gr89
        I29kGPR   gr90
        I29kGPR   gr91
        I29kGPR   gr92
        I29kGPR   gr93
        I29kGPR   gr94
        I29kGPR   gr95
        I29kGPR   gr96
        I29kGPR   gr97
        I29kGPR   gr98
        I29kGPR   gr99
        I29kGPR   gr100
        I29kGPR   gr101
        I29kGPR   gr102
        I29kGPR   gr103
        I29kGPR   gr104
        I29kGPR   gr105
        I29kGPR   gr106
        I29kGPR   gr107
        I29kGPR   gr108
        I29kGPR   gr109
        I29kGPR   gr110
        I29kGPR   gr111
        I29kGPR   gr112
        I29kGPR   gr113
        I29kGPR   gr114
        I29kGPR   gr115
        I29kGPR   gr116
        I29kGPR   gr117
        I29kGPR   gr118
        I29kGPR   gr119
        I29kGPR   gr120
        I29kGPR   gr121
        I29kGPR    gr122
        I29kGPR    gr123
        I29kGPR    gr124
        I29kGPR    lr2                    ; Set lr2-lr127 to known state = 0
        I29kGPR    lr3
        I29kGPR    lr4
        I29kGPR    lr5
        I29kGPR    lr6
        I29kGPR    lr7
        I29kGPR    lr8
        I29kGPR    lr9
        I29kGPR    lr10
        I29kGPR    lr11
        I29kGPR    lr12
        I29kGPR    lr13
        I29kGPR    lr14
        I29kGPR    lr15
        I29kGPR    lr16
        I29kGPR    lr17
        I29kGPR    lr18
        I29kGPR    lr19
        I29kGPR    lr20
        I29kGPR    lr21
        I29kGPR    lr22
        I29kGPR    lr23
        I29kGPR    lr24
        I29kGPR    lr25
        I29kGPR    lr26
        I29kGPR    lr27
        I29kGPR    lr28
        I29kGPR    lr29
        I29kGPR    lr30
        I29kGPR    lr31
        I29kGPR    lr32
        I29kGPR    lr33
        I29kGPR    lr34
        I29kGPR    lr35
        I29kGPR    lr36
        I29kGPR    lr37
        I29kGPR    lr38
        I29kGPR    lr39
        I29kGPR    lr40
        I29kGPR    lr41
        I29kGPR    lr42
        I29kGPR    lr43
        I29kGPR    lr44
        I29kGPR    lr45
        I29kGPR    lr46
        I29kGPR    lr47
        I29kGPR    lr48
        I29kGPR    lr49
        I29kGPR    lr50
        I29kGPR    lr51
        I29kGPR    lr52
        I29kGPR    lr53
        I29kGPR    lr54
        I29kGPR    lr55
        I29kGPR    lr56
        I29kGPR    lr57
        I29kGPR    lr58
        I29kGPR    lr59
        I29kGPR    lr60
        I29kGPR    lr61
        I29kGPR    lr62
        I29kGPR    lr63
        I29kGPR    lr64
        I29kGPR    lr65
        I29kGPR    lr66
        I29kGPR    lr67
        I29kGPR    lr68
```

```
          I29kGPR    lr69
          I29kGPR    lr70
          I29kGPR    lr71
          I29kGPR    lr72
          I29kGPR    lr73
          I29kGPR    lr74
          I29kGPR    lr75
          I29kGPR    lr76
          I29kGPR    lr77
          I29kGPR    lr78
          I29kGPR    lr79
          I29kGPR    lr80
          I29kGPR    lr81
          I29kGPR    lr82
          I29kGPR    lr83
          I29kGPR    lr84
          I29kGPR    lr85
          I29kGPR    lr86
          I29kGPR    lr87
          I29kGPR    lr88
          I29kGPR    lr89
          I29kGPR    lr90
          I29kGPR    lr91
          I29kGPR    lr92
          I29kGPR    lr93
          I29kGPR    lr94
          I29kGPR    lr95
          I29kGPR    lr96
          I29kGPR    lr97
          I29kGPR    lr98
          I29kGPR    lr99
          I29kGPR    lr100
          I29kGPR    lr101
          I29kGPR    lr102
          I29kGPR    lr103
          I29kGPR    lr104
          I29kGPR    lr105
          I29kGPR    lr106
          I29kGPR    lr107
          I29kGPR    lr108
          I29kGPR    lr109
          I29kGPR    lr110
          I29kGPR    lr111
          I29kGPR    lr112
          I29kGPR    lr113
          I29kGPR    lr114
          I29kGPR    lr115
          I29kGPR    lr116
          I29kGPR    lr117
          I29kGPR    lr118
          I29kGPR    lr119
          I29kGPR    lr120
          I29kGPR    lr121
          I29kGPR    lr122
          I29kGPR    lr123
          I29kGPR    lr124
          I29kGPR    lr125
          I29kGPR    lr126
          I29kGPR    lr127
                                           ; Set sp1,sp4-sp9 to known state = 0
          I29kSPR    OPS                   ; Set sp13 and sp14 to known state = 0
          I29kSPR    CHA
          I29kSPR    CHD
          I29kSPR    CHC
          I29kSPR    RBP
          I29kSPR    TMC
          I29kSPR    TMR
          I29kSPR    MMU
          I29kSPR    LRU
          I29kSPR    IPC                   ; Set sp128-135 to known state = 0
```

```
        I29kSPR   IPA
        I29kSPR   IPB
        I29kSPR   Q
        I29kSPR   ALU
        I29kSPR   BP
        I29kSPR   FC
        I29kSPR   CR
        const     gr66,0
        I29kMPR   0                    ; Set tr0-tr127 to known state = 0
        I29kMPR   1
        I29kMPR   2
        I29kMPR   3
        I29kMPR   4
        I29kMPR   5
        I29kMPR   6
        I29kMPR   7
        I29kMPR   8
        I29kMPR   9
        I29kMPR   10
        I29kMPR   11
        I29kMPR   12
        I29kMPR   13
        I29kMPR   14
        I29kMPR   15
        I29kMPR   16
        I29kMPR   17
        I29kMPR   18
        I29kMPR   19
        I29kMPR   20
        I29kMPR   21
        I29kMPR   22
        I29kMPR   23
        I29kMPR   24
        I29kMPR   25
        I29kMPR   26
        I29kMPR   27
        I29kMPR   28
        I29kMPR   29
        I29kMPR   30
        I29kMPR   31
        I29kMPR   32
        I29kMPR   33
        I29kMPR   34
        I29kMPR   35
        I29kMPR   36
        I29kMPR   37
        I29kMPR   38
        I29kMPR   39
        I29kMPR   40
        I29kMPR   41
        I29kMPR   42
        I29kMPR   43
        I29kMPR   44
        I29kMPR   45
        I29kMPR   46
        I29kMPR   47
        I29kMPR   48
        I29kMPR   49
        I29kMPR   50
        I29kMPR   51
        I29kMPR   52
        I29kMPR   53
        I29kMPR   54
        I29kMPR   55
        I29kMPR   56
        I29kMPR   57
        I29kMPR   58
        I29kMPR   59
        I29kMPR   60
        I29kMPR   61
```

```
I29kMPR    62
I29kMPR    63
I29kMPR    64
I29kMPR    65
I29kMPR    66
I29kMPR    67
I29kMPR    68
I29kMPR    69
I29kMPR    70
I29kMPR    71
I29kMPR    72
I29kMPR    73
I29kMPR    74
I29kMPR    75
I29kMPR    76
I29kMPR    77
I29kMPR    78
I29kMPR    79
I29kMPR    80
I29kMPR    81
I29kMPR    82
I29kMPR    83
I29kMPR    84
I29kMPR    85
I29kMPR    86
I29kMPR    87
I29kMPR    88
I29kMPR    89
I29kMPR    90
I29kMPR    91
I29kMPR    92
I29kMPR    93
I29kMPR    94
I29kMPR    95
I29kMPR    96
I29kMPR    97
I29kMPR    98
I29kMPR    99
I29kMPR    100
I29kMPR    101
I29kMPR    102
I29kMPR    103
I29kMPR    104
I29kMPR    105
I29kMPR    106
I29kMPR    107
I29kMPR    108
I29kMPR    109
I29kMPR    110
I29kMPR    111
I29kMPR    112
I29kMPR    113
I29kMPR    114
I29kMPR    115
I29kMPR    116
I29kMPR    117
I29kMPR    118
I29kMPR    119
I29kMPR    120
I29kMPR    121
I29kMPR    122
I29kMPR    123
I29kMPR    124
I29kMPR    125
I29kMPR    126
I29kMPR    127
jmpi       lr0                    ; return to caller
const      gr65,0
.end
```

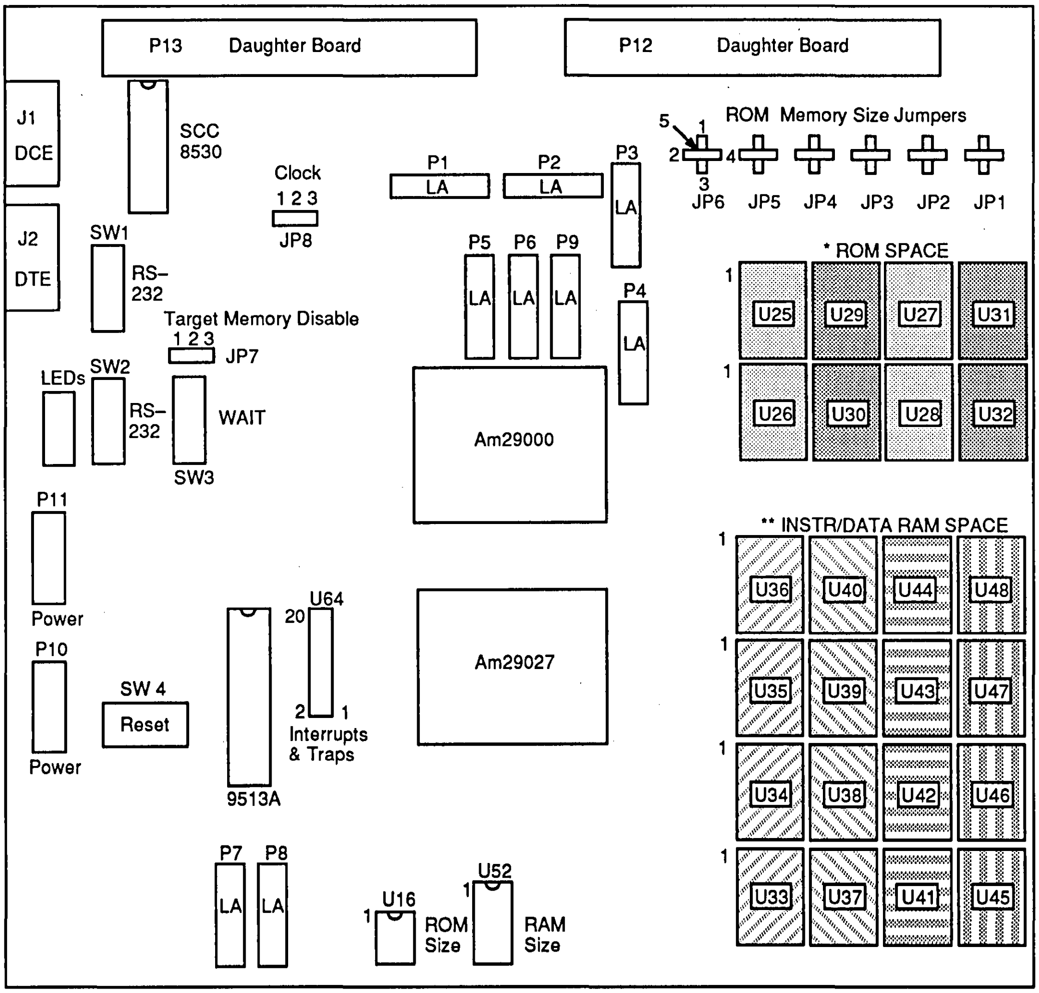
APPENDIX D: traps.s

```
;
;                       TRAPS.S
;
        .text
        .global spill,fill              ; Spill and fill process
        .include   "scregs.def"
spill:
        sub        tav,rab,rsp          ; compute spill: lower bound - sp
        sub        rfb,rfb,tav          ; adjust rfb pointer
        srl        tav,tav,2            ; shift to get number of words
        sub        tav,tav,1            ; count is one less
        mtsr       CR,tav               ; set Count Remaining register
        storem     0,0,lr0,rfb          ; spill
        sll        rfb,rab,0            ; adjust rfb pointer
        jmpi       tpc                  ; return to "caller"
        sll        rab,rsp,0            ; adjust rab
fill:
        const      tav,0x80<<2          ; local register bit
        or         tav,tav,rfb          ; in rfb for IPA
        mtsr       IPA,tav              ; IPA gets starting register number
        sub        tav,lr1,rfb          ; compute number of bytes to fill
        add        rab,rab,tav          ; push up the allocate bound
        srl        tav,tav,2            ; change byte count to word count
        sub        tav,tav,1            ; make count zero-based
        mtsr       CR,tav               ; set Count Remaining register
        sll        tav,rfb,0            ; save old rfb
        sll        rfb,lr1,0            ; push up the free bound
        jmpi       tpc                  ; return to "caller"
        loadm      0,0,gr0,tav          ; fill
        .end
```

APPENDIX E: scregs.def

```
        .reg      rsp,gr1                 ; register stack pointer
        .reg      msp,gr125               ; memory stack pointer
        .reg      rab,gr126               ; register allocate bound
        .reg      rfb,gr127               ; register free bound
        .reg      tpc,gr121               ; trap handler argument/temp
        .reg      tav,gr122               ; trap handler return address/temp
        .reg      tmp0,gr64               ; temp registers allocations
        .reg      tmp1,gr65
        .reg      tmp2,gr66
```

APPENDIX F: CONFIGURATION OF THE STEB



Footnotes:

░	* ROM Space Bank 0	▨	** RAM Space Bank 0	▤	** RAM Space Bank #2
▒	* ROM SPACE BANK 1	▧	** RAM Space Bank 1	▥	** RAM Space Bank #3

Note: The STEB uses PROMs (can be MON29K) in ROM space bank 0; otherwise can have RAMs in ROM space bank 0 for downloading programs using ADAPT29K.

11966A-13

Figure 13. Configuration of the STEB

Programming Standalone Am29000 Systems Application Note

by Jim Gibbons and Doug Walton

INTRODUCTION

Advanced Micro Devices is developing a complete line of Am29000™ simulators, hardware-target execution vehicles, and high-level language development tools for the Am29000 32-bit Streamlined Instruction Processor. These products are designed to support end-users who are building embedded system applications based on the Am29000 processor. For these users, often there is no existing operating system or kernel for their hardware design.

A standalone program runs independently of an operating system or other supporting software. As opposed to a program that runs under an operating system, a standalone program is concerned about the characteristics of the hardware environment. It controls hardware devices and must be aware of the system architecture. Consequently, the needed executive functions that would be performed by an operating system must be designed into the application program.

HOW TO USE THIS APPLICATION NOTE

This document covers some important issues in programming a standalone Am29000 system. Its purpose is not to explain every possible implementation of the Am29000, but to present a basic framework from which to start development.

Many sample sections of code are shown. Most are taken from the STARTUP files provided on the ASM29K™ software and are listed in the appendices. These files can be consulted for a complete example of the boot-up and initialization process. Be aware that the range of possible applications in which the Am29000 can be used is extensive, and it would be impossible to provide code that will work in every situation. The code samples have been tested in simple, limited applications, and should be used as a guideline, not as a finished solution.

The Effects of Memory Organization section discusses how the memory organization of an Am29000 system affects the design of a standalone program. Attention is given to the location from which code is executed and how it is accessed.

The Am29000 Calling Convention section summarizes the Am29000 run-time model. Writing good Am29000 assembly-language programs requires knowledge of the run-time model. Code samples used in this application note follow the convention established by the run-time model. Understanding the convention eases understanding the examples.

The Writing the Start-up Program section explains how an example startup program works. Each task done in the process is discussed, from configuring the Am29000 through calling _main.

SUGGESTED REFERENCE MATERIALS

This application note covers fundamental design issues involved in implementing a standalone Am29000 system. However, designing a standalone system is a complex task involving many areas. Knowledge of the Am29000 is necessary, as well as the subjects covered in the following reference materials.

Am29000 Streamlined Instruction Processor User's Manual, order #10620. It contains details regarding the instruction set and register organization of the Am29000.

Am29000 Streamlined Instruction Processor Data Sheet, order #09075. It embodies a great deal of information about the Am29000, including distinctive characteristics, general description, simplified system diagram, connection diagram, pin designations and descriptions, functional description, absolute maximum ratings, operational ranges, DC characteristics, switching characteristics and waveforms, and physical dimensions.

Am29000 Memory Design Handbook, order #10623. It discusses in detail the tradeoffs in designing an Am29000 memory system. Completely covers four different approaches to optimizing access speed versus cost and memory size.

Implementation of an Am29000 Stack Cache Application Note. It describes in detail how a stack cache would be used in a simple application.

Publication #	Rev.	Amendment	Issue Date:
11025	A	/0	11/89

These materials can be obtained by writing to:

Advanced Micro Devices, Inc.
901 Thompson Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

or by calling (800) 222-9323.

For questions that cannot be resolved with the current literature, further technical support can be obtained by writing or calling:

29K Support Products Engineering
Mail Stop 561
5900 E. Ben White Blvd.
Austin, TX 78741
(800) 2929-AMD (US)
0-800-89-1131 (UK)
0-031-11-1129 (Japan)

THE EFFECTS OF MEMORY ORGANIZATION

The organization of memory determines some of the duties that software must perform. The physical characteristics of the memory design have an impact on the system responsibilities in a standalone environment. Where the various types of memory are located and how they are accessed must be considered.

While many types of memory organization are possible in an Am29000 system, this discussion covers only a couple of the more widely known methods. The emphasis is not on describing all of the possibilities, but on showing how the duties of the standalone program change depending on how the system memory is arranged. For more information on the advantages and disadvantages of various Am29000 memory schemes, see the Am29000 Memory Design Handbook.

MEMORY SPACES

The Am29000 uses a three-bus Harvard architecture, which allows for many different types of memory organization. As shown in Figure 1, the Am29000 buses include the address bus, the data bus, and the instruction bus. All are 32 bits wide, but only the data bus is bidirectional. The address bus is output-only; the instruction bus is input-only. Using the buses and some control signals, the Am29000 supports five separate memory spaces. The available spaces are register, I/O, instruction ROM, coprocessor, and instruction/data RAM.

In any given system, the application program will reside and execute in some memory area(s), and will execute from some area(s). The areas can be the same, but they also can be different. Sometimes, the application program will need to be transcribed from one space into another before execution.

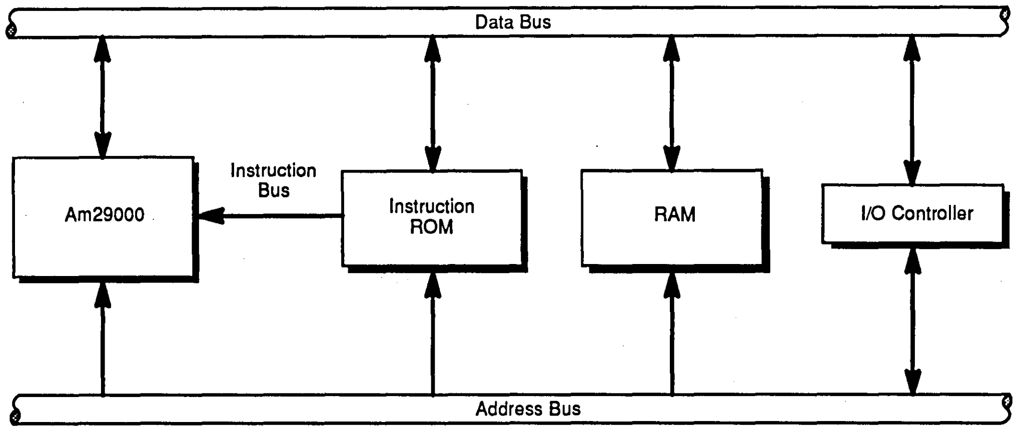
When code is transferred from one memory area to another, it is usually done so that a higher rate of execution can be achieved. Because the Am29000 is very fast, it can be limited by the access time of memory. Yet, high-speed PROMs are very expensive. Often it is more cost-effective to transcribe the code from slow PROMs to high-speed RAMs before execution.

BUS ARCHITECTURE

Bus architecture influences how data and instructions are transferred from one memory space to another. The Am29000 system in Figure 1 has separate Instruction ROM and instruction/data RAM areas. Code could be transcribed into the instruction RAM area from the instruction ROM using a series of const and consth instructions, but a problem would be evident: the Am29000 fetches the instructions from the instruction bus, regardless of the memory space in which the instructions reside. With the system in Figure 1, code transcribed to RAM cannot be executed because there is no access to the instruction bus.

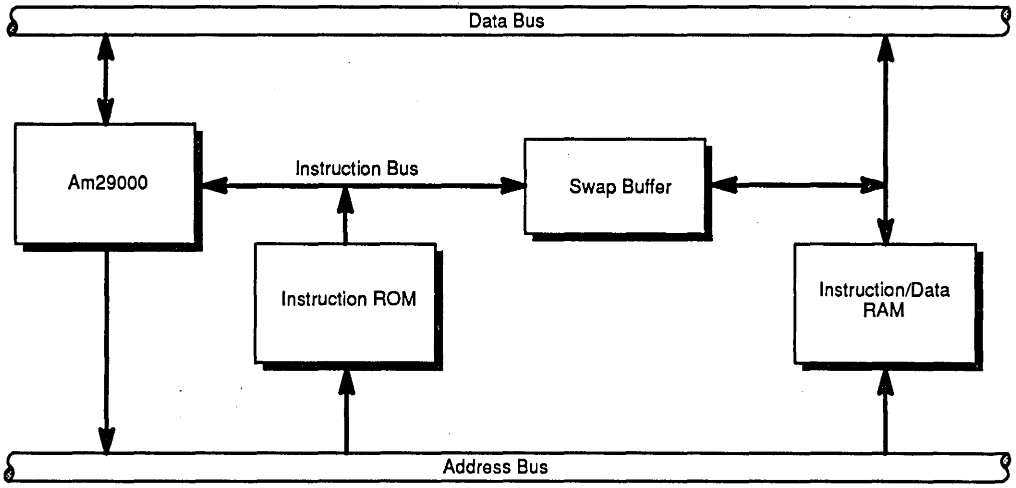
One method of resolving this problem is to establish a path between the data bus and the instruction bus. Such a path can be provided through a swap buffer, as shown in Figure 2. The swap buffer is bidirectional, which allows data or instructions on one bus to be moved to the other.

A different solution is used on AMD's PC Execution Board (PCEB29K™), where fixed storage for data and programs is on the host PC. When code is to be run, it is loaded into video DRAM (VDRAM) installed in the instruction/data RAM space. The dual-ported VDRAM has its shifter output connected to the Am29000 instruction bus and its data bus connected to the Am29000 data bus. In this way, the same physical address space exists on both buses, and data can be read or written via either the instruction bus or the data bus (see Figure 3).



11025A-01

Figure 1. A Typical Am29000 System



11025A-02

Figure 2. An Am29000 System with Swap Buffers

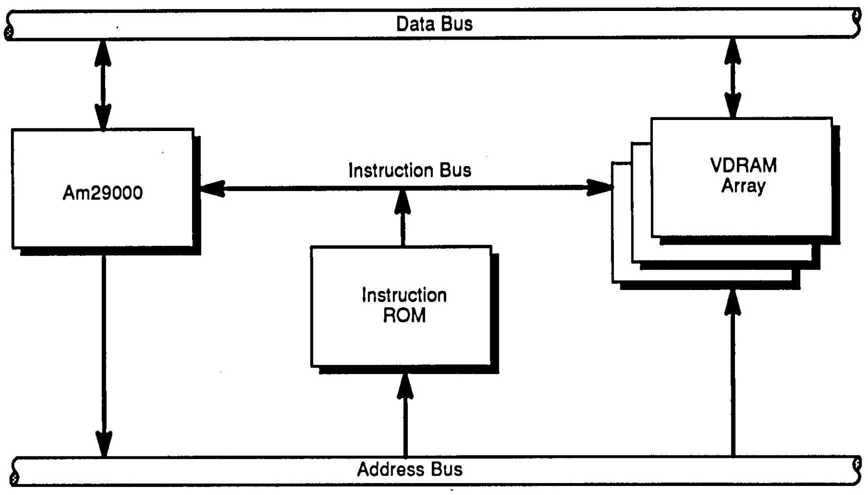


Figure 3. An Am29000 System with VDRAM

Am29000 CALLING CONVENTIONS

To enhance code readability and accuracy, the Am29000 run-time model convention is used. This convention defines standards for register declarations, parameters passing, spill and fill routines, and other topics.

There are many good reasons for using the Am29000 run-time model. First, it allows assembly-language programs to interface with C programs compiled by the HighC29K™ compiler. Second, it makes programs easier to understand, particularly for other developers making modifications or complementary products. Third, it has been tested thoroughly in many different environments. Using it from the start will likely save time later in the development process.

This section is a summary of the Am29000 run-time model. Because the code samples in the "Writing the Start-up Program" section follow the convention established by the run-time model, understanding it will make the code samples clearer. See also the *Am29000 Streamlined Instruction Processor User's Manual.*

DECLARATIONS

A file containing the declarations outlined in the convention normally is called into each module that uses the definitions. A declarations file can be called into an assembly-language source file by inserting a statement (usually at or near the top of the file) like:

```
.include "romdcl.h"
```

In this example, a declarations file named romdcl.h would be used with the program. For convenience, the declarations required to understand the code sections in this document are summarized in Table 1.

THE Am29000 RUN-TIME STORAGE ORGANIZATION

In a high-level language that supports nested function calls (such as C), specific information related to each function invocation often is stored on a run-time stack. The Am29000 run-time stack is actually two stacks. One is the register stack; the other is the memory stack.

Both stacks start at an arbitrary high address in memory and grow downward as function calls nest deeper. The "bottom" of the stack is the high address where the stack starts; the "top" of the stack is where the last stack item was placed, or the address of the lowest valid location.

Table 1. Summary of Am29000 Register Names

Protected Special Purpose Register Names		
vab	0	Vector Area Base Address
ops	1	Old Processor Status
cps	2	Current Processor Status
cfg	3	Configuration Register
cha	4	Channel Address
chd	5	Channel Data
chc	6	Channel Control
rbp	7	Register Bank Protect
tmc	8	Timer Counter
tmr	9	Timer Reload
pc0	10	Program Counter 0
pc1	11	Program Counter 1
pc2	12	Program Counter 2
mmu	13	MMU Configuration
lru	14	LRU Recommendation
Unprotected Special Purpose Register Names		
ipc	128	Indirect Pointer C
ipa	129	Indirect Pointer A
ipb	130	Indirect Pointer B
q	131	q
alu	132	ALU Status
bp	133	Byte Pointer
fc	134	Funnel Shift Count
cr	135	Load/Store Count Remaining

The register stack contains dynamically allocated information pertaining to the local state of a given function call, such as incoming arguments, local variables, and outgoing arguments being passed to another function. These function-specific data are organized into a series of overlapping structures called activation records or stack frames. A function is active when invoked, and each active function has an activation record somewhere on the register stack. When a function is entered, a new activation record, or register stack frame, is created; when the function is exited, its activation record is removed. An activation record is shown in Figure 4.

An important characteristic of activation records is that, because the outgoing arguments of a calling function ("caller") are the incoming arguments of the called function ("callee"), the callee's stack frame overlaps with the caller's stack frame. Consequently, except for the first activation record on the stack, the incoming arguments of the callee are identical to the outgoing arguments from the caller for each nested function. Figure 5 shows how activation records overlap on the register stack.

Because the Am29000 has a large, pointer-addressable internal local register file, it is possible to cache a portion of the register stack in local registers (see Figure 6). Where the next byte is placed is determined by *rsp* (the register stack pointer). The global register GR1 is assigned as the *rsp* because it can point to the current stack position in external memory, while bits 2–9 identify the current *lr0*. Activation records are allocated by subtracting the size of the frame needed from *rsp*, thus allocating a new block of local registers unique to this function invocation.

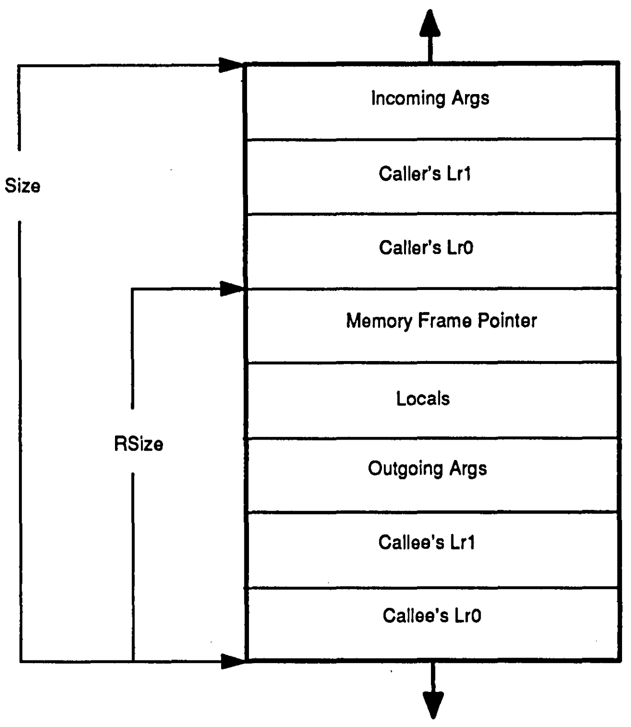


Figure 4. An Activation Record

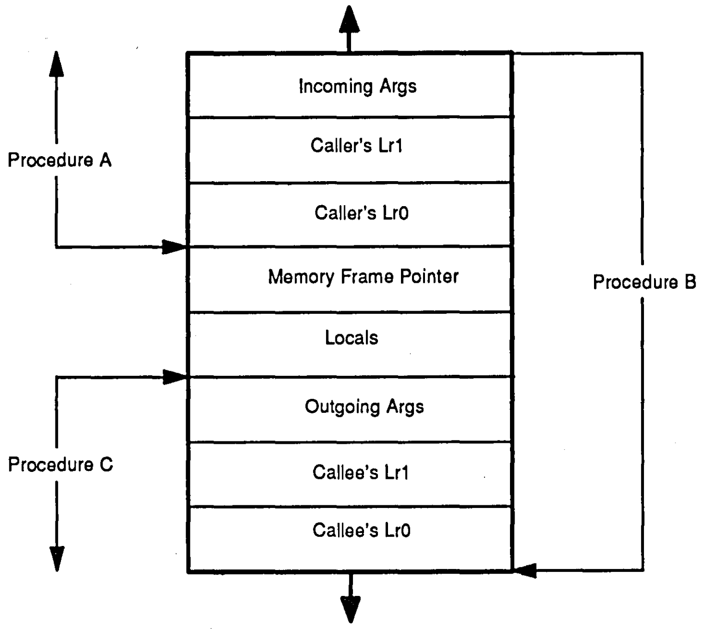
Caching the register stack introduces the operations described below:

Spill. The portion of the register stack cached in local registers cannot exceed 128; if it does, the oldest arguments are spilled to external memory. A spill occurs when *rsp* becomes less than *rab* (the register allocate bound).

Prologue. A prologue routine is an assembly-language macro that, given the number of incoming arguments, outgoing arguments, and local arguments, will allocate a register stack frame for the function.

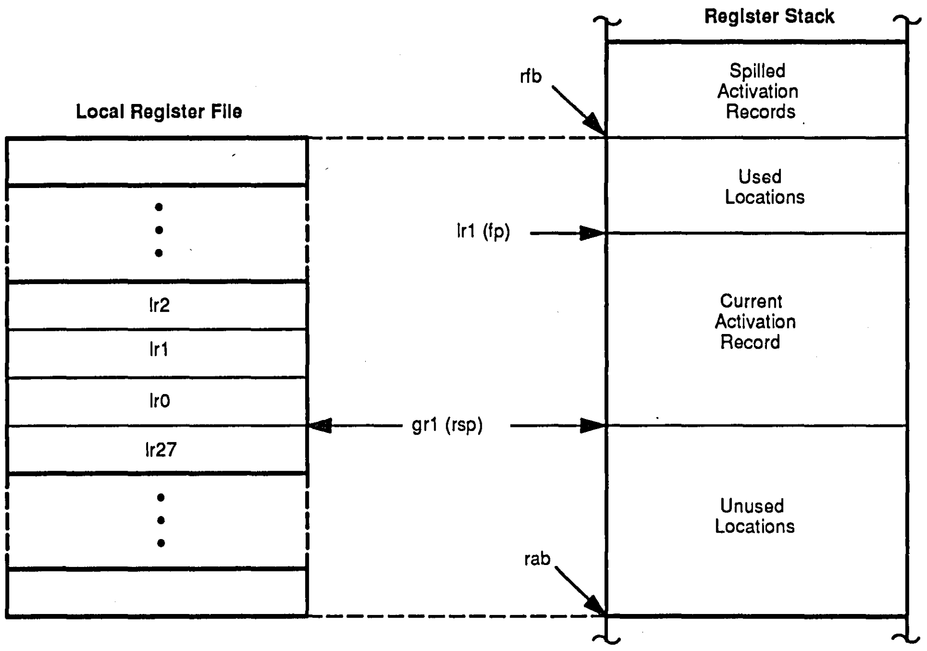
Epilogue. An epilogue routine is an assembly-language macro that deallocates the register stack frame and causes a jump to the return address.

Fill. When control is being returned to calling functions, a previously spilled activation record may not exist in the local register file. Then the register file needs to be filled from the register stack in external memory. A fill occurs when *rsp* is higher than *rfb* (the register free bound).



11025A-05

Figure 5. The Register Stack



11025A-06

Figure 6. The Stack Cube

WRITING THE START-UP PROGRAM

System initialization is one of the most critical duties performed by software in the standalone system. Devices must be configured, memory set up, and traps and vectors defined. In short, an execution environment must be prepared for the application program. If this is not done properly, the main application program will not function properly, and could contain difficult-to-find errors. So careful attention must be given to the routines that initialize the system.

This section discusses writing an assembly-language module that will establish the execution environment for a C application program. To demonstrate this, an example program is developed in a step-by-step fashion.

The example application is designed to run on an Am29000 system similar to the system shown in Figure 7. The system provides a generic Am29000 environment with instruction/data RAM (VDRAM), instruction ROM, and a dual-port 8530 serial communications controller (SCC). The dual-port VDRAM allows instructions to be read from RAM.

The example program consists of three assembly-language modules and a declarations file. The assembly-language module **START.S** (listed in Appendix B) is startup code that establishes the environment for a C-language program. The assembly-language module **BOOT.S** (listed in Appendix A) transfers the **START.S** and the C-language application code to RAM, as shown by the black arrows in Figure 7. **BOOT.S** then passes control to **START.S**. The final assembly-language program is **TEST.S** (listed in Appendix C). **TEST.S** simulates a C-language application and tests whether the startup has been properly performed. The declarations file (**ROMDCL.H**) and the linker command file (**TEST.LD**) are listed in Appendices D and E, respectively.

MAKING A BOOT.S MODULE TO TRANSCRIBE CODE

BOOT.S receives control first. It establishes serial communications, tests RAM, and transcribes the application code into RAM. The sequence performed by **BOOT.S** is:

1. Configure the Am29000.
2. Establish a register stack frame.
3. Initialize serial I/O for error reporting.
4. Test RAM.
5. Set pointers to invalid trap handler.
6. Call RAMInit (made by ROMCOFF) to transcribe code.
7. Transfer control to **START.S**.

Step 1— Configuring the Am29000

BOOT.S first configures the Am29000's current processor status register (*cps*) to a known state by executing the instruction:

```
mtsrim cps,0x173 ;RE,PD,PI,SM,DI,DA
```

This instruction enables instruction fetching from ROM (RE = 1), sets address translation for data and instruc-

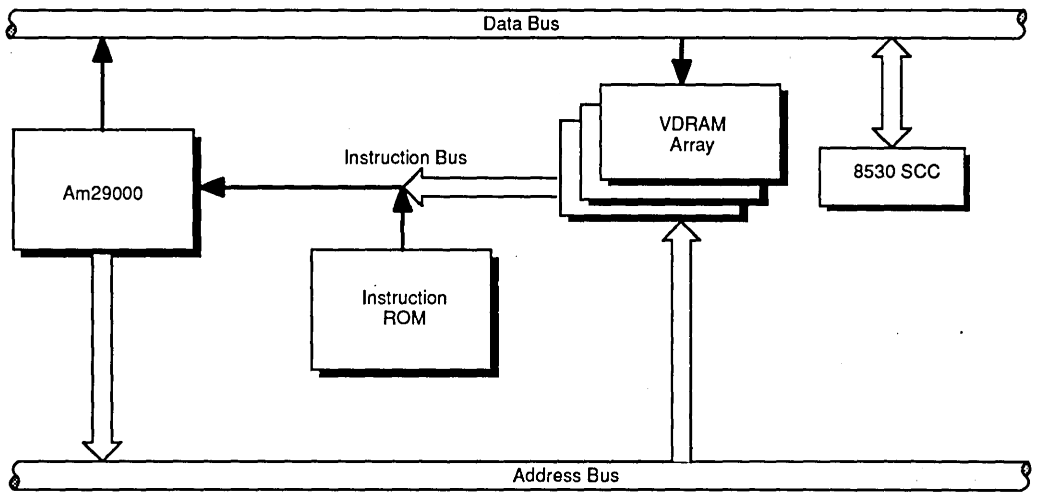


Figure 7. Example Am29000 System

11025A-07

tions off (PD,PI = 1), turns on supervisor mode (SM = 1), and disables all interrupts and traps (DI,DA = 1).

Step 2—Establishing a Simple Register Stack Frame

BOOT.S calls several procedures, so it establishes a Register Stack Frame. However, control will not return to **BOOT.S** after calling _main. Therefore, it only needs to use a limited stack frame. The frame is set up with:

```
const   rfb, 512      ;set up temp reg
frame
const   rab, 0
sub     rsp, rfb, 16 ;enough for p0 and
p1
add     lr1, rfb, 0
```

Step 3—Initializing I/O Devices

An I/O device is initialized early, so that it can be used to transmit error messages. The 8530 serial communications controller is initialized using the routine shown in Listing 1.

Listing 1. Initializing I/O Devices

```
SerInit:
          .reg        SI_CtAd, %%(TEMP_REG + 0)        ;control port address
          .reg        SI_CtVl, %%(TEMP_REG + 1)        ;control port value
          const       SI_CtAd, SCCCntlAd
          consth      SI_CtAd, SCCCntlAd
          const       SI_CtVl, 9                       ;reset the port
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0xc0
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 4                       ;x16, 1 stop, no parity
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x44
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 3                       ;8 bits receive
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0xc0
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 5                       ;8 bits xmit
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x60
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 9                       ;Int. disabled
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x0
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 10                      ;NRZ
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x0
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 11                      ;Tx & Rx BRG out
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x56
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 12                      ;9600 baud
          store       0, 0, SI_CtVl, SI_CtAd
          const       SI_CtVl, 0x6
          store       0, 0, SI_CtVl, SI_CtAd
```

Listing 1. Initializing I/O Devices (continued)

```
        const      SI_CtVl, 13                     ;9600 baud
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 0x0
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 14                     ;BRG in RTxC
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 0x0
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 14                     ;BRG on
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 0x1
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 3                      ;Rx enable
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 0xc1
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 5                      ;Tx enable
        store      0, 0, SI_CtVl, SI_CtAd
        const      SI_CtVl, 0xea
        store      0, 0, SI_CtVl, SI_CtAd
        EPILOGUE
```

Step 4—Testing RAM

The RAM is tested before code is transferred to it. **BOOT.S** calls a single test, an address pattern test. Other tests are included in the source listing shown in Appendix A. The test used by **BOOT.S** is shown in Listing 2.

Step 5—Setting the Vector Table Entries to the Invalid Trap Handler

START.S will set up the vector table, but **BOOT.S** guards against abnormal ends by making all of the vector table entries point to an invalid trap handler in ROM. This is done with the following routine, which is called from the main loop, as shown in Listing 3.

Listing 2. Testing RAM

```
        .sbttl     "RAM Address Pattern Test"
        FUNCTION   RAMAddr, 2, 0, 3
;
; This routine will run a two-pass test on RAM. It will be controlled by input values
; specifying the base address and the count of locations to be tested. In the first
; pass, the data will be set equal to the address.  In the second pass, the data
; will be set equal to the complement of the address.
;
;       In:        (see below)
;
;       Out:       (see below)
;
        .reg       RA_StrtAdd, %%(IN_PRM + 0)      ;starting address
        .reg       RA_WrdCnt, %%(IN_PRM + 1)       ;count of words
        .reg       RA_TmpCnt, %%(TEMP_REG + 0)     ;total test word count
        .reg       RA_StrtPat, %%(TEMP_REG + 1)    ;starting pattern
        .reg       RA_PtrnInc, %%(TEMP_REG + 2)    ;ptrn increment value
        .reg       RA_NxtAdd, %%(OUT_PRM + 0)      ;error address
        .reg       RA_WrtPat, %%(OUT_PRM + 1)      ;pattern written
        .reg       RA_RedPat, %%(OUT_PRM + 2)      ;pattern read
```

Listing 2. Testing RAM (continued)

```
        .reg      RA_Fail,    %%(RET_VAL + 0)        ;TRUE for fail
        add       RA_StrtPat, RA_StrtAdd, 0          ;start with address
        const     RA_PtrnInc, 4

RA_1:   ;fill memory with pattern
        add       RA_NxtAdd, RA_StrtAdd, 0           ;get start address
        sub       RA_TmpCnt, RA_WrdCnt, 2            ;for jmpfdec
        add       RA_WrtPat, RA_StrtPat, 0           ;set the pattern

RA_2:
        store     0, 0, RA_WrtPat, RA_NxtAdd
        add       RA_WrtPat, RA_WrtPat, RA_PtrnInc
        jmpfdec   RA_TmpCnt, RA_2
        add       RA_NxtAdd, RA_NxtAdd, 4            ;next test mem addr
        ;check memory for pattern
        add       RA_NxtAdd, RA_StrtAdd, 0           ;get start address
        sub       RA_TmpCnt, RA_WrdCnt, 2            ;for jmpfdec
        add       RA_WrtPat, RA_StrtPat, 0           ;set the pattern

RA_3:
        load      CD, DATA_CTL, RA_RedPat, RA_NxtAdd
        cpneq     RA_Fail, RA_RedPat, RA_WrtPat      ;err if neq
        jmpt      RA_Fail, RA_ERR
        nop
        add       RA_WrtPat, RA_WrtPat, RA_PtrnInc
        jmpfdec   RA_TmpCnt, RA_3
        add       RA_NxtAdd, RA_NxtAdd, 4            ;next test mem address
        ;invert ptrn for next pass
        nor       RA_StrtPat, RA_StrtPat, 0          ;invert initial
        cpneq     RA_Fail, RA_StrtPat, RA_StrtAdd
        jmpt      RA_Fail, RA_1
        subr      RA_PtrnInc, RA_PtrnInc, 0          ;negate inc value
        jmp       RA_EXIT
        nop

RA_ERR:
        call      lr0, RAMErr
        nop
        const     RA_Fail, TRUE                      ;set after call
        consth    RA_Fail, TRUE

RA_EXIT:
        EPILOGUE
```

Listing 3. Setting Vector Table Entries

```
        .sbttl      "Vector Initialization"
        LEAF        VectInit, 0

;
; This routine initializes the vector table and vab.  All vectors
; are set to point to the invalid trap handler in ROM.
;
        .reg        VI_Vect,    %%(TEMP_REG + 0)       ;vector value
        .reg        VI_VectSt,  %%(TEMP_REG + 1)       ;vector storage address
        .reg        VI_VectCnt, %%(TEMP_REG + 2)       ;vector count register
        mtsrim      vab, 0
        mfsr        VI_VectSt, vab
        const       VI_Vect, (InvalidTrapHandler | 2)
        consth      VI_Vect, InvalidTrapHandler
        const       VI_VectCnt, (256 - 2)              ;for jmpfdec

VI_Loop:
        store       0, 0, VI_VectSt, VI_Vect           ;store the vector
        jmpfdec     VI_VectCnt, VI_Loop
        add         VI_VectSt, VI_VectSt, 4
        EPILOGUE
```

Step 6—Transcribing Code to RAM

BOOT.S transcribes **START.S** and the C-language application (simulated by **TEST.S**) into instruction/data RAM by calling RAMInit.

RAMInit is a routine that is created by the ROMCOFF utility. When an executable Am29000 object file is submitted to ROMCOFF, the utility generates a relocatable object file of type RI_Text that (when called) establishes an image of the executable module in instruction/data RAM. **BOOT.S** transfers **START.S** and the C-language application to RAM by calling the RAMInit routine created by ROMCOFF.

RAMInit is called by:

```
call  RI_Ret,RAMInit  ;initialize RAM
```

Note that when RAMInit is called, the return address is not stored in a local register (such as *lr0*), and that RAMInit is called just before transferring control to _main. To transcribe data to RAM, RAMInit will create a stream of const and consth instructions that will load up the local registers starting from *lr0*. Then it will insert a store multiple command to transfer the data into memory. Consequently, any data in local registers will be overwritten.

Step 7—Calling START.S

As **BOOT.S** does not intend to have control returned to it, it calls **START.S** by simulating a return from interrupt. This is accomplished by setting the freeze (FRZ) bit ON in the old processor status (*ops*) and current processor status registers (*cps*), putting the starting address of **START.S** in *PC0*, and performing a return from interrupt (see Listing 4).

The Main Loop of BOOT.S

When all of the preceding steps are put together, the main loop appears as shown in Listing 5.

Listing 4. Calling START.S

```
        mtsrim      ops, 0x473                         ;FZ, PD, PI, SM, DI, DA
        mtsrim      cps, 0x473                         ;FZ, PD, PI, SM, DI, DA
        const       lr0, TextBas                       ;(using lr0 as temp)
        consth      lr0, TextBas
        mtsr        pc1, lr0
        add         lr0, lr0, 4
        mtsr        pc0, lr0
        iretinv                                        ;go to inst space, TextBas
```

Listing 5. Main Loop of BOOT.S

```
Boot:
          .reg        RI_Ret, %%(TEMP_REG + 0)    ;RAMInit return
          mtsrim      cps, 0x173                  ;RE, PD, PI, SM, DI, DA
          const       rfb, 512                    ;set up temp reg frame
          const       rab, 0
          sub         rsp, rfb, 16                ;enough for p0 and p1
          add         lr1, rfb, 0
          call        lr0, SerInit                ;initialize an 8530 to report errors
          nop
          const       p1, (RAM_SIZE >> 2)         ;test full RAM size
          consth      p1, (RAM_SIZE >> 2)
          call        lr0, RAMAddr                ;call a RAM address test
          const       p0, 0                       ;test from addr 0 (input parm) to RAM test
                                                  ;to RAM test
          call        lr0, VectInit               ;routine to initialize traps to
          nop                                     ;invalid trap handler
          call        RI_Ret, RAMInit             ;initialize RAM -- from ROMCOFF
          mtsrim      ops, 0x473                  ;FZ, PD, PI, SM, DI, DA
          mtsrim      cps, 0x473                  ;FZ, PD, PI, SM, DI, DA
          const       lr0, TextBas                ;(using lr0 as temp)
          consth      lr0, TextBas
          mtsr        pc1, lr0
          add         lr0, lr0, 4
          mtsr        pc0, lr0
```

CREATING THE EXECUTION ENVIRONMENT WITH START.S

The **START.S** file is used to prepare the execution environment for the application program (simulated by **TEST.S**). Although a given application certainly will have varied requirements in different hardware environments, the tasks that will be performed by **START.S** are needed to establish virtually any operating environment on the Am29000. These are:

1. Configure the Am29000.
2. Allocate the register and memory stacks.
3. Initialize vector table and trap handlers.
4. Initialize the TLB by marking all entries invalid.
5. Call "main."

Step 1—Configuring the Am29000

Code similar to that shown below can be used to set the contents of the cfg so that the vector area is a table of pointers (VF = 1) and the Branch Target Cache™ is disabled (CD = 1). Also, the cps register is set so that physical addressing is used for both instructions and data (PD = 1, PI = 1), all interrupts and traps are disabled (DI = 1), and supervisor mode is ON (SM = 1). The timer (tmr) is also set to 0 to avoid unwanted timer interrupts:

```
mtsrim  tmr,  0
mtsrim  cfg,  (VF|CD)
mtsrim  cps,  (PD|PI|SM|DI)
```

The setting of the VF bit has determined the structure of the vector area table. The vector area is a user-managed table in external instruction/data memory that starts at the address held in the vector area base (VAB) register. The vector area can have one of two different structures, as determined by the VF bit of the configuration register.

If VF = 1, then the vector area is organized as a list of 256 pointers to interrupt/trap handlers. If VF = 0, then the vector area is arranged as 256 64-instruction blocks, each corresponding to a given call. Each fixed block then contains the corresponding interrupt or trap handler. Figure 8 shows the two structures.

When the Am29000 receives an interrupt or trap, the location of the appropriate handler is determined by the vector area (VA). Each interrupt and trap has a vector number between 0 and 255 that corresponds to an entry in the vector area. Of the vector numbers, 0 to 63 are reserved for system and floating-point operations. The assigned vector numbers are given in the *Am29000 User's Manual.*

If the table is a list of pointers, control will be passed to the address at VAB + (vector number * 4). Multiplication by 4 adjusts the vector number to words. If the vector table is composed of handlers, control will be passed to a handler starting at VAB + (vector number * 64 * 4), where the vector number is adjusted to words and multiplied by the number of instructions per block (fixed) (see Table 2).

Table 2. The Location of a Pointer in the VAT

CFG:VF	ISR Address=
1	VAB + (vector number * 4)
0	VAB + (vector number * 256)

Step 2—Allocating Register and Memory Stack Frames

A full register stack frame is established by **START.S**, because it will call the application program (_main). Further, control could be passed back to the **START.S** return address (which then initiates a "warm start"). This should be done early in the main loop, as **START.S** will call some supporting assembly-language routines. The register stack frame can be established by the code shown in Listing 6.

Arguments that overflow the register stack will have to be placed in the memory stack (see Figure 8). The current position in the memory stack is pointed to by the memory stack pointer (*msp*).

The stack can be established by:

```
const   msp, MStkTop
consth  msp, MStkTop
```

Listing 6. Allocating Register and Memory Stack Frames

```
const     rfb, RStkTop              ;RStkTop is set to the
consth    rfb, RStkTop              ;desired address in the declarations file
const     rab, (RStkTop - 512)      ;128*4, maximum
consth    rab, (RStkTop - 512)      ;part that can
add       lr1, rfb, 0               ;be cached
sub       rsp, rfb, 16              ;adjusts for lr0, lr1, argc, and argv
```

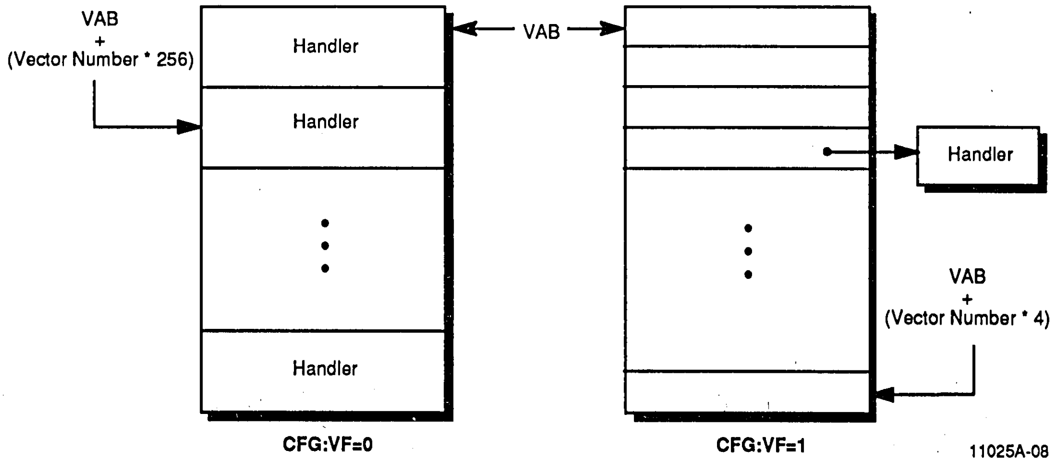


Figure 8. The Two Structures of the Vector Area

Step 3—Initializing the Vector Area and Vectors

Although the organization of the vector area is determined by the configuration register, the table and pointers still must be initialized. In the following example, the vector initialization code is kept compact, while permitting easy expansion of the vector set, by using a table in the .data section. Each entry in the table has two words. The first is the vector number; the second is the handler address (see Listing 7).

When the vector area base (vab) is supplied to the routine shown in Listing 8, it initializes the handlers.

Listing 7. Initializing the Vector Area and Vectors

```
          .data                                          ;switch to .data for table

VectInitTable:
          .word      V_SupInstTLB, SupInstTLBHandler
          .word      V_SupDataTLB, SupDataTLBHandler
          .word      V_MULTIPLY, MultiplyHandler
          .word      V_DIVIDE, DivideHandler
          .word      V_MULTIPLU, MultipluHandler
          .word      V_DIVIDU, DividuHandler
          .word      V_SPILL, SpillHandler
          .word      V_FILL, FillHandler
          .word      V_Timer, TimerHandler
          .equ       VINIT_CNT,((. - VectInitTable) / 8)
          .text                                          ;switch back to .text for code
```

Listing 8. Initializing Vector Handlers

```
VectInit:
          .reg       VI_Vect,%%(TMP_REG + 0)            ;vector value
          .reg       VI_St,%%(TMP_REG + 1)              ;vector storage address
          .reg       VI_Cnt,%%(TMP_REG + 2)             ;vector count
          .reg       VI_Base,%%(TMP_REG + 3)            ;vector base
          .reg       VI_TbPt,%%(TMP_REG + 4)            ;vector base
          mfsr       VI_Base, vab
          const      VI_Cnt, (VINIT_CNT - 2)            ;for jmpfdec
          const      VI_TbPt, VectInitTable
          consth     VI_TbPt, VectInitTable

VI_Loop:
          load       0, 0, VI_St, VI_TbPt               ;get the vector
          add        VI_TbPt, VI_TbPt, 4
          sll        VI_St, VI_St, 2                    ;convert to address
          add        VI_St, VI_St, VI_Base
          load       0, 0, VI_Vect, VI_TbPt             ;get the handler
          add        VI_TbPt, VI_TbPt, 4
          jmpfdec    VI_Cnt, VI_Loop
          store      0, 0, VI_Vect, VI_St
          jmp        raddr
          nop
```

Step 4—Initializing the Translation Look-Aside Buffer (TLB)

When the Am29000 is first powered-up, the TLB will not have valid entries. To prevent erroneous TLB misses, the entries should be marked invalid by the start-up sequence before control is passed to the application program. This can be done with an assembly-language sequence (see Listing 9).

Step 5—Calling "main"

Once the proper environment has been established for the application program, the main C program must be called. This is done by placing the address of the starting instruction in registers and performing a call. When the jump is "short," or less than 256 words, a *call* can be done directly. However, the jump often will be farther, and *calli* must be used in conjunction with an address stored in registers, as shown below:

```
const  raddr, _main ;store lower 16 bits
consth raddr, _main ;store upper 16 bits
calli  raddr, raddr ;call indirect
```

Notice that raddr signifies the return address, usually lr0, by convention. Once the call is made, the return address of the caller has replaced the target location, in the event there is a return from _main.

The START.S Main Loop

The complete **START.S** main loop, as developed in the previous sections, is shown in Listing 10. The routine receives control after being transcribed to RAM; once there, it initializes the vector handlers, clears the BSS area, initializes the TLBs, and establishes initial stack pointers and an initial register frame. Lastly, it invokes _main. Note that, in the event _main returns, a warm start is performed.

Listing 9. Initializing the TLB

```
         .reg       TI_Reg,%%(TEMP_REG + 0)      ;the TLB register number
         .reg       TI_Val,%%(TEMP_REG + 1)      ;the TLB value (0)
         .reg       TI_Cnt,%%(TEMP_REG + 2)      ;the TLB register count
         const      TI_Reg, 0
         const      TI_Val, 0
         const      TI_Cnt, (TLB_CNT - 2)        ;for jmpfdec

TI_Loop:
         mttlb      TI_Reg, TI_Val
         jmpfdec    TI_Cnt, TI_Loop
         add        TI_Reg, TI_Reg, 1
```

Listing 10. START.S Main Loop

```
Start:
          mtsrim     cps, 0x73                       ;set PD, PI, SM, DI, DA
          mtsrim     mmu, MMU_PS                     ;PID = 0
          mtsrim     cfg, 0x10                       ;VF
          const      rfb, RStkTop                    ;set up stack pointers
          consth     rfb, RStkTop
          const      rab, (RStkTop - 512)
          consth     rab, (RStkTop - 512)
          add        lr1, rfb, 0
          sub        rsp, rfb, 16                    ;make room for lr0, lr1, argc,
 argv
          const      msp, MStkTop
          consth     msp, MStkTop
          call       lr0, VectInit                   ;routine to install handled
vectors
          nop
          call       lr0, TLBInit                    ;routine to mark TLBs invalid
          nop
          mtsrim     cps, 0x10                       ;SM
          const      lr2, 0                          ;argc = 0
          const      lr3, 0                          ;argv = 0
          call       lr0, _main
          nop
          mtsrim     cps, 0x473                      ;set FZ, PD, PI, SM, DI, DA
          mtsrim     ops, 0x173                      ;set RE, PD, PI, SM, DI, DA
          mtsrim     cfg, 1                          ;cache disabled
          mtsrim     chc, 0                          ;contents invalid
          mtsrim     pcl, 0                          ;cold start address
          mtsrim     pc0, 4
          iretinv
```

APPENDIX A: boot.s

```
.title              "ROM Boot Code"
;
; Copyright 1988, Advanced Micro Devices
; Written by Gibbons and Associates, Inc.
;
; This module is intended to receive control at address 0. It handles a hardware
; reset or a simulation of that event in a "warm start" situation.
;
; Its purpose is to provide sufficient initializations for the operation of a program
; in RAM data/instruction space.  The initializations must include the transcription
; of the program and its initialized data. The code and initialized data are stored
; in ROM prior to transcription.
;
; To provide for orderly operation, C linkages are used. It is known that the register
; stack will never overflow. When certain calamities occur (e.g., invalid
; traps), the registers will be re-initialized to allow the use of subroutines in
; this module. There is no intention of ever returning under these circumstances.
;
; Some of the routines in this module have a rather tedious implementation because
; they do not assume the validity of RAM or the readability of ROM.  This is
; considered appropriate since it assures the validity of error handling.
;
; This module provides no global addresses for external use.  It is not intended to
; be called.  It is best thought of as bootstrap code.
;
; Some tests which are not actually used are included here for use in environments
; that may allow them.
;
; The external addresses named below are required.
;

          .extern    RAMInit                        ;romcoff generated

;
; This module needs the addresses for the control and data ports of the SCC.  These
; are declared below.
;
          .equ       SCCCntlAd,0xfffffff0           ;control port address
          .equ       SCCDataAd,0xfffffff4           ;data port address

;
; This module assumes that RAM begins at data address 0 and has the size declared
; below.

          .equ       RAM_SIZE,0x40000               ;256K bytes
          .include   "romdcl.h"
          .eject
          .sbttl     "Section Declarations"

;
; This module has only one section, which is called "rom."  It receives control at
; reset,i.e., it is an absolute segment based at address 0 (in ROM space).
;
          .sect      rom,text,absolute 0
          .use       rom
```

```
RomBase:
          jmp        Boot                                ;the RESET entry
          nop
          nop
          nop
          halt       ;the warn entry
          nop        ;Could be a report routine
; ......................................

          .eject
          .sbttl     "SCC Routines"

          LEAF       SerInit,0
;
; This routine initializes the serial port for non-interrupt driven access at 9600
; baud.
;
; In:     (nothing)
;
; Out:    (nothing)
;
          .reg       SI_CtAd,%%(TEMP_REG + 0)            ;control port address
          .reg       SI_CtVl,%%(TEMP_REG + 1)            ;control port value
          const      SI_CtAd,SCCCntlAd
          consth     SI_CtAd,SCCCntlAd
          const      SI_CtVl,9                           ;reset the port
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0xc0
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,4                           ;x16,1 stop,no parity
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x44
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,3                           ;8 bits receive
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0xc0
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,5                           ;8 bits xmit
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x60
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,9                           ;Int. disabled
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x0
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,10                          ;NRZ
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x0
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,11                          ;Tx & Rx BRG out
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x56
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,12                          ;9600 baud
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x6
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,13                          ;9600 baud
          store      0,0,SI_CtVl,SI_CtAd
          const      SI_CtVl,0x0
          store      0,0,SI_CtVl,SI_CtAd
```

```
        const     SI_CtVl,14                         ;BRG in RTxC
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,0x0
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,14                         ;BRG on
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,0x1
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,3                          ;Rx enable
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,0xc1
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,5                          ;Tx enable
        store     0,0,SI_CtVl,SI_CtAd
        const     SI_CtVl,0xea
        store     0,0,SI_CtVl,SI_CtAd
        EPILOGUE
; .......................................

        LEAF      SerXmt,1
;
; This routine transmits a single character via the SCC.  It will wait (forever) for
; the SCC to become ready.
;
; In:     (see below)
;
; Out:    (nothing)
;
        .reg      SX_Char,%%(IN_PRM + 0)             ;character
        .reg      SX_Ad,%%(TEMP_REG + 0)             ;port address
        .reg      SX_Vl,%%(TEMP_REG + 1)             ;port value
        const     SX_Ad,SCCCntlAd
        consth    SX_Ad,SCCCntlAd

SX_Wait:
        load      0,0,SX_Vl,SX_Ad                    ;get the status
        and       SX_Vl,SX_Vl,0x4                    ;check tx buf empty
        cpeq      SX_Vl,SX_Vl,0
        jmpf      SX_Vl,SX_Wait
        nop
        const     SX_Ad,SCCDataAd                    ;send the character
        consth    SX_Ad,SCCDataAd
        store     0,0,SX_Char,SX_Ad
        EPILOGUE
; .......................................

        LEAF      SerRcv,0
;
; This routine waits for a receive character to become ready, then reads and returns
; that character.
;
; In:     (nothing)
;
; Out:    (see below)
;
        .reg      SR_Ad,%%(TEMP_REG + 0)             ;port address
        .reg      SR_Char,%%(RET_VAL + 0)            ;character (stat tmp)
        const     SR_Ad,SCCCntlAd
        consth    SR_Ad,SCCCntlAd
```

```
SR_Wait:
        load     0,0,SR_Char,SR_Ad                  ;get the status
        and      SR_Char,SR_Char,0x1                ;check rcv buf ready
        cpeq     SR_Char,SR_Char,0
        jmpf     SR_Char,SR_Wait
        nop
        const    SR_Ad,SCCDataAd                    ;fetch the character
        consth   SR_Ad,SCCDataAd
        load     0,0,SR_Char,SR_Ad
        and      SR_Char,SR_Char,0xff
        EPILOGUE
; .......................................

        LEAF     SerChk,0
;
; This routine checks to determine if a receive character is ready at the serial
; port. It will return -1 if a character is ready and 0 if it is not.
;
; In:     (nothing)
;
; Out:    (see below)
;
        .reg     SC_Ad,%%(TEMP_REG + 0)             ;port address
        .reg     SC_Rdy,%%(RET_VAL + 0)             ;character
        const    SC_Ad,SCCCntlAd
        consth   SC_Ad,SCCCntlAd
        load     0,0,SC_Rdy,SC_Ad                   ;get the status
        and      SC_Rdy,SC_Rdy,0x1                  ;check rcv buf ready
        cpeq     SC_Rdy,SC_Rdy,0
        sra      SC_Rdy,SC_Rdy,31                   ;convert to 0 or -1
        EPILOGUE
; .......................................

        .eject
.sbttl  "Error Message Routines"

        FUNCTION SendErr,0,0,1
;
; This routine sends the text "Error - "
;
        .reg     SE_Char,%%(OUT_PRM + 0)            ;output character
        call     lr0,SerXmt
        const    SE_Char,'E'                        ;send a "E"
        call     lr0,SerXmt
        const    SE_Char,'r'                        ;send a "r"
        call     lr0,SerXmt
        const    SE_Char,'r'                        ;send a "r"
        call     lr0,SerXmt
        const    SE_Char,'o'                        ;send a "o"
        call     lr0,SerXmt
        const    SE_Char,'r'                        ;send a "r"
        call     lr0,SerXmt
        const    SE_Char,' '                        ;send a " "
        call     lr0,SerXmt
        const    SE_Char,'-'                        ;send a "-"
        call     lr0,SerXmt
        const    SE_Char,' '                        ;send a " "
        EPILOGUE
; .......................................

        FUNCTION SendNL,0,0,1
```

```
;
; This routine sends a CR-LF sequence.
;
        .reg        SN_Char,%%(OUT_PRM + 0)
        call        lr0,SerXmt
        const       SE_Char,0x0d                    ;send a "CR"
        call        lr0,SerXmt
        const       SE_Char,0x0a                    ;send a "LF"
        EPILOGUE
; .......................................

        FUNCTION    SendWord,1,1,1
;
; This routine sends a 32-bit word in ASCII hex
;
        .reg        SW_Word,%%(IN_PRM + 0)          ;the word to send
        .reg        SW_Shift,%%(LOC_REG + 0)        ;shift factor
        .reg        SW_T_Flag,%%(TEMP_REG + 0)
        .reg        SW_Char,%%(OUT_PRM + 0)         ;character to send
        const       SW_Shift,28                     ;right shift factor

SW_0:
        srl         SW_Char,SW_Word,SW_Shift
        and         SW_Char,SW_Char,0xf             ;isolate nibble
        cplt        SW_T_Flag,SW_Char,10            ;check decimal
        jmpt        SW_T_Flag,SW_1
        add         SW_Char,SW_Char,0x30            ;convert to ASCII digit
        add         SW_Char,SW_Char,0x27            ;convert to ASCII letter

SW_1:
        call        lr0,SerXmt                      ;send the character
        nop
        subs        SW_Shift,SW_Shift,4             ;next digit shift fact
        cpge        SW_T_Flag,SW_Shift,0            ;check if done
        jmpt        SW_T_Flag,SW_0                  ;continue if not
        nop
        EPILOGUE
; .......................................

        FUNCTION    RAMErr,3,0,1
;
; This routine reports RAM errors with the message,
; "Error - RAM at aaaaaaaa write bbbbbbbb read cccccccc\n"
;
        .reg        RE_ErrAdd,%%(IN_PRM + 0)
        .reg        RE_WrtPat,%%(IN_PRM + 1)
        .reg        RE_RedPat,%%(IN_PRM + 2)
        .reg        RE_Char,%%(OUT_PRM + 0)
        .reg        RE_Word,%%(OUT_PRM + 0)
        call        lr0,SendErr                     ;send "Error - "
        nop
        call        lr0,SerXmt
        const       RE_Char,'R'                     ;send a "R"
        call        lr0,SerXmt
        const       RE_Char,'A'                     ;send a "A"
        call        lr0,SerXmt
        const       RE_Char,'M'                     ;send a "M"
        call        lr0,SerXmt
        const       RE_Char,' '                     ;send a " "
        call        lr0,SerXmt
        const       RE_Char,'A'                     ;send a "A"
        call      lr0,SerXmt
        const     RE_Char,'T'                   ;send a "T"
        call      lr0,SerXmt
        const     RE_Char,' '                   ;send a " "
        call      lr0,SendWord                  ;send error address
        add       RE_Word,RE_ErrAdd,0
        call      lr0,SerXmt
        const     RE_Char,' '                   ;send a " "
        call      lr0,SerXmt
        const     RE_Char,'w'                   ;send a "w"
        call      lr0,SerXmt
        const     RE_Char,'r'                   ;send a "r"
        call      lr0,SerXmt
        const     RE_Char,'i'                   ;send a "i"
        call      lr0,SerXmt
        const     RE_Char,'t'                   ;send a "t"
        call      lr0,SerXmt
        const     RE_Char,'e'                   ;send a "e"
        call      lr0,SerXmt
        const     RE_Char,' '                   ;send a " "
        call      lr0,SendWord                  ;send good pattern
        add       RE_Word,RE_WrtPat,0
        call      lr0,SerXmt
        const     RE_Char,' '                   ;send a " "
        call      lr0,SerXmt
        const     RE_Char,'R'                   ;send a "R"
        call      lr0,SerXmt
        const     RE_Char,'e'                   ;send a "e"
        call      lr0,SerXmt
        const     RE_Char,'a'                   ;send a "a"
        call      lr0,SerXmt
        const     RE_Char,'d'                   ;send a "d"
        call      lr0,SerXmt
        const     RE_Char,' '                   ;send a " "
        call      lr0,SendWord                  ;send bad pattern
        add       RE_Word,RE_RedPat,0
        call      lr0,SendNL                    ;send a new line
        nop
        EPILOGUE
; ......................................

        FUNCTION  ROMErr,1,0,1
;
; This routine reports a ROM sum error with the message,
; "Error - ROM sum aaaaaaaa\n"
;
        .reg      ROM_Sum,%%(IN_PRM + 0)
        .reg      ROM_Char,%%(OUT_PRM + 0)
        .reg      ROM_Word,%%(OUT_PRM + 0)
        call      lr0,SendErr                   ;send "Error - "
        nop
        call      lr0,SerXmt
        const     ROM_Char,'R'                  ;send a "R"
        call      lr0,SerXmt
        const     ROM_Char,'O'                  ;send a "O"
        call      lr0,SerXmt
        const     ROM_Char,'M'                  ;send a "M"
        call      lr0,SerXmt
        const     ROM_Char,' '                  ;send a " "
        call      lr0,SerXmt
        const     ROM_Char,'s'                  ;send a "s"
        call      lr0,SerXmt
        const     ROM_Char,'u'                     ;send a "u"
        call      lr0,SerXmt
        const     ROM_Char,'m'                     ;send a "m"
        call      lr0,SerXmt
        const     ROM_Char,' '                     ;send a " "
        call      lr0,SerXmt
        const     ROM_Char,'='                     ;send a "="
        call      lr0,SerXmt
        const     ROM_Char,' '                     ;send a " "
        call      lr0,SendWord
        add       ROM_Word,ROM_Sum,0               ;send ROM check sum
        call      lr0,SendNL                       ;send a new line
        nop
        EPILOGUE
; .....................................

        FUNCTION  SizeErr,0,0,1
;
; This routine reports insufficient RAM size with the message
; "Error - RAM size\n"
;
        .reg      SIZ_Char,%%(OUT_PRM + 0)
        call      lr0,SendErr                      ;send "Error - "
        nop
        call      lr0,SerXmt
        const     SIZ_Char,'R'                     ;send a "R"
        call      lr0,SerXmt
        const     SIZ_Char,'A'                     ;send a "A"
        call      lr0,SerXmt
        const     SIZ_Char,'M'                     ;send a "M"
        call      lr0,SerXmt
        const     SIZ_Char,' '                     ;send a " "
        call      lr0,SerXmt
        const     SIZ_Char,'s'                     ;send a "s"
        call      lr0,SerXmt
        const     SIZ_Char,'i'                     ;send a "i"
        call      lr0,SerXmt
        const     SIZ_Char,'z'                     ;send a "z"
        call      lr0,SerXmt
        const     SIZ_Char,'e'                     ;send a "e"
        call      lr0,SendNL                       ;send a new line
        nop
        EPILOGUE
; .....................................

        FUNCTION  TrapErr,0,0,1
;
; This routine reports insufficient RAM size with the message
; "Error - Invalid trap\n"
;
        .reg      TE_Char,%%(OUT_PRM + 0)
        call      lr0,SendErr                      ;send "Error - "
        nop
        call      lr0,SerXmt
        const     TE_Char,'I'                      ;send a "I"
        call      lr0,SerXmt
        const     TE_Char,'n'                      ;send a "n"
        call      lr0,SerXmt
        const     TE_Char,'v'                      ;send a "v"
        call      lr0,SerXmt
        const     TE_Char,'a'                     ;send a "a"
        call      lr0,SerXmt
        const     TE_Char,'l'                     ;send a "l"
        call      lr0,SerXmt
        const     TE_Char,'i'                     ;send a "i"
        call      lr0,SerXmt
        const     TE_Char,'d'                     ;send a "d"
        call      lr0,SerXmt
        const     TE_Char,' '                     ;send a " "
        call      lr0,SerXmt
        const     TE_Char,'t'                     ;send a "t"
        call      lr0,SerXmt
        const     TE_Char,'r'                     ;send a "r"
        call      lr0,SerXmt
        const     TE_Char,'a'                     ;send a "a"
        call      lr0,SerXmt
        const     TE_Char,'p'                     ;send a "p"
        call      lr0,SendNL                      ;send a new line
        nop
        EPILOGUE
; ......................................

        .eject
        .sbttl    "ROM Checksum Test"

        FUNCTION  ROMSum,2,0,1
;
; This routine is used to ensure that the ROM is "intacted" correctly by using
; the checksum checking method.
;
; In:     (see below)
;
; Out:    (see below)
;
        .reg      RS_StrtAdd,%%(IN_PRM + 0)       ;start address
        .reg      RS_WrdCnt,%%(IN_PRM + 1)        ;word count
        .reg      RS_SumTmp,%%(TEMP_REG + 0)
        .reg      RS_ChkSum,%%(OUT_PRM + 0)
        .reg      RS_Fail,%%(RET_VAL + 0)         ;TRUE for fail
        xor       RS_ChkSum,RS_ChkSum,RS_ChkSum   ;clear ChkSum
        sub       RS_WrdCnt,RS_WrdCnt,2           ;for jmpfdec

RS_1:
        load      CD,ROM_CTL,RS_SumTmp,RS_StrtAdd
        add       RS_ChkSum,RS_ChkSum,RS_SumTmp   ;add to ChkSum
        jmpfdec   RS_WrdCnt,RS_1
        add       RS_StrtAdd,RS_StrtAdd,4         ;next ROM addr
                                                  ;if ChkSum == 0 then
        cpneq     RS_Fail,RS_ChkSum,0             ;RS_PASS else RS_ERR
        jmpf      RS_Fail,RS_EXIT
        nop
RS_ERR:
        call      lr0,ROMErr                      ;call ROMErr routine
        nop       ;O/P para -- ChkSum
        const     RS_Fail,TRUE                    ;TRUE for test fail
        consth    RS_Fail,TRUE
```

```
RS_EXIT:
        EPILOGUE
; ......................................

        .eject
        .sbttl      "RAM 01 Test"

        FUNCTION    RAM01,2,0,3
;
; This routine tests the RAM by the following method set all RAM area to 0 then check
; for 0. set all RAM area to 1 then check for 1.
;
; In:      (see below)
;
; Out:     (see below)
;
        .reg        R01_StrtAdd,%%(IN_PRM + 0)          ;starting address
        .reg        R01_WrdCnt,%%(IN_PRM + 1)           ;count of words
        .reg        R01_TmpCnt,%%(TEMP_REG + 0)         ;counter
        .reg        R01_NxtAdd,%%(OUT_PRM + 0)          ;error addres
        .reg        R01_WrtPat,%%(OUT_PRM + 1)          ;pattern written
        .reg        R01_RedPat,%%(OUT_PRM + 2)          ;pattern read
        .reg        R01_Fail,%%(RET_VAL + 0)            ;TRUE for fail
        xor         R01_WrtPat,R01_WrtPat,R01_WrtPat ;0 to start

R01_0:                                                  ;set 0's or 1's
        add         R01_NxtAdd,R01_StrtAdd,0            ;get strt RAM addr
        sub         R01_TmpCnt,R01_WrdCnt,2             ;for jmpfdec

R01_1:
        store       CD,DATA_CTL,R01_WrtPat,R01_NxtAdd
        jmpfdec     R01_TmpCnt,R01_1
        add         R01_NxtAdd,R01_NxtAdd,WRD_SIZ
                                                        ;check for 0's or 1's
        add         R01_NxtAdd,R01_StrtAdd,0            ;get strt RAM addr
        sub         R01_TmpCnt,R01_WrdCnt,2             ;for jmpfdec

R01_2:
        load        CD,DATA_CTL,R01_RedPat,R01_NxtAdd
        cpneq       R01_Fail,R01_RedPat,R01_WrtPat      ;err if neq
        jmpt        R01_Fail,R01_ERR
        nop
        jmpfdec     R01_TmpCnt,R01_2
        add         R01_NxtAdd,R01_NxtAdd,WRD_SIZ
        cpeq        R01_Fail,R01_WrtPat,0               ;if WrtPat = 0 then
        jmpt        R01_Fail,R01_0                      ;R01_0 else done
        nor         R01_WrtPat,R01_WrtPat,R01_WrtPat ;invert ptrn
        jmp         R01_EXIT                            ;pass 0 and 1 test
        nop

R01_ERR:                                                ;O/P Parms -- NxtAdd,WrtPat,RedPat
        call        lr0,RAMErr
        nop
        const       R01_Fail,TRUE                       ;TRUE for test fail
        consth      R01_Fail,TRUE
R01_EXIT:
        EPILOGUE
; ......................................

        .eject
        .sbttl      "RAM Checker Pattern Test"
        FUNCTION    RAMChkr,2,0,3
;
; This routine will run a two-pass checkerboard on RAM. It will be controlled by
; input values specifying the base address and the count of locations to be tested.
;
; In:       (see below)
;
; Out:      (see below)
;
        .reg        RC_StrtAdd,%%(IN_PRM + 0)          ;starting address
        .reg        RC_WrdCnt,%%(IN_PRM + 1)           ;count of words
        .reg        RC_TmpCnt,%%(TEMP_REG + 0)         ;total test word count
        .reg        RC_StrtPat,%%(TEMP_REG + 1)        ;starting pattern
        .reg        RC_NxtAdd,%%(OUT_PRM + 0)          ;error address
        .reg        RC_WrtPat,%%(OUT_PRM + 1)          ;pattern written
        .reg        RC_RedPat,%%(OUT_PRM + 2)          ;pattern read
        .reg        RC_Fail,%%(RET_VAL + 0)            ;TRUE for fail
        const       RC_StrtPat,CHKPAT_a5               ;start with a5
        consth      RC_StrtPat,CHKPAT_a5

RC_1:                                                  ;fill memory with pattern
        add         RC_NxtAdd,RC_StrtAdd,0             ;get start address
        sub         RC_TmpCnt,RC_WrdCnt,2              ;for jmpfdec
        add         RC_WrtPat,RC_StrtPat,0             ;set the pattern

RC_2:
        store       0,0,RC_WrtPat,RC_NxtAdd
        R_LEFT      RC_WrtPat                          ;rotate ptrn left
        jmpfdec     RC_TmpCnt,RC_2
        add         RC_NxtAdd,RC_NxtAdd,4              ;next test mem addr
                                                       ; check memory for pattern
        add         RC_NxtAdd,RC_StrtAdd,0             ;get start address
        sub         RC_TmpCnt,RC_WrdCnt,2              ;for jmpfdec
        add         RC_WrtPat,RC_StrtPat,0             ;set the pattern

RC_3:
        load        CD,DATA_CTL,RC_RedPat,RC_NxtAdd
        cpneq       RC_Fail,RC_RedPat,RC_WrtPat        ;err if neq
        jmpt        RC_Fail,RC_ERR
        nop
        R_LEFT      RC_WrtPat                          ;rotate ptrn left
        jmpfdec     RC_TmpCnt,RC_3
        add         RC_NxtAdd,RC_NxtAdd,4              ;next test mem addr
                                                       ; invert ptrn for next pass
        nor         RC_StrtPat,RC_StrtPat,0            ;invert initial
        jmpt        RC_StrtPat,RC_EXIT                 ;done if msb = 1
        nop
        jmp         RC_1                               ;try with inverted
        nop

RC_ERR:
        call        lr0,RAMErr
        nop
        const       RC_Fail,TRUE                       ;set after call
        consth      RC_Fail,TRUE

RC_EXIT:
        EPILOGUE
; .......................................
```

```
         .eject
         .sbttl      "RAM Address Pattern Test"

         FUNCTION    RAMAddr,2,0,3
;
; This routine will run a two-pass test on RAM.  It will be controlled by input values
; specifying the base address and the count of locations to be tested.  In the first
; pass,the data will be set equal to the address.  In the second pass, the data will
; be set equal to the complement of the address.
;
; In:      (see below)
;
; Out:     (see below)
;
         .reg        RA_StrtAdd,%%(IN_PRM + 0)         ;starting address
         .reg        RA_WrdCnt,%%(IN_PRM + 1)          ;count of words
         .reg        RA_TmpCnt,%%(TEMP_REG + 0)        ;total test word count
         .reg        RA_StrtPat,%%(TEMP_REG + 1)       ;starting pattern
         .reg        RA_PtrnInc,%%(TEMP_REG + 2)       ;ptrn increment value
         .reg        RA_NxtAdd,%%(OUT_PRM + 0)         ;error address
         .reg        RA_WrtPat,%%(OUT_PRM + 1)         ;pattern written
         .reg        RA_RedPat,%%(OUT_PRM + 2)         ;pattern read
         .reg        RA_Fail,%%(RET_VAL + 0)           ;TRUE for fail
         add         RA_StrtPat,RA_StrtAdd,0           ;start with address
         const       RA_PtrnInc,4

RA_1:                                                  ;fill memory with pattern
         add         RA_NxtAdd,RA_StrtAdd,0            ;get start address
         sub         RA_TmpCnt,RA_WrdCnt,2             ;for jmpfdec
         add         RA_WrtPat,RA_StrtPat,0            ;set the pattern
RA_2:
         store       0,0,RA_WrtPat,RA_NxtAdd
         add         RA_WrtPat,RA_WrtPat,RA_PtrnInc
         jmpfdec     RA_TmpCnt,RA_2
         add         RA_NxtAdd,RA_NxtAdd,4             ;next test mem addr

; check memory for pattern
         add         RA_NxtAdd,RA_StrtAdd,0            ;get start address
         sub         RA_TmpCnt,RA_WrdCnt,2             ;for jmpfdec
         add         RA_WrtPat,RA_StrtPat,0            ;set the pattern

RA_3:
         load        CD,DATA_CTL,RA_RedPat,RA_NxtAdd
         cpneq       RA_Fail,RA_RedPat,RA_WrtPat       ;err if neq
         jmpt        RA_Fail,RA_ERR
         nop
         add         RA_WrtPat,RA_WrtPat,RA_PtrnInc
         jmpfdec     RA_TmpCnt,RA_3
         add         RA_NxtAdd,RA_NxtAdd,4             ;next test mem addr
                                                       ; invert ptrn for next pass
         nor         RA_StrtPat,RA_StrtPat,0           ;invert initial
         cpneq       RA_Fail,RA_StrtPat,RA_StrtAdd
         jmpt        RA_Fail,RA_1
         subr        RA_PtrnInc,RA_PtrnInc,0           ;negate inc value
         jmp         RA_EXIT
         nop
```

```
RA_ERR:
          call      lr0,RAMErr
          nop
          const     RA_Fail,TRUE                        ;set after call
          consth    RA_Fail,TRUE

RA_EXIT:
          EPILOGUE
; .......................................

          .eject
          .sbttl    "Invalid Trap Handler"

InvalidTrapHandler:
;
; This routine receives control when an invalid trap occurs.  It will reinitialize
; a register frame for use in error reporting.  It then reports the fact that an
; invalid trap has occurred.  Reporting of specific trap numbers could be achieved,
; but at considerable cost in size.  The use of an instrument such as the ADAPT29K™
; is recommended for invalid trap identification.  If that is not practical, this
; handler (or some other) could be extended to report numbers.  It would require 2K
; bytes of additional code (jmp/const for each of 256 vectors).
;
          mtsrim    cps,0x173                           ;RE,PD,PI,SM,DI,DA
          const     rfb,512                             ;set up temp reg frame
          const     rab,0
          sub       rsp,rfb,8                           ;room for linkage
          call      lr0,SerInit                         ;ready to report errors
          add       lr1,rfb,0                           ;small frame required
          call      lr0,TrapErr                         ;show trap error
          nop
          halt
          nop
; .......................................

          .eject
          .sbttl    "Vector Initialization"

          LEAF      VectInit,0
;
; This routine initializes the vector table and vab.  All vectors
; are set to point to the invalid trap handler in ROM.
;
          .reg      VI_Vect,%%(TEMP_REG + 0)            ;vector value
          .reg      VI_VectSt,%%(TEMP_REG + 1)          ;vector storage address
          .reg      VI_VectCnt,%%(TEMP_REG + 2)         ;vector count register
          mtsrim    vab,0
          mfsr              VI_VectSt,vab
          const     VI_Vect,(InvalidTrapHandler | 2)
          consth    VI_Vect,InvalidTrapHandler
          const     VI_VectCnt,(256 - 2)                ;for jmpfdec

VI_Loop:
          store     0,0,VI_VectSt,VI_Vect               ;store the vector
          jmpfdec   VI_VectCnt,VI_Loop
          add       VI_VectSt,VI_VectSt,4
          EPILOGUE
; .......................................
```

```
         .eject
         .sbttl      "Boot"

Boot:
;
; This routine receives control upon a hardware reset.  Its purpose
; is to establish the execution environment for the main program. This involves
; transcriptions of data and possibly code.  The transcriptions may
; take the form of executing code since the ROM may not be readable.
;
         .reg        RI_Ret,%%(TEMP_REG + 0)            ;RAMInit return
         mtsrim      cps,0x173                          ;RE,PD,PI,SM,DI,DA
         const       rfb,512                            ;set up temp reg frame
         const       rab,0
         sub         rsp,rfb,16                         ;enough for p0 and p1
         add         lr1,rfb,0
         call        lr0,SerInit                        ;ready to report errors
         nop
         const       p1,(RAM_SIZE >> 2)                 ;test full RAM size
         consth      p1,(RAM_SIZE >> 2)
         call        lr0,RAMAddr                        ;just use one test
         const       p0,0                               ;test from address zero
         call        lr0,VectInit                       ;invalid traps
         nop
         call        RI_Ret,RAMInit                     ;initialize RAM
         mtsrim      ops,0x473                          ;FZ,PD,PI,SM,DI,DA
         mtsrim      cps,0x473                          ;FZ,PD,PI,SM,DI,DA
         const       lr0,TextBas                        ;(using lr0 as temp)
         consth      lr0,TextBas
         mtsr        pc1,lr0
         add         lr0,lr0,4
         mtsr        pc0,lr0
         iretinv     ;go to inst space,TextBas
;
;..............................................................
; end of boot.s
```

APPENDIX B: start.s

```
        .title      "Start and Other Assembly-language Routines"

; Copyright 1988, Advanced Micro Devices,Inc.
; Written by Gibbons and Associates,Inc.
; HISTORY:
; 1.3  29 July 88  E M Greenawalt  SPR 0001
; Fixed shift count on line 1034
;
; This module provides initializations and trap handling for a program written in C
; and operating in a stand alone environment.  It is designed for compatibility with
; the ADAPT29K and various Am29000 monitors.
;
; In this module, the first 16 system registers (gr64-gr79) are available for use as
; system statics.  They are not used in any of the routines in this file.  Their
; values are not saved and restored in the C interrupt handler interrupts, so they
; are truly static.
;
; The second 16 system registers (gr80-gr95) are used as temporary registers by trap
; handlers, etc., in this module.  No such trap handler is itself interruptable.  No
; presumption is made about the preservation of values in these registers by any
; program.

        .extern     _main                               ;the C main routine
        .global     V_SPILL                             ;the spill/fill vectors
        .global     V_FILL

; NOTE:   The equates below define the padding in the vector
;         section (to a full page), and constants related to
;         the page size.  The register and memory stack size
;         are also declared.
;
; When operating with a monitor, the VECT_PAD may need to be increased.
;
        .equ        PS,3                                ;page size designation
        .equ        RPN_SHIFT,(10 + PS)
        .equ        PAGE_SIZE,(1 << RPN_SHIFT)
        .equ        MMU_PS,(PS << 8)
        .equ        RPN_MASK,(~ (PAGE_SIZE - 1))
        .equ        VECT_PAD,(PAGE_SIZE - 0x400)
        .equ        RSTK_SIZE,PAGE_SIZE
        .equ        MSTK_SIZE,PAGE_SIZE
        .include    "romdcl.h"

;
; NOTE:   The equates below define traps for divide by zero
;         and divide overflow.  They are not standard.  They
;         are not handled here.

        .equ        V_DIV0,80                           ;divide by zero
        .equ        V_DIVOV,81                          ;divide overflow
        .eject
        .sbttl      "Section Declarations"
;
```

```
; Sections will be ordered in memory as shown below.
;
;         vectors (at 0)
;         rstack (register stack)
;         mstack (memory stack)
;         .data
;         .bss
;         .text
;         endsect (dummy for establishing bounds)
;
; Vectors will be initialized by start-up code with pointers to an invalid trap
; handler in ROM.  The initialization code will explicitly intercept those vectors
; that will be handled.
;
          .sect               vectors,bss
          .sect               rstack,bss
          .sect               mstack,bss
          .sect               endsect,bss

; The declarations that follow suggest the order of the segments, provide base
; names for each, and allocate sizes for the vectors and stacks.
;
; Jump instructions are also provided at the base of the .text section for ease
; in linkage to the Start routine and the special routine which provides for
; ADAPT29K initializations.
;
          .use        vectors
          .block      (4 * 256)
          .block      VECT_PAD
          .use        rstack

RStkBase:
          .block      RSTK_SIZE

RStkTop:
          .use        mstack

MStkBase:
          .block      MSTK_SIZE

MStkTop:
          .data

DataBase:             ;base of init data
          .bss

BSSBase:                                              ;base of BSS data
          .text

TextBase:             ;base of .text
          jmp         Start                           ;allows easy linkage to Start
          nop                                         ;for bootstrap code
          jmp         AdaptInit                       ;makes AdaptInit easier to find
          nop
          .use        endsect
```

```
EndBase:                                                   ;marks end of .text
         .block      4                                     ;dummy to assure existence
         .text                                             ;switch back to text

         .eject
         .sbttl      "Timer read/write functions"
         .global     _GetTmCnt
         .global     _SetTmCnt
         .global     _GetTmRld
         .global     _SetTmRld

         LEAF        _GetTmCnt,0
;
; This routine returns the timer/counter register value.  All the fields are returned;
; i.e., no mask is applied.
;
; In:      (nothing)
;
; Out:     (see below)
;
         .reg        GTC_Val,%%(RET_VAL + 0)               ;timer reg value
         mfsr        GTC_Val,tmc
         EPILOGUE
; .......................................

         LEAF        _SetTmCnt,1
;
; This routine sets the timer/counter register value.  All the fields are set;
; i.e., no mask is applied.
;
; In:      (see below)
;
; Out:     (nothing)
;
         .reg        STC_Val,%%(IN_PRM + 0)                ;timer reg value
         mtsr        tmc,STC_Val
         EPILOGUE
; .......................................

         LEAF        _GetTmRld,0
;
; This routine gets the current contents of the timer reload register.  No masks
; are applied.
;
; In:      (nothing)
;
; Out:     (see below)
;
         .reg        GTR_Val,%%(RET_VAL + 0)               ;timer reload value
         mfsr        GTR_Val,tmr
         EPILOGUE
; .......................................

         LEAF        _SetTmRld,1
;
; This routine sets the timer/counter reload value.  All the fields are set;
; i.e., no mask is applied.
;
; In:      (see below)
;
```

```
; Out:    (nothing)
          .reg        STR_Val,%%(IN_PRM + 0)          ;timer reload value
mtsr                tmr,STR_Val
          EPILOGUE
; ....................................

          .eject
          .sbttl      "32-bit Time Extensions"
;
; The routines below extend the timer counter to 32 bits via a trap handler.  The
; 32-bit value may be initialized and read by C-callable routines declared as
; globals. The trap handler is also included.  Note that the caller of the C routines
; must be running in supervisor mode.
;
          .global     _ClrTm32
          .global     _GetTm32
          .bss        ;switch to declare bss
TimeUpper:
          .block      4                               ;reserve a word for extension
          .text                                       ;switch back

          LEAF        _ClrTm32,0
;
; This routine clears the 32-bit extended counter by setting the tmc, tmr and
; software extension value.  The timer interrupt is also enabled in tmr.
;
; In:     (nothing)
;
; Out:    (nothing)                                   (timer initialized to zero)
;
; Temp:                                               (see below)
          .reg        CTVal,%%(TEMP_REG + 0)          ;timer reg value
          .reg        CTUpPt,%%(TEMP_REG + 1)         ;upper pointer
          const       CTVal,0xffffff                  ;for tc and TimeUpper
          consth      CTVal,0xffffff
          mtsr        tmc,CTVal                       ;should keep it busy
          consth      CTVal,0x1ffffff                 ;set ie
          mtsr        tmr,CTVal
          const       CTUpPt,TimeUpper
          consth      CTUpPt,TimeUpper
          const       CTVal,0                         ;no extension
          store       0,0,CTVal,CTUpPt
          EPILOGUE
; ....................................

          LEAF _GetTm32,0
;
; This routine returns a 32-bit clock counter.  The clock counter is implemented
; by extending the hardware counter in software and negating the value before it is
; returned. The negation causes the returned value to be an up counter of the time
; since the counter was last reset.  The low-level timer access routines may be used
; in initializations to assure a desired starting value.
;
; The software extension to 32 bits introduces a coordination problem in reading
; the counter's value.  This is resolved by reading the upper 8 bits both before
; and after the TC value.  If the TC value is greater than 2**23, the second upper
; value read is presumed to be correct.  Lengthy interruptions of this routine
; (> 2**21 clocks) could cause errors.
;
; In:     (nothing)
;
```

```
; Out:     (see below)
;
; Temp:                                                  (see below)
;
          .reg        TUpPt,%%(TEMP_REG + 0)            ;upper time pointer
          .reg        TUpr1,%%(TEMP_REG + 1)            ;upper time bits - 1st read
          .reg        TUpr2,%%(TEMP_REG + 2)            ;upper time bits - 2nd read
          .reg        TLwr,%%(TEMP_REG + 3)             ;lower time bits - from cntr
          .reg        TChk,%%(TEMP_REG + 4)             ;temp to check high bit
          .reg        T32,%%(RET_VAL + 0)               ;32-bit time value
          const       TUpPt,TimeUpper                   ;get upper 8 bits of timer
          consth      TUpPt,TimeUpper
          load        0,0,TUpr1,TUpPt
          add         TUpr1,TUpr1,0                     ;hold till load complete
          mfsr        TLwr,tmc
          load        0,0,TUpr2,TUpPt                   ;get upper 8 bits again
          sll         TChk,TLwr,8                       ;is upper TC bit set?
          jmpf        TChk,GT_Exit                      ;if not, use 1st read
          or          T32,TLwr,TUpr1                    ;poss ovfl before 2nd read
          or          T32,T32,TUpr2                     ;poss ovfl after 1st read

GT_Exit:
          subr        T32,T32,0                         ;negate to count up from zero
          EPILOGUE
; ........................................


TimerHandler:
;
; This routine handles the timer trap.  The timer trap will occur at intervals in the
; range of a second (depending on the actual clock speed).  The extension to 32 bits
; makes the  timer somewhat more useful for common benchmarks.  A different scheme
; would be required for longer intervals.
;
          .reg        THTr,%%(SYS_TEMP + 0)             ;temp for tmr (shared)
          .reg        THUpPt,%%(SYS_TEMP + 0)           ;pointer to upper 8 bits
          .reg        THUpVl,%%(SYS_TEMP + 1)           ;upper 8-bit value
          mfsr        THTr,tmr
          sll         THTr,THTr,7                       ;clear out upper tmr bits
          srl         THTr,THTr,7                       ;leaving ie alone
          mtsr        tmr,THTr
          const       THUpPt,TimeUpper                  ;decrement the upper bits
          consth      THUpPt,TimeUpper
          load        0,0,THUpVl,THUpPt
          srl         THUpVl,THUpVl,24
          sub         THUpVl,THUpVl,1
          sll         THUpVl,THUpVl,24
          store       0,0,THUpVl,THUpPt
          iret        ;done
; ........................................

          .eject
          .sbttl      "C Interrupt Handler Interface"
          .global     CIntf

CIntf:
;
; This routine is used to call a C routine that will handle an interrupt.  In order
; to accomplish this, the context of the current program must be saved prior to the
; call and restored after the call.  It is relatively expensive.  In many
; instances, it may be best to write the interrupt handlers in assembly-language. Note
; that assembly-language handlers will have the system statics available to retain
; state information. Note also that system statics are not saved and restored here.
; They are "static."
;
; This routine receives as inputs the address of the C routine and the vector number.
; It passes the vector number to the C routine as its only parameter.  An initial
; stack of 16 registers (including inputs) is provided to the C routine.
;
; In:     (SYS_TEMP + 0)                        C routine address
;         (SYS_TEMP + 1)                        vector number
;
; Out:    (nothing)
;
; Temp:   (SYS_TEMP 2-13)                       used to hold specials
;         (see below)
;
        .reg      CI_Rout,%%(SYS_TEMP + 0)      ;the C routine
        .reg      CI_Vect,%%(SYS_TEMP + 1)      ;the vector
        .reg      CI_Stk,%%(SYS_TEMP + 14)      ;stack check value
        .reg      CI_Frm,%%(SYS_TEMP + 14)      ;frame size (shared)
        mfsr      st2,ops                       ;save specials temps
        mfsr      st3,cha
        mfsr      st4,chd
        mfsr      st5,chc
        mfsr      st6,pc0
        mfsr      st7,pc1
        mfsr      st8,ipc
        mfsr      st9,ipa
        mfsr      st10,ipb
        mfsr      st11,q
        mfsr      st12,alu
        add       st13,rsp,0
        mtsrim    cps,0x73                      ;PD,PI,SM,DI,DA
        sub       msp,msp,((64 - 16) * 4)       ;allocate space for globals
        const     CI_Stk,MStkBase               ;check for overflow
        consth    CI_Stk,MStkBase
        asge      V_DataTLBProt,msp,CI_Stk      ;simulate Prot (no return on fail)
        store     0,0,gr80,msp                  ;flush for CPU bug
        mtsr      im                            CR,((64 - 16) - 1)
        storem    0,0,gr80,msp                  ;save the globals
        add       rfb,rsp,0                     ;move down the frame
        const     CI_Frm,512                    ;beneath rsp
        sub       rab,rfb,CI_Frm
        add       rsp,rab,(13 * 4)              ;set rsp in 16 reg frame
        sub       msp,msp,(16 * 4)              ;save the frame
        mtsr      im                            CR,(16 - 1)
        storem    0,0,rab,msp
        add       lr1,rfb,0                     ;require remaining locals
        add       p0,CI_Vect,0                  ;vector is output parm 0
        calli     lr0,CI_Rout                   ;call the handler
        mtsrim    cps,0x13                      ;with prot and no ints (no good
                                                ; for more complex TLB schemes)
        mtsrim    cps,0x73                      ;ready to reload
        sub       rab,rsp,(13 * 4)              ;reload locals in frame
        mtsrim    CR,(16 - 1)
        loadm     0,0,rab,msp
        add       msp,msp,(16 * 4)
        mtsrim    CR,((64 - 16) - 1)            ;reload globals
        loadm     0,0,gr64,msp
        add       msp,msp,((64 - 16) * 4)
        mtsr      ops,st2                       ;restore specials
```

```
         mtsr       cha,st3
         mtsr       chd,st4
         mtsr       chc,st5
         mtsr       pc0,st6
         mtsr       pc1,st7
         mtsr       ipc,st8
         mtsr       ipa,st9
         mtsr       ipb,st10
         mtsr       q,st11
         mtsr       alu,st12
         add        rsp,st13,0
         iret       ;return from int
; ....................................

         .eject
         .sbttl     "Multiply and Divide Handlers"

MultiplyHandler:
; This trap handler performs the (signed) operation:
;        DEST//Q <- SRCA * SRCB.
;
; IPC, IPA, and IPB are set by the MULTIPLY instruction prior to the invocation of
; this trap handler.
;
; In:    IPC        DEST
;        IPA        SRCA
;        IPB        SRCB
;
; Out:   DEST//Q    IPB = IPC                           (unimportant side effect)
;
; Temp:  (see below)
;
         .reg       MH_IP,%%(SYS_TEMP + 0)              ;temp for move operation
         mtsr       q,gr0                               ;SRCB (multiplier) to Q
         mfsr       MH_IP,ipc                           ;use a system temp to set
         mtsr       ipb,MH_IP                           ; ipb = ipc
         mul        gr0,gr0,0                           ;step 1. (no initial prod)
         mul        gr0,gr0,gr0                         ;step 2.
         mul        gr0,gr0,gr0                         ;step 3.
         mul        gr0,gr0,gr0                         ;step 4.
         mul        gr0,gr0,gr0                         ;step 5.
         mul        gr0,gr0,gr0                         ;step 6.
         mul        gr0,gr0,gr0                         ;step 7.
         mul        gr0,gr0,gr0                         ;step 8.
         mul        gr0,gr0,gr0                         ;step 9.
         mul        gr0,gr0,gr0                         ;step 10.
         mul        gr0,gr0,gr0                         ;step 11.
         mul        gr0,gr0,gr0                         ;step 12.
         mul        gr0,gr0,gr0                         ;step 13.
         mul        gr0,gr0,gr0                         ;step 14.
         mul        gr0,gr0,gr0                         ;step 15.
         mul        gr0,gr0,gr0                         ;step 16.
         mul        gr0,gr0,gr0                         ;step 17.
         mul        gr0,gr0,gr0                         ;step 18.
         mul        gr0,gr0,gr0                         ;step 19.
         mul        gr0,gr0,gr0                         ;step 20.
         mul        gr0,gr0,gr0                         ;step 21.
         mul        gr0,gr0,gr0                         ;step 22.
         mul        gr0,gr0,gr0                         ;step 23.
         mul        gr0,gr0,gr0                         ;step 24.
         mul        gr0,gr0,gr0                         ;step 25.
```

```
        mul         gr0,gr0,gr0                         ;step 26.
        mul         gr0,gr0,gr0                         ;step 27.
        mul         gr0,gr0,gr0                         ;step 28.
        mul         gr0,gr0,gr0                         ;step 29.
        mul         gr0,gr0,gr0                         ;step 30.
        mul         gr0,gr0,gr0                         ;step 31.
        mull        gr0,gr0,gr0                         ;step 32.
        iret        ;done
;
; This trap handler performs the (unsigned) operation
;       DEST//Q <- SRCA * SRCB.
;
; IPC,IPA,and IPB are set by the MULTIPLU instruction prior to
; the invocation of this trap handler.
;
; In:   IPC         DEST
;       IPA         SRCA
;       IPB         SRCB
;
; Out:  DEST//Q
;       IPB = IPC   (unimportant side effect)
;
; Temp: (see below)
;
        .reg        MU_IP,%%(SYS_TEMP + 0)              ;temp for move operation
        mtsr        q,gr0                               ;SRCB (multiplier) to Q
        mfsr        MU_IP,ipc                           ;use a system temp to set
        mtsr        ipb,MU_IP                           ; ipb = ipc
        mulu        gr0,gr0,0                           ;step 1. (no initial prod)
        mulu        gr0,gr0,gr0                         ;step 2.
        mulu        gr0,gr0,gr0                         ;step 3.
        mulu        gr0,gr0,gr0                         ;step 4.
        mulu        gr0,gr0,gr0                         ;step 5.
        mulu        gr0,gr0,gr0                         ;step 6.
        mulu        gr0,gr0,gr0                         ;step 7.
        mulu        gr0,gr0,gr0                         ;step 8.
        mulu        gr0,gr0,gr0                         ;step 9.
        mulu        gr0,gr0,gr0                         ;step 10.
        mulu        gr0,gr0,gr0                         ;step 11.
        mulu        gr0,gr0,gr0                         ;step 12.
        mulu        gr0,gr0,gr0                         ;step 13.
        mulu        gr0,gr0,gr0                         ;step 14.
        mulu        gr0,gr0,gr0                         ;step 15.
        mulu        gr0,gr0,gr0                         ;step 16.
        mulu        gr0,gr0,gr0                         ;step 17.
        mulu        gr0,gr0,gr0                         ;step 18.
        mulu        gr0,gr0,gr0                         ;step 19.
        mulu        gr0,gr0,gr0                         ;step 20.
        mulu        gr0,gr0,gr0                         ;step 21.
        mulu        gr0,gr0,gr0                         ;step 22.
        mulu        gr0,gr0,gr0                         ;step 23.
        mulu        gr0,gr0,gr0                         ;step 24.
        mulu        gr0,gr0,gr0                         ;step 25.
        mulu        gr0,gr0,gr0                         ;step 26.
        mulu        gr0,gr0,gr0                         ;step 27.
        mulu        gr0,gr0,gr0                         ;step 28.
        mulu        gr0,gr0,gr0                         ;step 29.
        mulu        gr0,gr0,gr0                         ;step 30.
        mulu        gr0,gr0,gr0                         ;step 31.
        mulu        gr0,gr0,gr0                         ;step 32.
        iret        ;done
```

```
; ......................................

DivideHandler:
;
; This trap handler performs the (signed) operation:
;         DEST <- (SRCA//Q) / SRCB
;
; IPC,IPA,and IPB are set by the DIVIDE instruction prior to
; the invocation of this trap handler.
;
;In:      IPC        DEST
;         IPA        SRCA
;         IPB        SRCB
;         Q
;
; Out:    DEST
;
; Temp:   (see below)

          .reg       D_Rmdr,%%(SYS_TEMP + 0)        ;shift area and remainder
          .reg       D_Dvsr,%%(SYS_TEMP + 1)        ;divisor
          .reg       D_Sign,%%(SYS_TEMP + 2)        ;0 for positive
          .reg       D_DvdHi,%%(SYS_TEMP + 3)       ;dividend high
          .reg       D_DvdLo,%%(SYS_TEMP + 4)       ;dividend low
          .reg       D_Quot,%%(SYS_TEMP + 5)
          .reg       D_Ovfl,%%(SYS_TEMP + 6)
          .reg       D_MnNg,%%(SYS_TEMP + 7)        ;most negative integer
          add        D_DvdHi,gr0,0                  ;SRCA is dividend high
          mfsr       D_DvdLo,q                      ;Q is dividend low
          sub        D_Dvsr,D_Dvsr,0                ;divisor is in SRCB
          add        D_Dvsr,D_Dvsr,gr0              ;any easier access?
          asneq      V_DIV0,D_Dvsr,0                ;check for divisor zero

DividendCheck:
          jmpf       D_DvdHi,DivisorCheck
          const      D_Sign,FALSE
          cpeq       D_Sign,D_Sign,0                ;toggle flag
          subr       D_DvdLo,D_DvdLo,0              ;negate dividend
          subrc      D_DvdHi,D_DvdHi,0

DivisorCheck:
          jmpf       D_Dvsr,DivideOp
          nop
          cpeq       D_Sign,D_Sign,0                ;toggle flag
          subr       D_Dvsr,D_Dvsr,0                ;negate divisor

DivideOp:
          mtsr       q,D_DvdLo                      ;dividend low to q
          div0       D_Rmdr,D_DvdHi                 ;D_Rmdr becomes shift high
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 1.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 2.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 3.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 4.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 5.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 6.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 7.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 8.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 9.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 10.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 11.
          div        D_Rmdr,D_Rmdr,D_Dvsr           ;step 12.
```

```
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 13.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 14.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 15.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 16.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 17.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 18.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 19.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 20.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 21.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 22.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 23.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 24.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 25.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 26.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 27.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 28.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 29.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 30.
        div        D_Rmdr,D_Rmdr,D_Dvsr              ;step 31.
;       divrem     D_Rmdr,D_Rmdr,D_Dvsr              ;don't need remainder
        mfsr       D_Quot,q                          ;get quotient out of q
        cplt       D_Ovfl,D_Quot,0                   ;check overflow
        jmpf       D_Sign,DivideCorrect
        cpeq       D_MnNg,D_MnNg,D_MnNg              ;set most neg
        cpeq       D_Ovfl,D_MnNg,D_Quot              ;check for most neg
        cpneq      D_Ovfl,D_Ovfl,D_Sign              ;allow if to be neg

DivideCorrect:
        jmpf       D_Sign,DivideExit                 ;done if positive
        aseq       V_DIVOV,D_Ovfl,0                  ;trap on overflow
        subr       D_Quot,D_Quot,0                   ;negate quotient
;       subr       D_Rmdr,D_Rmdr,0                   ;don't need remainder

DivideExit:
        add        gr0,D_Quot,0                      ;set DEST
        iret       ;done
; ......................................

DividuHandler:
;
; This trap handler performs the (unsigned) operation:
;       DEST <- (SRCA//Q) / SRCB
;
; IPC,IPA,and IPB are set by the DIVIDU instruction prior to
; the invocation of this trap handler.
;
; In:   IPC        DEST
;       IPA        SRCA
;       IPB        SRCB
;       Q
;
; Out:  DEST
;
; Temp: (see below)
;
        .reg       DU_Rmdr,%%(SYS_TEMP + 0)          ;shift area and remainder
        add        DU_Rmdr,gr0,0                     ;SRCA to DU_Rmdr
        div0       DU_Rmdr,DU_Rmdr                   ;DU_Rmdr becomes shift high
        div        DU_Rmdr,DU_Rmdr,gr0               ;step 1.
        div        DU_Rmdr,DU_Rmdr,gr0               ;step 2.
        div        DU_Rmdr,DU_Rmdr,gr0               ;step 3.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 4.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 5.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 6.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 7.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 8.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 9.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 10.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 11.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 12.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 13.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 14.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 15.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 16.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 17.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 18.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 19.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 20.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 21.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 22.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 23.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 24.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 25.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 26.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 27.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 28.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 29.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 30.
        div     DU_Rmdr,DU_Rmdr,gr0             ;step 31.
;       divrem  DU_Rmdr,DU_Rmdr,gr0             ;don't need remainder
        mfsr    gr0,q                           ;quotient to (ipc)
        iret    ;done
; ........................................

        .eject
        .sbttl  "Spill and Fill Handlers"

; The routines below handle the allocation and free assertions
; in subroutine prologues and epilogues.  The temps they use
; are given below.

        .reg    R_Cnt,%%(SYS_TEMP + 0)          ;temp for count (shared)
        .reg    R_Bnd,%%(SYS_TEMP + 0)          ;temp for boundary
        .reg    R_TmpPC0,%%(SYS_TEMP + 1)       ;temp for PC0
        .reg    R_TmpPC1,%%(SYS_TEMP + 2)       ;temp for PC1

SpillHandler:
;
; This routine handles a false assertion in the standard prologue
;
; In:   rab > rsp  (requiring an allocation)
;                                       lrl <= rfb
;                                       rfb == rab + 512
;
; Out:  rab == rsp (just enough allocated)
;                                       lrl <= rfb
;                                       rfb = rab + 512
        mfsr    R_TmpPC0,pc0                    ;save the PCs
        mfsr    R_TmpPC1,pc1
        mtsrim  cps,0x73                        ;PD,PI,SM,DI,DA
        sub     R_Cnt,rab,rsp                   ;R_Cnt = # of bytes to spill
        sub     rfb,rfb,R_Cnt                   ;move down the frame bound
```

```
          store       0,0,lr0,rfb                  ;flush for storem bug
          srl         R_Cnt,R_Cnt,2                ;R_Cnt = count of words to spill
          sub         R_Cnt,R_Cnt,1                ;correct for storem
          mtsr        cr,R_Cnt                     ;set up count for storem
          storem      0,0,lr0,rfb                  ;spill from the allocated area
          add         rab,rsp,0                    ;move down the allocate bound
          const       R_Bnd,RStkBase               ;check for possible overflow
          consth      R_Bnd,RStkBase
          asge        V_DataTLBProt,rab,R_Bnd      ;simulate TLB prot
                                                   ;NOTE: no return on fail
          mtsrim      cps,0x473                    ;FZ,PD,PI,SM,DI,DA
          mtsr        pc0,R_TmpPC0                 ;restore the PCs
          mtsr        pc1,R_TmpPC1
          iret
; .......................................

FillHandler:

; This routine handles a false assertion in the standard epilogue.
;
; In:     lr1 > rfb                                (requiring deallocation)
;                                            rsp >= rab
;                                            rfb == rab + 512
;
; Out:    lr1 == rfb                               (just enough freed)
;                                            rsp >= rab
;                                            rfb = rab + 512
          mfsr        R_TmpPC0,pc0                 ;save the PCs
          mfsr        R_TmpPC1,pc1
          mtsrim      cps,0x73                     ;PD,PI,SM,DI,DA
          const       R_Bnd,RStkTop                ;check for possible underflow
          consth      R_Bnd,RStkTop
          asle        V_DataTLBProt,rfb,R_Bnd      ;simulate TLB prot
                                                   ;NOTE: no return on fail
          const       R_Cnt,512                    ;make local reg ip
          or          R_Cnt,R_Cnt,rfb              ;                    from rfb
          mtsr        ipa,R_Cnt                    ;set up indirect ptr for loadm
          sub         R_Cnt,lr1,rfb                ;R_Cnt = # of bytes to fill
          add         rab,rab,R_Cnt                ;move up the allocate bound
          srl         R_Cnt,R_Cnt,2                ;R_Cnt = number of words to fill
          sub         R_Cnt,R_Cnt,1                ;correct for loadm
          mtsr        cr,R_Cnt                     ;set up count for loadm
          loadm       0,0,gr0,rfb                  ;fill area freed
          add         rfb,lr1,0                    ;move up frame bound
          mtsrim      cps,0x473                    ;FZ,PD,PI,SM,DI,DA
          mtsr        pc0,R_TmpPC0                 ;restore the PCs
          mtsr        pc1,R_TmpPC1
          iret
; .......................................

          .eject
          .sbttl      "TLB Miss Handler"

; The routines below provide one-for-one TLBs, i.e., the virtual address is set equal
; to the physical address.  A central routine is used to do the actual TLB update.
;
; Some enhancement would be appropriate to allow I/O access as data,; i.e.,
; memory-mapped I/O. Speed improvements could be realized (four instructions) by the
; allocation and initialization of system registers for the bounds.
;
; The temp registers used are indicated below.
```

```
;
          .reg      TH_Ad,%%(SYS_TEMP + 0)          ;the miss address
          .reg      TH_Ac,%%(SYS_TEMP + 1)          ;the required privileges
          .reg      TH_Bnd,%%(SYS_TEMP + 2)         ;access bound
          .reg      TH_Reg,%%(SYS_TEMP + 3)         ;TLB register number
          .reg      TH_Wd0,%%(SYS_TEMP + 4)         ;TLB word 0 value
          .reg      TH_Wd1,%%(SYS_TEMP + 5)         ;TLB word 1 value
;
; This routine handles supervisor instruction TLB misses.
;
; An attempted access out of range is treated as an instruction
; TLB protection violation.
;
          mfsr      TH_Ad,pc1
          const     TH_Bnd,TextBase
          consth    TH_Bnd,TextBase
          asge      V_InstTLBProt,TH_Ad,TH_Bnd
                                                    ;NOTE: no return on fail
          const     TH_Bnd,EndBase
          consth    TH_Bnd,EndBase
          aslt      V_InstTLBProt,TH_Ad,TH_Bnd
                                                    ;NOTE: no return on fail
          jmp       TLBHandler
          const     TH_Ac,0x4800                    ;VE,SE
; ......................................

SupDataTLBHandler:
;
; This routine handles the supervisor data TLB misses.  It should
; be enhanced to allow I/O access as well as data access.
          mfsr      TH_Ad,cha
          const     TH_Ac,0x7000                    ;VE,SR,SW
          const     TH_Bnd,MStkBase
          consth    TH_Bnd,MStkBase
          asge      V_DataTLBProt,TH_Ad,TH_Bnd
                                                    ;NOTE: no return on fail
          const     TH_Bnd,TextBase
          consth    TH_Bnd,TextBase
          aslt      V_InstTLBProt,TH_Ad,TH_Bnd
                                                    ;NOTE: no return on fail
;                                                (drop through to TLB handler)
; ......................................

TLBHandler:
;
; This routine handles TLB updates once it has been determined
; that the update is appropriate.
;
; NOTE:   This routine presumes an 8K-byte page size.
;
; In:     TH_Ad       the address where access is required
;         TH_Ac       the access that is required
;         lru         the recommended TLB for replacement
;
; Out:    (lru)       provides access to TH_Ad
          constn      TH_Wd1,RPN_MASK
          sll         TH_Wd0,TH_Wd1,5               ;shift for vtag
          and         TH_Wd1,TH_Wd1,TH_Ad           ;establish addr fields
          and         TH_Wd0,TH_Wd0,TH_Ad
          or          TH_Wd0,TH_Wd0,TH_Ac           ;establish access
          mfsr        TH_Reg,lru                    ;set the TLB entry
          mttlb      TH_Reg,TH_Wd0
          add        TH_Reg,TH_Reg,1
          mttlb      TH_Reg,TH_Wd1
          iret
; .......................................

          .eject
          .sbttl     "TLB Initialization"

LEAF      TLBInit,0
;
; This routine is used to initialize the TLBs.
;
; It clears all the TLB registers, thus marking all entries invalid.
;
; In:       (nothing)
;
; Out:      (nothing)
;
; Temps:    (see below)
;
          .reg       TI_Reg,%%(TEMP_REG + 0)          ;the TLB register number
          .reg       TI_Val,%%(TEMP_REG + 1)          ;the TLB value (0)
          .reg       TI_Cnt,%%(TEMP_REG + 2)          ;the TLB register count
          const      TI_Reg,0
          const      TI_Val,0
          const      TI_Cnt,(TLB_CNT - 2)             ;for jmpfdec

TI_Loop:
          mttlb      TI_Reg,TI_Val
          jmpfdec    TI_Cnt,TI_Loop
          add        TI_Reg,TI_Reg,1
          EPILOGUE
; .......................................

          .eject
          .sbttl     "Vector Initialization"

;
; In order that the vector initialization code might be compact
; and that the set of vectors initialized might be easily expanded,
; a table in .data is used.  Each entry in the table has two words.
; The first word is the number of the vector to be initialized.  The
; second word is the address of the handler.
;
          .data      ;switch to .data for table

VectInitTable:

          .word      V_SupInstTLB,SupInstTLBHandler
          .word      V_SupDataTLB,SupDataTLBHandler
          .word      V_MULTIPLY,MultiplyHandler
          .word      V_DIVIDE,DivideHandler
          .word      V_MULTIPLU,MultipluHandler
          .word      V_DIVIDU,DividuHandler
          .word      V_SPILL,SpillHandler
          .word      V_FILL,FillHandler
          .word      V_Timer,TimerHandler
          .equ       VINIT_CNT,((. - VectInitTable) / 8)
          .text                                       ;switch back to .text for code
; .......................................
```

```
VectInit:
;
; This routine initialzes the vectors for which handlers exist.
;
; In:     vab                                vector area base
;
; Out:    (vectors initialized)
;
; Temp:                                       (see below)
;
          .reg      VI_Vect,%%(TEMP_REG + 0)          ;vector value
          .reg      VI_St,%%(TEMP_REG + 1)            ;vector storage address
          .reg      VI_Cnt,%%(TEMP_REG + 2)           ;vector count
          .reg      VI_Base,%%(TEMP_REG + 3)          ;vector base
          .reg      VI_TbPt,%%(TEMP_REG + 4)          ;vector base
          mfsr      VI_Base,vab
          const     VI_Cnt,(VINIT_CNT - 2)            ;for jmpfdec
          const     VI_TbPt,VectInitTable
          consth    VI_TbPt,VectInitTable

VI_Loop:
          load      0,0,VI_St,VI_TbPt                 ;get the vector
          add       VI_TbPt,VI_TbPt,4
          sll       VI_St,VI_St,2                     ;convert to address (fixed v1.3)
          add       VI_St,VI_St,VI_Base
          load      0,0,VI_Vect,VI_TbPt               ;get the handler
          add       VI_TbPt,VI_TbPt,4
          jmpfdec   VI_Cnt,VI_Loop
          store     0,0,VI_Vect,VI_St
          jmpi      lr0
          nop
; ........................................

          .eject
          .sbttl    "ADAPT29K Initializations"
AdaptInit:
;
; This routine is for use in situations where the bootstrap process
; has not occurred.  Instead, the ADAPT29K has been used to load
; the program.  Initializations of the vectors, etc., will be
; required.
;
; As an aid to fault identification, the vector table is initialized
; with pointers to the words immediately following the vectors.  These
; words are initialized with HALT instructions.  When one of these
; halts executes, the ADAPT29K will report the event and the address
; of the halt.  This will allow the invalid trap that has occurred
; to be identified.
;
; CAUTION!  This requires that the vector pad be at least 1024.
;
          .reg      AI_Vect,%%(TEMP_REG + 0)          ;vector value
          .reg      AI_St,%%(TEMP_REG + 1)            ;vector storage address
          .reg      AI_Cnt,%%(TEMP_REG + 2)           ;vector count register
          .reg      AI_Halt,%%(TEMP_REG + 3)          ;halt instruction register
          mtsrim    cps,0x73                          ;PD,PI,SM,DI,DA
          mtsrim    vab,0
          mfsr      AI_St,vab
          const     AI_Vect,1024                      ;just beyond vectors
          const     AI_Halt,0x89000000
```

```
        consth      AI_Halt,0x89000000
        const       AI_Cnt,(256 - 2)                      ;for jmpfdec

AI_Loop:
        store       0,0,AI_St,AI_Vect                     ;store the vector
        add         AI_St,AI_St,4
        store       0,0,AI_Vect,AI_Halt                   ;store the HALT
        jmpfdec     AI_Cnt,AI_Loop
        add         AI_Vect,AI_Vect,4
        jmp         Start
        nop
; ........................................

        .eject
        .sbttl      "Start"

Start:
;
; This routine receives control after any required bootstrap processes.  It will
; initialize the vectors which are actually handled, clear the BSS area, initialize
; the TLBs, and establish initial stack pointers and an initial register frame.
; It will then invoke _main.
;
; In the event that _main returns, this routine will perform a warm start.
;
; In:     vab                                       indicates vector area
;
; Out:    (nothing)
        mtsrim      cps,0x73                              ;PD,PI,SM,DI,DA
        mtsrim      mmu,MMU_PS                            ;order # = 0
        mtsrim      cfg,0x10                              ;VF
        const       rfb,RStkTop                           ;set up stack pointers
        consth      rfb,RStkTop
        const       rab,(RStkTop - 512)
        consth      rab,(RStkTop - 512)
        add         lr1,rfb,0
        sub         rsp,rfb,16                            ;lr0,lr1,argc,argv
        const       msp,MStkTop
        consth      msp,MStkTop
        call        lr0,VectInit                          ;install handled vectors
        nop
        call        lr0,TLBInit                           ;establish TLBs invalid
        nop
        call        lr0,_ClrTm32                          ;clear and enable timer
        nop         ;    (leave to _main ???)
        mtsrim      cps,0x10                              ;SM
        const       lr2,0                                 ;argc = 0
        const       lr3,0                                 ;argv = 0
        call        lr0,_main
        nop
        mtsrim      cps,0x473                             ;FZ,PD,PI,SM,DI,DA
        mtsrim      ops,0x173                             ;RE,PD,PI,SM,DI,DA
        mtsrim      cfg,1                                 ;cache disabled
        mtsrim      chc,0                                 ;contents invalid
        mtsrim      pc1,0                                 ;cold start address
        mtsrim      pc0,4
        iretinv
;
;.........................................................................
; end of start.s
```

APPENDIX C: test.s

```
        .title      "Test of Assembly-language Utilities"
;
; Copyright 1988, Advanced Micro Devices, Inc.
; Written by Gibbons and Associates, Inc.
        .include    "romdcl.h"
        .extern     _GetTm32
        .data
        .word       0xDEADBEEF                      ;just to test
        .bss
        .block      1024                            ;verify zeros
        .text
        .eject
        .sbttl      "Multiply/Divide Test"
; ......................................

        LEAF        _MultDiv,0
;
; This routine gives a test of the multiply and divide trap
; handlers by the simple expedient of performing one of each.
; Using the debugger, it can be forced to loop, etc.
;
; In:     (nothing)
;
; Out:    (nothing)
;
; Temp:                                        (see below)
;
        .reg        MD_Mpd,%%(TEMP_REG + 0)         ;multiplicand
        .reg        MD_Mpr,%%(TEMP_REG + 1)         ;multiplier
        .reg        MD_PrLo,%%(TEMP_REG + 2)        ;product low
        .reg        MD_PrHi,%%(TEMP_REG + 3)        ;product high
        .reg        MD_Mlp,%%(TEMP_REG + 4)         ;BOOLEAN for looping
        .reg        MD_DvdHi,%%(TEMP_REG + 0)       ;dividend high
        .reg        MD_DvdLo,%%(TEMP_REG + 1)       ;dividend low
        .reg        MD_Dvsr,%%(TEMP_REG + 2)        ;divisor
        .reg        MD_Quot,%%(TEMP_REG + 3)        ;quotient
        .reg        MD_Dlp,%%(TEMP_REG + 4)         ;BOOLEAN for looping
        const       MD_Mlp,0                        ;FALSE
        const       MD_Mpd,3                        ;(full 32-bit for patching)
        consth      MD_Mpd,3
        const       MD_Mpr,5
        consth      MD_Mpr,5

M_Loop:
        multiply    MD_PrHi,MD_Mpd,MD_Mpr
        mfsr        MD_PrLo,q
        jmpt        MD_Mlp,M_Loop
        nop
        const       MD_Dlp,0                        ;FALSE
        const       MD_DvdHi,0                      ;(full setting for patch)
        consth      MD_DvdHi,0
        const       MD_DvdLo,15
        consth      MD_DvdLo,15
        const       MD_Dvsr,3
        consth      MD_Dvsr,3

D_Loop:
        mtsr        q,MD_DvdLo
        divide      MD_Quot,MD_DvdHi,MD_Dvsr
        jmpt        MD_Dlp,D_Loop
```

```
          nop
          EPILOGUE
; ........................................

          .eject
          .sbttl      "Spill/Fill Test"

          FUNCTION    _Recurse,1,29,1                     ;ALLOC_CNT = 32
;
; This routine is a simple recursive do-nothing that is used to test
; spill/fill.
;
; It accepts a count as its input, decrements that count, and, if the
; result is zero or greater, calls itself with the now decremented
; count.  Each instance of the routine allocates 32 new registers.
; Thus the total register requirement is 32 * (InCnt + 1) where InCnt
; is the input count.
;
; In:       (see below)
;
; Out:      (nothing in final return)
;
; Temp:     (allocated but not used)
;
          .reg        R_InCnt,%%(IN_PRM + 0)
          .reg        R_OutCnt,%%(OUT_PRM + 0)
          sub         R_OutCnt,R_InCnt,1
          jmpt        R_OutCnt,R_Exit
          nop
          call        lr0,_Recurse
          nop

R_Exit:
          EPILOGUE
; ........................................

          .eject
          .sbttl      "C Interrupt Interface Test"
          .extern     CIntf

          LEAF        _Trap70,1


;
; This "C" routine handles trap 70.  It increments the value of a global
; system register so that its effect may easily be seen.
;
; In:       (see below)
;
; Out:      st0         incremented
;           st1         set to input parameter value
;
          .reg        T70_V,%%(IN_PRM + 0)                ;the vector
          add         st0,st0,1
          add         st1,T70_V,0
          EPILOGUE
; ........................................

Trap70:
;
; This is the assembly-language routine that should get control on
; trap 70.  It invokes CIntf in such a way as to give control to
; _Trap70, the "C" routine above.  Note that control never returns
; to this routine.  CIntf performs the iret.
;
; In:      (nothing)
;
; Out:     (nothing)
;
        .reg        T70_Rout,%%(SYS_TEMP + 0)
        .reg        T70_Vect,%%(SYS_TEMP + 1)
        const       T70_Rout,_Trap70
        consth      T70_Rout,_Trap70
        jmp         CIntf
        const       T70_Vect,70
; .......................................

        .eject
        .sbttl      "_main"
        .global     _main

        FUNCTION    _main,2,2,1
;
; This routine plays the role of a C main routine.  It
; is coded in assembly language to ease testing with
; an absolute debugger.

        .reg        argc,%%(IN_PRM + 0)           ;argc (= 0)
        .reg        argv,%%(IN_PRM + 1)           ;argv (= NULL)
        .reg        StTm,%%(LOC_REG + 0)          ;start time
        .reg        EndTm,%%(LOC_REG + 1)         ;end time
        call        lr0,_GetTm32                  ;should return start time
        nop
        add         StTm,v0,0                     ;save the result
        call        lr0,_MultDiv                  ;test multiply/divide
        nop
        call        lr0,_Recurse                  ;test spill/fill
        const       p0,15                         ;require 1024 registers
        asneq       70,gr1,gr1                    ;force trap 70
        call        lr0,_GetTm32                  ;should return end time
        nop
        add         EndTm,v0,0                    ;save the result
        EPILOGUE
;
;....................................................................
; end of test.s
```

APPENDIX D: romdcl.h

```
        .eject
        .sbttl      "Register, constant, and Macro Declarations"
;
; Copyright 1988, Advanced Micro Devices
; Written by Gibbons and Associates, Inc.
;----------------------------------------------------------------------
; Global registers
;----------------------------------------------------------------------
;
        .reg        rsp,gr1                     ;local reg. var. stack pointer
        .equ        SYS_TEMP,64                 ;system temp registers
        .reg        st0,gr64
        .reg        st1,gr65
        .reg        st2,gr66
        .reg        st3,gr67
        .reg        st4,gr68
        .reg        st5,gr69
        .reg        st6,gr70
        .reg        st7,gr71
        .reg        st8,gr72
        .reg        st9,gr73
        .reg        st10,gr74
        .reg        st11,gr75
        .reg        st12,gr76
        .reg        st13,gr77
        .reg        st14,gr78
        .reg        st15,gr79
        .equ        SYS_STAT,80                 ;system static registers
        .reg        ss0,gr80
        .reg        ss1,gr81
        .reg        ss2,gr82
        .reg        ss3,gr83
        .reg        ss4,gr84
        .reg        ss5,gr85
        .reg        ss6,gr86
        .reg        ss7,gr87
        .reg        ss8,gr88
        .reg        ss9,gr89
        .reg        ss10,gr90
        .reg        ss11,gr91
        .reg        ss12,gr92
        .reg        ss13,gr93
        .reg        ss14,gr94
        .reg        ss15,gr95
        .equ        RET_VAL,96                  ;return registers
        .reg        v0,gr96
        .reg        v1,gr97
        .reg        v2,gr98
        .reg        v3,gr99
        .reg        v4,gr100
        .reg        v5,gr101
        .reg        v6,gr102
        .reg        v7,gr103
        .reg        v8,gr104
        .reg        v9,gr105
        .reg        v10,gr106
        .reg        v11,gr107
        .reg        v12,gr108
        .reg        v13,gr109
        .reg        v14,gr110
        .reg      v15,gr111
        .equ      TEMP_REG,96                     ;temp registers
        .reg      t0,gr96
        .reg      t1,gr97
        .reg      t2,gr98
        .reg      t3,gr99
        .reg      t4,gr100
        .reg      t5,gr101
        .reg      t6,gr102
        .reg      t7,gr103
        .reg      t8,gr104
        .reg      t9,gr105
        .reg      t10,gr106
        .reg      t11,gr107
        .reg      t12,gr108
        .reg      t13,gr109
        .reg      t14,gr110
        .reg      t15,gr111
        .equ      RES_REG,112                     ;reserved (for user)
        .reg      r0,gr112
        .reg      r1,gr113
        .reg      r2,gr114
        .reg      r3,gr115
        .equ      TEMP_EXT,116                    ;temp extension (and shared)
        .reg      x0,gr116
        .reg      x1,gr117
        .reg      x2,gr118
        .reg      x3,gr119
        .reg      x4,gr120
        .reg      x5,gr121
        .reg      x6,gr122
        .reg      x7,gr123
        .reg      x8,gr124
;-----------------------------------------------------------------------
; Global registers with special calling convention uses
;-----------------------------------------------------------------------
        .reg      tav,gr121                       ;trap handler argument (also x6)
        .reg      tpc,gr122                       ;trap handler return (also x7)
        .reg      lsrp,gr123                      ;large return pointer (also x8)
        .reg      slp,gr124                       ;static link pointer (also x9)
        .reg      msp,gr125                       ;memory stack pointer
        .reg      rab,gr126                       ;register alloc bound
        .reg      rfb,gr127                       ;register frame bound
;-----------------------------------------------------------------------
; Local compiler registers - output parameters, etc.
; (only valid if frame has been established)
;-----------------------------------------------------------------------
        .reg      p15,lr17                        ;parameter registers
        .reg      p14,lr16
        .reg      p13,lr15
        .reg      p12,lr14
        .reg      p11,lr13
        .reg      p10,lr12
        .reg      p9,lr11
        .reg      p8,lr10
        .reg      p7,lr9
        .reg      p6,lr8
        .reg      p5,lr7
        .reg      p4,lr6
        .reg      p3,lr5
        .reg      p2,lr4
```

```
         .reg       p1,lr3
         .reg       p0,lr2
;-----------------------------------------------------------------------
; TLB register count
;-----------------------------------------------------------------------

         .equ       TLB_CNT,128
         .eject

;-----------------------------------------------------------------------
; constants for general use
;-----------------------------------------------------------------------
         .equ       WRD_SIZ,4                          ;word size
         .equ       TRUE,0x80000000                    ;logical true -- bit 31
         .equ       FALSE,0x00000000                   ;logical false -- 0
         .equ       CHKPAT_a5,0xa5a5a5a5               ;check pattern
;-----------------------------------------------------------------------
; constants for data access control
;-----------------------------------------------------------------------
         .equ       CE,0b1                             ;co-processor enable
         .equ       CD,0b0                             ;co-processor disable
         .equ       AS,0b1000000                       ;set for I/O
         .equ       PA,0b0100000                       ;set for physical ad
         .equ       SB,0b0010000                       ;set for set BP
         .equ       UA,0b0001000                       ;set for user access
         .equ       ROM_OPT,0b100                      ;OPT values for acc
         .equ       DATA_OPT,0b000
         .equ       INST_OPT,0b000
         .equ       ROM_CTL,(PA + ROM_OPT)             ;control field
         .equ       DATA_CTL,(PA + DATA_OPT)
         .equ       INST_CTL,(PA + INST_OPT)
         .equ       IO_CTL,(AS + PA + DATA_OPT)

         .eject
;-----------------------------------------------------------------------
;defined vectors
;-----------------------------------------------------------------------
         .equ       V_IllegalOp,0
         .equ       V_Unaligned,1
         .equ       V_OutOfRange,2
         .equ       V_NoCoProc,3
         .equ       V_CoProcExcept,4
         .equ       V_ProtViol,5
         .equ       V_InstAccExcept,6
         .equ       V_DataAccExcept,7
         .equ       V_UserInstTLB,8
         .equ       V_UserDataTLB,9
         .equ       V_SupInstTLB,10
         .equ       V_SupDataTLB,11
         .equ       V_InstTLBProt,12
         .equ       V_DataTLBProt,13
         .equ       V_Timer,14
         .equ       V_Trace,15
         .equ       V_INTR0,16
         .equ       V_INTR1,17
         .equ       V_INTR2,18
         .equ       V_INTR3,19
         .equ       V_TRAP0,20
         .equ       V_TRAP1,21
; 22 - 31 reserved
         .equ       V_MULTIPLY,32
```

```
          .equ       V_DIVIDE,33
          .equ       V_MULTIPLU,34
          .equ       V_DIVIDU,35
          .equ       V_CONVERT,36
; 37 - 41 reserved
          .equ       V_FEQ,42
          .equ       V_DEQ,43
          .equ       V_FGT,44
          .equ       V_DGT,45
          .equ       V_FGE,46
          .equ       V_DGE,47
          .equ       V_FADD,48
          .equ       V_DADD,49
          .equ       V_FSUB,50
          .equ       V_DSUB,51
          .equ       V_FMUL,52
          .equ       V_DMUL,53
          .equ       V_FDIV,54
          .equ       V_DDIV,55
; 56 - 63 reserved
          .equ       V_SPILL,64
          .equ       V_FILL,65
          .equ       V_BSDCALL,66
          .equ       V_SYSVCALL,67
          .equ       V_BRKPNT,68
          .equ       V_EPI_OS,69

          .eject
          .macro     R_LEFT,REGVAR
;
; Rotate left
;
; Parameters:        REGVAR                             register to rotate
;
          add        REGVAR,REGVAR,REGVAR               ;shift left by 1 bit,C = MSB
          addc       REGVAR,REGVAR,0                    ;add C to LSB
          .endm
; .......................................

          .macro     FUNCTION,NAME,INCNT,LOCCNT,OUTCNT
;
; Introduces a non-leaf routine.
;
; This macro defines the standard tag word before the function,
; then establishes the statement label with the function's name
; and finally allocates a register stack frame.  It may not be used
; if a memory stack frame is required.
;
; Note also that the size of the register stack frame is limited.
; Neither this nor the lack of a memory frame is considered to be
; a severe restriction in an assembly-language environment.  The
; assembler will report errors if the requested frame is too large
; for this macro.
;
; It may be good practice to allocate an even number of both output
; registers and local registers.  This will help in maintaining
; double word alignment within these groups.  The macro will assure
; double word alignment of the stack frame as a whole, as required
; for correct linkage.
;
```

```
; Paramters:          NAME                              the function name
;                     INCNT                             input parameter count
;                     LOCCNT                            local register count
;                     OUTCNT                            output parameter count
;
          .set        ALLOC_CNT,((2 + OUTCNT + LOCCNT) << 2)
          .set        PAD_CNT,(ALLOC_CNT & 4)
          .set        ALLOC_CNT,(ALLOC_CNT + PAD_CNT)
          .if         (INCNT)
          .set        IN_PRM,(4 + OUTCNT + PAD_CNT + LOCCNT + 0x80)
          .endif
          .if         (LOCCNT)
          .set        LOC_REG,(2 + OUTCNT + PAD_CNT + 0x80)
          .endif
          .if         (OUTCNT)
          .set        OUT_PRM,(2 + 0x80)
          .endif
          .word       ((2 + OUTCNT + LOCCNT) << 16)

NAME:
          sub         rsp,rsp,ALLOC_CNT
          asgeu       V_SPILL,rsp,rab
          add         lr1,rsp,((4 + OUTCNT + LOCCNT + INCNT) << 2)
          .endm
; ......................................

          .macro      LEAF,NAME,INCNT
;
; Introduces a leaf routine
;
; This macro defines the standard tag word before the function,
; then establishes the statement label with the function's name.
;
; Paramters:          NAME                              the function name
;                     INCNT                             input parameter count
;
          .if         (INCNT)
          .set        IN_PRM,(2 + 0x80)
          .endif
          .set        ALLOC_CNT,0
          .word       0

NAME:
          .endm
; ......................................

          .macro      EPILOGUE
;
; Deallocates register stack frame (only and only if necessary).
;
          .if         (ALLOC_CNT)
          add         rsp,rsp,ALLOC_CNT
          nop
          jmpi        lr0
          asleu       V_FILL,lr1,rfb
          .else
          jmpi        lr0
          nop
          .endif
          .set        IN_PRM,(1024)                     ;illegal,to cause err on ref
          .set        LOC_REG,(1024)                    ;illegal,to cause err on ref
```

```
        .set      OUT_PRM,(1024)                     ;illegal,to cause err on ref
        .set      ALLOC_CNT,(1024)                   ;illegal,to cause err on ref
        endm


; Initial values for macro set variables to guard against misuse
        .set      IN_PRM,(1024)                      ;illegal,to cause err on ref
        .set      LOC_REG,(1024)                     ;illegal,to cause err on ref
        .set      OUT_PRM,(1024)                     ;illegal,to cause err on ref
        .set      ALLOC_CNT,(1024)                   ;illegal,to cause err on ref
;.........................................
; end of romdcl.h
```

APPENDIX E: test.ld

```
; test.ld    Linker Directives
;
; see test.s and start.s for descriptions of sections
;
load test.o,start.o
order vectors=0,rstack,mstack,.bss,.data,.text,endsect
```

Host Interface (HIF) v1.0 Specification Application Note

by E. M. Greenawalt

PREFACE

This document describes HIF (v1.0), the Am29000 Architectural Host Interface, and explains how to use it. HIF is the software standard that defines the interface between the user's high-level language program and the Am29000 processor. The document is written for experienced programmers and assumes a working knowledge of the Am29000 microprocessor.

INTRODUCTION

Advanced Micro Devices is developing a complete line of Am29000™ simulators, hardware target execution vehicles, and high-level language development tools for the Am29000 32-bit Streamlined Instruction Processor. These products are designed to support end-users who are building embedded system applications based on the Am29000 processor. For these users, often there is no existing operating system or kernel for their hardware design.

Before AMD could create development tools for the Am29000, a standard set of kernel services had to be defined that would interface a user-application program, written in a high-level language, to a host operating system or an Am29000 processor.

HIF, the host interface, is the software specification that defines this standard set of kernel services. Figure NO TAG shows the level where HIF resides. As implied by the figure, HIF does not describe any particular implementation; but rather each simulator, hardware vehicle, and high-level language implements HIF in its own way. The kernel services provide the minimum functionality needed to interface high-level language library functions to the user's operating system code.

Using HIF, program modules written in any of the languages available for the Am29000 can be combined, and the resulting program can run, without change, on any Am29000 simulator or hardware execution vehicle. Future AMD products will also use HIF, and AMD is actively encouraging third-party vendor support.

AMD is indebted to Embedded Performance, Incorporated (EPI), who originally developed the HIF concepts and then graciously placed them in the public domain.

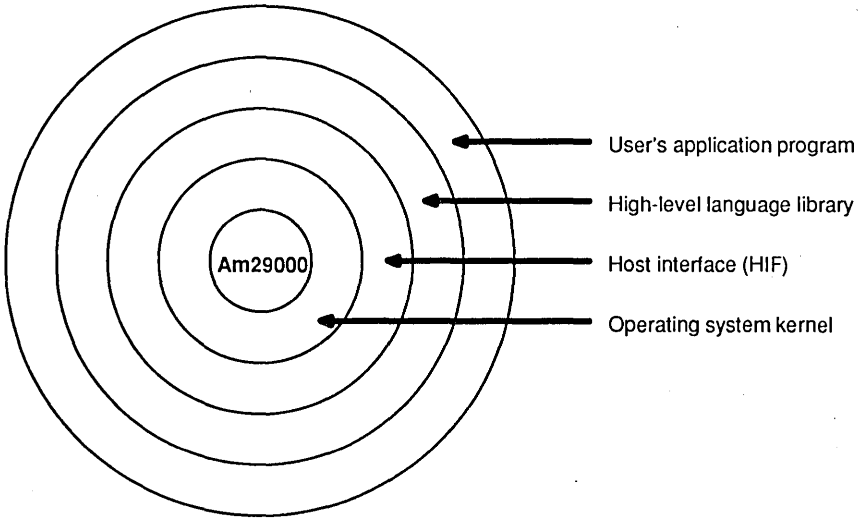


Figure 1. HIF Interface

Publication # 11014 Rev. A Amendment /0 Issue Date: 11/89

HIF APPLICATIONS

The HIF specification has broad applications; currently it provides the interface between the user's high-level language program and the following hardware and software products:

- *Am29000 Architectural Simulator.* This software product provides the means to simulate the operation of the Am29000 in a specified system environment. It provides detailed performance statistics by modeling the internal architecture of the Am29000, as well as system memory configurations and timing. The HIF specification is implemented to provide the interface between the user's program and the host operating system.
- *PC Execution Board (PCEB29K™).* This hardware/software product contains an Am29000 processor and memory and is an add-in board to IBM® PC-based systems. Part of the HIF specification is implemented on the board with another part implemented on the PC, to interface with the DOS operating system.
- *Standalone Execution Board (STEB).* This hardware product from STEP Engineering is intended to be an evaluation vehicle for the Am29000 and, optionally, Am29027™ Arithmetic Accelerator devices. The entire HIF specification is implemented on this board, which contains a resident monitor to implement the necessary kernel services.

Because HIF is a general-purpose standard, it can be used to interface any high-level language to the Am29000. User programs need not be written entirely in a high-level language; they may incorporate assembly-language functions when maximized performance is the primary concern.

HIF USERS

There are three categories of end-users who need to know the details of the host interface:

- Those using AMD-supplied hardware execution vehicles or simulators. This document defines the low-level mechanisms of HIF. With this information and the design concepts presented herein, end-users can extend the HIF environment to meet the needed degree of software functionality and sophistication.
- Those developing a custom kernel operating system for an Am29000 design. These users need access to AMD's high-level and assembly-language development tools. This document provides the information required to build a HIF-conforming kernel that uses the high-level language development tools directly. With this information, end-users can extend and customize the operating system code without interfering with the basic capabilities of the HIF.
- Those who are using the AMD-supplied high-level language development tools, but who must conform to another kernel operating system interface. There is sufficient information in this document to enable users to modify the development tools to properly interface with the target kernel's specifications.

HIF CONCEPTS

Programmers developing software in a high-level language do not work directly with the processor. Instead, they think in terms of a virtual machine ideally suited to the computational paradigm of the language. For instance, the C-language virtual machine has operations such as **fprintf()** and **strcpy()**, and the FORTRAN machine has operations such as **alog** and **sqrt**.

In actual practice, these virtual machines are implemented by libraries of object code that perform language-specific operations. As long as programmers use only the functions of the language's implied virtual machine, the programs will be portable across a broad range of implementations of the language.

However, computer systems generally provide another virtual machine to the world: one that is defined by the operating system software. This virtual machine requires system calls to perform the services that are implemented within the operating system code. Typical services are: process management, file system management, device management, and memory management.

The high-level language virtual machine usually consists of: (1) functions that can be implemented entirely within library routines, and (2) functions that require the services of the operating system. The functions of the first group (usually defined as the standard library for that language) are independent of the operating system virtual machine on which they are implemented. The functions of the second group must be coded in terms of the operating system virtual machine. In other words, they must make system calls.

It is often useful for end-users to also make system calls, even though this practice makes their programs less portable. This requirement can be accommodated by augmenting the language library with glue routines that specifically invoke the system calls, while providing the end user with suitable high-level syntax and semantics. (For detailed information on the glue routines for the C compiler, see the *HighC29K Reference Manual,* "Appendix A, Host Interface Definition.")

Given the above discussion, the required task is to create high-level language development tools that can be used easily and efficiently on a variety of execution vehicles. This task can be broken down into the following steps:

- Define an operating system virtual machine that provides sufficient functionality to support the fundamental requirements of each high-level language, but not so much as to require a massive development effort to create.
- Add appropriate glue routines to the standard libraries of the language so that the libraries are defined in terms of the operating system virtual machine.
- Implement the operating system's virtual machine services on the various execution vehicles. For hardware vehicles, the virtual machine is implemented by a kernel, typically contained in a resident monitor software program. For simulation vehicles, the virtual machine is implemented by code internal to the simulator and by code simulated by the simulator.

For the Am29000 hardware and software support products, HIF consists of the following operating system virtual machine definitions:

- A carefully defined, efficient system call mechanism. Accessing an HIF kernel service requires a transition from user mode to supervisor mode on the processor. This requires a specific mechanism, such as a trap handler, to be invoked.
- A set of services that support the primitive requirements of C, FORTRAN, and Pascal. Most of the services are defined according to UNIX® operating system interface specifications.
- A specification of the environment created by the kernel. This involves the definition of storage allocation and register initializations implemented by the kernel.

IMPLEMENTATION TYPES

Implementations of the HIF specification take two fundamental forms: self-hosted and embedded. Examples of each of these are provided in the *Standalone Execution Board* (STEB) manufactured by STEP Engineering and AMD's *PC Execution Board* (PCEB29K).

The STEB is a single-board computer that incorporates an Am29000 processor, an optional Am29027 arithmetic accelerator, program and data memory, serial ports, and timer-counter resources. The HIF implementation for this board consists of a resident monitor program that is downloaded into low-memory locations, and which implements the kernel services described in the "HIF Service Routine" section of this document. This is a self-hosted implementation.

In contrast to the STEB, the PCEB29K is an add-in board for IBM PC-compatible computers that incorporates an Am29000 processor, program and data memory, serial ports, and timer-counter resources. The HIF implementation for this board consists of two portions of code. One performs some of the kernel services on the board and the other performs some of the kernel services through the auspices of the DOS operating system. In the sense that the HIF is grafted onto the existing host operating system, it is called an embedded implementation. The architectural and instruction simulators are also embedded implementations because they share the HIF implementation between custom code and existing host-computer operating-system code.

There is no preference for either type of implementation as long as the services and features of the HIF specification are fully implemented in the target environment. With the standard interfaces that a HIF implementation presents, application programs written for one environment will run equally well in another.

HIF SERVICES PREVIEW

Table 1 lists the services defined by the HIF interface. Most are similar or identical to equivalent UNIX operating system calls. The titles given in column one are not the names that actually exist in a particular library but, instead, are the generic names of the services, for the purpose of this overview.

Table 1. HIF Services

Name	Description	Page
clock	Returns the elapsed processor time, in milliseconds	28
close	Closes a file	14
cycles	Returns processor cycle counts	29
exit	Terminates a program	10
getargs	Returns an argument address	27
getenv	Gets the environment	23
getpsize	Returns the memory page size	26
lseek	Sets a file position	17
open	Opens a file	11
read	Reads a buffer of data from a file	15
remove	Removes (deletes) a file	19
rename	Renames a file	20
sysalloc	Allocates memory space	24
sysfree	Frees allocated memory space	25
setvec	Sets user trap addresses	30
time	Returns number of seconds since Jan. 1, 1970	22
tmpname	Returns a temporary file name	21
write	Writes a buffer of data to a file	16

INTENDED AUDIENCE

This document is intended for systems designers and programmers who have a working knowledge of the Am29000 and its supporting peripheral hardware. It does not cover CPU design, the Am29000 instruction set, or any other hardware detail. Those topics are adequately covered in the reference documents listed below.

ABOUT THIS DOCUMENT

The contents of each section and appendix of this document are described below:

Section 1:	Introduction—discusses the important concepts underlying the host interface definition and previews the services that form the basis of the HIF specification.
Section 2:	System Call Mechanism—describes the mechanism used to make calls on the HIF services, and includes information on register usage for passing parameters and receiving results.
Section 3:	Service Routine Descriptions—describes each of the services defined in HIF and shows details of the code sequences, including examples, for invoking the services.
Section 4:	Process Environment—describes the standard memory allocation and register initializations performed by the HIF-conforming kernel prior to execution of a user program.
Appendix A:	HIF Quick Reference—lists all of the services and service parameters used in this document, in quick reference form.
Appendix B:	Error Messages—lists the error codes that HIF-conforming services may return.

REFERENCE DOCUMENTS

The user should have access to the following AMD documents:

- *Am29000 Streamlined Instruction Processor Users Manual*, order #10620
- *ADAPT29K User's Manual*
- *MON29K User's Manual*
- *MON29K Installation and Customization Manual*
- *Am29000 Execution Board and Monitor User's Manual*
- *ASM29K Utilities Manual* from the ASM29K documentation set
- *HighC29K Reference Manual* from the HighC29K documentation set

DOCUMENTATION CONVENTIONS

This specification assumes some familiarity with the UNIX operating system and the C language. In the following sections, the conventions presented in the subsections below are assumed.

Numeric Values

All numeric values are presumed to be expressed in decimal notation, unless otherwise stated. Hexadecimal values are prefaced by the characters "0x." Any value not prefaced by "0x" is defined to be a decimal number. For example:

`100092`	Decimal number
`0x100092`	Hexadecimal number

The first number, above, is a decimal value by implication, because it has not been prefaced by "0x." The second constant includes the explicit "0x" prefix, designating it as a hexadecimal value.

Character Strings

In the documentation, frequent mention is made of character strings that hold file names, path names, and environment variable names. In all cases, the HIF Specification requires that strings be constructed as a sequence of ASCII characters terminated by a NULL byte (an 8-bit character composed of all zero bits). This is the form in which strings are represented in the C language. Thus, the space reserved for a string must be one byte longer than the length of the string, to accommodate the NULL byte.

Languages such as Pascal, which require "counted" strings (that is, a single 8-bit byte in the first character of the string that specifies the number of bytes that follow), are required to convert these to NULL-terminated form before calling the HIF kernel services. In addition, languages other than C may need to convert strings passed back from the HIF kernel services to a compatible internal form. All returned strings are in NULL-terminated form.

SYSTEM CALL MECHANISM

System calls on Am29000-based systems are accomplished through invocation of a specific software trap. The Am29000 traps are roughly equivalent to software interrupts on other CPUs. System call traps are invoked through execution of an appropriate assert instruction whose assertion is FALSE at the time the instruction is executed.

Execution of an ASEQ, ASGE, ASGEU, ASGT, ASGTU, ASLE, ASLEU, ASLT, ASLTU, or ASNEQ instruction, where the result of the assertion is FALSE, will cause the trap specified in the instruction to be taken.

Once the trap is invoked, the Am29000 accesses a trap vector containing up to 256 separate trap handler addresses; or it may directly invoke a trap handler routine, depending on the implementation of the operating system trapping mechanism and the state of the Vector Fetch (VF) bit in the processor's Configuration Register. In most implementations, a table of vectors is used. However, the operating system software may implement direct trap execution for the increased efficiency it offers even though it requires the reservation of a much greater amount of system memory, but bypasses the need for vector table lookup.

When a trap is taken, the normal program execution sequence is interrupted and the trap handler is invoked. At this point, the current program's context is contained in Am29000 CPU registers. No saving or restoring of registers is performed by the processor when a trap occurs. HIF services are required to preserve the following registers and restore their contents before returning to the application program:

- All local registers
- Global registers *gr1*, *gr112*, *gr115*, and *gr125*
- Global registers *gr126* and *gr127* should be preserved according to AMD calling conventions. Their values may differ upon return from a HIF service, but the span between their values will remain the same.

The HIF services may modify the contents of certain registers without first saving their values, namely: *gr121*, *gr96*, and *gr97*; although, the application program should not count on *gr96* through *gr111* to be untouched by current and future HIF kernel services.

HIF SERVICE INVOCATION

Before invoking a HIF service, the service number and any input parameters to be passed must be loaded into Am29000 general registers. Both local and global registers are used for various HIF services, as shown in the HIF Quick Reference table in Appendix A of this document. Details for invoking specific services are contained in the Service Routine Descriptions section.

Service Number

Every HIF system service is identified by a unique number. Service numbers 0–127 and 256–383 are reserved for use by AMD and should not be used for user-supplied extensions.

```
const     lr2,input_file          ; set input file
consth    lr2,input_file          ; pathname address
const     lr3,O_RDONLY            ; set open mode
const     gr121,17                ; service number = 17 (open)
asneq     69,gr1,gr1              ; force trap 69 (system call)
const     lr2,input_file          ; set input file
consth    lr2,input_file          ; pathname address
const     lr3,O_RDONLY            ; set open mode
const     gr121,17                ; service number = 17 (open)
asneq     69,gr1,gr1              ; force trap 69 (system call)
```

The service number must be loaded into global register *gr121*, the trap-handler argument register. *Gr121* is a temporary register and its value is **not** preserved over a system call, nor will its value be preserved over any trap invoked by the running program.

Input Parameters

Any input parameters to be passed must be placed in local registers *lr2* through *lr17*. Input parameters are passed to HIF services using the parameter passing mechanism specified in the Am29000 calling conventions documentation (*Am29000 Streamlined Instruction Processor User's Manual*, order #10620).

Invoking a HIF Service

The HIF services are accessed by forcing trap 69 to occur, after the service number and parameters (if any) are loaded in the designated registers. Trap handler 69 executes the service in supervisor mode.

Returned Values

Most services return values, usually a single integer value (number of bytes read or written, number of clock ticks, size of a memory block, etc.), or a pointer (address of a file descriptor, address of a memory block, etc.). These values are returned in register *gr96*, per standard high-level language calling conventions.

If a service returns multiple values, the additional values are returned in *gr97*, *gr98*, and so forth. If the service fails to perform the requested task, the values contained in *gr96* and succeeding registers are not guaranteed to be valid.

See the documentation that accompanies your language processor for additional details on Am29000 high-level language calling conventions.

Status Reporting

In all cases, upon return from a HIF service, global register *gr121* contains either a TRUE value (0x80000000), or a positive non-zero integer error code indicating the reason for failure. Pre-defined error codes are listed in Appendix B of this document for existing HIF implementations.

HIF does not specify these error codes. They may be completely defined by an implementation, except for cases in which there is a corresponding, existing, UNIX error code. In these cases, the UNIX error code is expected to be used.

Example Assembly Code

The code fragment above shows how the definitions are implemented in Am29000 assembly-language to invoke the **open** HIF service to open a file:

In this example, local register *lr2* is loaded with the address of the filename constant; local register *lr3* contains the code: O_RDONLY, indicating that the file is to be opened for read-only access. The service number (17) is loaded into global register *gr121* and the service is executed by asserting that register *gr1* is not equal to itself. Since this is FALSE, the trap is invoked.

USER-MODE TRAPS

When a trap is invoked, the Am29000 switches from user mode to supervisor mode to execute the trap handler code. Most traps are properly executed in this mode, including the kernel services that implement the HIF specification. However, a few traps, such as the spill/fill handlers, are intended to execute in user mode. In these cases, the trap handler code is not part of the kernel, but is supplied by the particular high-level language product library and is linked with the user's application program.

In order to use a consistent trap handling mechanism, and to support the individual language products' methodologies for user-mode traps, a HIF service called **setvec**, is called with the address of the user-mode trap handler code for each of the traps handled in this way.

Once the user-mode handler addresses have been supplied, and the corresponding trap is invoked, the operating-system kernel receives control in supervisor mode. It then reinstates user mode and invokes the appropriate language library trap handler to complete the required operation. This bouncing from user mode to supervisor mode and back to user mode is referred to as a "trampoline" effect. When the trap handler's execution is complete, it returns directly to the user's application program, rather than back through the kernel.

The register stack spill/fill handlers are appropriate examples of code that is intended to execute in user mode. When a user's application program calls a function that requires a large number of local registers to execute, some currently unused registers may have to be written to main memory to free enough of the on-chip registers. In this case, the registers are spilled to memory via the spill-trap handler. When the function completes execution and intends to return to its caller, the spilled registers may have to be restored by calling the fill-trap handler. Since register stack management is unique for each application environment, individual spill/fill handlers are provided with each of the high-level language products.

HIF SERVICE ROUTINES

The HIF service routine calls currently defined are listed by decimal service number in Table 2 below and described in detail in the following pages.

Service numbers 0 through 127 and 256 through 383 are reserved by AMD and should not be used for user-supplied extensions. Table 3 describes the parameter names used in the service descriptions.

Table 2. HIF Service Calls

Number	Title	Description	Page
1	exit	Terminate a program	10
17	open	Open a file	11
18	close	Close a file	14
19	read	Read a buffer of data from a file	15
20	write	Write a buffer of data to a file	16
21	lseek	Seek file byte	17
22	remove	Remove a file	19
23	rename	Rename a file	20
33	tmpnam	Return a temporary name	21
49	time	Return seconds	22
65	getenv	Get environment	23
257	sysalloc	Allocate memory space	24
258	sysfree	Free memory space	25
259	getpsize	Return memory page size	26
260	getargs	Return base address	27
273	clock	Return milliseconds	28
274	cycles	Return processor cycles	29
289	setvec	Set user trap address	30

Table 3. Service Call Parameters

Parameter	Description
addrptr	A pointer to an allocated memory area, command-line-argument array, pathname buffer, or NULL-terminated environment variable name string.
baseaddr	The base address of command-line-argument vector.
buffptr	A pointer to buffer area which data is to be read from or written to during the execution of I/O services.
count	The number of bytes actually read from a file or written to a file.
cycles	The number of processor cycles returned.
errcode	The error code returned by the service, usually the same as the codes returned in the UNIX variable *errno*. See Appendix B, Table 8, starting at page 35, for a list of HIF error codes.
exitcode	The exit code of the application program.
filename	A pointer to a NULL-terminated ASCII string containing the directory path of a temporary filename.
fileno	The file descriptor, a small integer number. Descriptors 0, 1, and 2 are guaranteed to exist and correspond to open files on program entry (0 is UNIX equivalent of **stdin** and is opened for input, 1 is UNIX **stdout** and is opened for output, 2 is UNIX **stderr** and is opened for output). The *fileno* is returned when an **open** call is successful.
funaddr	A pointer to the address of a service.
mode	A series of option flags whose values represent the operation to be performed.
msecs	Milliseconds.
name	A pointer to a NULL-terminated ASCII string that contains an environment variable name.
nbytes	The number of data bytes requested to be read from or written to a file, or number of bytes to allocate from the heap.
newfile	A pointer to a NULL-terminated ASCII string that contains the directory path of a new filename.
offset	The number of bytes from a specified position (*orig*) in a file.
oldfile	A pointer to NULL-terminated ASCII string that contains the directory path of the old filename.
orig	A value of 0, 1, or 2 that refers to the beginning, current position, or the position of the end of a file.
pagesize	The memory page size in bytes returned.
pathname	A pointer to a NULL-terminated ASCII string that contains the directory path of a filename.
pflag	The UNIX file access permission codes.
retval	The return value that indicates success or failure.
secs	The seconds count returned.
trapno	The trap number.
where	The current position in a specified file.

Each service description on the pages that follow contains a concise explanation of the purpose of the service, the input and result register contents, and example assembly-language code to invoke the service. In all cases, operating system kernel services that meet the HIF specifications are invoked by forcing the software trap 69 to occur. The service number is always contained in general register *gr121* and parameters are passed, if necessary, in local registers, beginning with *lr2*.

When the service returns, general register *gr121* is required to report the success or failure of the service. If successful, *gr121* is expected to contain a TRUE boolean value (a 1 bit in the most significant bit position). If the service is not successful, a positive non-zero error code is returned in *gr121*. If the service returns results, the first result is held in *gr96*, the second in *gr97*, and so forth.

HIF implementations are required to return an error code when a requested operation is not possible. The codes from 0 to 255 are reserved for compatibility with current and future error return standards. The currently assigned codes and their meanings are listed in Appendix B, Table 8, starting on page 35. If a HIF implementation returns an error code in the range of 0 to 255, it **must** carry the identical meaning to the corresponding error code in this table. Error code values larger than 255 are available for implementation-specific errors.

In the examples, references are made to error handlers that are not part of the example code. These are assumed to be contained in the larger part of the user's code and are not supplied as part of the HIF specification. The JMPF instructions have been provided to show that interface glue routines should incorporate this error testing philosophy in order to be robust. In practice, error handling may be relegated to a single routine, or may be vested in individual sections of either in-line code, or as callable services by the glue routines.

Since HIF implementations may exist over a wide spectrum of systems, the capabilities of the HIF may vary from one system to the next. In the simplest case, the HIF implementation in an embedded Am29000 system, such as a printer controller, may contain no external file system. In this event, the input/output facilities specified in the kernel service descriptions need not be implemented. In more common cases, where the HIF will exist on systems that have full operating system capabilities, such as DOS or UNIX, it is assumed that all of the features of the HIF will be implemented. The service descriptions in this document provide a set of standard interfaces for commonly implemented operating system interfaces. If individual features are implemented, the interfaces are expected to follow the guidelines in this specification.

Descriptions of the individual services follow on the remaining pages of this section. They are listed in numeric sequence by service number. Appendix A, HIF Quick Reference, allows easy location of a service by its number.

Service 1—exit — Terminate a Program

Description

This service terminates the current program and returns a value to the system kernel, indicating the reason for termination. By convention, a zero passed in *lr2* indicates normal termination, while any non-zero value indicates an abnormal termination condition. There are no returned values in registers *gr96* and *gr121* since this service does not return.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	1 (0x1)	Service number
	lr2	exitcode	User-supplied exit code
Returns:	gr96	undefined	This service call does not return
	gr121	undefined	This service call does not return

Example Call

```
const    lr2, 1          ; exit code = 1
const    gr121,1         ; service = 1
asneq    69,gr1,gr1      ; call the operating system
```

In the above example, the operating system kernel is being called with service code 1 and an exit code of 1, which is interpreted according to the specifications of the individual operating system. The value of the exit code is not defined as part of the HIF specification.

In general, however, an exit code of zero (0) specifies a normal program termination condition, while a non-zero code specifies an abnormal termination resulting from detection of an error condition within the program.

Programs can terminate normally by falling through the curly brace at the end of the **main** function in a C-language program. Other languages may require an explicit call to the kernel's **exit** service.

Service 17—open

Open a File

Description

This service opens a named file in a requested mode. Files must be explicitly opened before any **read**, **write**, **close**, or other file positioning accesses can be accomplished. The **open** service, if successful, returns an integer token that is used to refer to the file in all subsequent service requests. In many high-level languages, the returned token is referred to as a "file descriptor."

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	17 (0x11)	Service number
	lr2	pathname	A pointer to a filename
	lr3	mode	See parameter descriptions below
	lr4	pflag	See parameter descriptions below
Returns:	gr96	fileno	Success: ≥ (file descriptor) Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Parameter Descriptions

Pathname is a pointer to a zero-terminated string that contains the full path and name of the file being opened.* Individual operating systems have different means to specify this information. With hierarchical file systems, individual directory levels are separated with special characters that can not be part of a valid filename or directory name. In UNIX-compatible file systems, directory names are separated by forward slash characters "/" (e.g., "/usr/jack/files/myfile"); where "usr," "jack," and "files" are succeedingly lower directory levels, beginning at the root directory of the file system. The name "myfile" is the filename to be opened at the specified level. The individual characteristics of files and pathnames are determined by the specifications of a particular operating system implementation.

Mode is composed of a set of flags, whose mnemonics and associated values are listed in Table 4.

Table 4. Open Service Parameters

Name	Value	Description
O_RDONLY	0x0000	Open for read only access
O_WRONLY	0x0001	Open for write only access
O_RDWR	0x0002	Open for read and write access
O_APPEND	0x0008	Always append when writing
O_NDELAY	0x0010	No delay
O_CREAT	0x0200	Create file if it does not exist
O_TRUNC	0x0400	If the file exists, truncate it to zero length
O_EXCL	0x0800	Fail if writing and the file exists
O_FORM	0x4000	Open in text format

The O_RDONLY mode provides the means to open a file and guarantee that subsequent accesses to that file will be limited to **read** operations. The operating system implementation will determine how errors are reported for unauthorized operations. The file pointer is positioned at the beginning of the file, unless the O_APPEND mode is also selected.

* The HIF specification intentionally refrains from defining the constituents of a legal pathname, or any intrinsic characteristics of the implemented file system. In this regard, the only requirement of a HIF-conforming kernel is that when the **open** service is successfully performed, that the routine returns a small integer value that can be used in subsequent input/output service calls to refer to the opened file.

The O_WRONLY mode provides the means to open a file and guarantee that subsequent accesses to that file will be limited to **write** operations. The operating system implementation will determine how errors are reported for unauthorized operations. The file pointer is positioned at the beginning of the file, unless the O_APPEND mode is also selected.

The O_RDWR mode provides the means to open a file for subsequent **read** and **write** accesses. The file pointer is positioned at the beginning of the file, unless the O_APPEND mode is also selected.

If O_APPEND mode is selected, the file pointer is positioned to the end of the file at the conclusion of a successful **open** operation, so that data written to the file is added following the existing file contents.

Ordinarily, a file must already exist in order to be opened. If the O_CREAT mode is selected, files that do not currently exist are created; otherwise, the **open** function will return an error condition in *gr121*.

If a file being opened already exists and the O_TRUNC mode is selected, the original contents of the file are discarded and the file pointer is placed at the beginning of the (empty) file. If the file does not already exist, the HIF service routine should return an error value in *gr121*, unless O_CREAT mode is also selected.

The O_EXCL mode provides a method for refusing to **open** the file if the O_WRONLY or O_RDWR modes are selected and the file already exists. In this case, the kernel service routine should return an error code in *gr121*.

O_FORM mode indicates that the file is to be opened as a text file, rather than a binary file. The nominal standard input, output, and error files (file descriptors 0, 1, and 2) are assumed to be open in text mode prior to commencing execution of the user's program.

When opening a FIFO (interprocess communication file) with O_RDONLY or O_WRONLY set, the following conditions apply:

- If O_NDELAY is set (i.e., equal to 0x0010):
  - A read-only open will return without delay.
  - A write-only open will return an error if no process currently has the file open for reading.
- If O_NDELAY is clear (i.e., equal to 0x0000):
  - A read-only open will block until a process opens a file for writing.
  - A write-only open will block until a process opens a file for reading.

When opening a file associated with a communication line (e.g., a remote modem or terminal connection), the following conditions apply:

- If O_NDELAY is set, the open will return without waiting for the carrier detect condition to be TRUE.
- If O_NDELAY is clear, the open will block until the carrier is found to be present.

The optional *pflag* parameter specifies the access permissions associated with a file; it is only required when O_CREAT is also specified (i.e., create a new file if it does not already exist). If the file already exists, *pflag* is ignored. This parameter specifies UNIX-style file access permission codes (*r*, *w*, and *x* for read, write, and execute, respectively) for the file's owner, the work group, and other users. If the parameter is missing, *pflag* will be set to –1 (all accesses allowed). See the UNIX operating system documentation for additional information on this topic.

Example Call

```
path:      .ascii   "/usr/jack/files/myfile\0"
           .set     mode,O_RDWR|O_CREAT|O_FORM
           .set     permit,0x180

fd:        .word    0                   ;
           const    lr2,path            ; address of pathname
           consth   lr2,path            ;
           const    lr3,mode            ; open mode settings
           const    lr4,permit          ; permissions
           const    gr121,17            ; service = 17 (open)
           asneq    69,gr1,gr1          ; perform OS call
           jmpf     gr121,open_err      ; jump if error on open
           const    gr120,fd            ; set address of
           consth   gr120,fd            ; file descriptor
           store    0,0,gr96,gr120      ; store file descriptor
```

In the above example, the file is being opened in read/write text mode. The UNIX permissions of the owner are set to allow reading and writing, but not execution, and all other permissions are denied. As indicated above in the parameter descriptions, the file permissions are only used if the file does not already exist. When the **open** service returns, the program jumps to the **open_err** error handler if the open was not successful; otherwise, the file descriptor returned by the service is stored for future use in **read**, **write**, **lseek**, **remove**, **rename**, or **close** service calls.

As described in the introduction to these services, the HIF can be implemented to several degrees of elaboration, depending on the underlying system hardware, and whether the operating system is able to provide the full set of kernel services. In the least capable instance (i.e., a standalone board with a serial port), it is likely that only the O_RDONLY, O_WRONLY and O_RDWR modes will be supported. In more capable systems, the additional modes should be implemented, if possible.

Service 18—close Close a File

Description

This service closes the open file associated with the file descriptor passed in *lr2*. Closing all files is automatic on program exit (see **exit**), but since there is an implementation-defined limit on the number of open files per process, an explicit **close** service call is necessary for programs that deal with many files.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	18 (0x12)	Service number
	lr2	fileno	File descriptor
Returns:	gr96	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fd:        .word    0

           const    gr96,fd          ; set address of
           consth   gr96,fd          ; file descriptor
           load     0,0,lr2,gr96     ; get file descriptor
           const    gr121,18         ; service = 18
           asneq    69,gr1,gr1       ; and call the OS
           jmpf     gr121,clos_err   ; handle close error
           nop                       ;
```

The above example illustrates loading a previously stored file descriptor (*fd*, in this case) and calling the kernel's **close** service to close the file associated with that descriptor. If an error occurs when attempting to close the file, the kernel will return an error code in *gr121* (the content of that register will not be TRUE) and the program will jump to an error handler; otherwise, program execution will continue.

Service 19—read

Read a Buffer of Data from a File

Description

This service reads a number of bytes from a previously opened file (identified by a small integer file descriptor in *lr2* that was returned by the **open** service) into memory starting at the address given by the buffer pointer in *lr3*. *Lr4* contains the number of bytes to be read. The number of bytes actually read is returned in *gr96*. Zero is returned in *gr96* if the file is already positioned at its end-of-file. If an error is detected, a small positive integer is returned in *gr121*, indicating the nature of the error.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	19 (0x13)	Service number
	lr2	fileno	File descriptor
	lr3	buffptr	A pointer to buffer area
	lr4	nbytes	Number of bytes to be read
Returns:	gr96	count	Success: > 0 (number of bytes actually read) EOF: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fd:         .word     0
            .set      BUFSIZE,256
buf:        .block    BUFSIZE
num:        .word     0

            const     gr96,fd             ; set address of
            consth    gr96,fd             ; file descriptor
            load      0,0,lr2,gr96        ; get file descriptor
            const     lr3,buf             ; set buffer address
            consth    lr3,buf             ;
            const     lr4,BUFSIZE         ; specify buffer size
            const     gr121,19            ; service = 19
            asneq     69,gr1,gr1          ; call the OS
            jmpf      gr121,rd_err        ; handle read errors
            const     gr120,num           ; set address of
            consth    gr120,num           ; 'num' argument
            store     0,0,gr96,gr120      ; store bytes read
```

The above example requests the HIF to return *BUFSIZE* bytes from the file descriptor contained in the variable *fd*. If the call is successful, *gr121* will contain a TRUE value and *gr96* will contain the number of bytes actually read. If the service fails, *gr121* will contain the error code.

Service 20—write

Write a Buffer of Data to a File

Description

This service writes a number of bytes from memory (starting at the address given by the pointer in *lr3*) into the file specified by the small positive integer file descriptor that was returned by the open service when the file was opened for writing. *Lr4* contains the number of bytes to be written. The number of bytes actually written is returned in *gr96.* If an error is detected, *gr121* will contain a small positive integer on return from the service, indicating the nature of the error.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	20 (0x14)	Service number
	lr2	fileno	File descriptor
	lr3	buffptr	A pointer to the buffer area
	lr4	nbytes	Number of bytes to be written
Returns:	gr96	count	Success: = *lr4* Failure: $0 \le gr96 < lr4$ Extreme: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fd:        .word    0
           .set     BUFSIZE,256
buf:       .block   BUFSIZE
num:       .word    0

           const    gr96,fd             ; set address of
           consth   gr96,fd             ; file descriptor
           load     0,0,lr2,gr96        ; get file descriptor
           const    lr3,buf             ; set buffer address
           consth   lr3,buf             ;
           const    lr4,BUFSIZE         ; specify buffer size
           const    gr121,20            ; service = 20
           asneq    69,gr1,gr1          ; call the OS
           jmpf     gr121,wr_err        ; handle write errors
           const    gr120,num           ; set address of
           consth   gr120,num           ; "num" variable
           store    0,0,gr96,gr120      ; store bytes written
```

The example, above, writes *BUFSIZE* bytes from the buffer located at *buf* to the file associated with the descriptor stored in *fd.* If errors are detected during execution of the service, the value returned in *gr121* will be FALSE. In this case, the **wr_err** error handler will be invoked. The number of bytes actually written is stored in the variable *num.*

Service 21—lseek

Seek a File Byte

Description

This service positions the file associated with the file descriptor in *lr2*, "*offset*" number of bytes from the position of the file referred to by the *orig* parameter. *Lr3* contains the number of bytes offset and *lr4* contains the value for *orig*. The parameter *orig* is defined as:

0 = Beginning of the file
1 = Current position of the file
2 = End of the file

The **lseek** service can be used to reposition the file pointer anywhere in a file. The offset parameter may either be positive or negative. However, it is considered an error to attempt to seek in front of the beginning of the file.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	21 (0x15)	Service number
	lr2	fileno	File descriptor
	lr3	offset	Number of bytes offset from orig
	lr4	orig	A code number indicating the point within the file from which the offset is counted
Returns:	gr96	where	Success: ≥ 0 (current position in the file) Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fd:         .word    6               ; file descriptor = 6
orig:       .word    0               ; origin = start of file
off:        .word    23              ; offset = 23 bytes

            const    gr96,fd         ; set address of
            consth   gr96,fd         ; file descriptor
            load     0,0,lr2,gr96    ; get file descriptor
            const    gr96,off        ; set address of
            consth   gr96,off        ; offset argument
            load     0,0,lr3,gr96    ; get offset
            const    gr96,orig       ; set address of
            consth   gr96,orig       ; origin argument
            load     0,0,lr4,gr96    ; get origin
            const    gr121,21        ; service = 21
            asneq    69,gr1,gr1      ; call the OS
            jmpf     gr121,seek_err  ; seek error if false
            nop                      ;
```

The above example shows how a file can be positioned to a particular byte address by specifying the *orig*, which is the starting point from which the file position is adjusted, and the offset, which is the number of bytes from the *orig* to move the file pointer. In this case, the file identified by file descriptor 6 is being repositioned to byte 23, measured from the beginning of the file (*orig* = 0).

The file descriptor, *offset*, and *orig* values are loaded from preset constants and **lseek** is called to perform the file positioning operation. If an error occurs when attempting to reposition the file, the value returned in *gr121* is not TRUE, containing an error code that indicates the reason for the error. Upon return, *gr96* also contains the file position measured from the beginning of the file. In this case, this value is not stored.

Service 22—remove

Remove a File

Description

This service deletes a file from the file system. *Lr2* contains a pointer to the pathname of the file. The path must point to an existing file, and the referenced file should not be currently open. The behavior of the remove service is undefined if the file is open.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	22 (0x16)	Service number
	lr2	pathname	A pointer to string that contains the pathname of the file
Returns:	gr96	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
path:       .ascii   "/usr/jack/files/myfile\0"

            const    lr2,path          ; set address of file
            consth   lr2,path          ; pathname.
            const    gr121,22          ; service = 22
            asneq    69,gr1,gr1        ; call the OS
            jmpf     gr121,rem_err     ; jump if error
            nop
```

In the above example, a file with a UNIX-style pathname stored in the string named path is being removed. The address (*pointer*) to the string is put into *lr2* and the kernel service 22 is called to remove the file. If the file does not exist, or if it has not previously been closed, an error code will be returned in *gr121*; otherwise, the value in *gr121* will be TRUE.

Service 23—rename   Rename a File

Description

This service moves a file to a new location within the file system. *Lr2* contains a pointer to the file's old pathname and *lr3* contains a pointer to the file's new pathname. When all components of the old and new pathnames are the same, except for the filename, the file is said to have been renamed. The file identified by the old pathname must already exist, or an error code will be returned and the rename operation will not be performed.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	23 (0x17)	Service number
	lr2	oldfile	A pointer to string containing the old pathname of the file
	lr3	newfile	A pointer to string containing the new pathname of the file
Returns:	gr96	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
old:        .ascii     "/usr/fred/payroll/report\0"
new:        .ascii     "/usr/fred/history/june89\0"

            const      lr2,old          ; set address of old pathname
            consth     lr2,old
            const      lr3,new          ; set address of new pathname
            consth     lr3,new
            const      gr121,23         ; service = 23 (rename)
            asneq      69,gr1,gr1       ; call the OS

            jmpf       gr121,ren_err    ; jump if rename error
            nop
```

The above example moves a file from its old path (renaming it in the process) to its new pathname location. The file will no longer be found at the old location.

Service 33—tmpnam

Return Temporary Name

Description

This service generates a string that can be used as a temporary file pathname. A different name is generated each time it is called. Generally, the name is guaranteed not to duplicate any existing filename. The argument passed in *lr2* should be a valid pointer to a buffer that is large enough to contain the constructed file name. HIF implementations are required to allocate a minimum of 128 bytes for this purpose.

If the argument in *lr2* contains a NULL pointer, the HIF service routine should treat this as an error condition and return a non-zero error number in global register *gr121*.

The HIF specification sets no standards for the format or content of legal pathnames; these are determined by individual operating system requirements. However, each implementation should undertake to construct a valid filename that is also unique.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	33 (0x21)	Service number
	lr2	addrptr	A pointer to buffer into which the filename is to be stored
Returns:	gr96	filename	Success: pointer to the temporary filename string. This will be the same as *lr2* on entry unless an error occurred Failure: = 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
fbuf:       .block   21                  ; buffer size = 21 bytes

            const    lr2,fbuf            ; set buffer pointer
            consth   lr2,fbuf            ;
            const    gr121,33            ; service = 33
            asneq    69,gr1,gr1          ; call the OS
            jmpf     gr121,tmp_err       ; jump if error
            nop
```

In the above example, the **tmpnam** service is called with a pointer to *fbuf*, which has been allocated to hold a name that is up to 21 bytes in length. If the service is able to construct a valid name, the filename will be stored in *fbuf* when the service returns. If the content of *gr121* on return is not TRUE, the program fragment jumps to **tmp_err** to handle the error condition.

Service 49—time Return Seconds Since 1970

Description

This service returns, in register *gr96*, the number of seconds elapsed since midnight, January 1, 1970, as an integer 32-bit value. It is assumed that the kernel service will have access to a counter whose contents can be preloaded that measures time, with at least a one-second resolution, for this purpose.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	49 (0x31)	Service number
Returns:	gr96	secs	Success: ≠ 0 (time in seconds) Failure: = 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
secs:       .word       0

            const       gr121,49          ; service = 49
            asneq       69,gr1,gr1        ; call the OS
            jmpf        gr121,tim_err     ; jump if error
            const       gr120,secs        ; set the address
            consth      gr120,secs        ; for storing 'secs'
            store       0,0,gr96,gr120    ; store the seconds
```

In the above example, the kernel service **time** is being called. If the value returned in *gr121* is TRUE, the number of seconds returned in *gr96* is stored in the *secs* variable; otherwise, the program jumps to **tim_err** to determine the cause of the error.

Service 65—getenv

Get Environment

Description

This service searches the system environment for a string associated with a specified symbol. *Lr2* contains a pointer to the symbol name. If the symbol name is found, a pointer to the string associated with it is returned in *gr96*; otherwise, a NULL pointer is returned.

In UNIX-hosted systems, the `setenv` command allows a user to associate a symbol with an arbitrary string. For example, the command

```
setenv TERM vt100
```

defines the string "vt100" to be associated with the symbol named *TERM*. Application programs can use this association to determine the type of terminal connected to the system, and, therefore, use the correct set of codes when outputting information to the user's screen. To access the string, **getenv** should be called with *lr2* pointing to a string containing the *TERM* symbol name. The address returned in *gr96* will point to the corresponding "vt100" string if *TERM* is found. In UNIX-hosted systems, entering a different `setenv` command lets the user select a different terminal name without requiring recompilation of the application program.

Operating system implementations that do not include provisions for environment variables should always return a NULL value in *gr96* when this service is requested.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	65 (0x41)	Service number
	lr2	name	A pointer to the symbol name
Returns:	gr96	addrptr	Success: pointer to the symbol name string Failure: = 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
mysym:      .ascii    "MYSYMBOL\0"
strptr:     .word     0

            const     lr2,mysym          ; set address of symbol
            consth    lr2,mysym          ; to be located in environment
            const     gr121,65           ; service = 65
            asneq     69,gr1,gr1         ; call the OS
            jmpf      gr121,env_err      ; jump if error
            const     gr120,strptr       ; set address of
            consth    gr120,strptr       ; string pointer
            store     0,0,gr96,gr120     ; store string pointer
```

The above example program calls the operating system **getenv** service to access a string associated with the environment variable *MYSYMBOL*. If the symbol is found, a pointer to the string associated with the symbol is returned in *gr96*. If the call is not successful (i.e., *gr121* holds a FALSE boolean value upon return), the program jumps to **env_err** to handle the error condition.

Service 257—sysalloc — Allocate Memory Space

Description

This service allocates a specified number of contiguous bytes from the operating-system-maintained heap and returns a pointer to the base of the allocated block. *Lr2* contains the number of bytes requested. If the storage is successfully allocated, *gr96* contains a pointer to the block; otherwise, *gr121* contains an error code indicating the reason for failure of the call.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	257 (0x101)	Service number
	lr2	nbytes	Number of bytes requested
Returns:	gr96	addrptr	Success: pointer to allocated bytes, Failure: = 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
blkptr:    .word     0

           const     lr2, 1200          ; request 1200 bytes
           const     gr121,257          ; service = 257
           asneq     69,gr1,gr1         ; call the OS
           jmpf      gr121,alloc_err    ; jump if error
           const     gr120,blkptr       ; set address to store
           consth    gr120,blkptr       ; pointer
           store     0,0,gr96,gr120     ; store the pointer
```

The above example requests a block of 1200 contiguous bytes from the system heap. If the call is successful, the program stores the pointer returned in *gr96* into a local variable called *blkptr*. If *gr121* contains a boolean FALSE value when the service returns, the program jumps to **alloc_err** to handle the error condition.

Service 258—sysfree

Free Memory Space

Description

This service returns memory to the system starting at the address specified in *lr2*. *Lr3* contains the number of bytes to be released. The pointer passed to the **sysfree** service in *lr2* and the byte count passed in *lr3* must match the address returned by a previous **sysalloc** service request for the identical number of bytes. No dynamic memory allocation structure is implied by this service. High-level language library functions such as **malloc()** and **free()** for the C language are required to manage random dynamic memory block allocation and deallocation, using the **sysalloc** and **sysfree** kernel functions as their basis.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	258 (0x102)	Service number
	lr2	addrptr	Starting address of area returned
	lr3	nbytes	Number of bytes to release
Returns:	gr96	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
blkptr:     .word    0

            const    gr120,blkptr       ; set address of previously
            consth   gr120,blkptr       ; block pointer
            load     0,0,lr2,gr120      ; fetch pointer to block
            const    lr3,1200           ; set number of bytes to release
            const    gr121,258          ; service = 258
            asneq    69,gr1,gr1         ; call the OS
            jmpf     gr121,free_err     ; jump if error
            nop                         ;
```

The above example calls **sysfree** to deallocate 1200 bytes of contiguous memory, beginning at the address stored in the *blkptr* variable. If the call is successful, the program continues; otherwise, if the return value in *gr121* is FALSE, the program jumps to **free_err** to handle the error condition.

Service 259—getpsize Return Memory Page Size

Description

This service returns, in register *gr96*, the page size, in bytes, used by the memory system of the HIF implementation.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	259 (0x103)	Service number
Returns:	gr96	pagesize	Success: memory page size, one of the following: 1024, 2048, 4096, and 8192 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
pagsiz:     .word     0

            const     gr121,259          ; service = 259
            asneq     69,gr1,gr1         ; call the OS
            jmpf      gr121,pag_err      ; jump if error
            const     gr120,pagsiz       ; set address to
            consth    gr120,pagsiz       ; store the page size
            store     0,0,gr96,gr120     ; store it!
```

The above example calls the operating system kernel to return the page size used by the virtual memory system. If the call was successful, *gr121* will contain a boolean TRUE result and the program will store the value in *gr96* into the *pagsiz* variable; otherwise, a boolean FALSE is returned in *gr121*. In this case, the program will jump to **pag_err** to handle the error condition.

Service 260—getargs

Return Base Address

Description

This service returns the base address of the command-line-argument vector *argv* in register *gr96*, as constructed by the operating system kernel when an application program is invoked.

Arguments are stored by the operating system as a series of NULL-terminated character strings. A pointer containing the address of each string is stored in an array whose base address (referred to as *argv*) is returned by the **getargs** HIF service. The last entry in the array contains a NULL pointer (an address consisting of all zero bits). The number of arguments can be computed by counting the number of pointers in the array, using the fact that the NULL pointer terminates the list.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	260 (0x104)	Service number
Returns:	gr96	baseaddr	Success: base address of argv Failure: = 0 (NULL pointer)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
argptr:     .word    0
            const    gr121,260          ; service = 260
            asneq    69,gr1,gr1         ; call the OS
            jmpf     gr121,bas_err      ; jump if error
            const    gr120,argptr       ; set address where base
            consth   gr120,argptr       ; pointer is to be stored
            store    0,0,gr96,gr120     ; store the pointer
```

The above example calls operating system service 260 to access the command-line-argument vector address. If the service executes without error, the program continues by storing the argument vector address in the variable *basptr*. If *gr121* contains a boolean FALSE value upon return, the program jumps to **bas_err** to handle the error condition.

Service 273—clock Return Time in Milliseconds

Description

This service returns the elapsed processor time in milliseconds. Operating system initialization procedures set this value to zero on startup. Successive calls to this service return times that can be arithmetically subtracted to accurately measure time intervals.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	273 (0x111)	Service number
Returns:	gr96	msecs	Success: ≠ 0 (time in milliseconds) Failure: = 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
time:       .word     0

            const     gr121,273          ; service = 273
            asneq     69,gr1,gr1         ; call the OS
            jmpf      gr121,clk_err      ; jump if error
            const     gr120,time         ; set the address where
            consth    gr120,time         ; time is to be stored
            store     0,0,gr96,gr120     ; store the time in ms.
```

The above example calls the operating system kernel to get the current value of the system clock in milliseconds. On return, if *gr121* contains a boolean FALSE value, the program jumps to **clk_err** to handle the error; otherwise, the time in milliseconds is stored in the variable *time.*

Service 274—cycles Return Processor Cycles

Description

This service returns an ascending positive number in registers *gr96* and *gr97* that is the number of processor cycles that have elapsed since the last hardware RESET was applied to the CPU. It provides a mechanism for user programs to access the contents of the internal Am29000 timer counter register. The cycle count can be multiplied by the speed of the processor clock to convert it to a time value. *Gr97* will contain the most significant bits of the cycle count, while *gr96* will contain the least significant bits. HIF implementations of this service are required to provide a cycle count with a minimum of 56 bits of precision.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	274 (0x112)	Service number
Returns:	gr96	cycles	Success: Bits 0-31 of processor cycles Failure: = 0 (in both *gr96* and *gr97*)
	gr97	cycles	Success: Bits 32-55 of processor cycles Failure: = 0 (in both *gr96* and *gr97*)
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
cycles:     .word    0                    ; MSBs of cycles
            .word    0                    ; LSBs of cycles

            const    gr121,274            ; service = 274
            asneq    69,gr1,gr1           ; call the OS
            jmpf     gr121,cyc_err        ; jump if error
            const    gr120,cycles         ; set the address where the
            consth   gr120,cycles         ; count is to be stored
            store    0,0,gr97,gr120       ; store the MSBs,
            add      gr120,gr120,4        ; increment the address,
            store    0,0,gr96,gr120       ; then store the LSBs of cycles.
```

The above example program fragment calls the operating system service 274 to access the number of CPU cycles that have elapsed since it was powered on. The cycle count (in *gr96* and *gr97*) is stored in the two words addressed by the variable *cycles* if the service call is successful. If *gr121* contains a boolean FALSE value on exit, the program jumps to **cyc_err** to handle the error condition.

Service 289—setvec — Set User Trap Address

Description

This service sets the address for user-level trap handler services that implement the local register stack spill and fill traps. It returns an indication of success or failure in register *gr96*. The method used to invoke these traps in user mode is described on page 6 of this specification, in the "User-Mode Traps" section.

Register Usage

Type	Regs	Contents	Description
Calling:	gr121	289 (0x121)	Service number
	lr2	trapno	trap number
	lr3	funaddr	address of user trap handler
Returns:	gr96	retval	Success: = 0 Failure: < 0
	gr121	0x80000000 errcode	Logical TRUE, service successful Error number, service not successful (implementation dependent)

Example Call

```
trpadr:     .word    0
            const    lr2,64          ; trap number = 64
            const    lr3,t64_hnd     ; set address of
            consth   lr3,t64_hnd     ; trap-64 handler
            const    gr121,289       ; service = 289
            asneq    69,gr1,gr1      ; call the OS
            jmpf     gr121,vec_err   ; jump if error
            const    gr120,trpadr    ; set address where to
            consth   gr120,trpadr    ; store the trap address
            store    0,0,gr96,gr120  ; and store it!
```

The above example calls the **setvec** service to pass the address to be used for the trap 64 trap handler routine. If the service returns with *gr121* containing a boolean TRUE result, the program continues by storing the trap address returned in *gr96*; otherwise, the program jumps to **vec_err** to handle the error condition.

PROCESS ENVIRONMENT

There are standard memory and register initializations that must be performed by a HIF-conforming kernel before entry to a user program. In C-language programs, this is usually performed by the module **crt0**. This module receives control when an application program is invoked, and executes prior to invocation of the user's main function. Other high-level languages have similar modules.

STARTUP INITIALIZATION

Initialization procedures must establish appropriate values for the general registers mentioned below. In addition, file descriptors for the standard input and output devices must be opened.

Register Stack Pointer (gr1)

The register stack pointer (*RSP*) register contains the main memory address in which the local register *lr0* will be saved, and from which it will be restored. The content of *RSP* is compared to the content of *RAB* to determine when it is necessary to spill part of the local register stack to memory. On startup, the values in *RAB, RSP* and *RFB* should be initialized to prevent a spill trap from occurring on entry to the **crt0** code, as shown by the following relation:

(*RAB* + 256) *RSP RFB*

This provides the **crt0** code with at least 64 registers on entry, which should be a sufficient number to accomplish its purpose.

Memory Stack Pointer (gr125)

The memory stack pointer (*MSP*) register points to the top of the memory stack, or the lowest-addressed entry on the memory stack. This register must be preserved (or, more conventionally, restored).

Register Allocate Bound (gr126)

The register allocate bound (*RAB*) register contains the register stack address of the lowest-addressed word contained within the register file. *RAB* is referenced in the prolog of most user program functions to determine whether a register spill operation is necessary to accommodate the local register requirements of the called function.

Register Free Bound (gr127)

The register free bound (*RFB*) register contains the register stack address of the lowest-addressed word not contained within the register file (and greater than *RAB*). *RFB* is referenced in the epilog of most user program functions to determine whether a register fill operation is necessary to restore previously spilled registers needed by the function's caller.

Open File Descriptors

File descriptor 0 (corresponding to the standard input device) must be opened for text mode input. File descriptors 1 and 2 (corresponding to standard output and standard error devices) must be opened for text mode output prior to entry to the user's program.

PROGRAM TERMINATION

The only valid way for an application to terminate execution is by calling the **exit** service. Most high-level languages provide this capability, even if the programmer does not explicitly invoke a corresponding library function.

TRAP HANDLERS

The trap vector entries shown in Table 5 must be installed, and corresponding handlers must be provided.

Table 5. Trap Handler Vectors

Trap	Description
32	MULTIPLY
33	DIVIDE
34	MULTIPLU
35	DIVIDU
36	CONVERT
42	FEQ
43	DEQ
44	FGT
45	DGT
46	FGE
47	DGE
48	FADD
49	DADD
50	FSUB
51	DSUB
52	FMUL
53	DMUL
54	FDIV
55	DDIV
64	Spill (Set up by the user's task through a setvec call)
65	Fill (Set up by the user's task through a setvec call)
69	HIF System Call

Note: The Spill (64) and Fill (65) traps are returned to the user's code to perform the trap handling functions in user mode, as described in the "User Mode Traps" section.

APPENDIX A: HIF QUICK REFERENCE

Table 6 lists the HIF service calls, calling parameters, and the returned values. If a column entry is blank, it means the register is not used or is undefined. Table 7 describes the parameters given in Table 6.

Table 6. HIF Service Calls

Service	Calling Parameters				Returned Values		
Title	GR121	LR2	LR3	LR4	GR96	GR97	GR121
exit	1	exitcode					
open	17	pathname	mode	pflag	fileno		errcode
close	18	fileno			retval		errcode
read	19	fileno	buffptr	nbytes	count		errcode
write	20	fileno	buffptr	nbytes	count		errcode
lseek	21	fileno	offset	orig	where		errcode
remove	22	pathname			retval		errcode
rename	23	oldfile	newfile		retval		errcode
tmpnam	33	addrptr			filename		errcode
time	49				secs		errcode
getenv	65	name			addrptr		errcode
sysalloc	257	nbytes			addrptr		errcode
sysfree	258	addrptr	nbytes		retval		errcode
getpsize	259				pagesize		errcode
getargs	260				baseaddr		errcode
clock	273				msecs		errcode
cycles	274				LSBs cycles	MSBs cycles	errcode
setvec	289	trapno	funaddr		retval		errcode

Table 7. Service Call Parameters

Parameter	Description
addrptr	A pointer to an allocated memory area, a command-line-argument array, a pathname buffer, or a NULL-terminated environment variable name string.
baseaddr	The base address of the command-line-argument vector.
buffptr	A pointer to the buffer area where data is to be read from or written to during the execution of I/O services.
count	The number of bytes actually read from file or written to a file.
cycles	The number of processor cycles (returned value).
errcode	The error code returned by the service. These are usually the same as the codes returned in the UNIX *errno* variable. See Appendix B, Table 8, for a list of HIF error codes.
exitcode	The exit code of the application program.
filename	A pointer to a NULL-terminated ASCII string that contains the directory path of a temporary filename.
fileno	The file descriptor which is a small integer number. File descriptors 0, 1, and 2 are guaranteed to exist and correspond to open files on program entry (0 refers to the UNIX equivalent of **stdin** and is opened for input; 1 refers to the UNIX **stdout**, and is opened for output; 2 refers to the UNIX **stderr**, and is opened for output).
funaddr	A pointer to the address of a function.
mode	A series of option flags whose values represent the operation to be performed.
msecs	Milliseconds.
name	A pointer to a NULL-terminated ASCII string that contains an environment variable name.
nbytes	The number of data bytes requested to be read from or written to a file, or the number of bytes to allocate from the heap.
newfile	A pointer to a NULL-terminated ASCII string that contains the directory path of a new filename.
offset	The number of bytes from a specified position (*orig*) in a file.
oldfile	A pointer to NULL-terminated ASCII string that contains the directory path of the old filename.
orig	A value of 0, 1, or 2 that refers to the beginning, the current position, or the position of the end of a file.
pagesize	The memory page size in bytes (returned val).
pathname	A pointer to a NULL-terminated ASCII string that contains the directory path of a filename.
pflag	The UNIX file access permission codes.
retval	The return value that indicates success or failure.
secs	The seconds count returned.
trapno	The trap number.
where	The current position in a specified file.

APPENDIX B: ERROR NUMBERS

HIF implementations are required to return error codes when a requested operation is not possible. The codes from 0 to 255 are reserved for compatibility with current and future error return standards. The currently assigned codes and their meanings are shown in Table 8. If a HIF implementation returns an error code in the range of 0 to 255, it must carry the identical meaning to the corresponding error code in this table. Error code values larger than 255 are available for implementation-specific errors.

Table 8. HIF Error Numbers Assigned

Number	Error Name	Description
0		Not used.
1	EPERM	Not owner This error indicates an attempt to modify a file in some way forbidden except to its owner.
2	ENOENT	No such file or directory This error occurs when a file name is specified and the file should exist but does not, or when one of the directories in a pathname does not exist.
3	ESRCH	No such process The process or process group whose number was given does not exist, or any such process is already dead.
4	EINTR	Interrupted system call This error indicates that an asynchronous signal (such as interrupt or quit) that the user has elected to catch occurred during a system call.
5	EIO	I/O error Some physical I/O error occurred during a read or write. This error may in some cases occur on a call following the one to which it actually applies.
6	ENXIO	No such device or address I/O on a special file refers to a sub-device that does not exist or is beyond the limits of the device.
7	E2BIG	Arg list is too long An argument list longer than 5120 bytes is presented to execve.
8	ENOEXEC	Exec format error A request is made to execute a file that, although it has the appropriate permissions, does not start with a valid magic number.
9	EBADF	Bad file number Either a file descriptor refers to no open file, or a read (write) request is made to a file that is open only for writing (reading).
10	ECHILD	No children Wait and the process has no living or unwaited-for children.
11	EAGAIN	No more processes In a fork, the system's process table is full, or the user is not allowed to create any more processes.
12	ENOMEM	Not enough memory During an execve or break, a program asks for more memory than the system is able to supply or else a process size limit would be exceeded.

Table 8. HIF Error Numbers Assigned (continued)

Number	Error Name	Description
13	EACCESS	Permission denied An attempt was made to access a file in a way forbidden by the protection system.
14	EFAULT	Bad address The system encountered a hardware fault in attempting to access the arguments of a system call.
15	ENOTBLK	Block device required A plain file was mentioned where a block device was required, such as in mount.
16	EBUSY	Device busy An attempt was made to mount a device that was already mounted, or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, or active text segment).
17	EEXIST	File exists An existing file was mentioned in an inappropriate context, e.g., link.
18	EXDEV	Cross-device link A hard link to a file on another device was attempted.
19	ENODEV	No such device An attempt was made to apply an inappropriate system call to a device, e.g., to read a write-only device, or the device is not configured by the system.
20	ENOTDIR	Not a directory A non-directory was specified where a directory is required, for example, in a path name or as an argument to *chdir.*
21	EISDIR	Is a directory An attempt to write on a directory.
22	EINVAL	Invalid argument This error occurs when some invalid argument for the call is specified. For example, dismounting a non-mounted device, mentioning an unknown signal in signal, or specifying some other argument that is inappropriate for the call.
23	ENFILE	File table overflow The system's table of open files is full, and temporarily no more open requests can be accepted.
24	EMFILE	Too many open files The configuration limit on the number of simultaneously open files has been exceeded.
25	ENOTTY	Not a typewriter The file mentioned in **stty** or **gtty** is not a terminal or one of the other devices to which these calls apply.
26	ETXTBSY	Text file busy The referenced text file is busy and the current request can not be honored.
27	EFBIG	File too large The size of a file exceeded the maximum limit.

Table 8. HIF Error Numbers Assigned (continued)

Number	Error Name	Description
28	ENOSPC	No space left on device A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks are available on the file system.
29	ESPIPE	Illegal seek A seek was issued to a socket or pipe. This error may also be issued for other non-seekable devices.
30	EROFS	Read-only file system An attempt to modify a file or directory was made on a device mounted read-only.
31	EMLINK	Too many links An attempt was made to establish a new link to the requested file and the limit of simultaneous links has been exceeded.
32	EPIPE	Broken pipe A write on a pipe or socket was attempted for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is caught or ignored.
33	EDOM	Argument too large The argument of a function in the math package is out of the domain of the function.
34	ERANGE	Result too large The value of a function in the math package is unrepresentable within machine precision.
35	EWOULDBLOCK	Operation would block An operation that would cause a process to block was attempted on an object in non-blocking mode.
36	EINPROGRESS	Operation now in progress An operation that takes a long time to complete was attempted on a non-blocking object.
37	EALREADY	Operation already in progress An operation was attempted on a non-blocking object that already had an operation in progress.
38	ENOTSOCK	Socket-operation on non-socket A socket-oriented operation was attempted on a non-socket device.
39	EDESTADDRREQ	Destination address required A required address was omitted from an operation on a socket.
40	EMSGSIZE	Message too long A message sent on a socket was larger than the internal message buffer or some other network limit.
41	EPROTOTYPE	Protocol wrong type for socket A protocol was specified that does not support the semantics of the socket type requested.

Table 8. HIF Error Numbers Assigned (continued)

Number	Error Name	Description
42	ENOPROTOOPT	Option not supported by protocol A bad option or level was specified when accessing socket options.
43	EPROTONOSUPPORT	Protocol not supported The protocol has not been configured into the system, or no implementation for it exists.
44	ESOCKTNOSUPPORT	Socket type not supported The support for the socket type has not been configured into the system, or no implementation for it exists.
45	EOPNOTSUPP	Operation not supported on socket For example, trying to accept a connection on a datagram socket.
46	EPFNOSUPPORT	Protocol family not supported The protocol family has not been configured into the system or no implementation for it exists.
47	EAFNOSUPPORT	Address family not supported by protocol family An address was used that is incompatible with the requested protocol.
48	EADDRINUSE	Address already in use Only one usage of each address is normally permitted.
49	EADDRNOTAVAIL	Cannot assign requested address This normally results from an attempt to create a socket with an address not on this machine.
50	ENETDOWN	Network is down A socket operation encountered a dead network.
51	ENETUNREACH	Network is unreachable A socket operation was attempted to an unreachable network.
52	ENETRESET	Network dropped connection on reset The host you were connected to crashed and rebooted.
53	ECONNABORTED	Software caused connection abort A connection abort was caused internal to your host machine.
54	ECONNRESET	Connection reset by peer A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot.
55	ENOBUFS	No buffer space available An operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.
56	EISCONN	Socket is already connected A connect request was made on an already connected socket; or a *sendto* or *sendmsg* request on a connected socket specified a destination when already connected.

Table 8. HIF Error Numbers Assigned (continued)

Number	Error Name	Description
57	ENOTCONN	Socket is not connected A request to send or receive data was disallowed because the socket was not connected and (when sending on a datagram socket) no address was supplied.
58	ESHUTDOWN	Cannot send after socket shutdown A request to send data was disallowed because the socket had already been shut down with a previous shutdown call.
59	ETOOMANYREFS	Too many references; cannot splice.
60	ETIMEDOUT	Connection timed out A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)
61	ECONNREFUSED	Connection refused No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.
62	ELOOP	Too many levels of symbolic links A pathname lookup involved more than the maximum limit of symbolic links.
63	ENAMETOOLONG	File name too long A component of a pathname exceeded the maximum name length, or an entire pathname exceeded the maximum path length.
64	EHOSTDOWN	Host is down A socket operation failed because the destination host was down.
65	EHOSTUNREACH	Host is unreachable A socket operation was attempted to an unreachable host.
66	ENOTEMPTY	Directory not empty A non-empty directory was supplied to a *remove* directory or **rename** call.
67	EPROCLIM	Too many processes The limit of the total number of processes has been reached. No new processes can be created.
68	EUSERS	Too many users The limit of the total number of users has been reached. No new users may access the system.
69	EDQUOT	Disk quota exceeded A write to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted; or the allocation of an *inode* for a newly created file failed because the user's quota of *inodes* was exhausted.
70	EVDBAD	RVD related disk error

CHAPTER 4
General Information

CHAPTER 4
RELATED LITERATURE

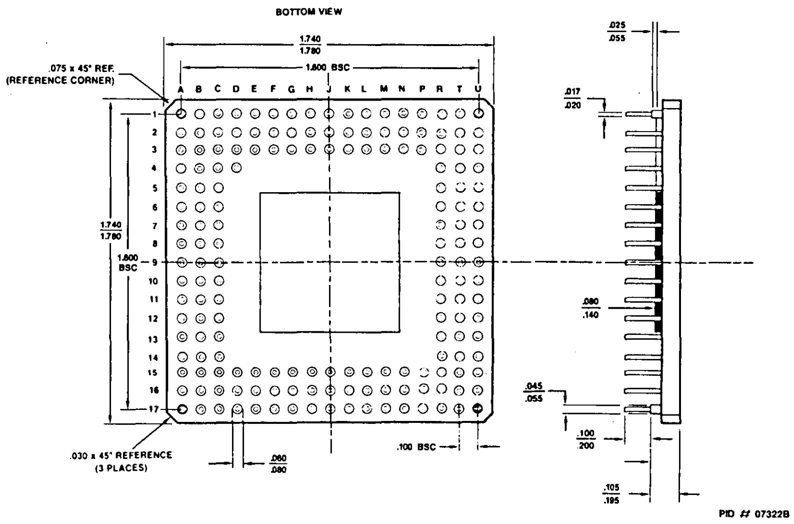
Additional Support Literature

The following is a list of AMD 29K Family literature that can be ordered from your local AMD Sales Representative or the Literature Distribution Center at (800) 222-9323, extension 5000; inside California, call (408) 749-5000. Technical and marketing information concerning the 29K Family also can be obtained by calling the 29K Hotline at (800) 2929-AMD.

Order No.	Title
09548	Am29000 Article Reprint Brochure
10344	Am29000 Family Overview Brochure
10345	29K Support Products Brochure
10620	Am29000 User's Manual
10621	Am29000 Performance Analysis Brochure
10623	Am29000 Memory Design Handbook
11426	Fusion 29K Catalog
11852	Am29027 Handbook

PACKAGE OUTLINES*

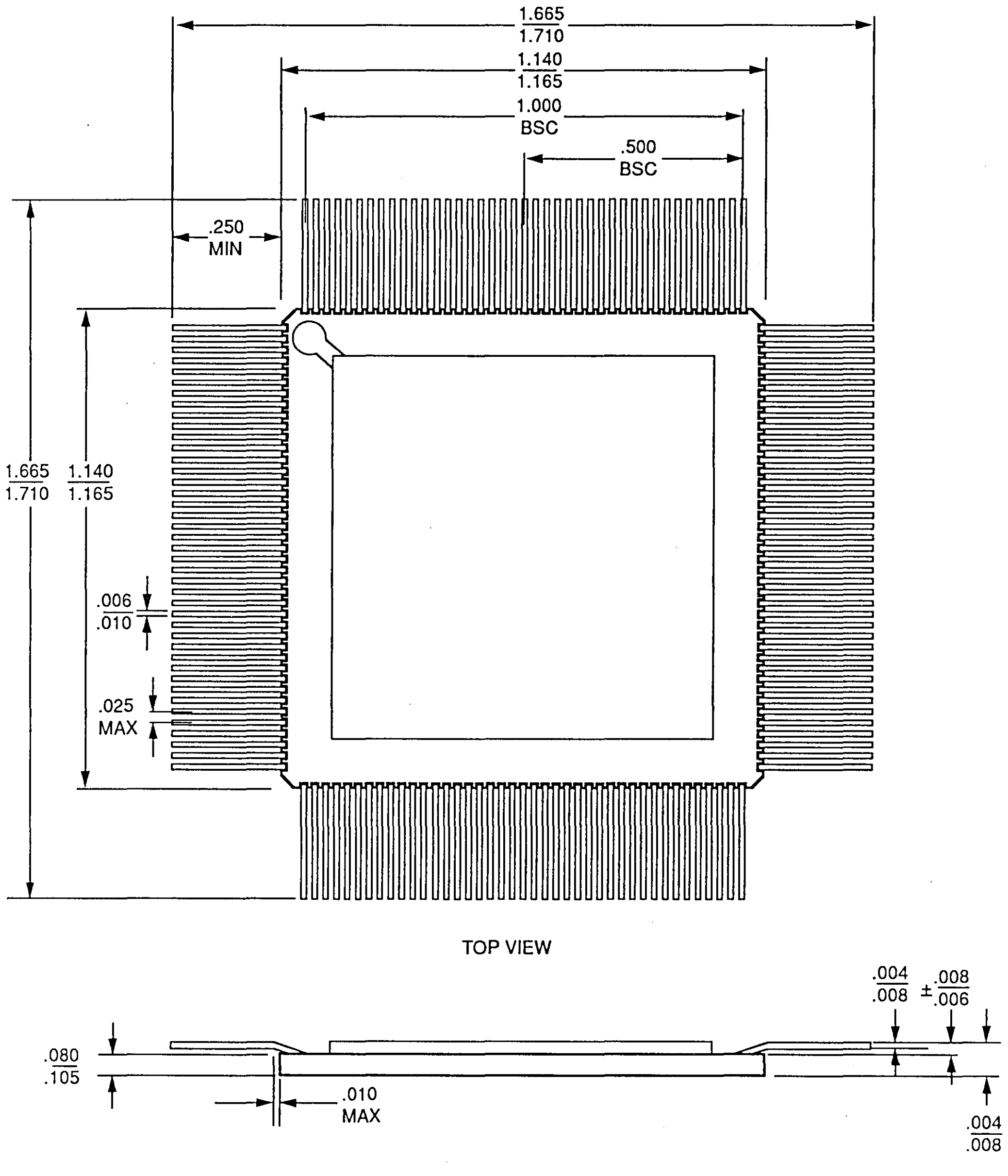
CGX169



*For reference only.

*For reference only. All dimensions are measured in inches. BSC is an ANSI standard for Basic Space Centering.

CQ164



1.665
1.710
1.140
1.165
1.000
BSC
.500
BSC
.250
MIN
1.665
1.710
1.140
1.165
.006
.010
.025
MAX
TOP VIEW
.004
.008
± .008
.006
.080
.105
.010
MAX
.004
.008
13092A

Notes

Notes

Sales Offices

North American

ALABAMA	(205) 882-9122
ARIZONA	(602) 242-4400
CALIFORNIA,	
Culver City	(213) 645-1524
Newport Beach	(714) 752-6262
Roseville	(916) 786-6700
San Diego	(619) 560-7030
San Jose	(408) 452-0500
Woodland Hills	(818) 992-4155
CANADA, Ontario,	
Kanata	(613) 592-0060
Willowdale	(416) 224-5193
COLORADO	(303) 741-2900
CONNECTICUT	(203) 264-7800
FLORIDA,	
Clearwater	(813) 530-9971
Ft. Lauderdale	(305) 776-2001
Orlando (Casselberry)	(407) 830-8100
GEORGIA	(404) 449-7920
ILLINOIS,	
Chicago (Itasca)	(312) 773-4422
Naperville	(312) 505-9517
KANSAS	(913) 451-3115
MARYLAND	(301) 796-9310
MASSACHUSETTS	(617) 273-3970
MICHIGAN	(313) 347-1522
MINNESOTA	(612) 938-0001
NEW JERSEY,	
Cherry Hill	(609) 662-2900
Parsippany	(201) 299-0002
NEW YORK,	
Liverpool	(315) 457-5400
Poughkeepsie	(914) 471-8180
Rochester	(716) 272-9020
NORTH CAROLINA	(919) 878-8111
OHIO,	
Columbus (Westerville)	(614) 891-6455
Dayton	(513) 439-0470
OREGON	(503) 245-0080
PENNSYLVANIA	(215) 398-8006
SOUTH CAROLINA	(803) 772-6760
TEXAS,	
Austin	(512) 346-7830
Dallas	(214) 934-9099
Houston	(713) 785-9001

International

BELGIUM, Bruxelles	TEL	(02) 771-91-42
	FAX	(02) 762-37-12
	TLX	846-61028
FRANCE, Paris	TEL	(1) 49-75-10-10
	FAX	(1) 49-75-10-13
	TLX	263282F
WEST GERMANY,		
Hannover area	TEL	(0511) 736085
	FAX	(0511) 721254
	TLX	922850
München	TEL	(089) 4114-0
	FAX	(089) 406490
	TLX	523883
Stuttgart	TEL	(0711) 62 33 77
	FAX	(0711) 625187
	TLX	721882
HONG KONG,	TEL	852-5-8654525
Wanchai	FAX	852-5-8654335
	TLX	67955AMDAPHX
ITALY, Milan	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	843-315286
JAPAN,		
Kanagawa	TEL	462-47-2911
	FAX	462-47-1729
Tokyo	TEL	(03) 345-8241
	FAX	(03) 342-5196
	TLX	J24064AMDTKOJ
Osaka	TEL	06-243-3250
	FAX	06-243-3253

International *(Continued)*

KOREA, Seoul	TEL	822-784-0030
	FAX	822-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY, Hovik	TEL	(03) 010156
	FAX	(02) 591959
	TLX	79079HBCN
SINGAPORE	TEL	65-3481188
	FAX	65-3480161
	TLX	55650 AMDMMI
SWEDEN,		
Stockholm	TEL	(08) 733 03 50
(Sundbyberg)	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TEL	886-2-7213393
	FAX	886-2-7723422
	TLX	886-2-7122066
UNITED KINGDOM,		
Manchester area	TEL	(0925) 828008
(Warrington)	FAX	(0925) 827693
	TLX	851-628524
London area	TEL	(0483) 740440
(Woking)	FAX	(0483) 756196
	TLX	851-859103

North American Representatives

CANADA	
Burnaby, B.C.	
DAVETEK MARKETING	(604) 430-3680
Calgary, Alberta	
DAVETEK MARKETING	(403) 291-4984
Kanata, Ontario	
VITEL ELECTRONICS	(613) 592-0060
Mississauga, Ontario	
VITEL ELECTRONICS	(416) 676-9720
Lachine, Quebec	
VITEL ELECTRONICS	(514) 636-5951
IDAHO	
INTERMOUNTAIN TECH MKTG, INC	(208) 888-6071
ILLINOIS	
HEARTLAND TECH MKTG, INC	(312) 577-9222
INDIANA	
Huntington - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
Indianapolis - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
IOWA	
LORENZ SALES	(319) 377-4666
KANSAS	
Merriam – LORENZ SALES	(913) 384-6556
Wichita – LORENZ SALES	(316) 721-0500
KENTUCKY	
ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3452
MICHIGAN	
Birmingham - MIKE RAICK ASSOCIATES	(313) 644-5040
Holland – COM-TEK SALES, INC	(616) 399-7273
Novi – COM-TEK SALES, INC	(313) 344-1409
MISSOURI	
LORENZ SALES	(314) 997-4558
NEBRASKA	
LORENZ SALES	(402) 475-4660
NEW MEXICO	
THORSON DESERT STATES	(505) 293-8555
NEW YORK	
East Syracuse – NYCOM, INC	(315) 437-8343
Woodbury – COMPONENT CONSULTANTS, INC	(516) 364-8020
OHIO	
Centerville – DOLFUSS ROOT & CO	(513) 433-6776
Columbus – DOLFUSS ROOT & CO	(614) 885-4844
Strongsville – DOLFUSS ROOT & CO	(216) 238-0300
PENNSYLVANIA	
DOLFUSS ROOT & CO	(412) 221-4420
PUERTO RICO	
COMP REP ASSOC, INC	(809) 746-6550
UTAH, R^2 MARKETING	(801) 595-0631
WASHINGTON	
ELECTRA TECHNICAL SALES	(206) 821-7442
WISCONSIN	
HEARTLAND TECH MKTG, INC	(414) 792-0920

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE TOLL FREE: (800) 222-9323 • (408) 749-5703

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale, California 94088-3453
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL-FREE
(800) 538-8450

APPLICATIONS HOTLINE
(800) 222-9323
(408) 749-5703

SUPPORT PRODUCTS ENGINEERING HOTLINE
USA (800) 2929-AMD
UK 0-800-89-1131
JAPAN 0-031-11-1129

Printed in USA
12175A