WEITEK

XL-8237
32-BIT RASTER IMAGE PROCESSOR

PRELIMINARY DATA
October 1988

The WEITEK XL-8237 is a fully-integrated CMOS 32-bit raster image processor. It is used with the WEITEK XL-8236 22-bit raster code sequencer to make the HyperScript-Processor, a high-performance graphics CPU capable of driving raster printers at up to 60 pages per minute. WEITEK's single-precision floating-point unit may also be used to produce a tightly-coupled raster image printing system.

Contents

The masters for this document were printed on an
XL-8200 development system

XL-8237 Raster Image Processor Data Sheet
October, 1988

WEITEK Corporation
1060 East Arques Avenue
Sunnyvale, California 94086
Telephone (408) 738-8400

WEITEK reserves the right to make changes to these specifications
at any time

Printed in the United States of America
90 89 88 6 5 4 3 2 1

XL-8237
32-BIT RASTER
IMAGE PROCESSOR

PRELIMINARY DATA
October 1988

Features

32-BIT, SINGLE-CHIP GRAPHICS PROCESSOR

32-bit integer ALU
Four-port 36×32 register file
Parallel multiply/divide unit for Bezier computation
32-bit shift/field merge unit for BitBlt
Single-cycle execution

HIGH PERFORMANCE

10 to 60 pages per minute running WEITEK's HyperScript interpreter
Peak BitBlt rate of over 65 million pixels per second
Bezier computations at 750 thousand endpoints per second

LOW SYSTEM COST

145-pin plastic PGA (pin grid array) package
Low power CMOS with TTL-compatible I/O

POWERFUL INSTRUCTION SET

Add, subtract, multiply, and divide
Complete set of logical operations
Shifts up to 31 bits in one cycle
Priority encode
Field extract/deposit/merge instructions
Perfect exchange (including bit reverse)

POWERFUL DEVELOPMENT TOOLS

PostScript-compatible interpreter
C compiler
Graphics development system

INTERFACES WITH OTHER XL-8200 PRODUCTS

Interfaces with the XL-8236 raster control sequencer
Interfaces with XL-8232 graphics floating point unit

Description

The XL-8237 is a RISC-architecture 32-bit raster image processor (RIP). It is used with the XL-8236 32-bit raster code sequencer (RCS) to form the XL-8200 HyperScript-Processor, a high-performance graphics processor that can run WEITEK's HyperScript interpreter and other page description languages. These chips also interface directly with WEITEK's 32-bit graphics floating point unit, the XL-3232.

The XL-8237 was designed specifically as a laser beam printer controller running a page description language. WEITEK supplies the HyperScript interpreter, a PostScript-compatible interpreter for its HyperScript-Processors. The architecture supports speeds from 10 to 60 pages per minute; thus it is a powerful and cost effective solution for a wide range of speeds, resolutions, colors, and page description languages.

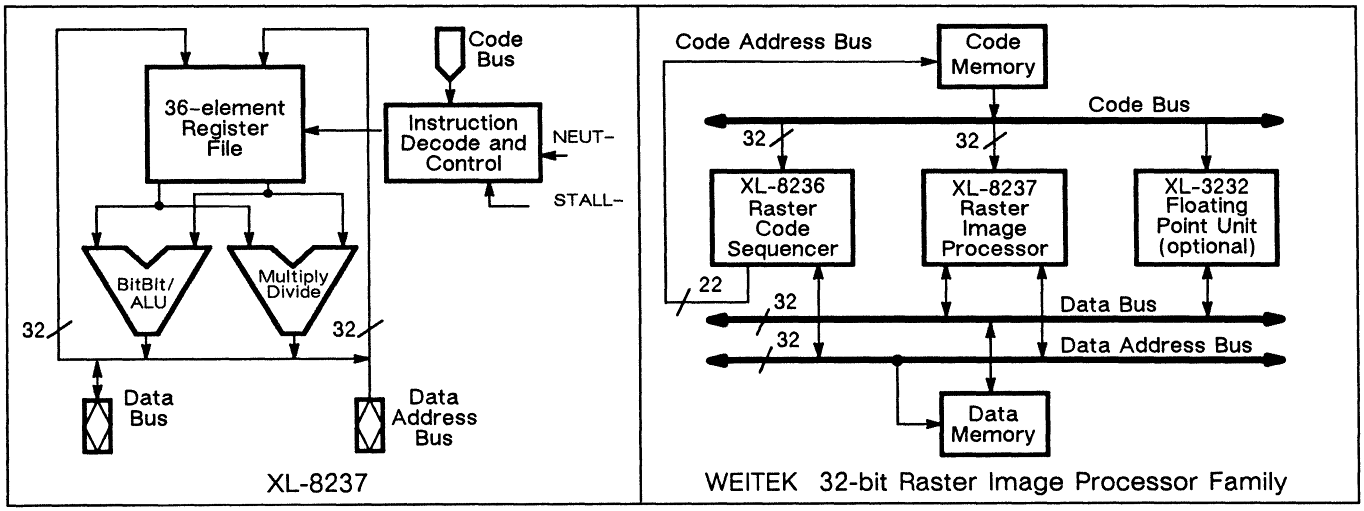


Figure 1. Simplified block diagrams

Description, continued

SPEED

6000 sans serif 10-point characters per second font placement rate

750 sans serif 10-point characters per second font-generation rate using URW's NIMBUS font-scaling from Bezier outlines

75 sans-serif 10-point characters per second font-generation rate using BITSTREAM FontWare font-scaling from Bezier outlines

Architecture

ALU

The heart of the XL-8237 RIP is the 32-bit ALU, which contains the hardware for arithmetic and logical functions. The ALU performs 32-bit addition and subtraction, sixteen different logical functions, and address generation. All ALU operations are performed in a single cycle.

BITBLT/SHIFTER/FIELD MERGE UNIT

The shift/merge unit provides a rich set of instructions for key raster image processing applications such as bit block transfer (BitBlt) and character placement.

The shifter unit provides 0–31 bits of shifting in either direction in the deposit, extract, and merge operations. This allows the RIP to extract bit fields of any length, operate on them, and replace them in the original word. Bit fields can also be combined through the bitwise merge instruction.

The perfect exchange function is used to rearrange the bits within a single word in a variety of ways. It can be used to swap fields of 2, 4, 8, or 16 bits, reverse the bit order within these fields, or both.

The priority encode function counts the number of zero bits before the first one bit is encountered.

MULTIPLY/DIVIDE UNIT

The multiply/divide unit make the RIP very effective in mathematically-intensive operations such as character generation and scaling from Bezier outlines.

Hardware multiply and divide functions give a 64-bit product of two 32-bit operands in 8 cycles, and a 32-bit quotient and 32-bit remainder from a mixed 64/32 bit division in 20 cycles. The multiply/divide unit operates independently of the rest of the ALU, so other operations can be performed in parallel with multiplication and division. The integer multiply/divide unit can emulate floating point operations in software at a 25-cycle rate (0.3 MFLOP at 120 ns).

Architecture, continued

REGISTER FILE

The four-port register file contains 36 registers, each 32 bits wide. This large register file allows frequently-used variables to be kept on-chip, reducing the number of memory accesses and increasing performance.

Registers 28–31 are duplicated in a second bank to give four temporary registers which can be used during interrupt handling.

BUS STRUCTURE

There are three independent 32-bit buses: Code, Data, and Data Address (C, D, and AD buses, respectively).

Independent code and data buses allow data-intensive operations such as BitBlt and character placement to run continuously, without being interrupted by code fetches.

The Code Bus provides the RIP with its instruction stream. When used with the XL-8236 raster code sequencer (RCS), both chips share the same 32-bit instruction stream. Many instructions also use the code word to provide immediate data fields.

The Data Bus provides bidirectional access to external memory, at the rate of one load per cycle or one store every two cycles. The Data Bus has individual write-select lines (WREN lines) for each byte in the word.

The Data Address bus is used to provide memory addresses and to transfer data to and from the RCS and I/O devices. A word can be transferred over the AD Bus every cycle.

MEMORY ACCESS

Loading and storing from memory is done with the address generation instructions and Load and Store Data instructions. The RIP uses a delayed load/delayed store scheme which overlaps memory access with other RIP operations in a straightforward way.

Memory access includes load and store instructions with features such as indexed addressing and pre- and post-increment addressing. The basic memory word is 32 bits wide, but bytes and halfwords can be accessed individually. Load and store operations take two instructions, but are pipelined to allow other operations to occur in parallel with memory access.

INSTRUCTION FORMAT

The RIP's instruction set is based on register-to-register operations specified in a 32-bit instruction word. The basic instruction format has three 5-bit register select fields, opcode and extended opcode fields, and a condition code select field. Thus a three-address instruction of the form $rc := ra + rb$ can be specified in a single word.

In many instructions, one of the operands can be replaced by an immediate value, allowing operations on constants to be specified in a single instruction without first loading the constant into a register.

Most instructions reserve the most-significant eight bits of the instruction word for an RCS opcode. When the RIP is used with the RCS, the two chips share the same 32-bit instruction stream.

Block Diagram

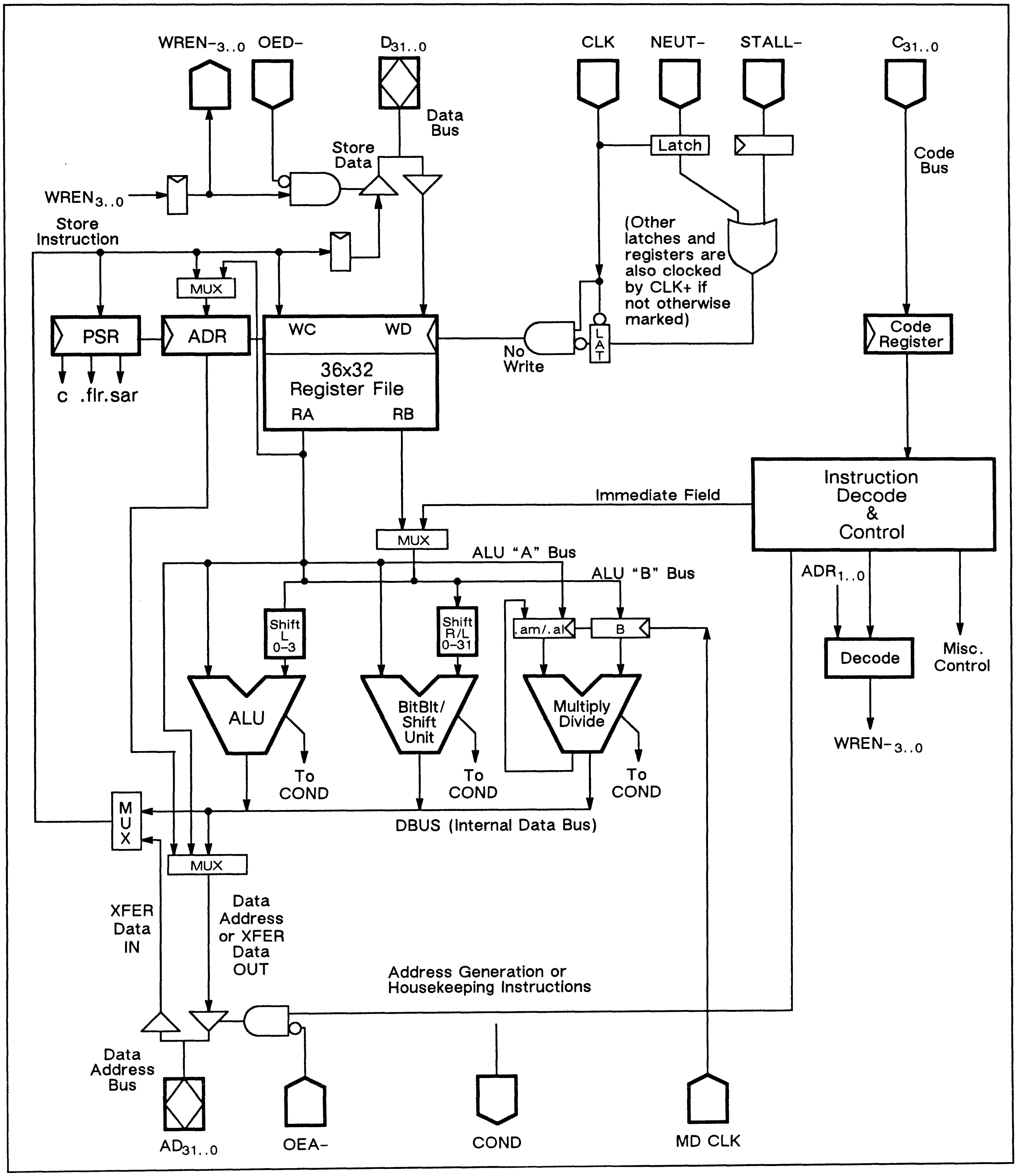


Figure 2. XL-8237 block diagram

Signal Description

C+

The $C_{31..0}$ Code Bus contains a 32-bit instruction word. Because it contains a built-in pipeline register, it is not necessary to use an external pipeline register between code memory and the XL-8237.

D+

The $D_{31..0}$ Data Bus is used as a bidirectional input/output bus. Data flow is in the form of memory-to-register and register-to-memory transfers. Tri-stating of the D Bus is controlled by the currently executing instruction and the OED− signal.

AD+

The $AD_{31..0}$ Data Address Bus provides addresses for data memory operations. It is driven with either the contents of the address register (.adr) or the result of an address computation instruction. Tri-stating of the AD Bus is controlled by the currently executing instruction and the OEA− signal. It can also be used as a bidirectional data bus for transfers from the RCS or other hardware.

COND+

The COND output is a single-bit condition code signal that indicates one of several possible one-bit status values derived from the result of the current instruction. See *Condition Code Generation* section on page 13.

CLK

The clock input, CLK, is a single-phase TTL-level clock signal. One instruction is executed per clock cycle. The CLK signal selects whether the current clock cycle is to be "phase one" (CLK high) or "phase two" (CLK low). Many of the external signals are synchronized to either the rising or falling edge of this signal.

MDCLK

The Multiply/Divide clock input, MDCLK, is a single-phase TTL-level clock signal. The internal multiply/divide registers are synchronized to the positive-going edge of this clock. This signal must be synchronized to the rising and falling edges of the CLK signal, and runs at twice the frequency of CLK.

NEUT−

The NEUT− input causes the currently executing instruction to be neutralized or canceled; that is, any internal effects that the instruction was to have (such as modification of register contents or status bits) are canceled.

STALL−

The STALL− input cancels the next instruction.

WREN−

The $WREN^{-}_{3..0}$ outputs indicate which bytes of the data word are to be stored to the data memory. This control information is driven when a Store Data instruction is executed by the RIP, otherwise these signals are high.

OEA−, OED−

OEA− and OED− are asynchronous output enable signals for the AD and D buses respectively. The buses drive when their respective output enables are low, and float when output enables are high.

If the OEA− signal is de-asserted, then the AD Bus is tri-stated regardless of the OEA signal or the executing instruction. If the OEA− signal is asserted, then the AD Bus is driven under control of the OEA+ signal or the currently executing instruction. Note that OEA− is *not* simply the complement of OEA+.

The OED− signal functions similarly. If the OED− signal is de-asserted, then the D Bus is tri-stated.

VCC, GND

The VCC and GND pins provide a supply voltage of +5.0 volts, and system ground of 0 volts, respectively. All VCC and GND pins must be connected.

TIE HIGH, TIE LOW

Signals marked "Tie High" should be tied to VCC. Signals marked "Tie Low" should be tied to GND. Future versions of the XL-8237 may redefine these as signal pins, so it's advisable to tie them through traces rather than directly to power and ground planes.

Memory Addressing

The XL-8237 provides address generation functions, including addressing of bytes, halfwords, or words in a word-wide memory. These functions determine byte and halfword positioning within a word from the least significant two bits of the memory address. This is illustrated in figure 3.

Halfword addresses ending with "11" and word addresses ending with "01," "10," or "11" are not defined for Load and Store operations—that is, data to be loaded cannot straddle a word boundary. Data that straddles a word boundary can be obtained using two loads and a merge.

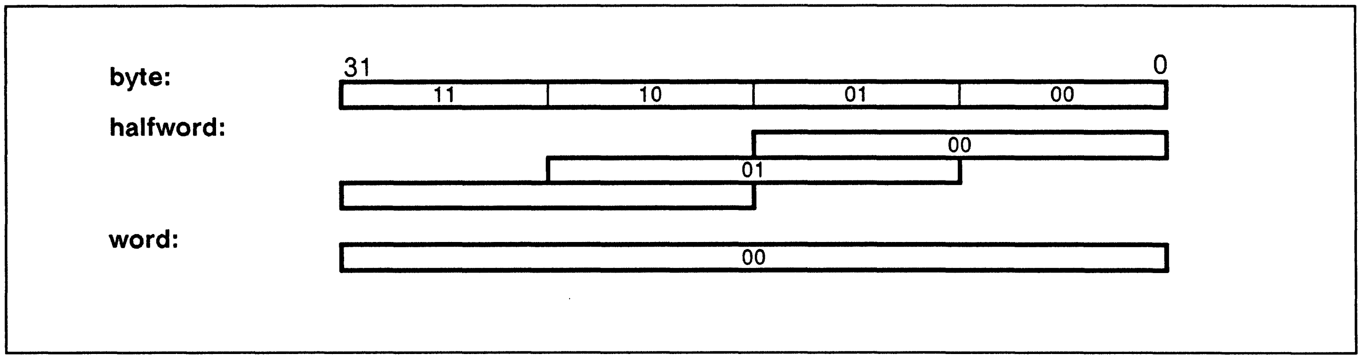


Figure 3. Memory addressing

Registers

REGISTER FILE

The register file contains 36 registers, each 32 bits wide, which are accessed through four independently addressable ports.

The 36 registers are numbered 0–31 and 28′–31′. (See figure 4.) Only registers 0–31 can be directly accessed through the five-bit register numbers contained in an instruction. A special instruction, swap (one of the housekeeping instructions), exchanges the contents of registers 28–31 and 28′–31′ in a single cycle. Normally the four extra registers are used only by interrupt routines for temporary working storage.

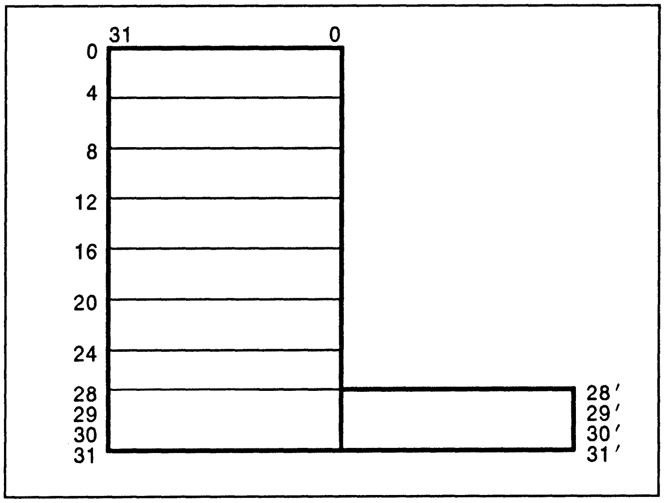


Figure 4. Data registers

ADDRESS HOLDING REGISTER

The XL-8237 retains the last address generated by any of the address generation instructions in the .adr register. The .adr register serves two purposes. It is used by the interrupt mechanism to aid in saving and restoring the state of the system. It is used by the byte alignment instructions to indicate the beginning byte offset. The format is given in figure 6.

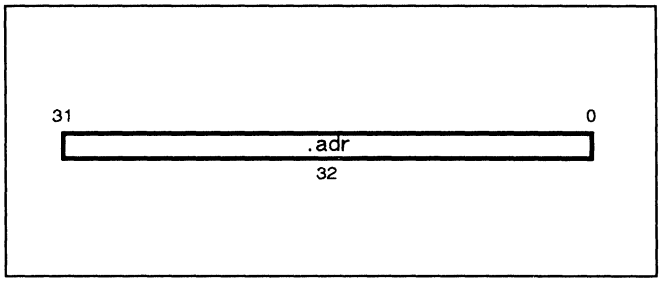


Figure 5. Address register

PRODUCT REGISTERS

There are two 32-bit product registers: .am and .al. They are used by the multiply and divide hardware and the bitwise merge instruction. During the operation of the multiply and divide hardware the contents of these registers are undefined. This implies that the bitwise merge instruction cannot be used during a multiply or divide operation. Several instructions in the housekeeping instruction set explicitly manipulate the contents of these registers. The format is given in figure 6.

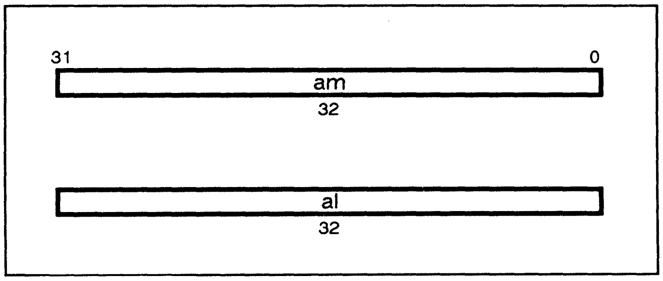


Figure 6. Product registers

Registers, continued

PROCESSOR STATUS REGISTER

The RIP retains some control information in the *processor status register,* .psr. The format is given in figures 7 and 8.

31					0
reserved	be	z	c	flr	sar
20	1	1	1	5	5

Figure 7. Processor status register

Symbol	Meaning
sar	shift amount register
flr	field length register
c	carry bit
z	register bank select (for registers 28-31 or 28'-31')
be	reserved for future extension. Must be set to zero
reserved	reserved for future extension. Must be set to zero.

Figure 8. Processor status register bit fields

Instruction Set

TERMS AND SYMBOLS

The instructions are listed on page 11, then described in detail on the following pages. Each description includes a pseudo-code definition of the instruction. The following symbols are used:

|| Concatenate fields. abc || def gives abcdef.

| Indicates that operations separated by this symbol occur in parallel.

a dup b Duplicate b a times. 3 dup 0 gives 000.

reg(ra) Register number ra

COND Condition Code

{ } Begin and end comment

≪ ixs Shift left by ixs bits

tcovf Two's complement overflow

usovf Unsigned overflow

unadd Unsigned add

unsub Unsigned subtract

result The result of any internal operation that is available on the *internal* DBUS (see Simplified Block Diagram, figure 2). Typically, result will be driven out on the AD Bus but can also be driven out on the D Bus.

[31..0] Specifies the bit field from bits 31 to bit 0, inclusive. For example, reg (ra) [3..0] gives the lower four bits of register ra.

a *op* b Perform an operation on operands a and b

Instruction Format

A 32-bit instruction word is used to control the operation of the XL-8237. This instruction word is designed to be directly shared by the XL-8236 RCS. Therefore, the two parts should be considered together.

Normally the instruction word is divided into two sections. The first, the most significant 8 bits, is used to control I/O operation of the RIP as well as perform many RCS operations. The second, the least significant 24 bits, is normally used to control the internal operations of the RIP. However, this second field can be used by certain, so-called "long" RCS operations and by inter-chip transfer instructions.

The following table gives the abbreviations used for bit fields. The instruction formats are given in figure 10 on the next page.

Field	Meaning
RCS	RCS control field
ra	register select
rb	register select
rc	register select
rd	register select
shf	controls amount of shift in Extract/Deposit/Merge
len	controls field length in Extract/Deposit/Merge
p	controls Perfect Exchange
imm16	16-bit signed immediate data field
imm11	11-bit signed immediate data field
imm10	10-bit signed immediate data field
imm5	5-bit signed immediate data field
cn	selects condition to be generated
e,ext,m	operation code extensions
ixs	shift specification
s	specifies signed or unsigned
siz	size of data item
processor operation	any 24-bit RIP instruction
x	Reserved for future definition*

* See section *Compatibility With Future Enhancements*

Figure 9. Instruction fields

Instruction Format, continued

	8	3	5	5	6	5	Detailed Description Page #
Arithmetic and Logical Instructions							
Arithmetic instructions	RCS	100	ra	rc	ext c	rb/imm5	13
Add Signed Immediate	RCS	101	ra	rc	1 immediate10		15
Logical instructions	RCS	101	ra	rc	0 ext c	rb	16
Field Manipulation Instructions							
Deposit/Deposit and Merge	RCS	000	ra	len	shf m	rb	19
Merge Immediate	RCS	001	ra	len	shf 1	imm5	20
Extract	RCS	001	ra	len	shf 0	rb	21
Dynamic Extract/Deposit/Merge	RCS	100	ra	len	111 ext	rb	22
Merge Halfword High	RCS	011	ra	immediate16			23
Bitwise Merge	RCS	111	ra	rc	1111 01	rb	24
Perfect Exchange	RCS	111	ra	p	1111 10	rb	25
Address Generation Instructions							
Load/Store Address with Signed Displacement	RCS	110	ra	immediate16			27
Load/Store Address with Index/Signed Displacement	RCS	111	ra	rc	1 ext ixs	rb/imm5	28
Load/Store and Alignment Instructions							
Load Halfword Immediate	RCS	010	ra	immediate16			31
Load Data to RIP	110 rd	RIP operation					32
Store Data from RIP	00001001	RIP operation (result is stored)					33
Byte Align for Load Data	RCS	111	ra	x 0 s siz	1111 00	rb	34
Byte Align and Store Data	RCS	111	ra	e 1 s siz	1111 00	rb	36
Miscellaneous							
Multiply/Divide/Priority Encode/Housekeeping	RCS	111	ra	ext	1111 11	rb/imm5	38
Coprocessor/Sequencer Operations							
Coprocessor Operation	RCS	111	x	x	0 x x	x	47
Store Data from Coprocessor	101 rd	RIP operation					48
Load Data to Coprocessor	111 rd	RIP operation					49
Transfer to/from RIP	00000000	ext	ra	x	x		50
Long RCS instruction	00000001	RCS					
Long RCS instruction	00000010	RCS					
Long RCS instruction	00000011	RCS					
Long RCS instruction	0001	RCS					

Figure 10. XL-8237 Instruction formats

Condition Code Generation

All instructions that could produce a meaningful condition code, generate one automatically. These are single-cycle instructions. Specific condition codes generated by each single-cycle instruction are summarized in the table below. The condition code generated by an instruction is available on the COND pin at the end of the cycle during which the instruction is executed. Instructions not listed in the table do not generate any condition. See descriptions of individual instructions for details.

Condition	Format		Function
cond	ra	:= ra deposit rb[shf,len]	deposit
cond	ra	:= ra extract rb[shf,len]	extract
cond	ra	:= ra deposit imm5[shf,len]	deposit imm5
cond	ra	:= imm16	load imm16
cond	ra	:= ra deposit imm16[16,16]	merge imm16
cond1	c,rc	:= rb/imm5 ± ra+c	arithmetic
cond	ra	:= ra deposit rb [.sar,.flr]	dynamic deposit
cond	ra	:= extract rb[.sar,.flr]	dynamic extract
cond2	ra	:= ra *op* rb	logical
cond3	rc	:= ra+imm10	add imm10
cond4	.adr	:= ra+imm16	load/store address generation
cond3	rc	:= ra+(rb<<ixs)	add with index
cond4	ra,.adr	:= ra+(rb/imm5<<ixs)	load/store indexed address generation
cond	mem[.adr]	:= rb align[signed siz]	byte align for store
cond	rc	:= (ra and not al) or (rb and al)	bitwise merge
cond	ra	:= p exchange rb	perfect exchange
cond	ra	:= priority encode rb	priority encode
cond	ra	:= am+rb+al[31]	retrieve multiply/divide result

Conditions:

cond	non-zero result
cond1	>0, ≥ 0, =0, overflow (unsigned or two's complement)
cond2	non-zero, or all bytes non-zero
cond3	two's complement overflow
cond4	unsigned overflow, or two's complement shift overflow

Figure 11. Generated condition codes

Arithmetic Instructions

RCS	100	ra	rc	ext	cn	rb/imm5
8	3	5	5	4	2	5

FORMAT

instruction	ext	meaning
c,rc := unsigned(ra + rb)	0000	unsigned add
c,rc := unsigned(rb − ra)	0001	unsigned subtract
c,rc := unsigned(ra + rb + c)	0010	unsigned add with carry
c,rc := unsigned(rb − ra − 1 + c)	0011	unsigned subtract with borrow
rc := ra + imm5	0100	two's complement add immediate
rc := imm5 − ra	0101	two's complement subtract immediate
rc := ra + rb	0110	two's complement add
rc := rb − ra	0111	two's complement subtract
c,rc := ra + rb	1000	two's complement add
c,rc := rb − ra	1001	two's complement subtract
c,rc := ra + rb + c	1010	two's complement add with carry
c,rc := rb − ra − 1 + c	1011	two's complement subtract with borrow
c,rc := ra + imm5	1100	two's complement add immediate
c,rc := imm5 − ra	1101	two's complement subtract immediate

DESCRIPTION

Arithmetic instructions include signed and unsigned add (with and without carry), and signed and unsigned subtract (with and without borrow).

Depending on the value of the *ext* field, the contents of register *ra* is either added or subtracted from either the contents of register *rb*, or a sign-extended immediate value. Four forms of this instruction also add in the *c* bit from the .psr, and ten forms update the *c* bit. The result is placed in register *rc*.

CONDITION

The condition generated depends on the value of the *ext* and *cn* fields of the instruction. For most operations, the condition generated assumes that the result is a two's complement value; however, for the unsigned add and subtract operations, the condition generated assumes that the result is an unsigned quantity. The unsigned and two's complement equal-to conditions and less-than conditions remain arithmetically valid for all valid input values, even if unsigned or two's complement overflow occurs as a result of the addition or subtraction operation.

cn	condition signal generated
00	two's complement/unsigned not equal to zero
01	two's complement/unsigned greater than or equal to zero
10	two's complement/unsigned overflow
11	two's complement/unsigned greater than zero

Arithmetic Instructions, continued

OPERATION

```
temp := 28 dup imm[4] || imm[3..0];
case ext of
      0000b: c || result := reg(ra) + reg(rb);
      0001b: c || result := (not reg(ra)) + reg(rb) + 1;
      0010b: c || result := reg(ra) + reg(rb) + c;
      0011b: c || result := (not reg(ra)) + reg(rb) + c;
      0100b:      result := reg(ra) + temp;
      0101b:      result := (not reg(ra)) + temp + 1;
      0110b:      result := reg(ra) + reg(rb);
      0111b:      result := (not reg(ra)) + reg(rb) + 1;
      1000b: c || result := reg(ra) + reg(rb);
      1001b: c || result := (not reg(ra)) + reg(rb) + 1;
      1010b: c || result := reg(ra) + reg(rb) + c;
      1011b: c || result := (not reg(ra)) + reg(rb) + c;
      1100b: c || result := reg(ra) + temp;
      1101b: c || result := (not reg(ra)) + temp + 1;
endcase;
reg(rc) := result;
if ext [3..2] ≠ 00b then
      tcovf:= c32 xor c31
      case c of
            00b: cond := (result ≠ 0) or tcovf; *
            01b: cond := result[31] xnor tcovf;
            10b: cond := tcovf;
            11b: cond := (result ≠ 0 or tcovf) and (result[31] xnor tcovf);
      endcase;
else
      usovf := unsub xor c32;
      case c of
            00b: cond := result ≠ 0 or usovf;
            01b: cond := unadd or c32;
            10b: cond := usovf;
            11b: cond := (result ≠ 0 or usovf) and (unadd or c32);
      endcase;
endif;
COND := cond;
```

* See *Overflow Detection.*

Arithmetic Instructions, continued

ADD SIGNED IMMEDIATE

RCS	101	ra	rc	1	imm10
8	3	5	5	1	10

FORMAT

rc := ra + imm10

DESCRIPTION

The 10-bit signed immediate value is added to the contents of register *ra*. The result is placed in register *rc*. This instruction does not affect the c bit of the .psr. The condition generated is TRUE (1) if a signed 32-bit overflow is encountered, otherwise the condition generated is FALSE (0).

OPERATION

```
result   := reg(ra) + (23 dup imm10[9]) || imm10[8..0];    {sign-extend and add}
reg(rc)  := result;
COND     := tcovf;
```

Logical Instructions

LOGICAL INSTRUCTIONS

RCS	101	ra	rc	0	ext	c	rb
8	3	5	5	1	4	1	5

FORMAT

instruction	ext	meaning
rc := zeros	1111	clear all bits
rc := ra and rb	1110	logical and
rc := ra and (not rb)	1101	logical and-not
rc := ra	1100	pass
rc := (not ra) and rb	1011	logical not-and
rc := rb	1010	pass
rc := ra xor rb	1001	logical xor
rc := ra or rb	1000	logical or
rc := (not ra) and (not rb)	0111	logical nor
rc := ra xnor rb	0110	logical xnor
rc := not rb	0101	logical not
rc := ra or (not rb)	0100	logical or-not
rc := not ra	0011	logical not
rc := (not ra) or rb	0010	logical not-or
rc := (not ra) or (not rb)	0001	logical nand
rc := ones	0000	set all bits

DESCRIPTION

The contents of register *ra* and the contents of register *rb* are combined in a logical or bitwise function. The function performed depends on the value of the *ext* field. The result is placed in register *rc*.

CONDITION

The condition generated depends on the value of the the c field. The condition "all bytes non-zero" permits quick scanning through byte data, using word operations.

Logical Instructions, continued

condition	c	condition signal generated
≠ 0	0	not equal to zero
all bytes of rc ≠ 0	1	all bytes not equal to zero

OPERATION

```
case ext of
      1111b: result := 32 dup 0;
      1110b: result := reg(ra) and reg(rb);
      1101b: result := reg(ra) and (not reg(rb));
      1100b: result := reg(ra);
      1011b: result := (not reg(ra)) and reg(rb);
      1010b: result := reg(rb);
      1001b: result := reg(ra) xor reg(rb);
      1000b: result := reg(ra) or reg(rb);
      0111b: result := (not reg(ra)) and (not reg(rb));
      0110b: result := reg(ra) xnor reg(rb);
      0101b: result := not reg(rb);
      0100b: result := reg(ra) or (not reg(rb));
      0011b: result := not reg(ra);
      0010b: result := (not reg(ra)) or reg(rb);
      0001b: result := (not reg(ra)) or (not reg(rb));
      0000b: result := 32 dup 1;
endcase;
case c of
      0b: cond := (result ≠ 0);
      1b: cond  :=  not((result[31..24]=0) or (result[23..16]=0) or (result[15..8]=0)  or (result[7..0]=0));
endcase;
reg(rc)  := result;
COND   := cond;
```

Field Manipulation Instructions

The Extract, Deposit, and Merge instructions are used to perform computations on portions of a word. Typically, the desired bit field is converted into a full word, using an Extract instruction, operated on and converted back into a bit field by the Deposit or Merge instructions. These instructions can also be used to perform simple left and right shifts as well as rotations. Figure 12 shows the operation of these instructions.

The Extract/Deposit/Merge instructions have two forms: static and dynamic. The static form specifies the field length and shift amount in the instruction as constants. The dynamic form uses the .flr and .sar fields from the .psr.

The Extract instruction converts a bit field within a register into a 32-bit value which is stored into another register. The extracted bit field is aligned to the least-significant bit of the destination register. The high order bits of the destination are filled with zeros or sign extended, controlled by the field length and shift amount. If the sum of the field length and shift amount is greater than 32, sign extension is performed; otherwise zero-fill is selected.

The Deposit and Merge instructions perform the inverse operation: a 32-bit register is inserted into a specified field of a destination register. For Deposit instructions, all other bits of the destination are set to zero. For Merge instructions, the other bits of the destination are not modified.

There is a special form of the Merge instruction: Merge Immediate, which uses a 5-bit signed constant instead of a register as the value to be inserted. This instruction allows convenient bit set and reset as well as many other useful operations.

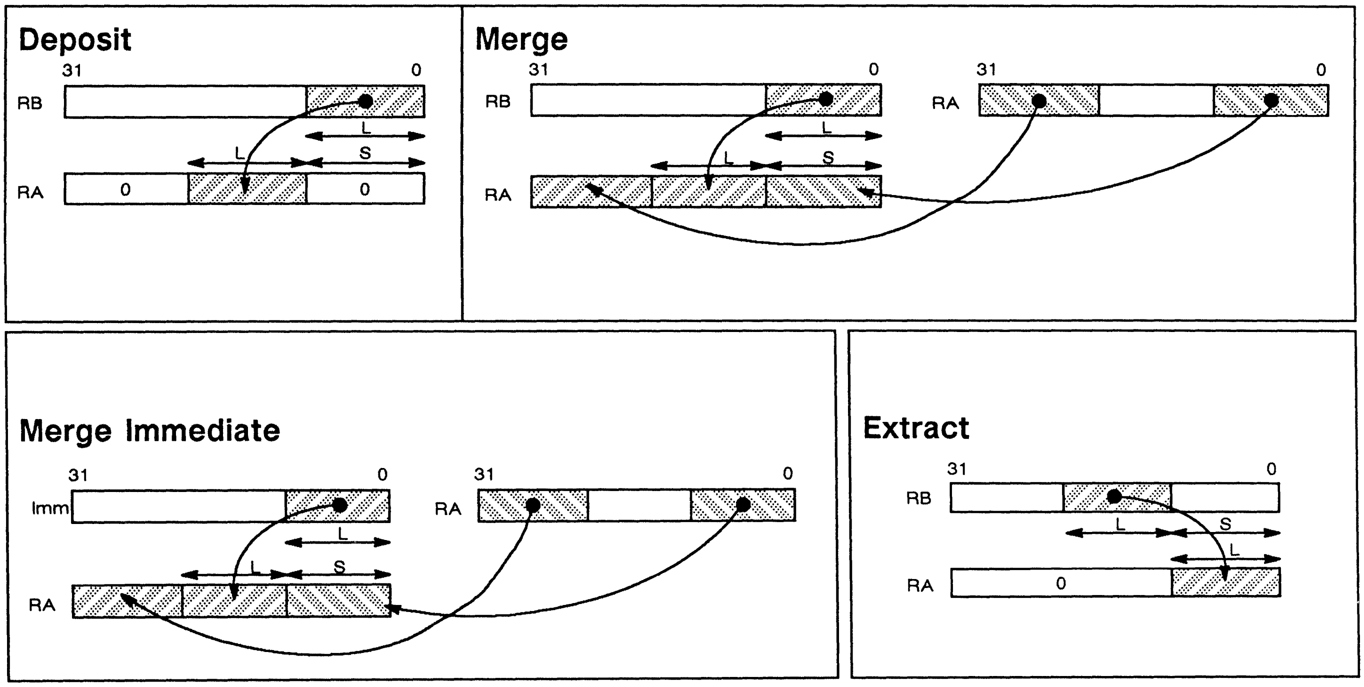


Figure 12. Deposit, extract, and merge instructions

Field Manipulation Instructions, continued

DEPOSIT/DEPOSIT AND MERGE

RCS	000	ra	len	shf	m	rb
8	3	5	5	5	1	5

FORMAT

ra := deposit rb [shf, len]
ra := ra deposit rb [shf, len]

DESCRIPTION

A right justified field of length specified by *len* is taken from the contents of register *rb*. The field is left-shifted by *shf* bits. If the sum of *shf* and *len* is greater than 32, the field is truncated. If the *m* bit is zero, the result is the field, otherwise the field is merged with the contents of register *ra*. The result is placed in register *ra*.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-zero and is FALSE (0) if the result is zero.

OPERATION

```
if len > 0 then
        l := len;
else
        l := 32;
endif;
f := l + shf;
if f > 32 then
        f := 32;
        l := 32 - shf;
endif;
if m = 0 then
        result := (32-f dup 0) || reg(rb)[l-1..0] || (shf dup 0);          {shift rb left by shf bits}
else
        result := reg(ra)[31..f] || reg(rb)[l-1..0] || reg(ra)[shf-1..0];  {overlay field from rb on
                                                                             top of ra}
endif;
reg(ra) := result;
COND := (result ≠ 0);  {condition is TRUE if result is non-zero}
```

Field Manipulation Instructions, continued

MERGE IMMEDIATE

RCS	001	ra	len	shf	1	imm5
8	3	5	5	5	1	5

FORMAT

ra := ra deposit imm5 [shf, len]

DESCRIPTION

A right justified field of length specified by *len* is taken from the sign extended value contained in the *imm5* field. The field is left shifted by *shf* bits. If the sum of *shf* and *len* is greater than 32, the field is truncated. The field is merged with the contents of register *ra*. The result is placed in register *ra*.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO and is FALSE (0) if the result is ZERO.

OPERATION

```
if len > 0 then
        l := len;
else
        l := 32;
endif;
f := l + shf;
if f > 32 then
        f := 32;
        l := 32 - shf;
endif;
temp := (28 dup imm5[4]) || imm5[3..0];                          {sign-extend the immediate field to 32 bits}
result := reg(ra)[31..f] || temp[l-1..0] || reg(ra)[shf-1..0];   {shift and merge on top of ra}
reg(ra) := result;
COND := (result ≠ 0);
```

Field Manipulation Instructions, continued

EXTRACT

RCS	001	ra	len	shf	0	rb
8	3	5	5	5	1	5

FORMAT

ra := extract rb [shf, len]

DESCRIPTION

The contents of register *rb* is right-shifted by the number of bits specified by *shf,* and a right-justified field of length specified by *len* is extracted from it. If the sum of *shf* and *len* is greater than 32, the extracted field is sign-extended. The result is the extracted field, which is placed in register *ra*.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO and is FALSE (0) if the result is ZERO.

OPERATION

```
if len > 0 then
      l  := len;
else
      l  := 32;
endif;
temp     := (shf dup reg(rb)[31]) || reg(rb)[31..shf];   {shift rb right by shf bits}
result   := (32-l dup 0) || temp[l-1..0];                {zero all the bits outside the selected
                                                          field}
reg(ra)  := result;
COND     := (result≠0);
```

Field Manipulation Instructions, continued

DYNAMIC EXTRACT/DEPOSIT/MERGE

RCS	100	ra	len	111	ext	rb
8	3	5	5	3	3	5

FORMAT

ext	format	meaning
0	ra := deposit rb [sar, len]	dynamic deposit, fixed length
1	ra := deposit rb [sar, flr]	dynamic deposit
2	ra := ra deposit rb [sar, len]	dynamic merge, fixed length
3	ra := ra deposit rb [sar, flr]	dynamic merge
4	ra := extract rb [sar, len]	dynamic extract, fixed length
5	ra := extract rb [sar, flr]	dynamic extract

DESCRIPTION

These instructions perform Deposit, Deposit and Merge, Extract, and Deposit Immediate and Merge instructions with the shift amount determined by the contents of the *sar* register and the field length controlled either by the *flr* register of the *psr* or by the *len* field in the instruction. See the Extract and Deposit instructions for details on the function of these operations.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO, otherwise the condition generated is FALSE (0).

Field Manipulation Instructions, continued

MERGE HALFWORD HIGH

RCS	011	ra	imm16
8	3	5	16

FORMAT

ra := ra deposit imm16 [16, 16]

DESCRIPTION

The 16-bit immediate value is merged into the most significant 16 bits of register *ra*, and the result is placed in register *ra*.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO and is FALSE (0) if the result is ZERO.

OPERATION

```
result    := imm16[15..0] || reg(ra)[15..0];     {merge onto ra after shifting by 16 bits}
reg(ra)   := result;
COND      := (result ≠ 0);
```

Field Manipulation Instructions, continued

BITWISE MERGE

RCS	111	ra	rc	1111	01	rb
8	3	5	5	4	2	5

FORMAT

rc := (ra and not al) or (rb and al)

DESCRIPTION

This instruction performs a so called Bitwise Merge between the bits of the contents of register *rb* and register *ra*, controlled by the contents of register *al*. The result is placed in register *rc*.

Note that Multiply and Divide also use the al register. Therefore, a Bitwise Merge should not be executed while a Multiply or Divide operation is in progress.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO, otherwise the condition generated is FALSE (0).

OPERATION

```
result  := (reg(ra) and not al) or (reg(rb) and al);
reg(rc) := result;
COND    := (result ≠ 0);
```

Field Manipulation Instructions, continued

PERFECT EXCHANGE

RCS	111	ra	p	1111	10	rb
8	3	5	5	4	2	5

FORMAT

ra := p exchange rb

DESCRIPTION

This flexible bit manipulation command is used to swap fields or reverse the bit order on 2, 4, 8, 16, or 32-bit fields. One use of bit reversal is to calculate addresses in Fast Fourier Transforms. The Perfect Exchange operation is controlled by the 5-bit *p* field in the instruction.

This instruction performs a perfect exchange among the bits of the contents of register *rb*. The result is placed in register *ra*. Each bit, *p[i]*, of the *p* field controls the exchange of pair-wise adjacent fields of size 2^i bits. For example, when *p[0]* is set, each even-odd pair of bits is exchanged, and when *p[4]* is set, the upper halfword is exchanged with the lower halfword.

This general capability provides several important special cases. For example, setting *p[4..0]* to 11111 causes all bits in a word to be placed in reverse order (radix-2 bit reverse), and setting *p[4..0]* to 11110 causes all pairs of bits to be reversed (radix-4 bit reverse). Setting *p[4..0]* to 11000 will reverse the order of bytes in a word.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO, otherwise the condition generated is FALSE (0).

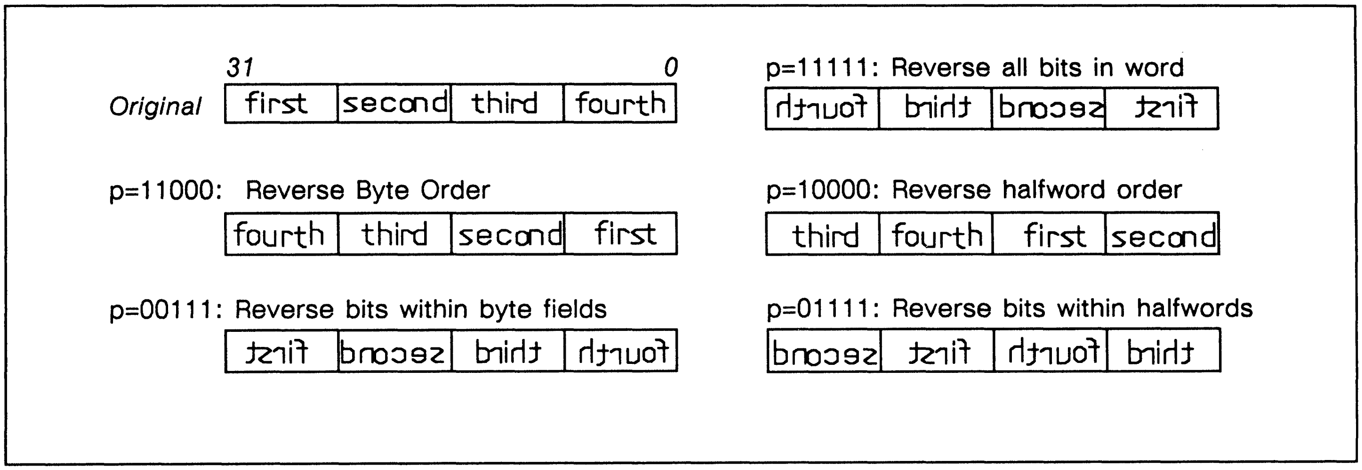

31
0
Original
first
second
third
fourth
p=11111: Reverse all bits in word
p=11000: Reverse Byte Order
fourth
third
second
first
p=10000: Reverse halfword order
third
fourth
first
second
p=00111: Reverse bits within byte fields
p=01111: Reverse bits within halfwords

Figure 13. Examples of perfect exchange

Field Manipulation Instructions, continued

OPERATION

```
t := reg(rb);
if p[4] then
        t := t[15..0] || t[31..16];
endif;
if p[3] then
        for i := 0 to 16 by 16 do
            t[i+15..i] := t[i+7..i] || t[i+15..i+8];
        enddo;
endif;
if p[2] then
        for i := 0 to 24 by 8 do
            t[i+7..i] := t[i+3..i] || t[i+7..i+4];
        enddo;
endif;
if p[1] then
        for i := 0 to 28 by 4 do
            t[i+3..i] := t[i+1..i] || t[i+3..i+2];
        enddo;
endif;
if p[0] then
        for i := 0 to 30 by 2 do
            t[i+1..i] := t[i] || t[i+1];
        enddo;
endif;
reg(ra)    := t;
COND       := (t ≠ 0);
```

Address Generation Instructions

LOAD/STORE ADDRESS WITH SIGNED DISPLACEMENT

RCS	110	ra	imm16
8	3	5	16

FORMAT

adr := ra + imm16

DESCRIPTION

The 16-bit immediate value is sign extended and added to the unsigned base address in register *ra*. The result is passed out the AD Bus and placed in the *adr* register. This instruction does not affect the c bit of the *psr*.

CONDITION

The condition generated is TRUE (1) if an unsigned 32-bit overflow is encountered, otherwise the condition generated is FALSE (0).

OPERATION

```
result := reg(ra) + (17 dup imm16[15]) || imm16[14..0];   {add the sign-extended displacement}
adr := result;                                             {internal address register}
AD := result;                                              {external address bus}


if imm16[15] = 0 then
      COND :=usovf;
else
      COND := not (usovf);
endif;
```

Address Generation Instructions, continued

LOAD/STORE ADDRESS OR ADD WITH INDEX/SIGNED DISPLACEMENT

RCS	111	ra	rc	1	ext	ixs	rb/imm5
8	3	5	5	1	3	2	5

FORMAT

instruction	ext	meaning
adr := ra, rc := ra + (rb << ixs)	000	load/store indexed, modify after
adr := ra, rc := ra + (imm5 <<ixs)	001	load/store signed displacement, modify after
adr := rc := ra + (rb << ixs)	010	load/store indexed, modify before
adr := rc := ra + (imm5 <<ixs)	011	load/store signed displacement, modify before
adr := ra + (rb << ixs)	100	load/store indexed, no modify
adr := ra + (imm5 <<ixs)	101	load/store signed displacement, no modify
rc := ra + (rb << ixs)	110	add indexed

DESCRIPTION

Address generation instructions take a left-shifted (0–3 bits), signed value from an immediate field or register and add it to a base register, optionally writing the result to another register. The address driven onto the AD Bus may be the result of the addition or the contents of the base register *before* the addition. This corresponds to pre-increment and post-increment indexing. Again, the shifting facility simplifies the generation of halfword, word, and doubleword array addresses in a byte-addressable environment.

```
{Post-modify:}
adr := ra, rc := ra + imm5
adr := ra, rc := ra + rb

{Pre-modify:}
adr := (rc := ra + imm5)
adr := (rc := ra + rb)

{No modify:}
adr := ra + imm5
adr := ra + rb
```

The contents of register *rb* (index) or a 5-bit signed displacement is shifted left the number of bits specified by *ixs* (a value of 0 causes no shift), and added to the unsigned base address in register *ra*. If modification is requested, the result is stored in register *rc*. The calculated address is the result of the add operation unless modify after is requested, in which case it is the contents of register *ra*. The calculated address is placed in the *adr* register and driven on the AD Bus. These instructions do not affect the *c* bit of the *psr*.

Address Generation Instructions, continued

CONDITION

The condition generated is TRUE (1) if a two's complement overflow is encountered when shifting or an unsigned 32-bit overflow is encountered when adding; otherwise the condition generated is FALSE (0). For the Add With Shift instruction (ext 6), the addition operation tests for two's complement addition overflow.

OPERATION

```
if ext[0] = 0 then
      temp := reg(rb);
else
      temp := (28 dup imm5[4]) || imm5[3..0];         {sign-extend the immediate field}
endif;
result := reg(ra) + (temp[31-ixs..0] || (ixs dup 0));   {shift the displacement by ixs bits and add}
case ext[2..1] of

      00b: adr := AD := reg(ra); reg(rc) := result;   {do the operation}
      01b: adr := AD := reg(rc) := result;
      10b: adr := AD := result;
      11b: reg(rc) := result;
endcase;
if ext [2..0] = 110 then
      COND := tcovf;
else
      COND := (reg(rb)[31..32-ixs] ≠ (ixs dup reg(rb)[31-ixs])) or usovf;
endif;
```

Load/Store and Alignment Instructions

Data transfers to and from memory take two operations: address generation and data transfer.

To load data, the RIP first executes a Load/Store Address instruction, which calculates an address and drives it onto the AD Bus. The RIP executes a Load Data instruction during a subsequent cycle, which takes the contents of the D Bus and puts it into a register.

Another instruction can be performed at the same time as the Load Data instruction, since Load Data uses only the RCS field of the instruction word. For instance, you can put an address generation instruction in the field, reducing the time for consecutive loads to one cycle per word.

Storing data is similar. Addresses are again generated with a Load/Store Address command, and data is stored with the Store Data command. The Store Data command takes up the RCS field, and the instruction in the RIP field generates the data to be stored. For instance, if the instruction was *rc := ra+rb,* the sum of ra+rb would be stored in *rc and* stored into memory.

The Byte Align and Store command can be used to store bytes, halfwords, and words. Because this is a separate instruction, it requires an extra cycle.

Load/Store and Alignment Instructions, continued

LOAD HALFWORD IMMEDIATE

RCS	010	ra	imm16
8	3	5	16

FORMAT

ra := imm16

DESCRIPTION

The 16-bit immediate value is sign-extended and placed into register *ra*.

CONDITION

The condition generated is TRUE (1) if the result of the operation is non-ZERO and is FALSE (0) if the result is ZERO.

OPERATION

```
result  := (17 dup imm16[15]) || imm16[14..0];     {sign-extend to 32 bits}
reg(ra) := result;
COND    := (result ≠ 0);
```

Load/Store and Alignment Instructions, continued

LOAD DATA

110	rd	RIP/coprocessor operation
3	5	24

FORMAT

rd := mem[adr]

DESCRIPTION

This instruction specifies that data is to be loaded from the D Bus into register rd in the register file. Because this instruction uses the RCS field of the instruction word, it can be performed simultaneously with other RIP operations. The loaded data is not available for use until the next instruction. Care must be taken to avoid writing of the data into the same register as specified by the operation in the remainder of the instruction word. If the other RIP instruction specifies that register *rd* is to be modified, then the contents of *rd* becomes undefined at the end of this instruction.

CONDITION

This instruction does not generate any condition. However, a condition may be generated by any instruction that is combined with this instruction; the condition so generated will not be affected by this instruction.

OPERATION

reg(rd) := D;

PROGRAMMING EXAMPLES

```
adr := ra+imm;                          {any RIP address instruction}
rd  := mem[adr] | rc := rd op rb;       {here the old value of rd is used in the calculation}


adr := ra+imm;
rd  := mem[adr] | <other instruction>;
rc  := rd op rb;                        {here the new, loaded value of rd is used in the calculation}


adr := ra + imm;
rd  := mem[adr] | rd := rc op rb;       {value of rd becomes undefined, not allowed}
```

Load/Store and Alignment Instructions, continued

STORE DATA

00001001	RIP operation
8	24

FORMAT

```
mem[adr] := result;      {RIP operation}
```

DESCRIPTION

This instruction specifies that the result of an RIP operation is to be stored to the previously addressed memory location. The Store Data instruction is specified in the uppermost 8 bits of the instruction. The lower 24 bits are used to specify any RIP operation that produces a result.

The Store Data instruction places the result of the current RIP operation onto the D Bus, and asserts all four WREN– bits.

CONDITION

The Store Data does not generate any condition. However, a condition may be generated by the RIP operation that is combined with this instruction; the condition thus generated will not be affected by this instruction.

OPERATION

```
D := result;
```

PROGRAMMING EXAMPLES

```
adr      := ra+imm;            {any RIP address instruction}
mem[adr]:= rc := ra op rb;     {write result to rc and memory}
```

Load/Store and Alignment Instructions, continued

BYTE ALIGN FOR LOAD DATA

RCS	111	ra	x	0	s	siz	1111	00	rb
8	3	5	1	1	1	2	4	2	5

FORMAT

ra := rb align [unsigned siz]
ra := rb align [signed siz]

siz	Size of operand
00	byte
01	halfword
10	tri-byte
11	word

DESCRIPTION

This instruction extracts a byte, halfword, tri-byte, or a word from a previously loaded word in the *rb* register. The instruction uses the byte address in the *adr* register together with the two-bit *siz* field from the instruction to extract the correct byte(s). The extracted value is zero-extended or sign-extended to a full 32-bit value. Zero- or sign-extension is controlled by the s bit in the instruction. The resulting 32-bit value is stored in the destination register *ra*.

The typical sequence of instructions to load a byte requires two instructions and three cycles. (The extra cycle is used to load the word containing the desired byte. This does not require the ALU and it could perform any other operation on this cycle.)

This instruction is defined to be register-to-register only; condition and AD Bus outputs are undefined.

CONDITION

This instruction does not generate a condition output.

Load/Store and Alignment Instructions, continued

OPERATION

```
a           := adr[1..0] • 8;
l           := (siz + adr[1..0]) • 8 + 7;
size        := siz • 8 + 7;
case s of
  0b: res   := (31-size dup 0) || reg(rb)[l..a];
  1b: res   := (31-size dup reg(rb)[l]) || reg(rb)[l..a];
endcase;
reg(ra)     := res;
COND        := undefined;                    {NOTE: siz + adr[1..0] ≤ 3}
result      := undefined;
```

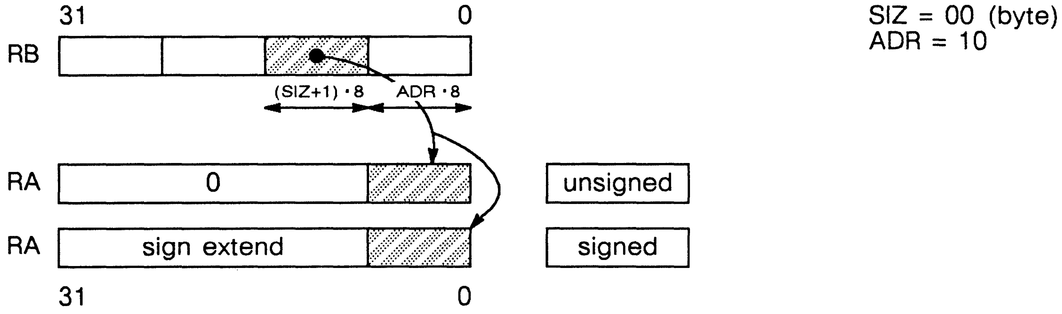


Figure 14. Byte align for load data instruction

Load/Store and Alignment Instructions, continued

BYTE ALIGN AND STORE DATA

RCS	111	ra	e	1	s	siz	1111	00	rb
8	3	5	1	1	1	2	4	2	5

FORMAT

instruction	e	s	meaning
mem[adr] := rb align [unsigned siz]	0	0	align for store unsigned
mem[adr] := rb align [signed siz]	0	1	align for store signed
mem[adr] := ra	1	x	no alignment

siz	Size of operand
00	byte
01	halfword
10	tri-byte
11	word

DESCRIPTION

These operations transfer from the register file to the external (D+) data bus, while aligning and truncating or sign extending the value to allow byte, halfword, and tri-byte addressing into word-wide memory. The correct alignment is specified by the *adr* register, the data size by the *siz* field, and sign extension by the *s* field. To implement this instruction properly the external memory must be capable of writing individual bytes as controlled by the WREN- bus.

The Byte Align and Store instruction is used to store bytes, halfwords, tri-bytes, and words in a byte addressable environment. The instruction performs two operations: data alignment and byte-write control.

The instruction takes as input a register number, the *adr* register, the data size (a constant in the instruction), and other miscellaneous controls in the instruction. The register number designates the data to be stored (the rightmost byte or halfword.) The *adr* register indicates the particular byte alignment to use; this register is automatically set by any address generation instruction. The data size is used to indicate whether a byte or halfword is being stored.

The input data to be stored is shifted the correct number of places to align it to the correct byte as specified by the low order two bits of the *adr* register. The correct byte write controls are driven so that the external memory subsystem will only write the correct bytes. The condition pin is also driven to indicate if the signed or unsigned data value would not fit into the destination format (overflow).

CONDITION

The generated is TRUE (1) if the operation truncates significant bits from the operand, otherwise the condition generated is FALSE (0).

Load/Store and Alignment Instructions, continued

OPERATION

```
a     := adr[1..0] • 8;
l     := (siz + adr[1..0]) • 8 + 7;
size  := siz • 8 + 7;
if e = 0 then
    case s of
    0b:  result := (31-l dup 0) || reg(rb)[size..0] || (a dup 0);
         D      := result;
         COND   := (reg(rb)[31..size+1] = 0);
    1b:  result := (31-l dup 0) || reg(rb)[size..0] || (a dup 0);
         D      := result;
         COND   := (reg(rb)[31..size+1] = (31-size dup reg(rb)[size]));
    endcase;
else
    result  := reg (ra);
    D       := result;
    COND    := undefined;
endif;
for i := 0 to 3 do
    WREN-[i] := FALSE;
endfor;
for i := 0 to siz do
    WREN-[adr [1..0] + i] := TRUE; {NOTE: siz + adr[1..0] ≤ 3}
endfor;
```

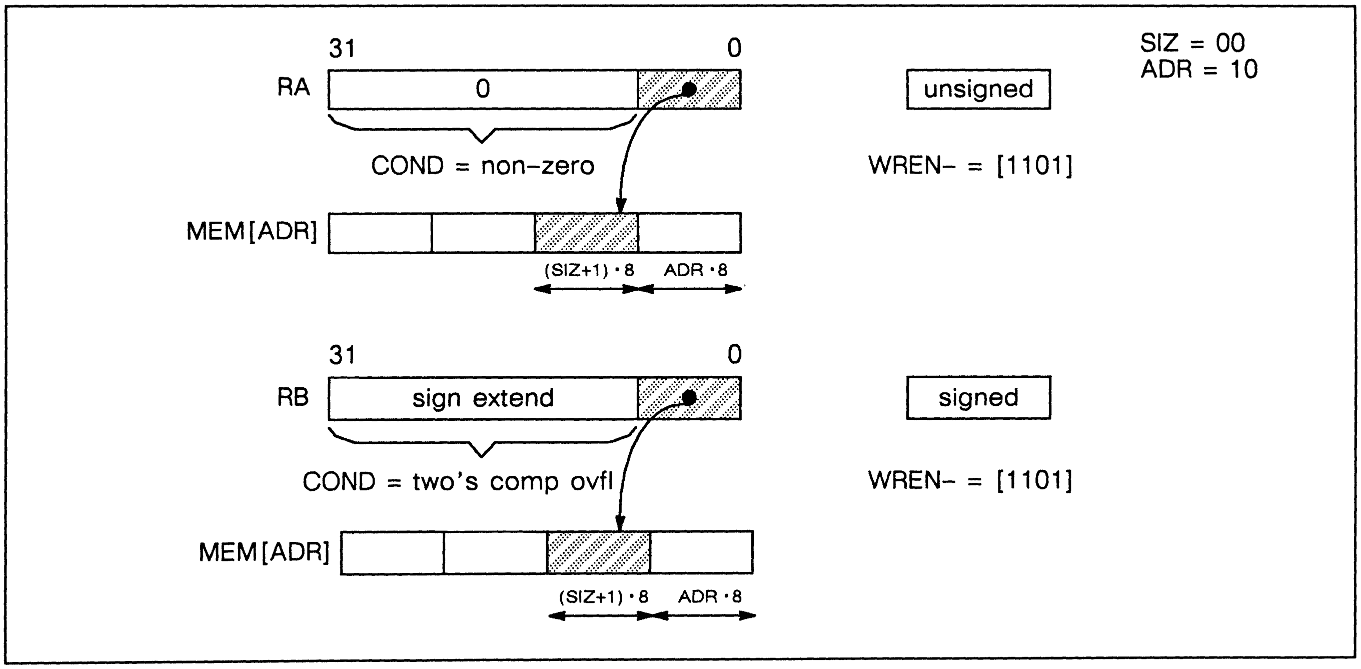


Figure 15. Byte align and store instruction

Multiply/Divide/Priority Encode/Housekeeping Instructions

MULTIPLY/DIVIDE/PRIORITY ENCODE/HOUSEKEEPING INSTRUCTIONS

RCS	111	ra	ext	1111	11	rb/imm5
8	3	5	5	4	2	5

FORMAT

instruction	ext	meaning
am,al := ra• rb	00000	start 32-bit two's complement multiply
-	00001	-
al,am := am||ra ÷ rb	00010	start 64-bit/32-bit unsigned integer divide
am := rb, al := ra	00011	load/reload am and al
ra := al	00100	unload quotient/ls product
ra := am + rb	00101	unload remainder/ms product plus register
ra := am + imm5 + al[31]	00110	unload remainder/ms product plus immediate plus sign
ra := am + imm5	00111	unload remainder/ms product plus immediate
ra := priority encode rb	01000	priority encode
-	01001	-
flr := rb	01010	load field length register
sar := rb	01011	load shift amount register
-	01100	-
-	01101	-
-	01110	-
-	01111	-
ra := psr	10000	save processor status register
ra := adr	10001	save address register
ra := psr, psr := rb	10010	save and load processor status register
psr := rb, adr := ra	10011	restore processor status register, address register
-	10100	-
-	10101	-
-	10110	-
-	10111	-
AD := adr	11000	load/store using address register
-	11001	-
-	11010	-
-	11011	-
psr.z := not psr.z, ra := psr	11100	swap register banks and save psr
psr.z := not psr.z	11101	swap register banks
-	11110	-
-	11111	-

DESCRIPTION

The first group of operations controls the multiply and divide hardware and allows access to the two 32-bit product registers. Note that the contents of the product registers are undefined if a multiply or divide operation is currently in progress. A Multiply requires an additional 11 MDCLK+ cycles to complete; due to optimization on the chip this only requires 5 CLK+ cycles. A Divide requires an additional 32 MDCLK+ cycles to complete, or 16 CLK+ cycles (see the section on Multiply and Divide operations for details).

The Multiply instruction (*ext* 0) loads the contents of register *ra* into the multiplier register and the contents of register *rb* into the multiplicand register and initiates a multiplication operation. The multiplier (*ra* operand) and the multiplicand (*rb* operand) are assumed to be two's complement values. A correction term, which may be computed while the multiplication operation is going on, may be added to the resulting product to perform purely unsigned or mixed unsigned and two's complement multiplication.

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

The Divide instruction loads the contents of register *ra* into the least significant word dividend register, the contents of register *rb* into the divisor register, and then initiates an unsigned divide operation. The value contained in the *am* register is used as the most significant word of the dividend. For 64-bit division, overflow may be checked while the division is going on ($am \geq rb$). For signed division, correction factors need to be applied to the result.

The second group of operations includes the Priority Encode instruction. The Priority Encode instruction gives, as a result, the number of ZERO bits which precede the most significant ONE bit of register *rb*. If all bits are ZERO, the value returned is 32, and the condition generated is FALSE (0), otherwise the condition is TRUE (1).

The third and fourth group of operations perform various housekeeping functions on the *adr* register, the *psr*, and portions of the *psr*.

CONDITION

The COND+ output is driven HIGH if the result is nonzero for one of the three instructions that unload *am* (*ext* 5,6,7) and the Priority Encode instruction (*ext* 8). This allows testing for signed or unsigned overflow from a multiplication. COND+ is not defined for the other five Multiply/Divide instructions (*ext* 0, 1, 2, 3, 4). The COND+ output is not defined for any of the Housekeeping instructions (*ext* = 1xxxx).

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

OPERATION

```
case ext of
      00000b: result := undefined; {initiate 32-bit two's complement multiply (ra • rb)}
      00010b: result := undefined; {start 64-bit/32-bit unsigned integer divide (am || ra÷rb)}
      00011b: result := undefined; am := rb; al := ra; {load/reload am and al}
      00100b: result := (ra := am + rb); {unload remainder/ms product plus register}
      00101b: result := (ra := am + rb); {unload remainder/ms product plus register}
      00110b: result := (ra := am + imm5 + al[31]); {unload remainder/ms product plus sign}
            {this condition detects two's complement multiply overflow}
      00111b: result := (ra := am + imm5); {unload remainder/ms product}
            {this condition detects unsigned multiply overflow}
      01000b: result := ra; ra := priority encode rb;
      01010b: result := undefined; flr := reg(rb)[4..0];
      01011b: result := undefined; sar := reg(rb)[4..0];
      10000b: result := reg(ra) := psr;
      10001b: result := reg(ra) := adr;
      10010b: result := reg(ra) := psr; psr := reg(rb);
      10011b: result := undefined; psr := reg(rb); adr := reg(ra);
      11000b: AD     := result := adr
      11001b: AD     := result := adr
      11100b: result := psr; psr.z := not psr.z;  reg(ra) := result;
      11101b: result := undefined; psr.z := not psr.z
endcase;
if ext [4..0] = 00101, 00110, 00111, 01000 then
      COND := (result ≠ 0);
else
      COND := undefined;
endif;
```

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

MULTIPLY AND DIVIDE OPERATIONS

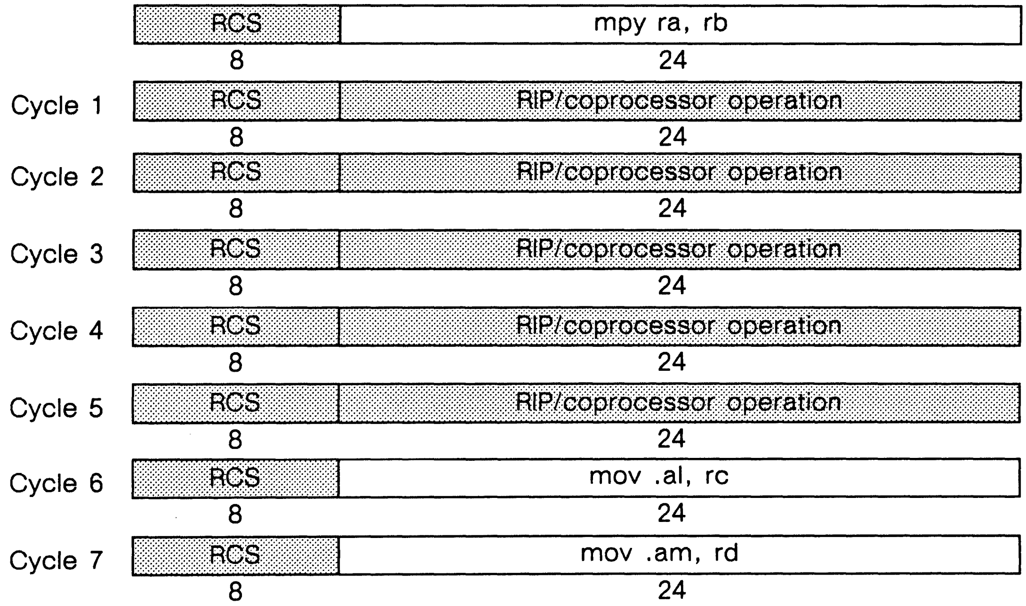
A 32-bit signed Multiply is performed in eight cycles; a 64/32-bit mixed-precision division is done in 20 cycles. Multiplication gives a 64-bit product; division gives a 32-bit quotient and 32-bit remainder.

The Multiply and Divide operations use dedicated hardware so that other operations may be performed in the RIP simultaneously. A Multiply or Divide operation is initiated, and a fixed number of cycles later the result is placed into the *am* and *al* (product) registers. The *al* register is loaded on cycle 6, and the *am* register is loaded on cycle 7. The contents of the *al* and *am* registers are undefined prior to cycles 6 and 7, respectively. The result can be removed from the product registers at any time after the operation has been completed. If a Multiply or Divide operation is attempted while another is currently in progress, it will be ignored.

Note that the Multiply and Divide operations are dependent on the number of cycles, not instructions, that are executed. Instructions that have been neutralized still count for the purposes of determining when a multiply or divide operation is finished.

The Multiply/Divide operations themselves have no effect on the condition code. However, the templates for a 32-bit unsigned or two's complement multiply use an addition instruction to generate a condition on the last cycle. This condition indicates if an overflow has occurred on the Multiply operation.

32-Bit Two's Complement Multiply with 64-Bit Result



Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

32-Bit Two's Complement Multiply with 32-Bit Result

RCS	mpy ra, rb
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	mov .al, rc
8	24
RCS	addamis 0, rd
8	24

32-Bit Unsigned Multiply with 32-Bit or 64-Bit Result

–	mpy ra, rb
8	24
br .gez, $+2	addi 0, ra, ra
8	24
br $+2	mov rb, rc
8	24
–	clr rc
8	24
br .gez, $+2	addi 0, rb, rb
8	24
–	add rc, ra, rc
8	24
–	mov .al, rd
8	24
–	addam rc, re
8	24

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

UNSIGNED DIVIDE

The divide operation shown here is for 32-bit unsigned numbers only. When the dividend and divisor are both 32-bit unsigned numbers the .am register must be loaded with zero. The 32-bit unsigned quotient and remainder may be unloaded after the divide operation is complete. When the dividend is a 64-bit unsigned number and the divisor is a 32-bit unsigned number, the results of the Divide operation are undefined.

64-Bit Dividend, 32-Bit Divisor, 32-Bit Quotient, Remainder

RCS	mov rc, .am
8	24
RCS	div ra, rb
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	RIP/coprocessor operation
8	24
RCS	mov .al, rd (quotient)
8	24
RCS	mov .am, re (remainder)
8	24

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

32-BIT SIGNED DIVIDE

If a single length two's complement divide is desired then additional instructions need to be added to convert the input operands to unsigned numbers, calculate the resulting sign and convert the results to the correct sign. Depending on the desired definition of the modulo operation extra code to convert the remainder may also be required. The code example in figure 16 shows how this may be achieved when the dividend and divisor are both 32-bit two's complement numbers.

64-BIT SIGNED DIVIDE

If a double length two's complement divide is desired then additional instructions need to be added to convert the input operands to unsigned numbers, calculate the resulting sign and convert the results to the correct sign. Depending on the desired definition of the modulo operation extra code to convert the remainder may also be required. The code example NO TAG shows how this may be achieved when the dividend is a 64-bit two's complement number and the divisor is a 32-bit two's complement number. This is the only case when the the .am register can be loaded with a non-zero value and correct results be obtained.

Multiply/Divide/Priority Encode/Housekeeping Instructions, continued

```
        .native

        .text

        .globl  _s_32_by_32_div

_s_32_by_32_div:
#
#       This routine performs a signed integer divide with a 32-bit
#       dividend and a 32-bit divisor. It produces a 32-bit quotient,
#       and a 32-bit remainder.
#
#       Input:
#                       .r0     dividend
#                       .r1     divisor
#
#       Output:
#                       .r2     quotient
#                       .r3     remainder
#
        .reg            .r2, q
        .reg            .r3, rem
        .reg            .r4, trash
        .reg            .r5, dividend
        .reg            .r6, divisor
        .reg            .r7, offset
        .reg            .r8, sign
        .reg            .r9, zero
        .reg            .r10, maxneg

# initialize zero
        movi            0, zero

# check negative dividend
        addi            0, .r0, dividend; br .gez check_divisor

# negate dividend
        subi            0, dividend, dividend

check_divisor:
        addi            0, .r1, divisor; br .gez do_divide

# negate divisor
        subi            0, divisor, divisor

do_divide:
        ldamal          zero, trash
        div             dividend, divisor

# check for max negative number in divisor or dividend
        movi            0x80000000,  maxneg
        movih           0x80000000 >> 16, maxneg
        sub             maxneg, .r1, trash; br .nez $+2
        br              max_neg_divisor
        sub             maxneg, .r0, trash; br .nez not_max_neg
        br              overflow

# check for divide by zero
not_max_neg:
        addi            0, divisor, trash; br .nez $+2
        br              divide_by_zero

# calculate sign of quotient
        xor             .r0, .r1, sign
        movi            1, offset
        addi            0, .r0, trash; br .gez $+2
        addi            3, offset, offset       # negative remainder
        addi            0, sign, trash; br .gez $+2
        addi            6, offset, offset       # negative quotient
        pushs           offset
        brstkp

q_pos_r_pos:
        mov             .al, q                  # positive quotient
        rts; mov        .am, rem                # positive remainder
        nop
```

```
q_pos_r_neg:
        mov             .al, q                  # positive quotient
        rts; mov        .am, rem
        subi            0, rem, rem; ovneut     # negative remainder

q_neg_r_pos:
        mov             .al, q
        rts; subi       0, q, q                 # negative quotient
        subi            0, .am, rem; ovneut     # positive remainder

q_neg_r_neg:
        mov             .al, q
        subi            0, q, q                 # negative quotient
        rts; mov        .am, rem
        subi            0, rem, rem; ovneut     # negative remainder

max_neg_divisor:
# divisor is max neg — i.e. -2^32 — so the quotient is 0
# (max/max -> 1), and the remainder is neg (dividend)
        sub             maxneg, .r0, trash; br .nez not_max_dividend
        nop                                     # kill cycles from div
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        rts; movi       1, q
        movi            0, rem; ovneut

not_max_dividend:
        nop                                     # kill cycles from div
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        rts: neg        dividend, rem
        movi            0, q; ovneut

overflow:
#
#
# overflow exception handler goes here
# NOTE: 9 cycles (from div) must be used before returning
#
#
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        rts

divide_by_zero:
#
#
# division by zero exception handler goes here
# NOTE: 7 cycles (from div) must be used before returning
#
#
        nop
        nop
        nop
        nop
        nop
        nop
        rts
```

Figure 16. 32-bit/32-bit signed divide

```
        .native

        .text

        .globl   _s_64_by_32_div

 _s_64_by_32_div:
 #
 #      This routine performs a signed integer divide with a 64-bit
 #      dividend and a 32-bit divisor. It produces a 32-bit quotient,
 #      and a 32-bit remainder.
 #
 #      Input:
 #                      .r0     dividend — msw
 #                      .r1     dividend — lsw
 #                      .r2     divisor#
 #      Output:
 #                      .r3     quotient
 #                      .r4     remainder
 #
        .reg            .r3, q
        .reg            .r4, rem
        .reg            .r5, trash
        .reg            .r6, div1
        .reg            .r7, div2
        .reg            .r8, divisor
        .reg            .r9, offset
        .reg            .r10, sign
        .reg            .r11, zero
        .reg            .r12, maxneg

 # check for negative dividend/divisor
        addi            0, .r0, div1; br .ltz neg_dividend
        mov             .r1, div2; shbr check_divisor

 neg_dividend
        movi            0, zero
        usub            zero, .r1, div2
        usubc           zero, div1, div1

 check_divisor:
        addi            0, .r2, divisor; br .gez do_divide
        subi            0, divisor, divisor         # negate divisor

 do_divide:
        ldamal          div1, trash
        div             div2, divisor

 # check for max negative number in divisor or dividend
        movi            0x80000000,  maxneg
        movih           0x80000000 >> 16, maxneg
        sub             maxneg, .r2, trash; br .nez $+2
        br              max_neg_divisor
        sub             maxneg, .r0, trash; br .nez not_max_neg
        addi            .r1, 0, trash; br .nez, not_max_neg
        br              overflow

 not_max_neg:
 # check for divide by zero, overflow
        addi            0, divisor, trash; br .nez $+2
        br              divide_by_zero

        usub            div1, divisor, trash; br .ltz $+2
        br              overflow

        xor             .r0, .r2, sign
        movi            1, offset
        addi            0, .r0, trash; br .gez $+2
        addi            4, offset, offset           # negative remainder
        addi            0, sign, trash; br .gez $+2
        addi            8, offset, offset           # negative quotient
        pushs           offset
        brstkp
```

```
 q_pos_r_pos:
        mov             .al, q                          # positive quotient
        ext             q, 31, 1, trash; br .nez overflow
        rts; mov        .am, rem                        # positive remainder
        nop

 q_pos_r_neg:
        mov             .al, q                          # positive quotient
        ext             q, 31, 1, trash; br .nez overflow
        rts; mov        .am, rem
        subi            0, rem, rem; ovneut             # negative remainder

 q_neg_r_pos:
        mov             .al, q
        ext             q, 31, 1, trash; br .nez overflow
        rts; subi       0, q, q                         # negative quotient
        subi            0, .am, rem; ovneut             # positive remainder

 q_neg_r_neg:
        mov             .al, q
        ext             q, 31, 1, trash; br .nez overflow
        subi            0, q, q                         # negative quotient
        rts; mov        .am, rem
        subi            0, rem, rem; ovneut             # negative remainder

 overflow:
 #
 #
 # overflow exception handler goes here
 # NOTE: 8 cycles (from div) must be used before  returning
 #
 #
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        rts

 max_neg_divisor:
 # divisor is max neg — i.e. -2^32 — so the quotient is neg
 # (dividend msw), and the remainder is 0
        sub             maxneg, .r0, trash; br .nez not_max
        addi            0, .r1, trash; br .nez not_max
        br              overflow                        #dividend is max neg

 not_max:
 # negate the dividend and use the msw
        nop                                             # kill cycles from div
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        usub            zero, .r1, trash
        rts; usubc      zero, .r0, q
        clr             rem; ovneut

 divide_by_zero:
 #
 #
 # division by zero exception handler goes here
 # NOTE: 6 cycles (from div) must be used before  returning
 #
 #
        nop
        nop
        nop
        nop
        nop
        rts
```

Figure 17. 64-bit/32-bit signed divide

Coprocessor/RCS Operations

COPROCESSOR OPERATIONS

RCS	x	x	0	x	x	x

DESCRIPTION

This instruction is a no-op. It is designed to allow a coprocessor to execute an instruction without changing the state of the RIP.

CONDITION

This instruction does not generate a condition.

Coprocessor/RCS Operations, continued

STORE DATA FROM COPROCESSOR

101	rd	RIP operation
3	5	24

FORMAT

mem[adr] := coprocessor register (rd);   {RIP operation}

DESCRIPTION

This instruction specifies that coprocessor register rd is to be stored to the previously addressed memory location. The Store Data From Coprocessor instruction is specified in the uppermost 8 bits of the instruction. The lower 24 bits may be used to specify any RIP operation. The coprocessor should place the result on the D bus, and the XL-8237 will assert all four WREN- bits.

CONDITION

This instruction does not generate a condition. However, a condition may be generated by the RIP operation that is combined with this instruction; the condition thus generated will not be affected by this instruction.

OPERATION

D := coprocessor register (rd);

Coprocessor/RCS Operations, continued

LOAD DATA TO COPROCESSOR

111	rd	RIP operation
3	5	24

FORMAT

coprocessor register (rd) := mem[adr]; {RIP operation}

DESCRIPTION

This instruction specifies that the contents of the previously addressed memory location are to be loaded into the coprocessor register *rd.* The Load Data To Coprocessor instruction is specified in the uppermost 8 bits of the instruction. The lower 24 bits may be used to specify any RIP operation.

CONDITION

This instruction does not generate a condition.

OPERATION

coprocessor register (rd) := mem[adr];

Coprocessor/RCS Operations, continued

TRANSFER TO/FROM RIP

00000000	ext	ra	x	x
8	3	5	5	11

FORMAT

```
AD := ra
ra  := AD
```

DESCRIPTION

This instruction allows single cycle transfers to be made between an XL-8237 register and an external register (including one in the XL-8236 RCS). The transfer is made via the AD+ bus. This provides a path between the RIP and other system blocks, such as a RCS or floating point coprocessor, without having to use "mailboxes" in system memory.

CONDITION

This instruction does not generate a condition.

OPERATION

```
case ext[2..0] of
      000b:      AD := reg(ra);
      001b:      AD := reg(ra);
      010b:      reg(ra) := AD;
      011b:      reg(ra) := AD;
      100b:      nop;
      101b:      nop;
      110b:      nop;
      111b:      nop;
endcase;
```

Memory Operations

This section deals with operations needed to move data to and from memory. Loading and storing data requires two steps: address generation and data transfer. Loading and storing can be performed on aligned 32-bit words, or on any contiguous set of bytes within a word. Addresses are generated using one of the RIP's address generation instructions. The address calculated by these instructions is latched into the *adr* register and driven onto the AD Bus.

A word load operation can be performed in conjunction with any other RIP operation. The Load Data instruction is specified in the instruction field normally reserved for RCS control. The data is written into the register file at the end of this instruction, and is available for use in the next instruction. If the other RIP operation specifies the same register as a source in the word load operation (*rd*), the old value of *rd* is used (see programming examples for the Load Data instruction). Note that it is legitimate for this instruction to initiate another Read operation by executing an RIP address operation. This gives a maximum pipelined rate of one load per cycle with a 2-cycle latency for each individual word.

A byte-aligned load operation starts in the same manner as a word load. When the word is in the register file, an extra instruction, Byte Align For Load, is executed, using the current value of the *adr* register and the value of the *siz* field to extract the appropriate byte, half-word, or tri-byte from the loaded word. In order for this to execute properly the *adr* register must not have been modified, since the subsequent Byte Align For Load instruction would then use the new, incorrect, value. This gives a maximum rate of one byte-aligned load per two cycles with a three cycle latency for each load.

A word-aligned store operation is performed by storing the result of a simultaneous register-to-register operation in the RIP. The four WREN– write enables are asserted with the result on the D Bus.

A data store operation can also be initiated by a Byte Align For Store instruction. This instruction takes a register, extracts the selected data (byte, halfword, tribyte, or word), and drives the result data onto the D bus. The RIP uses the *siz* field of the store instruction and the contents of the *adr* register to determine the value of the WREN– bus.

It has been assumed here that read accesses to memory have a single cycle of latency. If slower memory is used then the STALL– input signal may be used to stall the XL-8237 to provide the necessary delay; extra instructions can be inserted between the RIP address operation and the subsequent data transfer operation; or CLK can be stopped until the data is ready. MDCLK can be kept running while CLK is stopped, so long as the skew specifications are otherwise maintained.

Generally, STALL– is used on code memory misses, and the clock is stopped on data memory misses.

Memory Operations, continued

Load and Store Operations

Load template (word-aligned)

RCS	RIP address operation
8	24
load rd	RIP/coprocessor operation
8	24

Load template (byte-aligned)

RCS	RIP address operation
8	24
load rd	RIP/coprocessor operation
8	24
RCS	align rb, ra, size
8	24

Store template (word-aligned)

RCS	RIP address generation
8	24
store	RIP operation
8	24

Store template (byte-aligned)

RCS	RIP address generation
8	24
RCS	store rc, size
8	24

Instruction Neutralization

Normally the XL-8237 executes one instruction per clock cycle. Under certain circumstances this flow needs to be modified; perhaps because external data or code is unavailable or a coprocessor requests a stall condition. The RIP has two input signals, NEUT– and STALL–, that are used to cancel the current, the next, or both the current and next instructions, respectively.

NEUT–

The NEUT– signal suppresses the results of the current instruction. If this signal is asserted, the current instruction is cancelled without modifying any state in the RIP. This signal is meant to be used in conjunction with the XL-8236 RCS to make optimal use of the effects of delayed branching.

STALL–

The STALL– input signal cancels the next instruction. This signal is intended to be used with a code cache or dynamic RAM to signal the delay or absence of code.

NOP

The XL-8237 does not have an explicit NOP instruction. Many instructions can be used to achieve the effect of a NOP, for example, adding 0 to a register. Care should be taken that a Load to that register is not also being performed on the same cycle. Any *long RCS instruction* (in the XL-8236 RCS) is treated by the RIP as a NOP.

Overflow Detection

The XL-8237 checks for overflow in certain instructions. If an overflow is detected, then the COND signal is modified to indicate that this has occurred. Refer to the specific instructions to determine which instructions generate overflow and how that overflow affects the COND signal.

The RIP recognizes two types of overflow: two's complement (tcovf) and unsigned (usovf). The definition of a two's complement overflow is when the carry-in to the most significant bit is different from the carry-out from the most significant bit. The definition of an unsigned overflow is when the result is larger than $2^{32} - 1$ or less than 0.

Development Tools

The HyperScript-Processors are part of WEITEK's XL-Series of processors. They are largely compatible with the XL-8100 series of processors, and use the same development tools.

WEITEK provides a family of software tools to aid applications development and debugging, using the XL-8236 and its companion processors, the XL-8237 32-bit Raster Image Processor and the XL-3232 32-bit Graphics Floating Point Data Path Unit.

HYPERSCRIPT INTERPRETERS

WEITEK supplies a PostScript-compatible interpreter that offers form, fit, function, and image compatibility with that offered by Adobe Systems Corporation. Both a C version of the software, and a assembly-coded graphics library are available.

Third-party PostScript-compatible interpreters will also be available for the XL-8200.

The interpreter supports both Bitstream FontWare and URW's NIMBUS font-scaling software. Fonts are fully compatible with Adobe Font Metrics and are represented in Bezier outline form.

HIGH-LEVEL LANGUAGE COMPILERS

The XL-Series supports an industry-standard C compiler. Industry-standard implementations allow existing programs to be ported to the XL-Series without modification. These compilers all share an optimizing code generator which employs optimization techniques found on mainframe compilers.

The compiler-generated code is refined through the XL-Series' unique parallelizer. The XL-Series parallelizer takes the sequential code and compacts integer and floating point operations into every instruction. The parallelizer provides the code-packing efficiency that otherwise could be achieved only through hand-written assembly code.

COMPLETE DEVELOPMENT SYSTEM SUPPORT

The design of an XL-based product is simplified by the XL software and hardware development tools. The application programmer is able to develop and debug software on a VAX, SUN 3, or Compaq Deskpro 386 system with the XL-Series Software Development Environment, which includes a software simulator. A development board set, which includes a software monitor and I/O drivers, allows low-level debugging and final tuning of software on an XL-Series processor. For the hardware designer, XL-Series functional simulators and complete engineering documentation, including an example PC-board layout, are available.

The design of raster image processors is also facilitated by a graphics development system which is composed of a RIP board with the XL-8200, 3 Mbytes of page buffer and font memory, 256 kwords of code memory for the interpreter, PC/AT-bus system interface, and Canon LBP-SX video interface card. This graphics development system provides a stable hardware environment on which PDLs can be debugged independently of the final target hardware.

Design Requirements

The XL-8237 is designed to be upgradeable to enhanced parts while retaining instruction set compatibility. In order to assure compatibility with future WEITEK processor devices, the following restrictions, which do not degrade performance in any way, should be observed:

> Set all fields marked "x" to zero in instructions which contain them. This assures that operations which are added in future designs will not modify the function of current instructions.
>
> Set all processor status register bits which are marked as zero to a zero value. This assures that additional *psr* bits which may be added will not impact compatibility with the RIP.

Do not attempt to read the *al* and *am* registers before a Multiply or Divide operation has completed, as the values returned or functions performed in these cases may be implementation-dependent.

Pins marked "NC" (not connected) on the pin configuration diagram may be defined as *signal pins* in future enhancements to the RIP. Therefore, to preserve future upward compatibility, these pins should indeed be left unconnected.

Specifications

ABSOLUTE MAXIMUM RATINGS

Supply voltage	−0.5 to 7.0 V
Input voltage	−0.5 to Vcc
Output voltage	−0.5 to Vcc
Operating temperature range T_{CASE}	−55°C to 125°C
Storage temperature range	−65°C to 150°C
Lead temperature (10 seconds)	300°C
Junction temperature	175°C

Recommended Operating Conditions

PARAMETER		COMMERCIAL			UNIT
		MIN	NOM	MAX	
V_{CC}	Supply voltage	4.75	5.0	5.25	V
I_{OH}	High-level output current			-1.0	mA
I_{OL}	Low-level output current			4.0	mA
T_{CASE}	Operating case temperature	0		85	°C

DC Specfications

PARAMETER		TEST CONDITIONS	COMMERCIAL		UNIT
			MIN	MAX	
V_{IH}	High-level input voltage	V_{CC} = MIN	2.0		
V_{IHC}	High-level input voltage for CLK and MDCLK only	V_{CC} = MIN	2.4		
V_{IL}	Low-level input voltage	V_{CC} = MIN		0.8	V
V_{ILC}	Low-level input voltage for CLK and MDCLK only	V_{CC} = MIN		0.8	V
V_{OH}	High-level output voltage	V_{CC} = MIN, I_{OH} = −1.0 mA	2.8		
V_{OL}	Low-level output voltage	V_{CC} = MIN, I_{OL} = 5.0 mA		0.4	
I_{LI}	Input leakage current	V_{CC} = MAX, V_{IN} = 0 − V_{CC}		±10	
I_{LO}	Output leakage current (output disabled)	V_{CC} = MAX, V_{OUT} = 0 − V_{CC}		±10	µA
I_{CC}	Standby current	V_{CC} = MAX, DC conditions,TTL inputs		250	
I_{CC}	Switching current	V_{CC} = MAX, T_{CY} = MIN, TTL inputs		300	mA
C_{IN}	Input capacitance *	T_A = 25° C		8	pF
C_{CLK}	Clock capacitance *	f = 1 MHz		20	
C_{OUT}	Output capacitance *	V_{CC} = 5.0 V		10	

* Capacitance not tested.

AC Timing Description

An instruction cycle is one CLK cycle, broken into two parts named phase one and phase two. The CLK signal controls the selection of the two phases, as shown in figure 19.

The MDCLK signal is used only for controlling the multiply/divide unit. It is driven at twice the frequency of the CLK cycle and is synchronized to it. It does not control any other logic on the chip.

The timing of all XL-8237 signals during the execution of a single instruction (cycle 1) is shown in figure 21. The instruction received just prior to cycle 1 is decoded and executed during the cycle. If STALL- is asserted at the beginning of the cycle, then the instruction is interpreted as a NOP. If NEUT- is asserted (during or at the end of the cycle, respectively), then the instruction is executed but the results are discarded.

The AD Bus output is controlled by the executing instruction. Three different parameters are specified (T_1, T_2, T_3, and T_4) which characterize the output delay for three different classes of instructions (corresponding to three different major paths through the device). The T_1 parameter applies to the instructions that drive *ra* directly out onto the AD Bus—intra-processor transfer and post-increment forms of address generation. The T_2 and T_3 parameters are for those instructions that compute results in the ALU—arithmetic, logical, and address generation. The T_4 parameter applies to all other instructions with meaningful results: Deposit, Merge, Extract, Byte Align, Perfect Exchange, Priority Encode, etc.. Note that the AD Bus output is not affected by the assertion of NEUT- in the current cycle.

If a store instruction is executed in the current cycle, the D Bus is driven with the results during the next (delayed) cycle, with an output delay of T_{11}. In this mode, if NEUT- is asserted in the current cycle, then the D Bus is tri-stated during the delayed cycle. If STALL- is asserted at the end of the previous cycle, the D Bus is also tri-stated during the delayed cycle.

The WREN- Bus tracks the D Bus. When the D Bus is tri-stated, the WREN- Bus is forced HIGH. When the D Bus is driven, one or more of the WREN- Bus signals will be asserted to indicate the valid bytes on the D Bus (and thus the bytes to be written).

AC Specifications

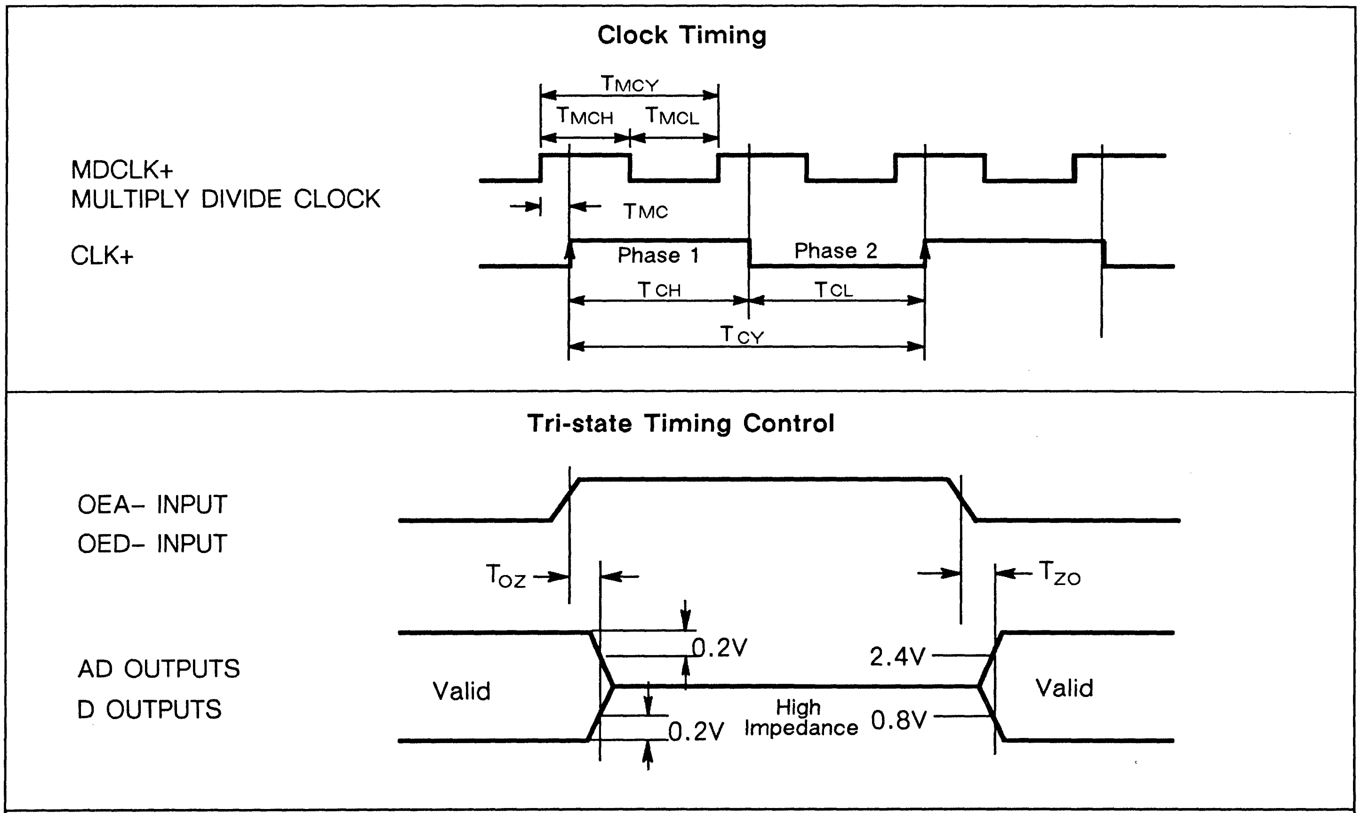
AC TEST CONDITIONS:			
V_{CC} = MIN	V_{IH} = 3.5V V_{OH} = 2.8V, I_{OH} = -1.0 mA V_{IL} = 0.4V V_{OL} = 0.4V, I_{OL} = 4.0 mA	T_{CASE} = 85 °C	C_{LOAD} = 40 pF

DESCRIPTION	NOTES	XL-8237-40		XL-8237-20		XL-8237-10		COMMENTS
		MIN	MAX	MIN	MAX	MIN	MAX	
T_{CY} Clock cycle time		120		200		350		
T_{CH} Clock high time		55		90		165		
T_{CL} Clock low time		55		90		165		
T_{MCY} MDCLK cycle time		60		100		175		
T_{MCH} MDCLK high time		25		45		80		
T_{MCL} MDCLK low time		25		45		80		
T_{MC} MDCLK rising edge to CLK transition		0	10	0	10	0	10	
T_R, T_F CLK rise and fall times			5		5		5	
T_1 CLK rising edge to AD valid			95		150		250	Driving a register out on the AD Bus (Intrasystem Transfer)
T_2 CLK rising edge to AD[31..3] valid			95		150		250	Driving the ALU result on the AD Bus. Also applies to address generation
T_3 CLK rising edge to AD[2..0] valid			85		150		250	As above
T_4 CLK rising edge to AD valid			110		165		265	Driving the result from the Field Merge Unit on the AD Bus. Applies to Priority Encode, Shift, Merge, Extract, Byte Align And Store.
T_{11} CLK rising edge to D and to WREN- outputs			50		60		70	
T_{12} D Bus turn-on time		5		5		5		
T_{13} D Bus turn-off time			50		60		70	
T_{14} CLK rising edge to COND output			105		165		315	
T_{15} AD Bus turn-on time		15		15		15		
T_{16} AD Bus turn-off time			55		65		75	
T_{S1} Input setup time for data on the AD and D D buses and misc.		25		30		35		
T_{S2} Input setup time for C Bus		25		30		35		
T_{S3} Set-up time for NEUT- input		15		25		40		
T_{H1} Hold time		3		3		3		Hold time for all inputs except C bus
T_{H2} Hold time		5		5		5		Hold time for the C bus only
T_{VO} Output valid time		5		5		5		
T_{ZO} Output enable time			30		40		50	
T_{OZ} Output disable time			30		40		50	

All units in nanoseconds

Figure 18. Clock and tri-state timing. Contact your WEITEK sales representative for XL-8237-60 AC Specifications.

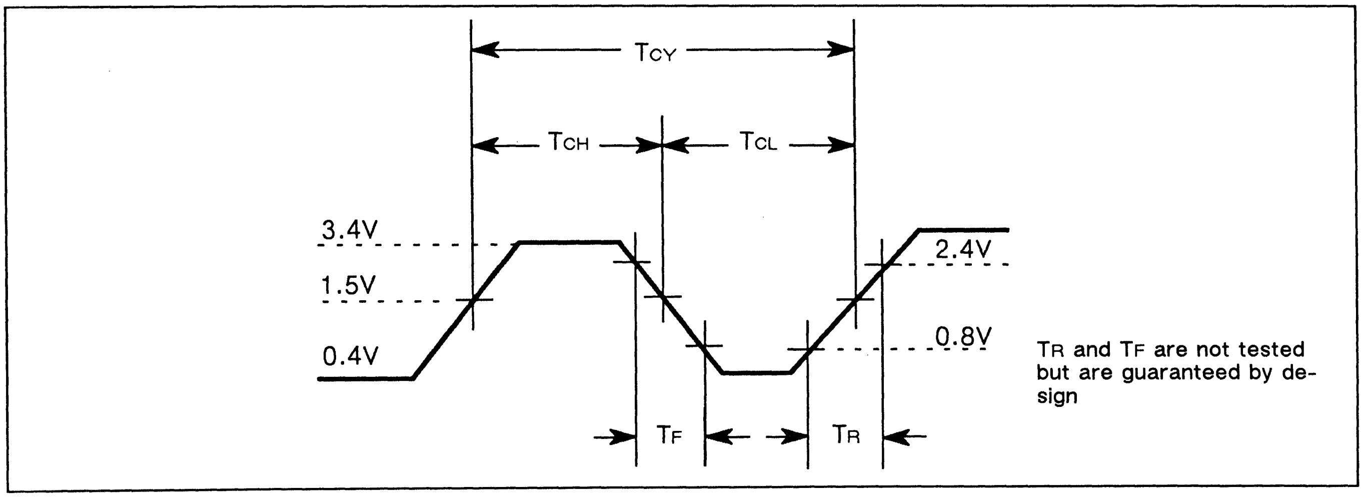
Timing Diagrams



Notes:

1. T_{OZ} is a slowest of two values: T_{HZ} and T_{LZ}, which are the times from high voltage to tri-state and low-voltage to tri-state, respectively. Similarly, T_{ZO} is the slowest of T_{ZH} and T_{ZL}. T_{OZ} is not measured but is guaranteed by design.
2. TTL inputs are 0.4V and 3.5 V
3. Timing transitions are measured at 1.5V unless otherwise specified

Figure 19. Clock and tri-state timing



T_R and T_F are not tested but are guaranteed by design

Figure 20. Clock timing, showing rise and fall times

Timing Diagrams, continued

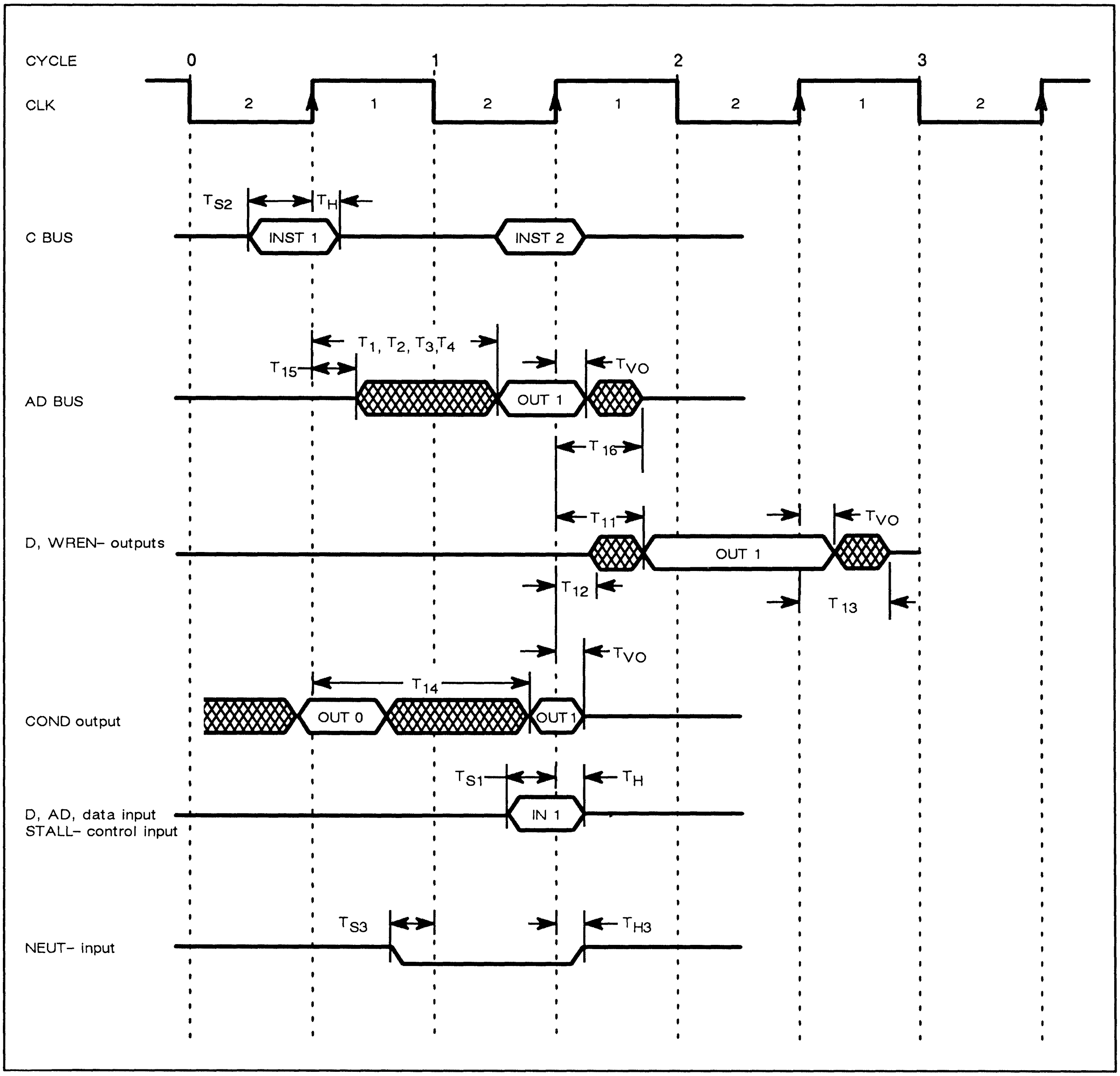
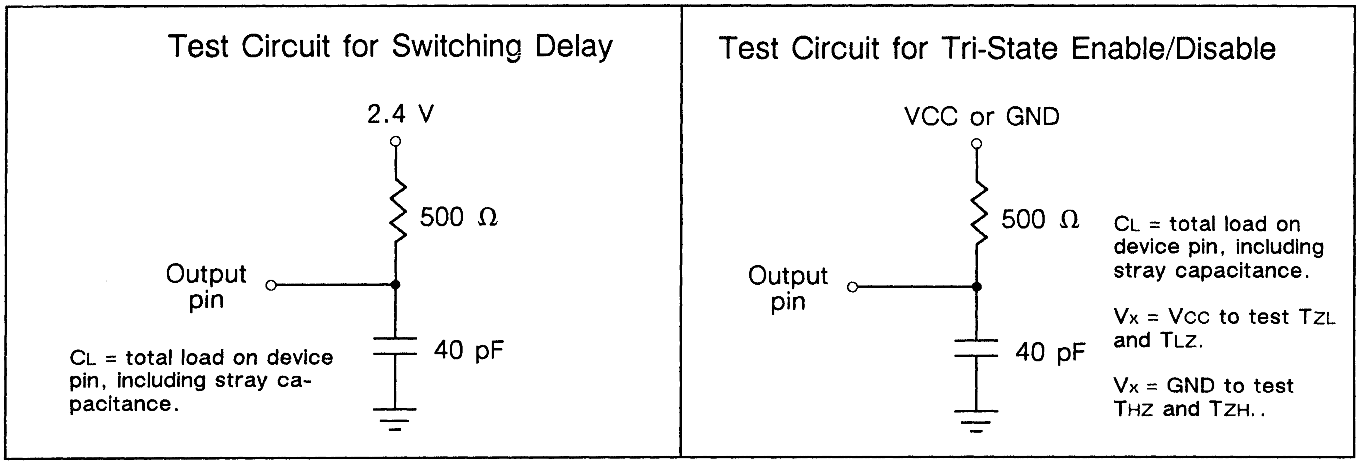


Figure 21. Timing diagram

I/O Characteristics



Test Circuit for Switching Delay

Test Circuit for Tri-State Enable/Disable

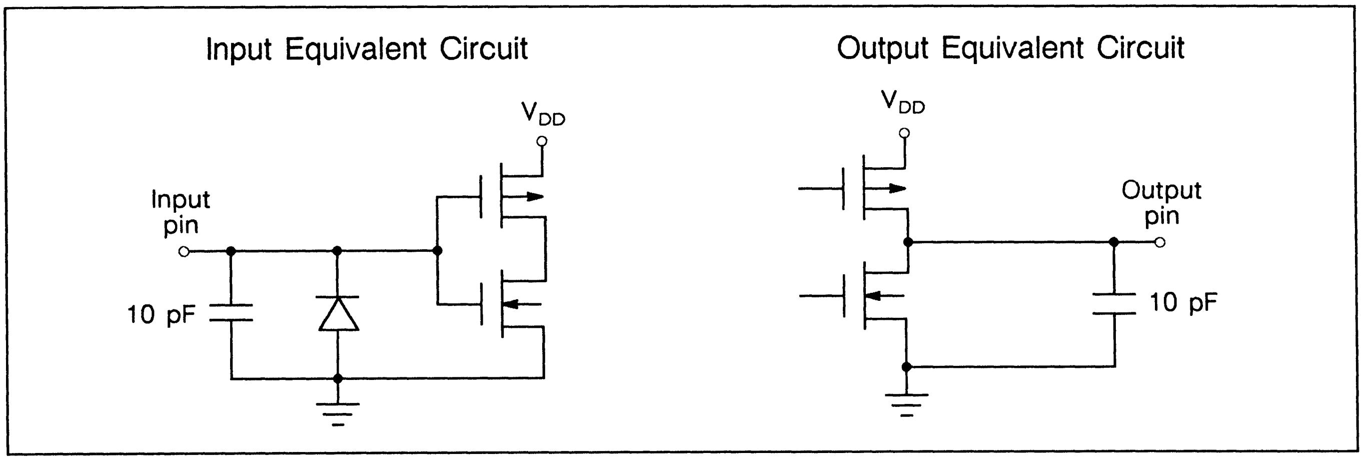
CL = total load on device pin, including stray capacitance.

CL = total load on device pin, including stray capacitance.

Vx = Vcc to test TZL and TLZ.

Vx = GND to test THZ and TZH. .

Figure 18. Test Load For Delay Measurement



Input Equivalent Circuit

Output Equivalent Circuit

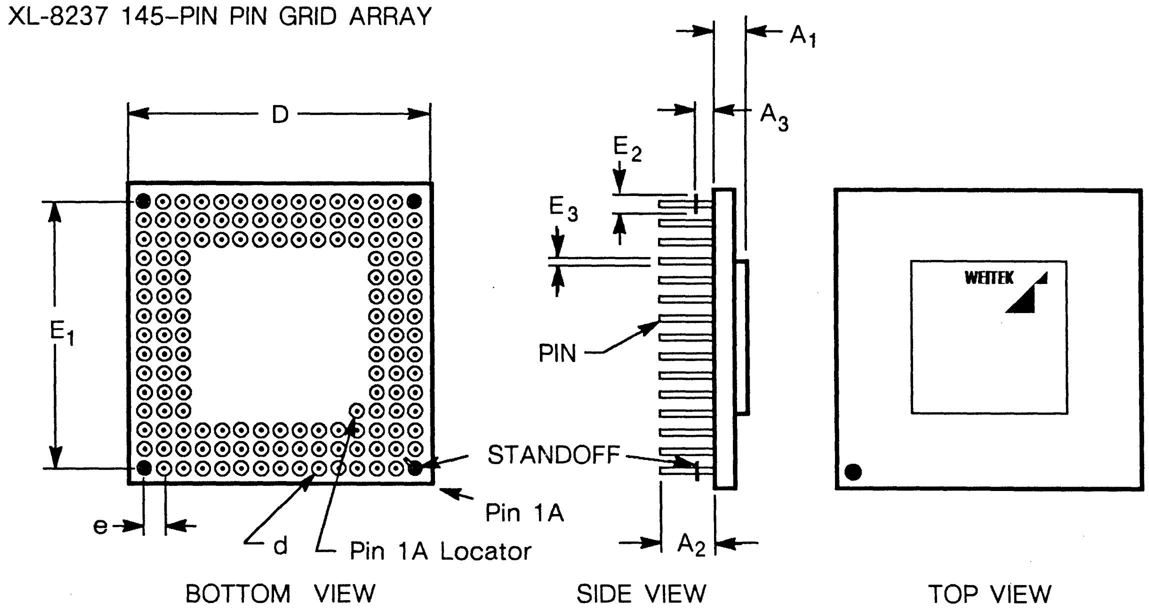
Figure 22. Input and output equivalent circuits

Pin Configuration

15	VCC	D26	AD24	C24	NC	D23	AD22	AD21	D21	AD20	AD19	AD18	D18	D17	D16
14	VCC	GND	AD25	D25	D24	AD23	C23	C21	D20	C19	C18	C17	AD16	VCC	OED–
13	AD27	GND	AD26	C26	C25	COND	C22	D22	C20	D19	AD17	C16	GND	VCC	GND
12	D28	D27	GND										OEA–	VCC	TIE LOW
11	AD28	NC	C27										TIE HIGH	GND	AD15
10	C29	NC	C28										C15	D15	C14
9	NC	NC	D29										NC	AD14	D14
8	D30	AD29	NC					Top View (cavity up)					D13	AD13	C13
7	AD30	D31	C30										NC	C12	NC
6	NC	NC	C31										D11	NC	D12
5	AD31	NC	WREN 1–										NC	AD11	AD12
4	GND	WREN 2–	NC	KEY PIN									VCC	STALL–	C11
3	WREN 3–	WREN 0–	MDCLK	GND	AD0	D2	C3	D5	C5	D7	D9	D10	C10	NEUT–	NC
2	VCC	CLK	GND	C0	C1	C2	D3	C4	C6	AD6	AD7	AD8	C9	AD10	TIE HIGH
1	VCC	D0	D1	AD1	AD2	AD3	D4	AD4	AD5	D6	C7	D8	C8	AD9	GND
	A	B	C	D	E	F	G	H	J	K	L	M	N	P	R

Figure 23. Pin configuration (pinouts are identical for ceramic and plastic pin-grid array packages)

Physical Dimensions



Symbol	INCHES		MM	
	MAX	MIN	MAX	MIN
A1	0.135	0.080	3.43	2.03
A2	0.210	0.175	5.33	4.46
A3	0.080	0.040	2.03	1.14
D	1.657	1.555	42.1	39.4
E1	0.140 TYP		3.56 TYP	
E2	0.050 TYP		1.27 TYP	
E3	0.020	0.016	0.51	0.41
d	0.075	0.035	1.91	0.89
e	0.100 TYP		2.54	

Figure 24. XL-8237 physical dimensions

Ordering Information

PACKAGE TYPE	SPEED	TEMP. RANGE (CASE)	ORDER NUMBER
145-pin plastic PGA	–10	T = 0°C to 85°C	XL-8237-010-GPU
145-pin plastic PGA	–20	T = 0°C to 85°C	XL-8237-020-GPU
145-pin plastic PGA	–40	T = 0°C to 85°C	XL-8237-040-GPU
145-pin plastic PGA	–60	T = 0°C to 85°C	XL-8237-060-GPU

PACKAGE TYPE	SPEED	TEMP. RANGE (CASE)	ORDER NUMBER
145-pin ceramic PGA	–10	T = 0°C to 85°C	XL-8237-010-GCU
145-pin ceramic PGA	–20	T = 0°C to 85°C	XL-8237-020-GCU
145-pin ceramic PGA	–40	T = 0°C to 85°C	XL-8237-040-GCU
145-pin ceramic PGA	–60	T = 0°C to 85°C	XL-8237-060-GCU

Revision Summary

T_{16} was corrected in the AC timing diagram

T_R and T_F were added to the AC Specifications table

T_H was split into T_{H1} and T_{H2}

The –60 part grade was added to the Order ing Information section

New examples of the divide instruction were added

Typographical errors were corrected

XL-8237
32-BIT RASTER
IMAGE PROCESSOR

PRELIMINARY DATA
October 1988

For additional information on WEITEK products, please fill out the form below and mail.

Name ____________________ Title ____________________

Company ____________________ Phone ____________________

Address ____

Comments ____

I am currently involved in a design with the following Weitek products ____________________ and wish to be added to your design data base to insure that I receive status updates.

APPLICATION:

□ ENGINEERING WORKSTATIONS

□ GRAPHICS

□ PERSONAL COMPUTERS

□ SCIENTIFIC COMPUTERS

□ OTHER ____________________

Check the products on which you wish to receive data sheets:

□ Have a sales person call

ATTACHED PROCESSORS

□ XL-SERIES OVERVIEW

□ XL-8200 OVERVIEW

COPROCESSORS

□ 1167

□ 1164/1165

□ 3164/3364

□ 3167

BUILDING BLOCKS

□ 2264/2265

□ 3132/3332

□ 1232/1233

□ 1066

□ 2010

□ 2245

□ 2516

□ 2517

WEITEK use: Rec'd Out TPT Source: DS

Status ____

WEITEK XL-8237 32-BIT RASTER IMAGE PROCESSOR

Please Comment On The Quality Of This Data Sheet.

Even though we have tried to make this data sheet as complete as possible, it is conceivable that we have missed something that may be important to you. If you believe this is the case, please describe what the missing information is, and we will consider including it in the next printing of the data sheet.

Fold, Staple and Mail to Weitek Corp.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1374 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

WEITEK Corporation
1060 E. Arques Ave.
Sunnyvale, CA 94086-BRM-9759

ATTN: Ed Masuda

WEITEK

WEITEK'S CUSTOMER COMMITMENT:

Weitek's mission is simple: to provide you with VLSI solutions to solve your compute-intensive problems. We translate that mission into the following corporate objectives:

1. To be first to market with performance breakthroughs, allowing you to develop and market systems at the edge of your art.

2. To understand your product, technology, and market needs, so that we can develop Weitek products and corporate plans that will help you succeed.

3. To price our products based on the fair value they represent to you, our customers.

4. To invest far in excess of the industry average in Research and Development, giving you the latest products through technological innovation.

5. To invest far in excess of the industry average in Selling, Marketing, and Technical Applications Support, in order to provide you with service and support unmatched in the industry.

6. To serve as a reliable, resourceful, and quality business partner to our customers.

These are our objectives. We're committed to making them happen. If you have comments or suggestions on how we can do more for you, please don't hesitate to contact us.

Art Collmeyer
President

Headquarters
Weitek Corporation
1060 E. Arques Avenue
Sunnyvale, CA 94086
TWX 910-339-9545
WEITEK SVL
FAX (408) 738-1185
TEL (408) 738-8400

Domestic Sales Offices
Weitek Corporation
1060 E. Arques Avenue
Sunnyvale, CA 94086
TWX 910-339-9545
WEITEK SVL
FAX (408) 738-1185
TEL (408) 738-8400

Corporate Place IV
111 South Bedford St.
Suite 200
Burlington, MA 01803
FAX (617) 229-4902
TEL (617) 229-8080

European Sales Headquarters
Greyhound House, 23/24 George St.
Richmond, Surrey, TW9 1JY
England
TELEX 928940 RICHBI G
FAX 011-441 940 6208
TEL 011-441 549 0164

Japanese Representative
4-8-1 Tsuchihashi
Miyamae-Ku
Kawasaki, Kanagawa-Pre
213 Japan
FAX 044-877-4268
TEL 044-852-1135