Memory Management
Applications Handbook

Contributors

VLSI Logic Applications Engineering
VLSI Marketing Department

TEXAS
INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes in the devices or the device specifications identified in this publication without notice. TI advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems such testing necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

In the absence of written agreement to the contrary, TI assumes no liability for TI applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Contents

ABEL is a trademark of DATA I/O
CUPL is a trademark of Personal CAD Systems, Inc.

List of Illustrations

List of Tables

1 Introduction

Texas Instruments (TI) is pleased to make available this collection of application reports and application briefs on several of our new single-chip memory management products. Designed to help you attain your systems goals, these products can help you achieve minimal memory access times for maximum system throughput. While there may be other alternatives available, few can provide speed enhancement while reducing both design effort and component cost. Many of the new memory management products from TI accelerate the performance of VME, VERSAbus®, MULTIBUS®, and PC Bus architectures even further.

Single chip solutions to complex functions such as cache tag control, DRAM address multiplexing and refresh, and soft error correction are all part of the TI memory management family of products. This family offers many benefits such as greater ease of use, improved system performance, and a reduction in cost over discrete logic options.

This document contains information on each of the new memory management products from TI. For example, the cache tag application report details how you can store frequently accessed data and instructions in a few high speed SRAMs and then "tag" them by using the TI family of CMOS cache controllers to achieve high speed system throughput.

High-performance DRAM controllers are discussed in "Memory Timing Controllers Using the 'ALS2967/'ALS2968 & 'ALS6301/'ALS6302." The TI SN74ALS6301 DRAM controller incorporates address multiplexing and refresh circuitry in a single chip. The wide address capability and IMPACT™ speed performance of the SN74ALS6301 allow it to support the newer 1M-bit DRAM chips.

Maintaining data integrity in larger memory arrays can be accomplished using error detection and correction circuits. The application report on the TI SN74AS632 32-bit Error Detection and Correction (EDAC) circuit describes how the 'AS632 can detect both hard and soft errors in memory arrays and guarantee system reliability by correcting these errors while avoiding processor wait states.

Every memory management function from TI uses a "universal architecture" design to allow for easy compatibility with any microprocessor, keeping design effort to a minimum. These single-chip solutions are designed to be especially easy to use with Motorola VME and VERSAbus® architectures as well as Intel MULTIBUS®, and PC Bus based systems.

2 Memory Timing Controllers

2.1 Introduction

As processor and memory speeds increase, so do dynamic memory controller requirements. Typical processor speeds today range from 8 to 10 MHz. This increase in processor speed has created a need for faster memories, as well as faster memory timing controllers. The 'ALS2967, 'ALS2968, 'ALS6301, and 'ALS6302 are Memory Timing Controllers that are designed to meet the need of high performance memory systems.

In addition to offering better system performance, a faster memory controller typically allows the designer to use slower-rated dynamic random access memories (DRAMs). This results in significant cost savings because of the large number of DRAMs required. In other words, a faster dynamic memory controller can reduce overall dynamic memory costs.

The 'ALS2967, 'ALS2968, 'ALS6301, and 'ALS6302 feature address multiplexing, memory bank selection, and an address latch for systems which multiplex both data and address on the same bus. A row counter is provided for normal refresh operations. Column and bank counters are available for systems which use memory scrubbing.

This Section describes the functional operation of the 'ALS2967, 'ALS2968, 'ALS6301, and 'ALS6302 and shows how they can be interfaced to a typical processor. For illustration purposes, a simple timing controller generated from programmable logic is used to interface both the 'ALS2967 and the 'ALS6301 to the microprocessor. The 'ALS2967 is interfaced with an Intel 8086 and the 'ALS6301 is interfaced with a Motorola 68000.

This Section also presents a circuit configuration which interfaces the MC68000 to DRAM memory using the THCT4502B dynamic RAM Controller. The memory array is organized as 4 banks of 256K memory (TMS4256/4257) providing a 1M byte deep system architecture.

2.2 Memory Timing Controllers Using the SN54/74ALS2967, SN54/74ALS2968

2.2.1 Functional Description

The 'ALS2967 and 'ALS2968 are capable of controlling 16K, 64K, and 256K DRAMs. The two devices typically operate in a read/write or a refresh mode. During normal read/write operations, the row and column addresses are multiplexed to the DRAM, and the corresponding $\overline{RAS}$ and $\overline{CAS}$ signals are activated to strobe the addresses into memory. In the refresh mode, the two counters cycle through the refresh addresses. If memory scrubbing is not being implemented, only the row counter is used. When memory scrubbing is being performed, both the row and column counters are used to perform read-modify-write cycles using an error detection and correction circuit such as the 'ALS632A. In this mode, all $\overline{RAS}$ outputs will be active (low) while only one $\overline{CAS}$ output is active at a time.

Two device types are offered to help simplify interfacing with the system dynamic timing controller. The 'ALS2967 offers active-low row address strobe input ($\overline{RASI}$) and column address strobe input ($\overline{CASI}$) signals, while the 'ALS2968 offers active-high RASI and CASI inputs. Figure 2-1 is a functional block diagram of the two devices.

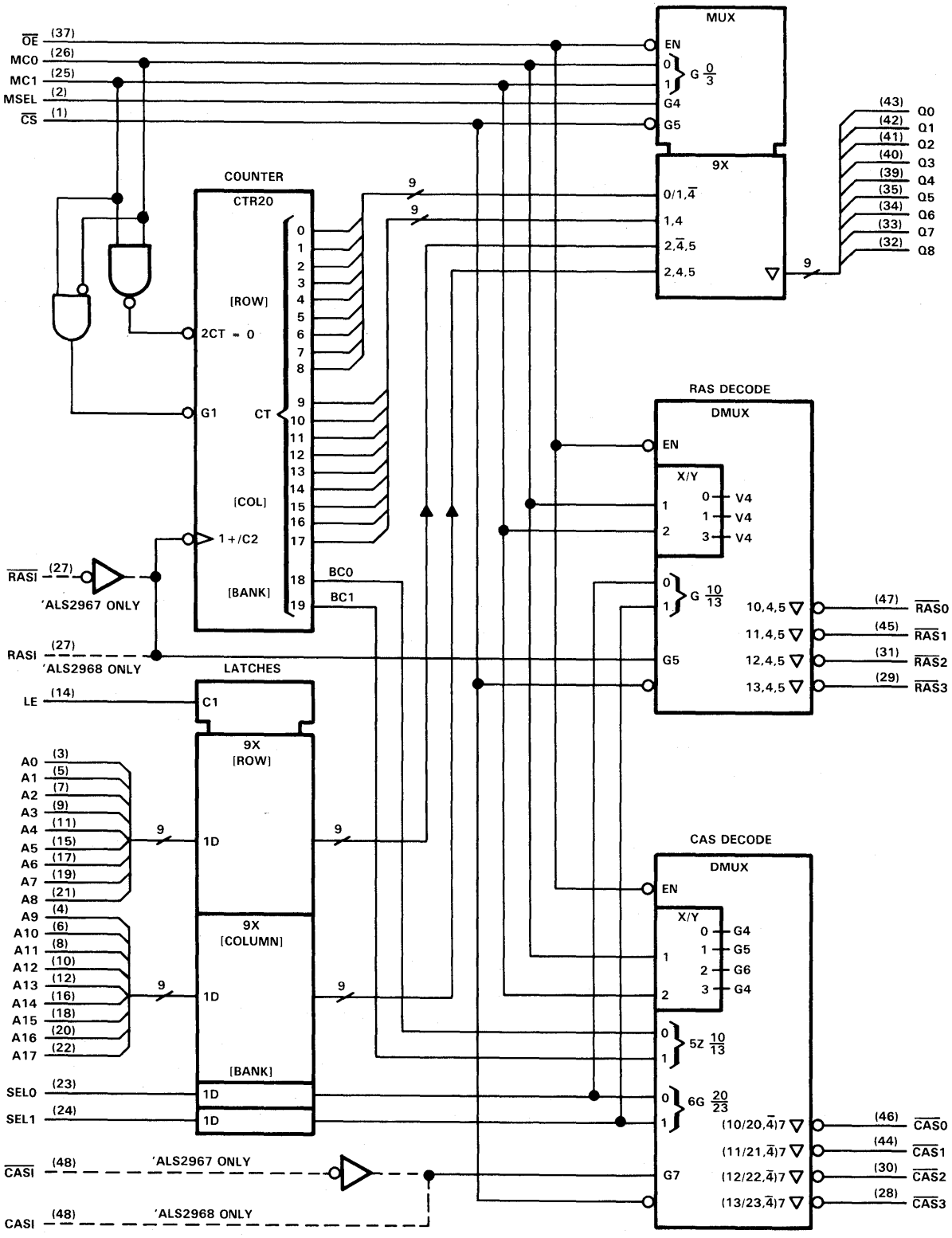


Figure 2-1. 'ALS2967, 'ALS2968 Functional Block Diagram

Table 2-1 describes the four operating modes of the 'ALS2967 and 'ALS2968 as controlled by inputs MC0 and MC1. During normal read/write operations, the row and column addresses are multiplexed to the DRAM. When MSEL is high, the column address is selected; when MSEL is low, the row address is selected. The corresponding $\overline{RAS}$n and $\overline{CAS}$n output signals strobe the addresses into the selected memory bank or banks. A single 'ALS2967 or 'ALS2968 can control as many as four banks of 256K memory. Additional banks of memory can be controlled by using additional 'ALS2967 or 'ALS2968 devices and decoding each chip select ($\overline{CS}$) input.

Table 2-1. 'ALS2967, 'ALS2968 Mode-Control Function Table

SIGNAL		MODE SELECTED
MC1	MC0	
L	L	Refresh without Scrubbing. Refresh cycles are performed using the row counter to generate the addresses. In this mode, all four $\overline{RAS}$ outputs are active while the four $\overline{CAS}$ outputs remain high.
L	H	Refresh with Scrubbing/Initialize. Refresh cycles are performed using both the row and column counters to generate the addresses. MSEL selects the row or the column counter. All four $\overline{RAS}$ outputs go low in response to $\overline{RASI}$ ('ALS2967) or RASI ('ALS2968), while only one $\overline{CAS}$n output goes low in response to $\overline{CASI}$ ('ALS2967) or CASI ('ALS2968). The bank counter keeps track of which $\overline{CAS}$ output goes active. This mode can also be used during system power-up so that the memory can be written with a known data pattern.
H	L	Read/Write. This mode is used to perform read/write cycles. Both the row and column addresses are multiplexed to the address output lines using MSEL. SEL0 and SEL1 are decoded to determine which $\overline{RAS}$n and $\overline{CAS}$n outputs will be active.
H	H	Clear Refresh Counters. This mode clears the three refresh counters (row, column, and bank) on the inactive transition of $\overline{RASI}$ ('ALS2967) or RASI ('ALS2968), putting them at the beginning of the refresh sequence. In this mode, all four $\overline{RAS}$ outputs are driven low after the active edge of $\overline{RASI}$ ('ALS2967) or RASI ('ALS2968) so that DRAM wake-up cycles can also be performed.

In systems where addresses and data are both multiplexed onto a single bus, the 'ALS2967 and 'ALS2968 use latches (row, column and bank) to hold the address information. The 20 input latches are transparent when the latch enable input (LE) is high; the input data is latched whenever LE goes low. For systems in which the processor has separate address and data buses, LE may be tied high.

The two 9-bit counters in the 'ALS2967 and 'ALS2968 support 128, 256, and 512 line refresh operations. Transparent, burst, synchronous, or asynchronous refresh modes are all possible as determined by the memory timing controller. The refresh counters are advanced on the low-to-high transition of $\overline{RASI}$ on the 'ALS2967, and on the high-to-low transition of RASI on the 'ALS2968. This is true in either refresh mode. In the clear refresh counter mode, the refresh counters (row, column, and bank) can be reset to zero on the low-to-high transition of $\overline{RASI}$ on the 'ALS2967 or on the high-to-low transition of RASI on the 'ALS2968.

2.2.2 Typical Implementation

Figure 2-2 shows a system interface using the 'ALS2967 between an Intel 8086 and four banks of 256K DRAMs. Addresses A18 and A19 are used to select one of the four memory banks. Since members of the 8086 processor family multiplex both data and addresses onto the same data bus, input latches on the 'ALS2967 must be used to store the row, column, and bank information. The ALE signal from the 8086 can be directly connected to the latch enable (LE) input on the 'ALS2967.

The $\overline{RASI}$, $\overline{CASI}$, MSEL and mode control (MC0, MC1) inputs on the 'ALS2967 must be generated by the memory timing controller. The memory timing controller functions as an arbitrator between refresh cycles and 8086 access cycles. It also guarantees that timing requirements of the DRAM will be met.

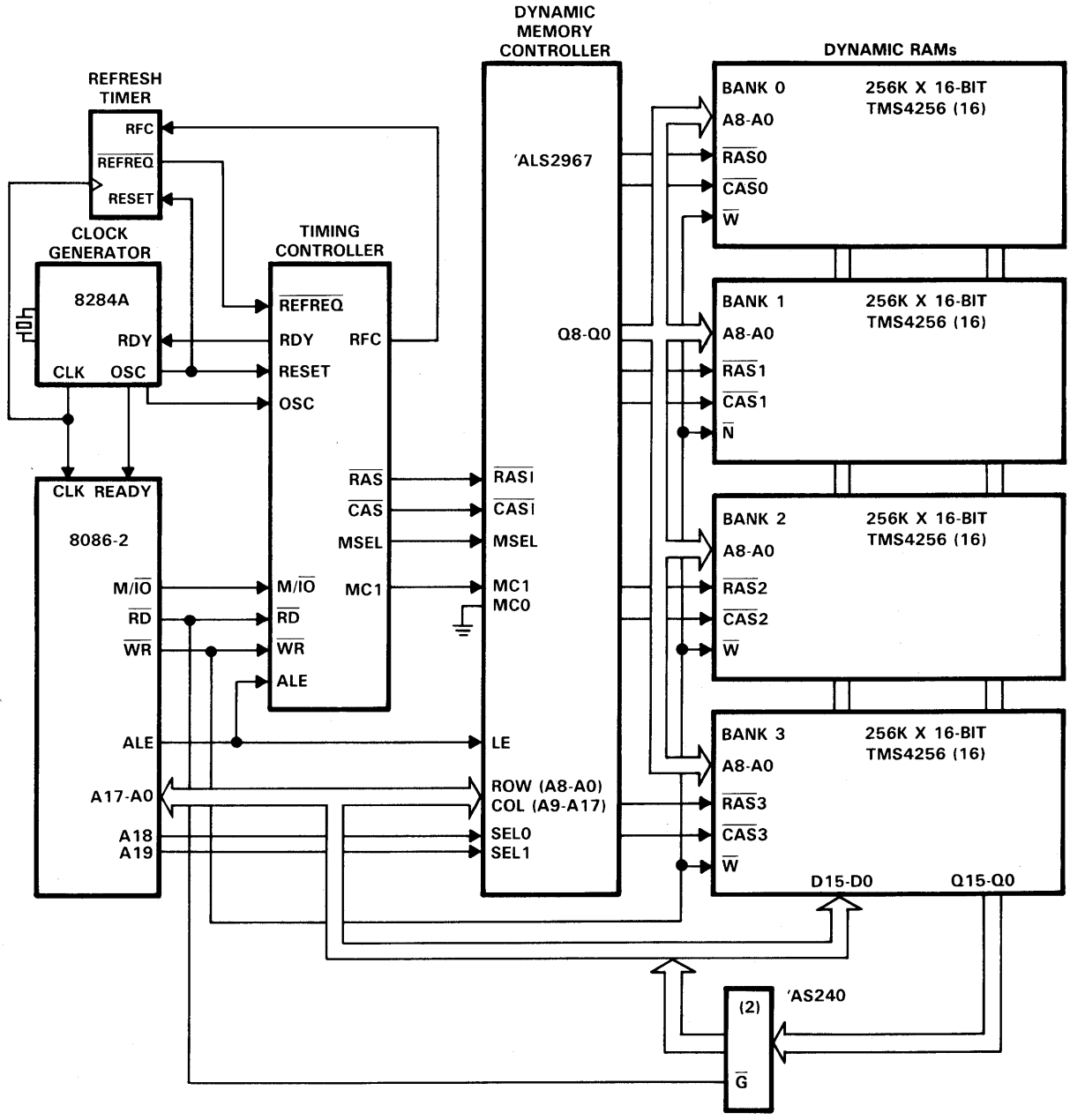


Figure 2-2. 'ALS2967, 'ALS2968 Timing Controller Interface

2.2.3 Timing Controller Details

Figure 2-3 is a timing diagram for a typical 8086 access cycle. The 'ALS2967 control signals required to execute the access cycle are also shown. Control signals for the 'ALS2967 are referenced from the OSC output of the 8284A clock generator. The timing controller in this example is generated from a state machine referenced from the OSC output of the 8284A. In critical timing situations, it may be necessary to tightly control the phase relationship of the system clock to the OSC signal. This can be accomplished by using a phase lock loop or similar method to generate the OSC signal.

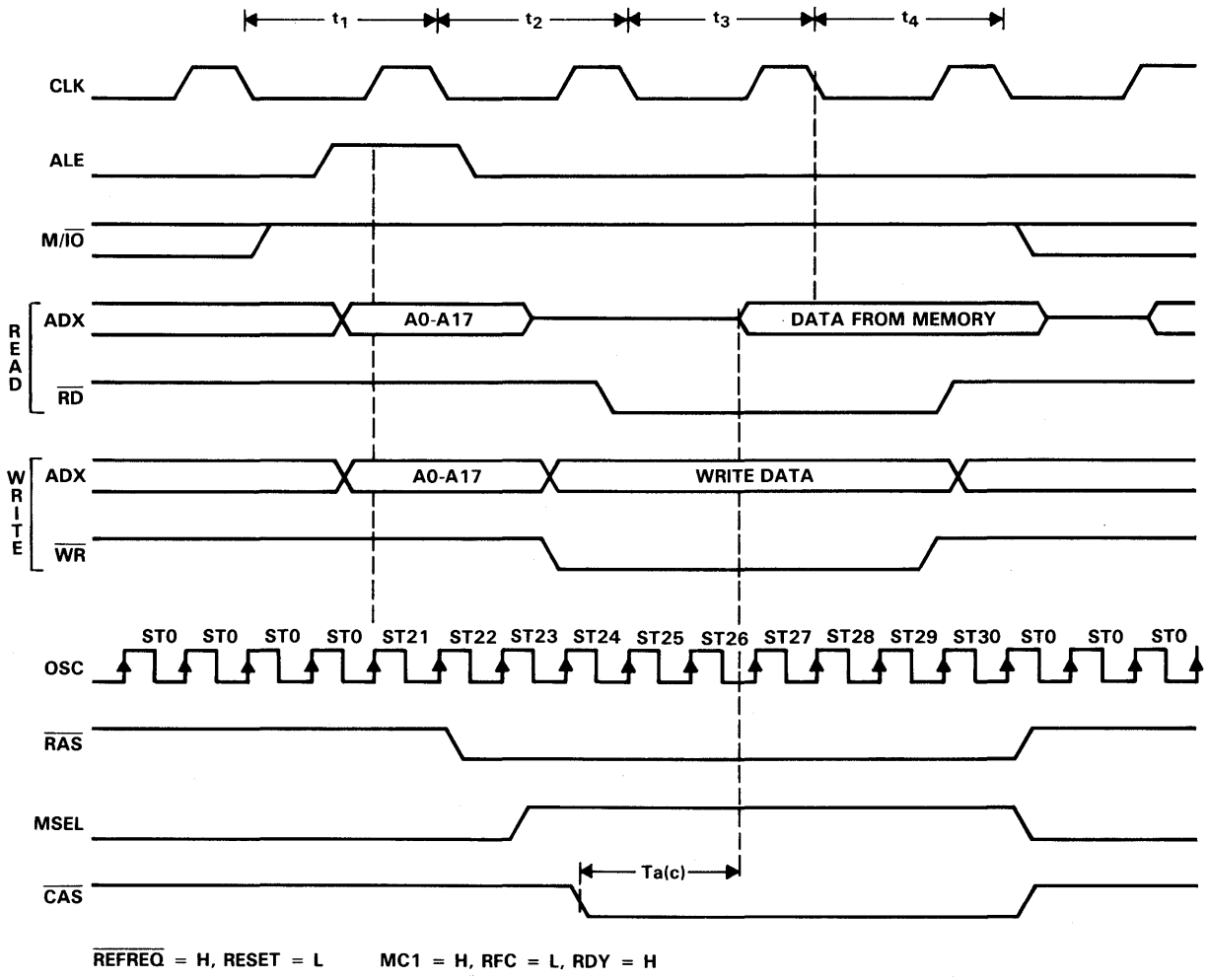


Figure 2-3. 8086 Access Cycle

In this example, refresh requests ($\overline{\text{REFREQ}}$) are generated every 122 clock cycles. The timing controller will perform the refresh cycle ($\overline{\text{RAS}}$ only) immediately if the processor is not in the middle of an access cycle. If the controller is in the middle of an access cycle, the refresh cycle will be delayed until the access cycle is complete. If the controller is asked to perform an access cycle during a refresh, the controller will place the processor in a wait state (RDY low) until the refresh is complete. Figure 2-4 shows the timing diagram for a refresh/access cycle as explained above. To implement memory scrubbing, the controller must execute a read/write cycle during the refresh cycle and then place the 'ALS2967 in the memory scrubbing mode (This example executes $\overline{\text{RAS}}$ only refresh).

Figure 2-5 is a flow chart for the timing controller. ABEL™ and CUPL™ software was used to generate fuse maps from the present state of the inputs and present condition of the state machine. These fuse maps were then used in programming the field programmable logic devices. The files used to generate the fuse maps have been included for reference at the end of this application note.

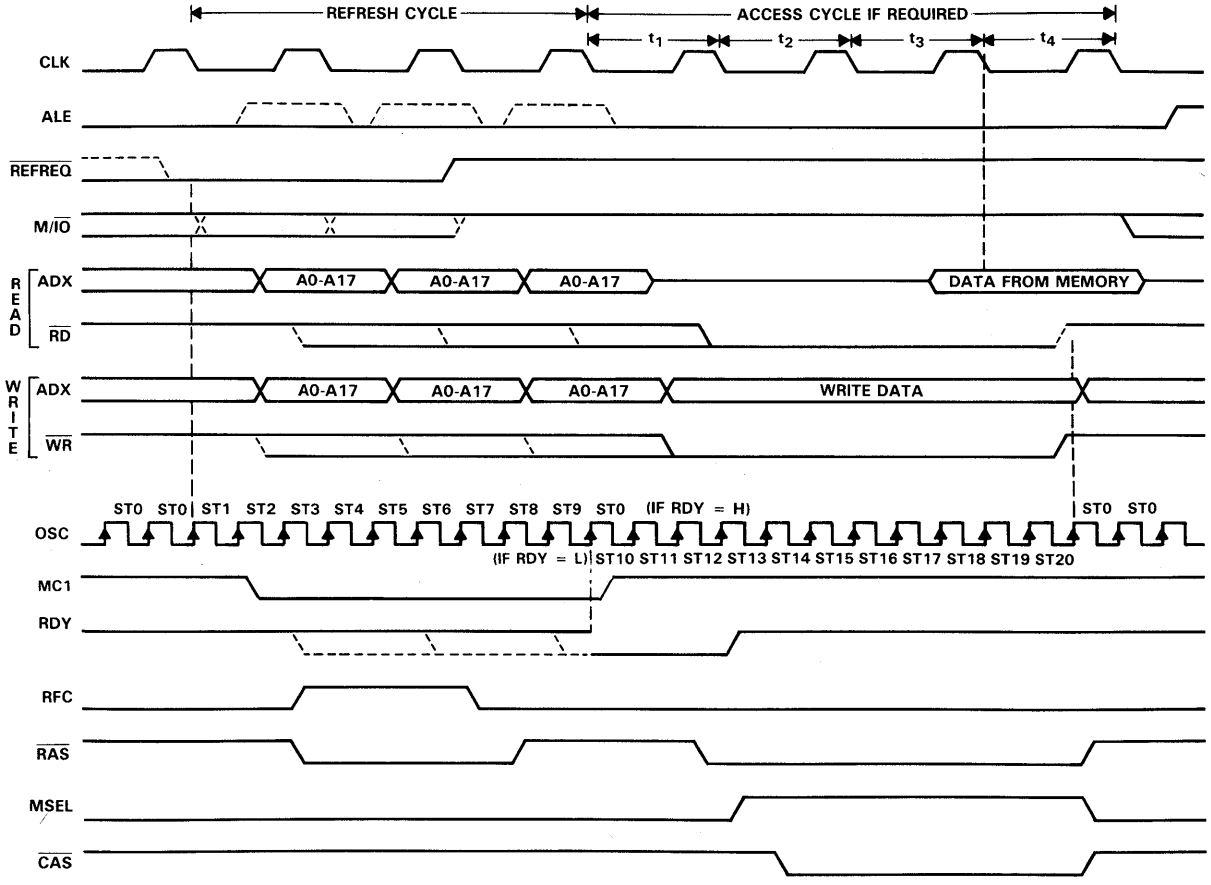


Figure 2-4. Refresh/Access Cycle

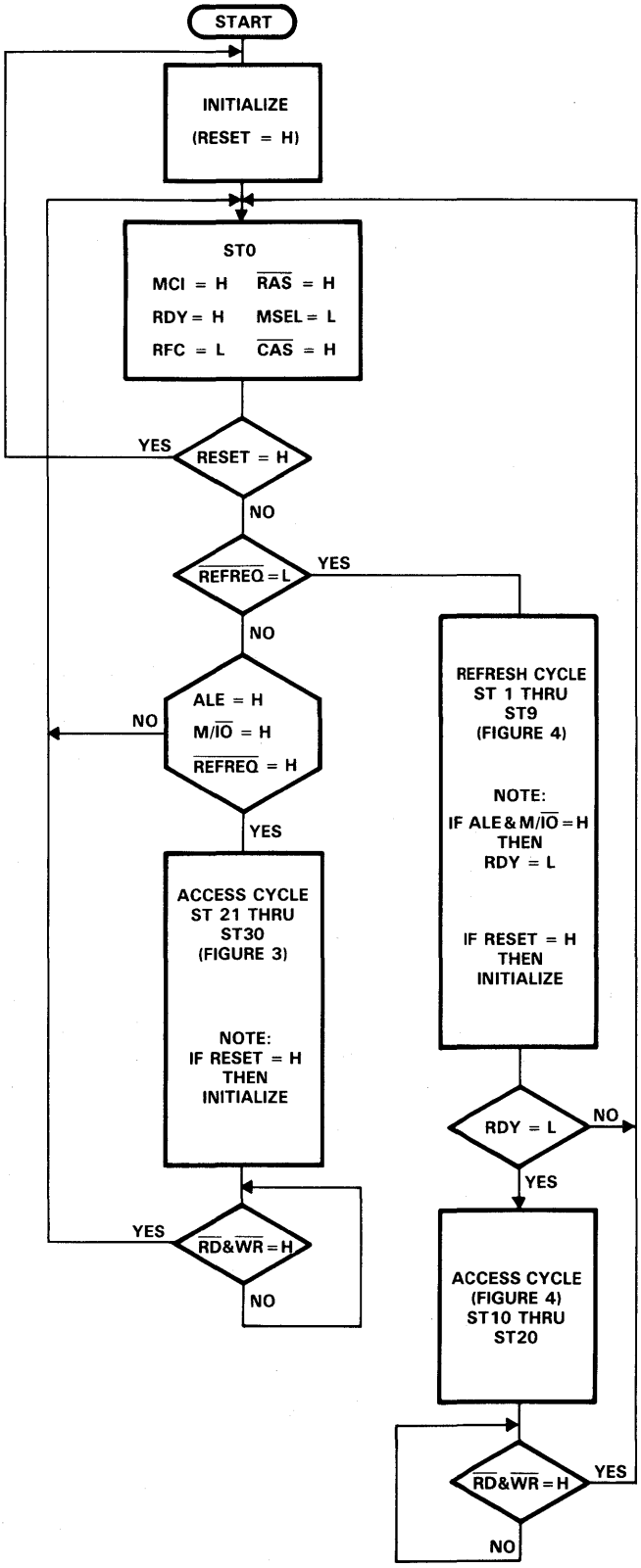


Figure 2-5. 'ALS2967, 'ALS2968 Memory Timing Controller Flow Chart

Figure 2-6 shows the actual circuit implementation of the refresh and memory timing controller. The refresh timer signals the controller whenever it is time to execute a refresh cycle. As required by memory, every row (256 on the TMS4256 DRAM) must be addressed every 4 ms. This implies that one row should be refreshed at least once every 15.6 μs. With an 8-MHz system clock, the refresh timer should use approximately a division factor of 122. This results in a refresh request every 15.3 μs. The refresh complete input (RFC) is used to signal the refresh timer that the refresh has been completed. It is important that the timer not stop so that the 4-ms memory requirement is maintained.

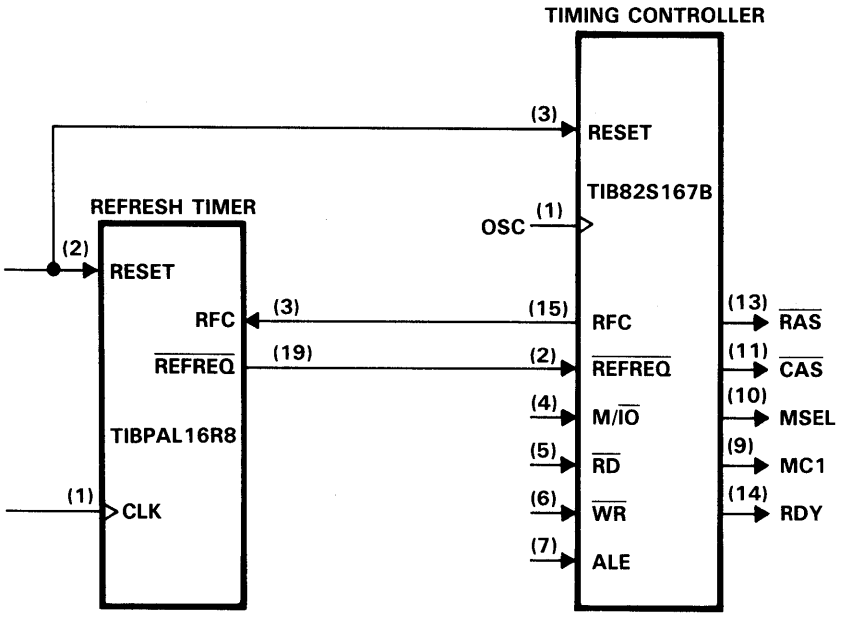


Figure 2-6. Refresh/Memory Timing Controller

The TIBPAL16R8 circuit shown in Figure 2-6 is used to generate the refresh request signal every 122 clock cycles. The refresh request signal (active low) will remain active (low) until a refresh complete (RFC) signal is received from the timing controller. During a system reset, the refresh request output is set to a high-logic level. When using different clock rates or memory sizes, the division circuit in the refresh timer should be adjusted accordingly.

The TIB82S167B field programmable sequencer shown in Figure 2-6 is configured as a state machine to execute the flow chart shown in Figure 2-5. In cases with different system timings, the CUPL™ file can be modified to fit the processor requirements. In addition, a slight modification to the file will allow an 'ALS2968 to be used instead of an 'ALS2967.

A preprogrammed sample of the refresh and timing controllers shown in Figure 2-6 can be obtained by calling PAL/PROM Applications, 214/995-2980.

2.2.4 Summary

The 'ALS2967 and 'ALS2968, coupled with programmable logic, offer the system designer a solution to high-speed dynamic memory requirements. Programmable logic allows the designer to tailor the timing controller to a selected processor and memory. In many cases, the generation of a high-speed timing controller from programmable logic will allow the designer to use slower DRAMs without affecting system speed. This results in lower total system cost because of the large number of memory devices used.

2.2.5 ABEL™ and CUPL™ Files

2.2.5.1 ABEL™ File

```
module  RF_TIMER
title  'REFRESH TIMER
        R. K. BREUNINGER  TEXAS INSTRUMENTS, DALLAS, 08/12/86'

        RFT   DEVICE   'P16R8';

"input declarations

CLK             pin 1;              "  SYSTEM CLOCK (8086)
RESET           pin 2;              "  RESETS WHEN HIGH
RFC             pin 3;              "  REFRESH COMPLETE
OE              pin 11;             "  MUST BE TIED LOW

"output declarations

Q0,Q1,Q2        pin 12,13,14;       "  COUNTER STATES
Q3,Q4,Q5,Q6     pin 15,16,17,18;    "  COUNTER STATES
REFREQ_         pin 19;             "  REFRESH REQUEST - ACTIVE LOW

"intermediate variables

CNT_REF   =  Q0 & !Q1 & !Q2 & Q3 & Q4 & Q5 & Q6;
SCLR      =  RESET # CNT_REF;
count     = [Q6,Q5,Q4,Q3,Q2,Q1,Q0];
C,H,L     = .C.,1,0;

equations

REFREQ_   :=  RFC # !CNT_REF &  REFREQ_ #  RESET;
Q0        :=  (!Q0 ) & !SCLR;
Q1        :=  ( Q1 $  Q0) & !SCLR;
Q2        :=  ( Q2 $  Q1 &  Q0) & !SCLR;
Q3        :=  ( Q3 $ (Q2 &  Q1 &  Q0)) & !SCLR;
Q4        :=  ( Q4 $ (Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;
Q5        :=  ( Q5 $ (Q4 &  Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;
Q6        := !(!Q5 & !Q6 # !Q4 & !Q6 # !Q3 & !Q6
              # !Q0 & !Q6 # !Q2 & !Q6 # !Q1 & !Q6 # SCLR);

test_vectors                    ([OE,RESET,CLK,RFC] -> [count,REFREQ_])
                                 [ 0,  1  , C , 0 ] -> [  0  ,   H   ];
@CONST cnt = 1; @REPEAT 121   { [ 0,  0  , C , 0 ] -> [ cnt ,   H   ];
                                 @CONST cnt = cnt + 1;}

                                 [ 0,  0  , C , 0 ] -> [  0  ,   L   ];
@CONST cnt = 1; @REPEAT 20    { [ 0,  0  , C , 0 ] -> [ cnt ,   L   ];
                                 @CONST cnt = cnt + 1;}

                                 [ 0,  0  , C , 1 ] -> [ 21  ,   H   ];
                                 [ 0,  0  , C , 0 ] -> [ 22  ,   H   ];
                                 [ 0,  0  , C , 0 ] -> [ 23  ,   H   ];
                                 [ 0,  0  , C , 0 ] -> [ 24  ,   H   ];

end RF_TIMER
```

2.2.5.2 *CUPL™ Source File*

```
Partno          MTC-S167;
Name            MTC-S167;
Date            08/13/86;
Revision        03;
Designer        BREUNINGER;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*                  DYNAMIC TIMING CONTROLER                     */
/*                      (FOR ALS2967)                            */
/*****************************************************************/
/* Allowable Target Device Types: TIB82S167B                     */
/*****************************************************************/

/**  Inputs  **/
pin 1 =  OSC;              /*  OSCILLATOR (8284A)                */
pin 2 = REFREQ;            /*  REFRESH REQUEST                   */
pin 3 = RESET;             /*  RESET - INITIALIZES WHEN HIGH     */
pin 4 = MIO;               /*  MEMORY I/O                        */
pin 5 = RD;                /*  READ                              */
pin 6 = WR;                /*  WRITE                             */
pin 7 = ALE;               /*  ADDRESS LATCH ENABLE              */
pin 16 = GND;              /*  PIN 16 MUST BE TIED LOW           */

/**  Outputs  **/
pin  9 = MC1_;             /*  MODE CONTROL                      */
pin 10 = MSEL;             /*  MULTIPLEXER SELECT                */
pin 11 = CAS;              /*  COLUMN ADDRESS STROBE             */
pin 13 = RAS;              /*  ROW ADDRESS STOBE                 */
pin 14 = RDY;              /*  READY                             */
pin 15 = RFC;              /*  REFRESH COMPLETE                  */

/** Internal Node Group  -   State bits declared as nodes       **/
node [P4_,P3_,P2_,P1_,P0_];

/** Declarations and Intermediate Variable Definitions          **/
Field State  = [P4_,P3_,P2_,P1_,P0_];

$define  ST0 'b'00000
$define  ST1 'b'00001
$define  ST2 'b'00010
$define  ST3 'b'00011
$define  ST4 'b'00100
$define  ST5 'b'00101
$define  ST6 'b'00110
$define  ST7 'b'00111
$define  ST8 'b'01000
$define  ST9 'b'01001
$define ST10 'b'01010
$define ST11 'b'01011
$define ST12 'b'01100
```

```
$define ST13 'b'01101
$define ST14 'b'01110
$define ST15 'b'01111
$define ST16 'b'10000
$define ST17 'b'10001
$define ST18 'b'10010
$define ST19 'b'10011
$define ST20 'b'10100
$define ST21 'b'10101
$define ST22 'b'10110
$define ST23 'b'10111
$define ST24 'b'11000
$define ST25 'b'11001
$define ST26 'b'11010
$define ST27 'b'11011
$define ST28 'b'11100
$define ST29 'b'11101
$define ST30 'b'11110
$define ST31 'b'11111

/** Logic Equations **/
Sequence State
{Present ST0 IF  RESET            NEXT ST0 OUT [MC1_, RDY,!RFC, RAS,!MSEL, CAS];
             IF !RESET & !REFREQ  NEXT ST1;
             IF !RESET &  REFREQ & ALE & MIO NEXT ST21;
             DEFAULT              NEXT ST0;

/** REFRESH CYCLE **/
Present ST1 IF  ALE & MIO&!RESET NEXT ST2 OUT [!MC1_,!RDY];
            IF !RESET            NEXT ST2 OUT [!MC1_];
Present ST2 IF  ALE & MIO&!RESET NEXT ST3 OUT [!RDY, RFC,!RAS];
            IF !RESET            NEXT ST3 OUT [ RFC,!RAS];
Present ST3 IF  ALE & MIO&!RESET NEXT ST4 OUT [!RDY];
            IF !RESET            NEXT ST4;
Present ST4 IF  ALE & MIO&!RESET NEXT ST5 OUT [!RDY];
            IF !RESET            NEXT ST5;
Present ST5 IF  ALE & MIO&!RESET NEXT ST6 OUT [!RDY];
            IF !RESET            NEXT ST6;
Present ST6 IF  ALE & MIO&!RESET NEXT ST7 OUT [!RDY,!RFC];
            IF !RESET            NEXT ST7 OUT [!RFC];
Present ST7 IF  ALE & MIO&!RESET NEXT ST8 OUT [!RDY, RAS];
            IF !RESET            NEXT ST8 OUT [ RAS];
Present ST8 IF  ALE & MIO&!RESET NEXT ST9 OUT [!RDY];
            IF !RESET            NEXT ST9;
Present ST9 IF  RDY & !RESET     NEXT ST0 OUT [ MC1_, RDY,!RFC, RAS,!MSEL, CAS];
            IF !RDY & !RESET     NEXT ST10 OUT [ MC1_];
```

```
/** ACCESS IMMEDIATELY AFTER REFRESH WHEN REQUESTED **/
Present ST10 IF !RESET                NEXT ST11;
Present ST11 IF !RESET                NEXT ST12 OUT [!RAS];
Present ST12 IF !RESET                NEXT ST13 OUT [ RDY, MSEL];
Present ST13 IF !RESET                NEXT ST14 OUT [!CAS];
Present ST14 IF !RESET                NEXT ST15;
Present ST15 IF !RESET                NEXT ST16;
Present ST16 IF !RESET                NEXT ST17;
Present ST17 IF !RESET                NEXT ST18;
Present ST18 IF !RESET                NEXT ST19;
Present ST19 IF !RESET                NEXT ST20;
Present ST20 IF RD & WR & !RESET NEXT ST0 OUT [ MC1_, RDY,!RFC, RAS,!MSEL, CAS];
             IF !RESET                NEXT ST20;

/** ACCESS TIMING CYCLE **/
Present ST21 IF !RESET                NEXT ST22 OUT [!RAS];
Present ST22 IF !RESET                NEXT ST23 OUT [ MSEL];
Present ST23 IF !RESET                NEXT ST24 OUT [!CAS];
Present ST24 IF !RESET                NEXT ST25;
Present ST25 IF !RESET                NEXT ST26;
Present ST26 IF !RESET                NEXT ST27;
Present ST27 IF !RESET                NEXT ST28;
Present ST28 IF !RESET                NEXT ST29;
Present ST29 IF !RESET                NEXT ST30;
Present ST30 IF RD & WR & !RESET NEXT ST0 OUT [ MC1_, RDY,!RFC, RAS,!MSEL, CAS];
             IF !RESET                NEXT ST30;}

APPEND  MC1_.s = RESET;   APPEND  RDY.s = RESET;   APPEND  RFC.r = RESET;
APPEND  RAS.s = RESET;    APPEND  MSEL.r= RESET;   APPEND  CAS.s = RESET;
APPEND  P0_.r = RESET;    APPEND  P1_.r = RESET;   APPEND  P2_.r = RESET;
APPEND  P3_.r = RESET;    APPEND  P4_.r = RESET;
```

2.2.5.3 *CUPL™ Simulation File*

```
Partno          MTC-S167;
Name            MTC-S167;
Date            08/13/86;
Revision        03;
Designer        BREUNINGER;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*          DYNAMIC TIMING CONTROLLER SIMULATION FILE            */
/*                        (FOR ALS2967)                          */
/*****************************************************************/
/* Allowable Target Device Types: TIB82S167B                     */
/*****************************************************************/

ORDER:
GND,%3,OSC,%3,RESET,%6,REFREQ,%4,MIO,%3,RD,%2,WR,%2,ALE,%5,
MC1_,%4,MSEL,%3,CAS,%3,RAS,%3,RDY,%3,RFC;

VECTORS:

$msg"REFRESH WITH ACCESS FOLLOWING";
$msg"     ------------ INPUT --------------    -------- OUTPUT --------";
$msg"     GND OSC RESET REFREQ MIO RD WR ALE    MC1 MSEL CAS RAS RDY RFC";
$msg"     ---------------------------------------------------------------";
/*RESET*/ 0   C   1      X     X   X  X  X      H   L    H   H   H   L
/* ST0*/  0   C   0      0     X   X  X  X      H   L    H   H   H   L
/* ST1*/  0   C   0      X     0   X  X  0      L   L    H   H   H   L
/* ST2*/  0   C   0      X     0   X  X  0      L   L    H   L   H   H
/* ST3*/  0   C   0      X     0   X  X  0      L   L    H   L   H   H
/* ST4*/  0   C   0      X     1   X  X  1      L   L    H   L   L   H
/* ST5*/  0   C   0      X     X   X  X  X      L   L    H   L   L   H
/* ST6*/  0   C   0      X     X   X  X  X      L   L    H   L   L   L
/* ST7*/  0   C   0      X     X   X  X  X      L   L    H   H   L   L
/* ST8*/  0   C   0      X     X   X  X  X      L   L    H   H   L   L
/* ST9*/  0   C   0      X     X   X  X  X      H   L    H   H   L   L
/*ST10*/  0   C   0      1     X   X  X  X      H   L    H   H   L   L
/*ST11*/  0   C   0      X     X   X  X  X      H   L    H   L   L   L
/*ST12*/  0   C   0      X     X   X  X  X      H   H    H   L   H   L
/*ST13*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST14*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST15*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST16*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST17*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST18*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST19*/  0   C   0      X     X   X  X  X      H   H    L   L   H   L
/*ST20*/  0   C   0      X     X   0  0  X      H   H    L   L   H   L
/*ST20*/  0   C   0      X     X   1  1  X      H   L    H   H   H   L
```

```
$msg"    ";
$msg"REFRESH WITHOUT ACCESS FOLLOWING";
$msg"    ---------- INPUT ---------------    -------- OUTPUT --------";
$msg"    GND OSC RESET REFREQ MIO RD WR ALE   MC1 MSEL CAS RAS RDY RFC";
/*RESET*/ 0   C    1      X    X  X  X  X     H    L   H   H   H   L
/* ST0*/  0   C    0      0    X  X  X  X     H    L   H   H   H   L
/* ST1*/  0   C    0      X    0  X  X  0     L    L   H   H   H   L
/* ST2*/  0   C    0      X    0  X  X  0     L    L   H   L   H   H
/* ST3*/  0   C    0      X    0  X  X  0     L    L   H   L   H   H
/* ST4*/  0   C    0      X    0  X  X  0     L    L   H   L   H   H
/* ST5*/  0   C    0      X    0  X  X  0     L    L   H   L   H   H
/* ST6*/  0   C    0      X    0  X  X  0     L    L   H   L   H   L
/* ST7*/  0   C    0      X    0  X  X  0     L    L   H   H   H   L
/* ST8*/  0   C    0      X    0  X  X  0     L    L   H   H   H   L
/* ST9*/  0   C    0      X    0  X  X  0     H    L   H   H   H   L

$msg"    ";
$msg"ACCESS TIMING CYCLE ";
$msg"    ---------- INPUT ---------------    -------- OUTPUT --------";
$msg"    GND OSC RESET REFREQ MIO RD WR ALE   MC1 MSEL CAS RAS RDY RFC";
/*RESET*/ 0   C    1      X    X  X  X  X     H    L   H   H   H   L
/* ST0*/  0   C    0      1    1  X  X  1     H    L   H   H   H   L
/*ST21*/  0   C    0      X    X  X  X  X     H    L   H   L   H   L
/*ST22*/  0   C    0      X    X  X  X  X     H    H   H   L   H   L
/*ST23*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST24*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST25*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST26*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST27*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST28*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST29*/  0   C    0      X    X  X  X  X     H    H   L   L   H   L
/*ST30*/  0   C    0      X    X  0  0  X     H    H   L   L   H   L
/*ST30*/  0   C    0      X    X  1  1  X     H    L   H   H   H   L
```

2.3 Memory Timing Controllers Using the SN54/74ALS6301, SN54/74ALS6302

2.3.1 Functional Description

The 'ALS6301 and 'ALS6302 are capable of controlling any DRAM up to 1M. The two devices typically operate in a read/write or a refresh mode. During normal read/write operations, the row and column addresses are multiplexed to the DRAM, and the corresponding $\overline{RAS}$ and $\overline{CAS}$ signals are activated to strobe the addresses into memory. In the refresh mode, the two counters cycle through the refresh addresses. If memory scrubbing is not being implemented, only the row counter is used. When memory scrubbing is being performed, both the row and column counters are used to perform read-modify-write cycles using an error detection and correction circuit such as the 'ALS632A. In this mode, all $\overline{RAS}$ outputs will be active (low) while only one $\overline{CAS}$ output is active at a time.

Two device types are offered to help simplify interfacing with the system dynamic timing controller. The 'ALS6301 offers active-low row address strobe input ($\overline{RASI}$) and column address strobe input ($\overline{CASI}$) signals, while the 'ALS6302 offers active-high RASI and CASI inputs. Figure 2-7 is a functional block diagram of the two devices.

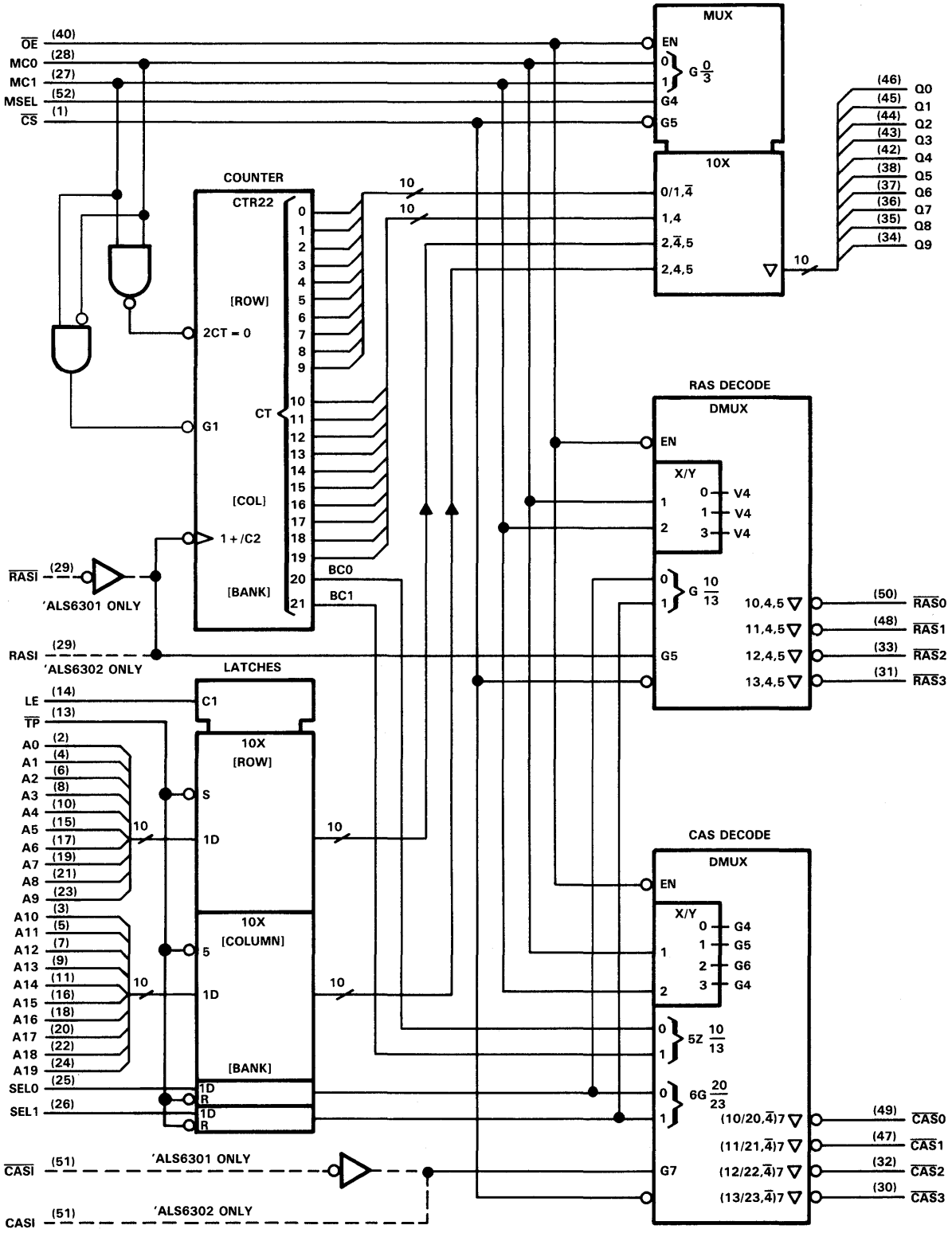


Figure 2-7. 'ALS6301, 'ALS6302 Functional Block Diagram

Table 2-2 describes the four operating modes of the 'ALS6301 and 'ALS6302 as controlled by inputs MC0 and MC1. During normal read/write operations, the row and column addresses are multiplexed to the DRAM. When MSEL is high, the column address is selected; when MSEL is low, the row address is selected. The corresponding $\overline{RAS}$n and $\overline{CAS}$n output signals strobe the addresses into the selected memory bank or banks. A single 'ALS6301 or 'ALS6302 can control as many as four banks of 1M memory. Additional banks of memory can be controlled by using additional 'ALS6301 or 'ALS6302 devices and decoding each chip select ($\overline{CS}$) input.

Table 2-2. 'ALS6301, 'ALS6302 Mode-Control Function Table

SIGNAL		MODE SELECTED
MC1	MC0	
L	L	Refresh without Scrubbing. Refresh cycles are performed using the row counter to generate the addresses. In this mode, all four $\overline{RAS}$ outputs are active while the four $\overline{CAS}$ outputs remain high.
L	H	Refresh with Scrubbing/Initialize. Refresh cycles are performed using both the row and column counters to generate the addresses. MSEL selects the row or the column counter. All four $\overline{RAS}$ outputs go low in response to $\overline{RASI}$ ('ALS6301) or RASI ('ALS6302), while only one $\overline{CAS}$n output goes low in response to $\overline{CASI}$ ('ALS6301) or CASI ('ALS6302). The bank counter keeps track of which $\overline{CAS}$ output goes active. This mode can also be used during system power-up so that the memory can be written with a known data pattern.
H	L	Read/Write. This mode is used to perform read/write cycles. Both the row and column addresses are multiplexed to the address output lines using MSEL. SEL0 and SEL1 are decoded to determine which $\overline{RAS}$n and $\overline{CAS}$n outputs will be active.
H	H	Clear Refresh Counters. This mode clears the three refresh counters (row, column, and bank) on the inactive transition of $\overline{RASI}$ ('ALS6301) or RASI ('ALS6302), putting them at the beginning of the refresh sequence. In this mode, all four $\overline{RAS}$ outputs are driven low after the active edge of $\overline{RASI}$ ('ALS6301) or RASI ('ALS6302) so that DRAM wake-up cycles can also be performed.

In systems where addresses and data are both multiplexed onto a single bus, the 'ALS6301 and 'ALS6302 use latches (row, column and bank) to hold the address information. The 22 input latches are transparent when the latch enable input (LE) is high; the input data is latched whenever LE goes low. For systems in which the processor has separate address and data buses, LE may be tied high.

The two 10-bit counters in the 'ALS6301 and 'ALS6302 support 128, 256, and 512 line refresh operations. Transparent, burst, synchronous, or asynchronous refresh modes are all possible as determined by the memory timing controller. The refresh counters are advanced on the low-to-high transition of $\overline{RASI}$ on the 'ALS6301, and on the high-to-low transition of RASI on the 'ALS6302. This is true in either refresh mode. In the clear refresh counter mode, the refresh counters (row, column, and bank) can be reset to zero on the low-to-high transition of $\overline{RASI}$ on the 'ALS6301 or on the high-to-low transition of RASI on the 'ALS6302.

2.3.2 Typical Implementation

Figure 2-8 shows a system interface using the 'ALS6301 between a Motorola 68000L10 and four banks of 1M DRAMs. Addresses A21 and A22 are used to select one of the four memory banks. Since members of the 68000 processor family have separate address and data busses, the input latches on the 'ALS6301 are left transparent by tying the latch enable (LE) input high. The $\overline{CAS0}$ thru $\overline{CAS3}$ outputs of the 'ALS6301 are fed into the byte controller along with processor signals $\overline{LDS}$ and $\overline{UDS}$. The byte controller made from programmable logic allows the processor to determine whether upper, lower or both bytes are accessed.

The $\overline{RASI}$, $\overline{CASI}$, MSEL and mode control (MC0, MC1) inputs on the 'ALS6301 must be generated by the memory timing controller. The memory timing controller functions as an arbitrator between refresh cycles and 68000L10 access cycles. It also guarantees that timing requirements of the DRAM will be met.

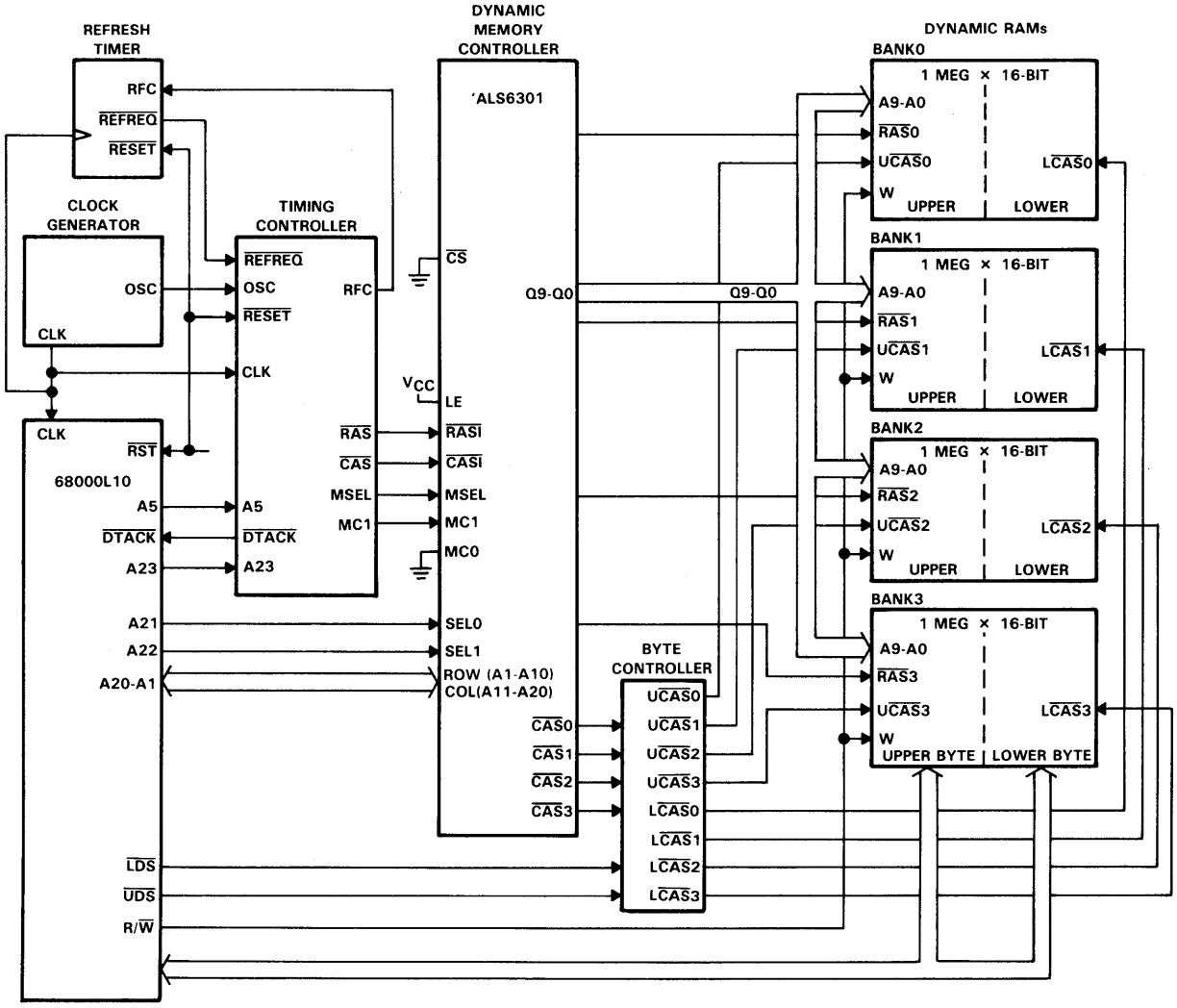
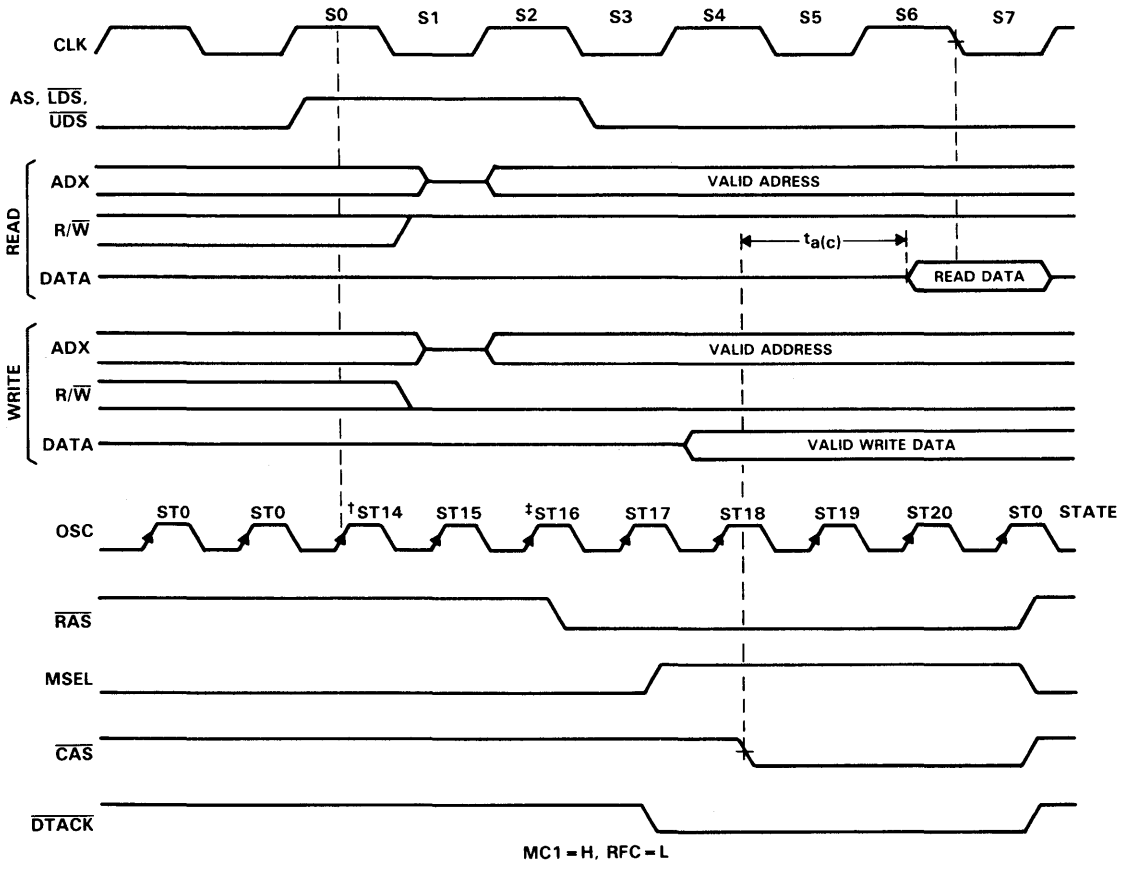


Figure 2-8. 'ALS6301, 'ALS6302 Timing Controller Interface

2.3.3 Timing Controller Details

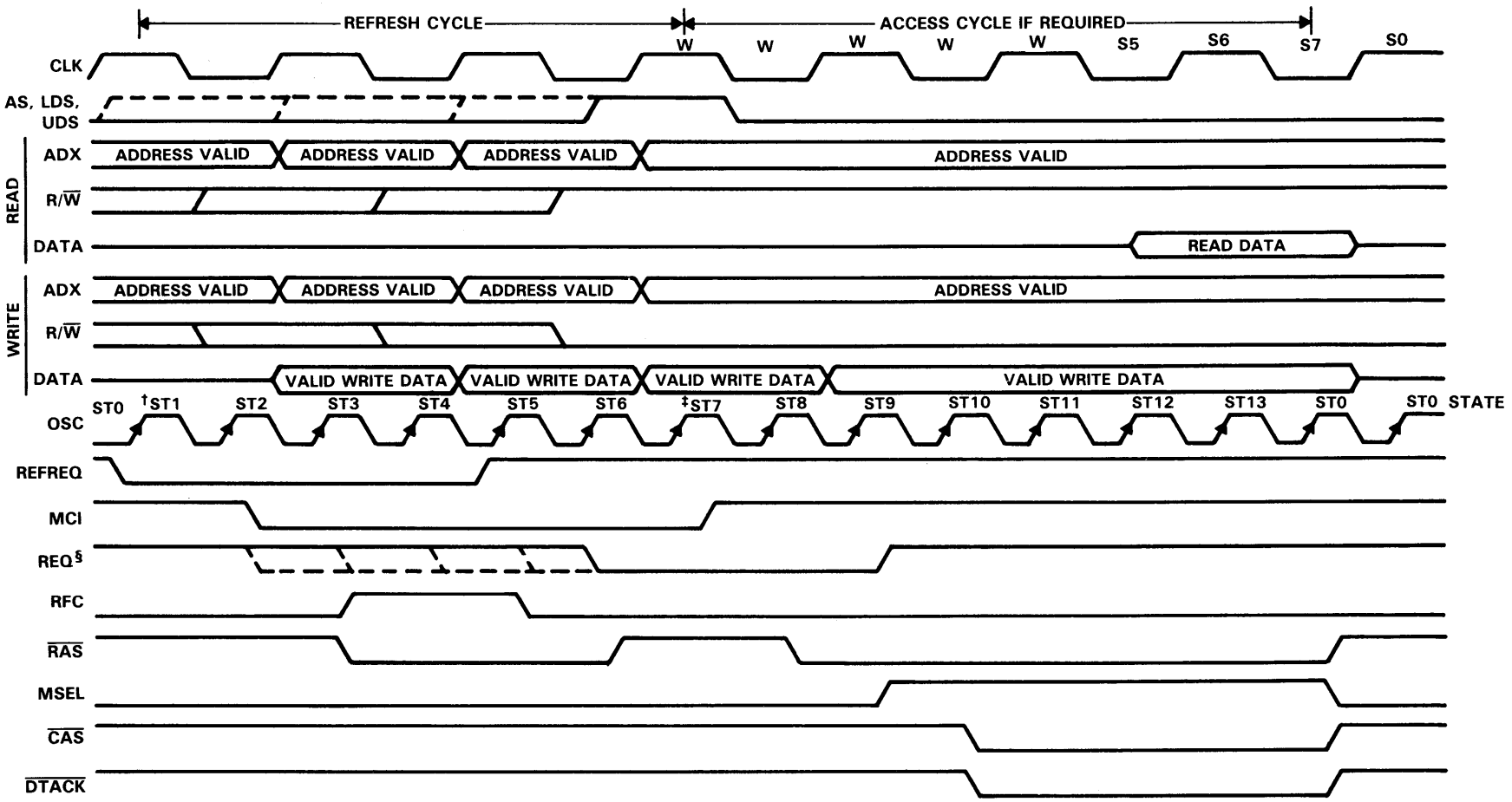
Figure 2-9 is a timing diagram for a typical 68000L10 access cycle. The 'ALS6301 control signals required to execute the access cycle are also shown. Control signals for the 'ALS6301 are referenced from the OSC output of the 8284A clock generator. OSC runs at 2 times the speed of the system clock, that is CLK = 10 MHz and OSC = 20 MHz. By running the timing controller at a higher speed than the system clock, the system performance is improved. A programmable logic sequencer, the TIB82S167B, was programmed for use as the timing controller.

In this example, refresh requests ($\overline{\text{REFREQ}}$) are generated every 155 clock cycles. The timing controller will perform the refresh cycle ($\overline{\text{RAS}}$ only) immediately if the processor is not in the middle of an access cycle. If the controller is in the middle of an access cycle, the refresh cycle will be delayed until the access cycle is complete. If the controller is asked to perform an access cycle during a refresh, the access cycle will begin immediately after the refresh cycle is completed. Address bit A23 indicates whether the access requested is a memory access (A23 = L) or an I/O access (A23 = H). The timing controller will perform an access cycle only if Address bit A23 is low. Figure 2-10 is a timing diagram of the refresh/access cycle as explained above. To implement memory scrubbing, the controller must execute a read/write cycle during the refresh cycle and then place the 'ALS6301 in the memory scrubbing mode. (This example executes a $\overline{\text{RAS}}$ only refresh.) The flowchart in Figure 2-11 outlines the required functionality of the timing controller. This flowchart was used along with the timing diagrams in Figures 2-9 and 2-10 to design the timing controller.



†Start sequence when AS = H, CLK = H, $\overline{\text{REFREQ}}$ = H, STATE = 0
‡Return to ST0 if A23 is high

Figure 2-9. 68000 Access Cycle



† Start sequence when $\overline{\text{REFREQ}}$ = L, STATE = 0
‡ Return to STATE 0 if REQ = H or A23 = H
§ REQ is internal status register used to store an access request during a refresh cycle. (If AS = H during refresh cycle ST1-ST5)

Figure 2-10. Refresh/Access Cycle

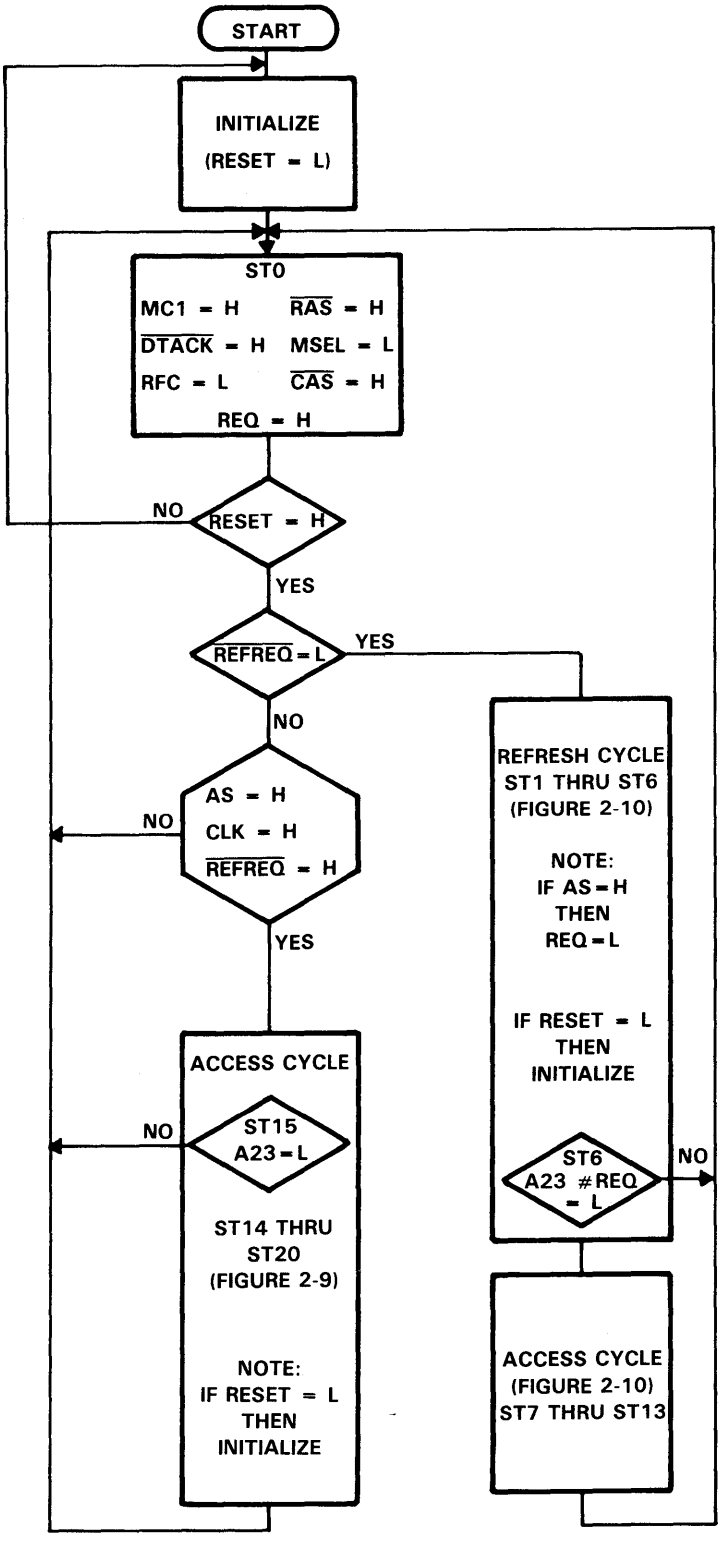


Figure 2-11. 'ALS6301, 'ALS6302 Memory Timing Controller Flow Chart

2.3.4 Refresh Timer Details

Figure 2-12 shows the actual circuit implementation of the refresh and memory timing controller. The refresh timer signals the controller whenever it is time to execute a refresh cycle. As required by memory, every row (512 on the TMS4C1025 DRAM) must be addressed every 8 ms. This implies that one row should be refreshed at least once every 15.6 ms. With a 10-MHz system clock, the refresh timer should use approximately a division factor of 155. This results in a refresh request every 15.3 ms. The refresh complete input (RFC) is used to signal the refresh timer that the refresh has been completed. It is important that the timer not stop so that the 8 ms memory requirement is maintained.

The TIBPAL22V10 circuit shown in Figure 2-12 is used to generate the refresh request signal every 155 clock cycles. The refresh request signal (active low) will remain active (low) until a refresh complete (RFC) signal is received from the timing controller. During a system reset, the refresh request output is set to a high logic level. When using different clock rates or memory sizes, the division circuit in the refresh timer should be adjusted accordingly.

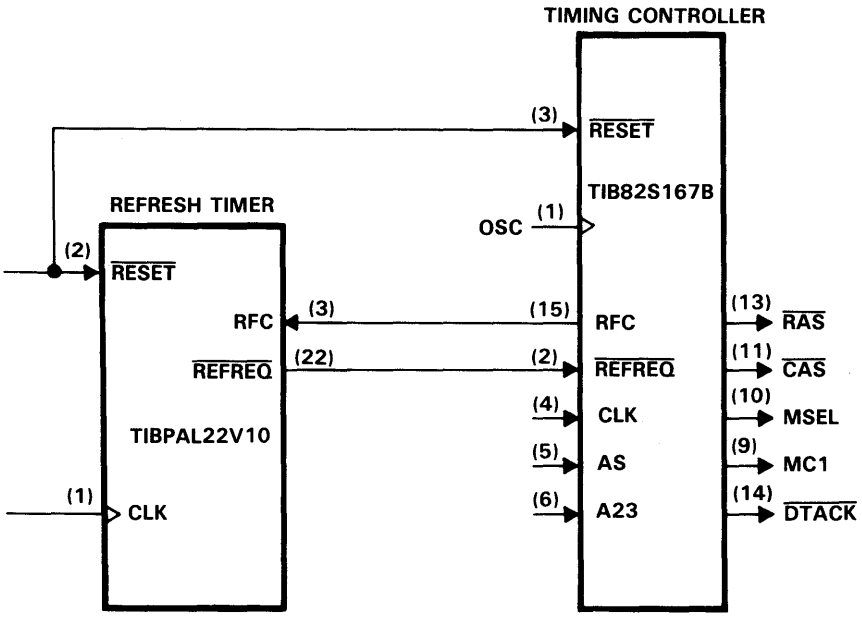


Figure 2-12. Refresh/Memory Timing Controller

2.3.5 Programmable Logic Designs

As mentioned previously, the timing controller, byte controller, and the refresh timer used in this example are created using programmable logic. ABEL™ and CUPL™ software packages have been used to reduce equations and generate the fuse maps needed to program these devices. The files used to generate the fuse maps have been included for reference at the end of this application report. Test vectors are included with the device files so software simulation can be performed on the computer. If the proper instruction is provided, the software will attach the test vectors to the end of the fuse map. This allows programming equipment to run a functional test on each device immediately after programming. To help familiarize the reader with these software tools, the timing controller design was done in both ABEL™ and CUPL™.

The TIB82S167B field programmable sequencer shown in Figure 2-12 is configured as a state machine to execute the flow chart shown in Figure 2-11. As shown in the flowchart, the timing controller is initialized by taking the reset input low. From the initialization state, state 0, the timing controller can perform either an access or a refresh cycle depending on the signals AS, CLK, and $\overline{\text{REFREQ}}$. If an access is requested (AS = H) during a refresh cycle, an internal status register, REQ, will flag the request and as soon as the refresh cycle is completed, an access cycle will be started. At the start of an access cycle, the timing controller checks the state of the A23 address bit. If A23 is high, indicating an I/O access, the timing controller terminates the access cycle and returns to state 0.

As seen in Figures 2-9, 2-10, and 2-11, a state, ST0-ST30, has been assigned to each clock cycle. The appended ABEL™ and CUPL™ files can be easily understood by comparing the state equations to the states shown in these figures. Since the only difference between the 'ALS6301 and the 'ALS6302 is that the $\overline{\text{RASI}}$ and the $\overline{\text{CASI}}$ inputs are active-high instead of active-low, a slight modification to the timing controller software file will allow an 'ALS6302 to be used instead of an 'ALS6301. The TIBPAL22V10 refresh timer and the TIBPAL16L8 byte controller designs are straight forward and easily achieved as can be seen in the appended files.

In applications with different systems timings, the ABEL™ and CUPL™ files can be modified to fit the processor requirements. A preprogrammed sample of the timing controller, refresh timer and byte controller can be obtained by calling LSI/PAL/PROM Applications, 214/995-2980. If a basic understanding of programmable logic is needed, see the Texas Instruments Field Programmable Logic Applications note.

2.3.6 Summary

The 'ALS6301 and 'ALS6302, coupled with programmable logic, offer the system designer a solution to high speed dynamic memory requirements. Programmable logic allows the designer to tailor the timing controller to a selected processor and memory. In many cases, the generation of a high speed timing controller from programmable logic will allow the designer to use slower DRAMs without affecting system speed. This results in lower total system cost because of the large number of memory devices used.

2.3.7 ABEL™ Files

module DMC S167

```
module DMC_S167 flag '-KY','-R2' "leave unused OR terms connected
title 'DYNAMIC MEMORY CONTROLLER FOR THE ALS6301 APPLICATION
Loren Schiele   Texas Instruments,  August 15, 1986'

        DMC  device 'F82S167';

" Input pin assignments

OSC     pin   1;                        " OSCILLATOR
REFREQ  pin   2;                        " REFRESH REQUEST
RESET   pin   3;                        " RESET - INITIALIZES WHEN LOW
CLK     pin   4;                        " OSC DIVIDED BY 2
AS      pin   5;                        " ADDRESS STROBE
A23     pin   6;                        " MOST SIGNIFICANT ADDRESS BIT
GND     pin  16;                        " PIN 16 MUST BE TIED LOW

" Output pin and node assignments

MC1     pin   9;  MC1_R    node 25;     " MODE CONTROL
MSEL    pin  10;  MSEL_R   node 26;     " MULTIPLEXER SELECT
CAS     pin  11;  CAS_R    node 27;     " COLUMN ADDRESS STROBE
RAS     pin  13;  RAS_R    node 28;     " ROW ADDRESS STROBE
DTACK   pin  14;  DTACK_R  node 29;     " DATA ACKNOWLEDGE
RFC     pin  15;  RFC_R    node 30;     " REFRESH COMPLETE

" Internal status and counter nodes

P0      node 36;  P0_R     node 42;     " INTERNAL COUNTER REGISTER
P1      node 35;  P1_R     node 41;     " INTERNAL COUNTER REGISTER
P2      node 34;  P2_R     node 40;     " INTERNAL COUNTER REGISTER
P3      node 33;  P3_R     node 39;     " INTERNAL COUNTER REGISTER
P4      node 32;  P4_R     node 38;     " INTERNAL COUNTER REGISTER
REQ     node 31;  REQ_R    node 37;     " REFRESH REQUEST STATUS REGISTER

" Define Set and Reset inputs to output and status flip-flops
MC1_    = [MC1,MC1_R];
MSEL_   = [MSEL,MSEL_R];
CAS_    = [CAS,CAS_R];
RAS_    = [RAS,RAS_R];
DTACK_  = [DTACK,DTACK_R];
RFC_    = [RFC,RFC_R];
REQ_    = [REQ,REQ_R];

" 'high' and 'low' are used to set or reset the output and status
"  registers.  Example: MC1_ := high & RESET; will cause pin 9 to
"  go high on the next clock edge if input pin 3 is high.

high    = [ 1,  0];
low     = [ 0,  1];
Count   = [P4,P3,P2,P1,P0];             " STATE REGISTER SET DEFINED
Cnt     = [P4,P3,P2,P1,P0];             " STATE REGISTER SET DEFINED
H,L,clk,X = 1, 0, .C., .X.;

@page
```

```
state_diagram Count                                          " NEXT
        State 0:        case                                 " STATE
                         !REFREQ   & RESET                    : 1;
                          REFREQ   & AS & CLK & RESET         :14;
                          REFREQ   & (!AS # !CLK)             : 0;
                        endcase;

" REFRESH TIMING CYCLE
        State 1:        MC1_  := low  & RESET;
                        REQ_  := low  & (AS & RESET);
                        case     RESET==1                    : 2; endcase;
        State 2:        RFC_  := high & RESET;
                        RAS_  := low  & RESET;
                        REQ_  := low  & (AS & RESET);
                        case     RESET==1                    : 3; endcase;
        State 3:        REQ_  := low  & (AS & RESET);
                        case     RESET==1                    : 4; endcase;
        State 4:        RFC_  := low  & RESET;
                        REQ_  := low  & (AS & RESET);
                        case     RESET==1                    : 5; endcase;
        State 5:        RAS_  := high;
                        REQ_  := low  & (AS & RESET);
                        case     RESET == 1                  : 6; endcase;

" DETERMINE IF ACCESS HAS BEEN REQUESTED
        State 6:        REQ_  := high & A23;
                        MC1_  := high & RESET;
                        case     REQ # A23                   : 0;
                                !A23  & !REQ & RESET         : 7;
                        endcase;

" ACCESS AFTER REFRESH
        State 7:        RAS_  := low  & RESET;
                        case     RESET==1                    : 8; endcase;
        State 8:        REQ_  := high & RESET;
                        MSEL_ := high & RESET;
                        case     RESET==1                    : 9; endcase;
        State 9:        CAS_  := low  & RESET;
                        DTACK_:= low  & RESET;
                        case     RESET==1                    :10; endcase;
        State 10:       case     RESET==1                    :11; endcase;

        State 11:       case     RESET==1                    :12; endcase;

        State 12:       case     RESET==1                    :13; endcase;

        State 13:       RAS_  := high;
                        MSEL_ := low;
                        CAS_  := high;
                        DTACK_:= high;
                        case     RESET==1                    : 0; endcase;
@page
```

```
" ACCESS TIMING CYCLE
        State 14:       case     RESET==1                :15; endcase;
        State 15:       RAS_  := low  & !A23 & RESET;
                        case A23 == 1                    : 0;
                             !A23 & RESET                :16;
                        endcase;
        State 16:       MSEL_ := high & RESET;
                        DTACK_:= low  & RESET;
                        case     RESET==1                :17; endcase;
        State 17:       CAS_  := low  & RESET;
                        case     RESET==1                :18; endcase;
        State 18:       case     RESET==1                :19; endcase;
        State 19:       case     RESET==1                :20; endcase;
        State 20:       RAS_  := high;
                        MSEL_ := low;
                        CAS_  := high;
                        DTACK_:= high;
                        case     RESET==1                : 0; endcase;

equations
        enable MC1 = 1;  "always enabled, pin 19 is preset

" INITIALIZATION WHEN RESET IS LOW
        [ MC1,RAS,DTACK,REQ,CAS]                  :=  !RESET;
        [ P0_R,P1_R,P2_R,P3_R,P4_R,MSEL_R,RFC_R] :=  !RESET;

test_vectors ' REFRESH WITH ACCESS FOLLOWING'

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   0  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 1 ];
 [ 0 ,clk,  1  ,   X  , X , 1, X ] -> [ L ,  L , H , H ,  H  , L , L , 2 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ L ,  L , H , L ,  H  , H , L , 3 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ L ,  L , H , L ,  H  , H , L , 4 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ L ,  L , H , L ,  H  , L , L , 5 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ L ,  L , H , H ,  H  , L , L , 6 ];
 [ 0 ,clk,  1  ,   X  , X , X, 0 ] -> [ H ,  L , H , H ,  H  , L , L , 7 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , L ,  H  , L , L , 8 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , H , L ,  H  , L , H , 9 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,10 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,11 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,12 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,13 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 0 , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 0 , 1, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 1 , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];

test_vectors ' REFRESH WITHOUT ACCESS FOLLOWING'

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   0  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 1 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , H ,  H  , L , H , 2 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , H , H , 3 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , H , H , 4 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , L , H , 5 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ L ,  L , H , H ,  H  , L , H , 6 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   0  , X , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 1 ];
@page
```

```
test_vectors ' REFRESH WITH ACCESS REQUEST BUT DATA NOT IN DRAM (A23=H) '

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   0  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 1 ];
 [ 0 ,clk,  1  ,   X  , X , 1, X ] -> [ L ,  L , H , H ,  H  , L , L , 2 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , H , L , 3 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , H , L , 4 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , L ,  H  , L , L , 5 ];
 [ 0 ,clk,  1  ,   X  , X , 0, X ] -> [ L ,  L , H , H ,  H  , L , L , 6 ];
 [ 0 ,clk,  1  ,   X  , X , 0, 1 ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 0 , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];

test_vectors ' ACCESS TIMING CYCLE '

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 1 , 1, X ] -> [ H ,  L , H , H ,  H  , L , H ,14 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H ,15 ];
 [ 0 ,clk,  1  ,   X  , X , X, 0 ] -> [ H ,  L , H , L ,  H  , L , H ,16 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , H , L ,  L  , L , H ,17 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,18 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,19 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,20 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 0 , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];

test_vectors ' ACCESS TIMING CYCLE BUT DATA NOT IN DRAM (A23=H) '

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 1 , 1, X ] -> [ H ,  L , H , H ,  H  , L , H ,14 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H ,15 ];
 [ 0 ,clk,  1  ,   X  , X , X, 1 ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 0 , 0, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];

test_vectors ' RESET DURING ACCESS TIMING CYCLE '

([GND,OSC,RESET,REFREQ,CLK,AS,A23] -> [MC1,MSEL,CAS,RAS,DTACK,RFC,REQ,Cnt])
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  1  ,   1  , 1 , 1, X ] -> [ H ,  L , H , H ,  H  , L , H ,14 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H ,15 ];
 [ 0 ,clk,  1  ,   X  , X , X, 0 ] -> [ H ,  L , H , L ,  H  , L , H ,16 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , H , L ,  L  , L , H ,17 ];
 [ 0 ,clk,  1  ,   X  , X , X, X ] -> [ H ,  H , L , L ,  L  , L , H ,18 ];
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];
 [ 0 ,clk,  0  ,   X  , X , X, X ] -> [ H ,  L , H , H ,  H  , L , H , 0 ];

end DMC_S167
```

module TIMER154

```
module  TIMER154 flag '-r2','-f'
title  'REFRESH TIMER
        LOREN SCHIELE  TEXAS INSTRUMENTS, DALLAS, 08/15/86'

        T154  DEVICE   'P22V10';

"input declarations

CLK             pin 1;              "  SYSTEM CLOCK
RESET           pin 2;              "  RESETS WHEN LOW
RFC             pin 3;              "  REFRESH COMPLETE

"output declarations

Q0,Q1,Q2,Q3     pin 14,15,16,17;      "  COUNTER STATES
Q4,Q5,Q6,Q7     pin 18,19,20,21;      "  COUNTER STATES
REFREQ_         pin 22;               "  REFRESH REQUEST - ACTIVE LOW

"intermediate variables

CNT_154_   =  !Q0 & Q1 & !Q2 & Q3 & Q4 & !Q5 & !Q6 & Q7;
SCLR       =  !RESET # CNT_154_;
count      = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
C,H,L,X    = .C.,1,0,.X.;

equations

REFREQ_   :=  RFC # !CNT_154_ &  REFREQ_ # !RESET;
Q0        :=  (!Q0 ) & !SCLR;
Q1        :=  ( Q1 $  Q0) & !SCLR;
Q2        :=  ( Q2 $  Q1 &  Q0) & !SCLR;
Q3        :=  ( Q3 $ (Q2 &  Q1 &  Q0)) & !SCLR;
Q4        :=  ( Q4 $ (Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;
Q5        :=  ( Q5 $ (Q4 &  Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;
Q6        :=  ( Q6 $ (Q5 &  Q4 &  Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;
Q7        :=  ( Q7 $ (Q6 &  Q5 &  Q4 &  Q3 &  Q2 &  Q1 &  Q0)) & !SCLR;

test_vectors              ([RESET,CLK,RFC] -> [count,REFREQ_])
                           [  0  , C , 0 ] -> [  0  ,   H   ];
@CONST cnt = 1;
@REPEAT 154 {              [  1  , C , 0 ] -> [ cnt ,   H   ];
@CONST cnt = cnt + 1;}
                           [  1  , C , 0 ] -> [  0  ,   L   ];
@CONST cnt = 1;
@REPEAT 20  {              [  1  , C , 0 ] -> [ cnt ,   L   ];
@CONST cnt = cnt + 1;}
                           [  1  , C , 1 ] -> [ 21  ,   H   ];
                           [  1  , C , 0 ] -> [ 22  ,   H   ];
                           [  1  , C , X ] -> [ 23  ,   H   ];
                           [  0  , C , X ] -> [  0  ,   H   ];
end TIMER154
```

2.3.8 CUPL™ Files

DYNAMIC MEMORY CONTROLLER

```
Partno          DMC-S167;
Name            DMC-S167;
Date            08/15/86;
Revision        01;
Designer        SCHIELE;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*                    DYNAMIC MEMORY CONTROLLER                  */
/*                          FOR ALS6301                          */
/*****************************************************************/
/* Allowable Target Device Types: TIB82S167B                     */
/*****************************************************************/

/**  Inputs  **/
pin 1 =  OSC;             /*  OSCILLATOR                         */
pin 2 = REFREQ;           /*  REFRESH REQUEST                    */
pin 3 = RESET;            /*  RESET - INITIALIZES WHEN LOW       */
pin 4 = CLK;              /*  OSC DIVIDED BY 2                   */
pin 5 = AS;               /*  ADDRESS STROBE                     */
pin 6 = A23_;             /*  MOST SIGNIFICANT ADDRESS BIT       */
pin 16 = GND;             /*  PIN 16 MUST BE TIED LOW            */

/**  Outputs  **/
pin  9 = MC1_;            /*  MODE CONTROL                       */
pin 10 = MSEL;            /*  MULTIPLEXER SELECT                 */
pin 11 = CAS;             /*  COLUMN ADDRESS STROBE              */
pin 13 = RAS;             /*  ROW ADDRESS STOBE                  */
pin 14 = DTACK;           /*  DATA ACKNOWLEDGE                   */
pin 15 = RFC;             /*  REFRESH COMPLETE                   */

/** Internal Node Group  -   State bits declared as nodes      **/
node [REQ,P4_,P3_,P2_,P1_,P0_];

/** Declarations and Intermediate Variable Definitions         **/
Field State  = [P4_,P3_,P2_,P1_,P0_];

$define  ST0 'b'00000
$define  ST1 'b'00001
$define  ST2 'b'00010
$define  ST3 'b'00011
$define  ST4 'b'00100
$define  ST5 'b'00101
$define  ST6 'b'00110
$define  ST7 'b'00111
$define  ST8 'b'01000
$define  ST9 'b'01001
$define ST10 'b'01010
$define ST11 'b'01011
$define ST12 'b'01100
$define ST13 'b'01101
$define ST14 'b'01110
$define ST15 'b'01111
$define ST16 'b'10000
$define ST17 'b'10001
$define ST18 'b'10010
$define ST19 'b'10011
$define ST20 'b'10100
```

```
/** Logic Equations **/

Sequence State
{Present ST0 IF  RESET & !REFREQ            NEXT ST1;
             IF  RESET &  REFREQ & AS & CLK NEXT ST14;
             DEFAULT                        NEXT ST0;

/*  REFRESH TIMING CYCLE */
Present ST1 IF  AS & RESET  NEXT ST2  OUT [!MC1_,!REQ];
            IF  RESET       NEXT ST2  OUT [!MC1_];
Present ST2 IF  AS & RESET  NEXT ST3  OUT [ RFC,!RAS,!REQ];
            IF  RESET       NEXT ST3  OUT [ RFC,!RAS];
Present ST3 IF  AS & RESET  NEXT ST4  OUT [!REQ];
            IF  RESET       NEXT ST4;
Present ST4 IF  AS & RESET  NEXT ST5  OUT [!RFC,!REQ];
            IF  RESET       NEXT ST5  OUT [!RFC];
Present ST5 IF  AS & RESET  NEXT ST6  OUT [ RAS,!REQ];
            IF  RESET       NEXT ST6  OUT [ RAS];
/** DETERMINE IF ACCESS HAS BEEN REQUESTED **/
Present ST6 IF  A23_ # REQ  NEXT ST0  OUT [ MC1_,REQ];
            IF !A23_ & RESET & !REQ NEXT ST7 OUT [ MC1_];

/** ACCESS AFTER REFRESH **/
Present ST7  IF  RESET       NEXT ST8  OUT [!RAS];
Present ST8  IF  RESET       NEXT ST9  OUT [ REQ, MSEL];
Present ST9  IF  RESET       NEXT ST10 OUT [!CAS,!DTACK];
Present ST10 IF  RESET       NEXT ST11;
Present ST11 IF  RESET       NEXT ST12;
Present ST12 IF  RESET       NEXT ST13;
Present ST13                 NEXT ST0  OUT [ RAS,!MSEL, CAS, DTACK];

/** ACCESS TIMING CYCLE **/
Present ST14 IF  RESET       NEXT ST15;
Present ST15 IF  A23_        NEXT ST0;
             IF !A23_ & RESET NEXT ST16 OUT [!RAS];
Present ST16 IF  RESET       NEXT ST17 OUT [ MSEL,!DTACK];
Present ST17 IF  RESET       NEXT ST18 OUT [!CAS];
Present ST18 IF  RESET       NEXT ST19;
Present ST19 IF  RESET       NEXT ST20;
Present ST20                 NEXT ST0  OUT [ RAS,!MSEL, CAS, DTACK];}

APPEND MC1_.s  = !RESET;  APPEND REQ.s  = !RESET;  APPEND RFC.r = !RESET;
APPEND RAS.s   = !RESET;  APPEND MSEL.r = !RESET;  APPEND CAS.s = !RESET;
APPEND DTACK.s = !RESET;  APPEND P0_.r  = !RESET;  APPEND P1_.r = !RESET;
APPEND P2_.r   = !RESET;  APPEND P3_.r  = !RESET;  APPEND P4_.r = !RESET;
```

DYNAMIC MEMORY SIMULATION

```
Partno          DMC-S167;
Name            DMC-S167;
Date            08/15/86;
Revision        01;
Designer        SCHIELE;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*                  DYNAMIC TIMING CONTROLLER                    */
/*                       SIMULATION FILE                         */
/*                         FOR ALS6301                           */
/*****************************************************************/
/* Allowable Target Device Types: TIB82S167B                     */
/*****************************************************************/

ORDER:  GND,%3,OSC,%3,RESET,%6,REFREQ,%4,CLK,%3,AS,%2,A23_,%6,
        MC1_,%4,MSEL,%3,CAS,%3,RAS,%4,DTACK,%4,RFC,%4,REQ;

VECTORS:
$msg" REFRESH WITH ACCESS FOLLOWING";
$msg"     ";
$msg"    ------------ INPUT -------------   -------- OUTPUT ---------- ACCESS";
$msg"    GND OSC RESET REFREQ CLK AS A23    MC1 MSEL CAS RAS DTACK RFC  REQ";
$msg"    ---------------------------------------------------------------------";
/*RESET*/ 0   C    0      X    X  X  X      H    L   H   H    H    L    H
/* ST0*/  0   C    1      0    X  X  X      H    L   H   H    H    L    H
/* ST1*/  0   C    1      X    X  1  X      L    L   H   H    H    L    L
/* ST2*/  0   C    1      X    X  X  X      L    L   H   L    H    H    L
/* ST3*/  0   C    1      X    X  X  X      L    L   H   L    H    H    L
/* ST4*/  0   C    1      X    X  X  X      L    L   H   L    H    L    L
/* ST5*/  0   C    1      X    X  X  X      L    L   H   H    H    L    L
/* ST6*/  0   C    1      X    X  X  0      H    L   H   H    H    L    L
/* ST7*/  0   C    1      X    X  X  X      H    L   H   L    H    L    L
/* ST8*/  0   C    1      X    X  X  X      H    H   H   L    H    L    H
/* ST9*/  0   C    1      X    X  X  X      H    H   L   L    L    L    H
/*ST10*/  0   C    1      X    X  X  X      H    H   L   L    L    L    H
/*ST11*/  0   C    1      X    X  X  X      H    H   L   L    L    L    H
/*ST12*/  0   C    1      X    X  X  X      H    H   L   L    L    L    H
/*ST13*/  0   C    1      X    X  X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    0  0  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    0  1  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    1  0  X      H    L   H   H    H    L    H

$msg"     ";
$msg"     ";
$msg"REFRESH WITHOUT ACCESS FOLLOWING";
$msg"     ";
$msg"    ------------ INPUT -------------   -------- OUTPUT ---------- ACCESS";
$msg"    GND OSC RESET REFREQ CLK AS A23    MC1 MSEL CAS RAS DTACK RFC  REQ  ";
$msg"    ---------------------------------------------------------------------";
/*RESET*/ 0   C    0      X    X  X  X      H    L   H   H    H    L    H
/* ST0*/  0   C    1      0    X  X  X      H    L   H   H    H    L    H
/* ST1*/  0   C    1      X    X  0  X      L    L   H   H    H    L    H
/* ST2*/  0   C    1      X    X  0  X      L    L   H   L    H    H    H
/* ST3*/  0   C    1      X    X  0  X      L    L   H   L    H    H    H
/* ST4*/  0   C    1      X    X  0  X      L    L   H   L    H    L    H
/* ST5*/  0   C    1      X    X  0  X      L    L   H   H    H    L    H
/* ST6*/  0   C    1      X    X  X  0      H    L   H   H    H    L    H
/* ST0*/  0   C    1      X    X  0  X      H    L   H   H    H    L    H
```

```
$msg"     ";
$msg"     ";
$msg"REFRESH WITH ACCESS REQUEST BUT DATA NOT IN DRAM (A23=H)";
$msg"     ";
$msg"     ------------ INPUT -------------    -------- OUTPUT ---------- ACCESS";
$msg"     GND OSC RESET REFREQ CLK AS A23     MC1 MSEL CAS RAS DTACK RFC  REQ  ";
$msg"     ------------------------------------------------------------------------";
/*RESET*/ 0   C    0      X    X   X  X      H    L   H   H    H    L    H
/* ST0*/  0   C    1      0    X   X  X      H    L   H   H    H    L    H
/* ST1*/  0   C    1      X    X   1  X      L    L   H   H    H    L    L
/* ST2*/  0   C    1      X    X   0  X      L    L   H   L    H    H    L
/* ST3*/  0   C    1      X    X   0  X      L    L   H   L    H    H    L
/* ST4*/  0   C    1      X    X   0  X      L    L   H   L    H    L    L
/* ST5*/  0   C    1      X    X   0  X      L    L   H   H    H    L    L
/* ST6*/  0   C    1      X    X   0  1      H    L   H   H    H    L    H
/* ST0*/  0   C    1      1    0   0  X      H    L   H   H    H    L    H

$msg"     ";
$msg"     ";
$msg"ACCESS TIMING CYCLE ";
$msg"     ";
$msg"     ------------ INPUT -------------    -------- OUTPUT ---------- ACCESS";
$msg"     GND OSC RESET REFREQ CLK AS A23     MC1 MSEL CAS RAS DTACK RFC  REQ  ";
$msg"     ------------------------------------------------------------------------";
/*RESET*/ 0   C    0      X    X   X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    1   1  X      H    L   H   H    H    L    H
/*ST14*/  0   C    1      X    X   X  X      H    L   H   H    H    L    H
/*ST15*/  0   C    1      X    X   X  0      H    L   H   L    H    L    H
/*ST16*/  0   C    1      X    X   X  X      H    H   H   L    L    L    H
/*ST17*/  0   C    1      X    X   X  X      H    H   L   L    L    L    H
/*ST18*/  0   C    1      X    X   X  X      H    H   L   L    L    L    H
/*ST19*/  0   C    1      X    X   X  X      H    H   L   L    L    L    H
/*ST20*/  0   C    1      X    X   X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    0   0  X      H    L   H   H    H    L    H

$msg"     ";
$msg"     ";
$msg"ACCESS TIMING CYCLE BUT DATA NOT IN DRAM (A23=H)";
$msg"     ";
$msg"     ------------ INPUT -------------    -------- OUTPUT ---------- ACCESS";
$msg"     GND OSC RESET REFREQ CLK AS A23     MC1 MSEL CAS RAS DTACK RFC  REQ  ";
$msg"     ------------------------------------------------------------------------";
/*ST0 */  0   C    1      1    1   1  X      H    L   H   H    H    L    H
/*ST14*/  0   C    1      X    X   X  X      H    L   H   H    H    L    H
/*ST15*/  0   C    1      X    X   X  1      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    0   0  X      H    L   H   H    H    L    H

$msg"     ";
$msg"     ";
$msg"RESET DURING ACCESS TIMING CYCLE ";
$msg"     ";
$msg"     ------------ INPUT -------------    -------- OUTPUT ---------- ACCESS";
$msg"     GND OSC RESET REFREQ CLK AS A23     MC1 MSEL CAS RAS DTACK RFC  REQ  ";
$msg"     ------------------------------------------------------------------------";
/*RESET*/ 0   C    0      X    X   X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    1      1    1   1  X      H    L   H   H    H    L    H
/*ST14*/  0   C    1      X    X   X  X      H    L   H   H    H    L    H
/*ST15*/  0   C    1      X    X   X  0      H    L   H   L    H    L    H
/*ST16*/  0   C    1      X    X   X  X      H    H   H   L    L    L    H
/*ST17*/  0   C    1      X    X   X  X      H    H   L   L    L    L    H
/*ST18*/  0   C    0      X    X   X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    0      X    X   X  X      H    L   H   H    H    L    H
/*ST0 */  0   C    0      X    X   X  X      H    L   H   H    H    L    H
```

BYTE CONTROLLER

```
Partno          BYTE_CON;
Name            BYTE_CON;
Date            08/15/86;
Revision        01;
Designer        SCHIELE;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*                      BYTE CONTROLLER                          */
/*              FOR ALS6301/MC68000L10 APPLICATION               */
/*****************************************************************/
/* Allowable Target Device Types: TIBPAL16L8                     */
/*****************************************************************/
/**  Inputs  **/
pin 1 = CAS0;           /*  CAS BANK SELECT                      */
pin 2 = CAS1;           /*       "                               */
pin 3 = CAS2;           /*       "                               */
pin 4 = CAS3;           /*       "                               */
pin 5 = LDS;            /*  LOWER DATA STROBE                    */
pin 6 = UDS;            /*  UPPER DATA STROBE                    */

/**  Outputs  **/
pin 12 = UCAS0;         /*  UPPER BYTE SELECT - BANK 0           */
pin 13 = LCAS0;         /*  LOWER BYTE SELECT - BANK 0           */
pin 14 = UCAS1;         /*  UPPER BYTE SELECT - BANK 1           */
pin 15 = LCAS1;         /*  LOWER BYTE SELECT - BANK 1           */
pin 16 = UCAS2;         /*  UPPER BYTE SELECT - BANK 2           */
pin 17 = LCAS2;         /*  LOWER BYTE SELECT - BANK 2           */
pin 18 = UCAS3;         /*  UPPER BYTE SELECT - BANK 3           */
pin 19 = LCAS3;         /*  LOWER BYTE SELECT - BANK 3           */

/*  equations     */
UCAS0  = CAS0 # UDS;
LCAS0  = CAS0 # LDS;
UCAS1  = CAS1 # UDS;
LCAS1  = CAS1 # LDS;
UCAS2  = CAS2 # UDS;
LCAS2  = CAS2 # LDS;
UCAS3  = CAS3 # UDS;
LCAS3  = CAS3 # LDS;
```

BYTE CONTROLLER SIMULATION

```
Partno          BYTE_CON;
Name            BYTE_CON;
Date            08/15/86;
Revision        01;
Designer        SCHIELE;
Company         TEXAS INSTRUMENTS;
Assembly        None;
Location        DALLAS, TEXAS;
/*****************************************************************/
/*                BYTE CONTROLLER SIMULATION FILE                */
/*              FOR ALS6301/MC68000L10  APPLICATION              */
/*****************************************************************/
/* Allowable Target Device Types: TIBPAL16L8                     */
/*****************************************************************/

ORDER:
CAS0,%2,CAS1,%2,CAS2,%2,CAS3,%3,LDS,%2,UDS,%4,LCAS0,%2,UCAS0,%2,
LCAS1,%2,UCAS1,%2,LCAS2,%2,UCAS2,%2,LCAS3,%2,UCAS3;

VECTORS:

$msg"      ----- INPUT -----    ------- OUTPUT -------   ";
$msg"                                L  U  L  U  L  U  L  U   ";
$msg"      C  C  C  C               C  C  C  C  C  C  C  C   ";
$msg"      A  A  A  A  L  U         A  A  A  A  A  A  A  A   ";
$msg"      S  S  S  S  D  D         S  S  S  S  S  S  S  S   ";
$msg"      0  1  2  3  S  S         0  0  1  1  2  2  3  3   ";
$msg"      ----------------------------------------------   ";
           1  1  1  1  X  X         H  H  H  H  H  H  H  H
           X  X  X  X  1  1         H  H  H  H  H  H  H  H

           0  1  1  1  0  0         L  L  H  H  H  H  H  H
           1  0  1  1  0  0         H  H  L  L  H  H  H  H
           1  1  0  1  0  0         H  H  H  H  L  L  H  H
           1  1  1  0  0  0         H  H  H  H  H  H  L  L

           0  1  1  1  0  1         L  H  H  H  H  H  H  H
           1  0  1  1  0  1         H  H  L  H  H  H  H  H
           1  1  0  1  0  1         H  H  H  H  L  H  H  H
           1  1  1  0  0  1         H  H  H  H  H  H  L  H

           0  1  1  1  1  0         H  L  H  H  H  H  H  H
           1  0  1  1  1  0         H  H  H  L  H  H  H  H
           1  1  0  1  1  0         H  H  H  H  H  L  H  H
           1  1  1  0  1  0         H  H  H  H  H  H  H  L
```

2.4 THCT4502B/MC68000L8 Interface

2.4.1 Introduction

This application report presents a circuit configuration which interfaces the MC68000L8 to DRAM memory via the THCT4502B dynamic RAM controller. The memory array is four banks of 256K-byte memory (TMS4256/4257) that provides a 1M byte deep system architecture.

Figure 2-13 is a schematic diagram of the circuit and Figure 2-14 a timing diagram for two consecutive read cycles. Figure 2-15 shows a write access, followed by a refresh, followed by a read-access grant. The THCT4502B uses the MC68000L8 system clock and requires no wait states on normal access cycles. When incorporating DRAMs and a DRAM controller into a microprocessor based system, the following timing specifications should be satisfied to guarantee a correct match between processor and memory.

ALE-to-Clock Relationship
DRAM Refresh Time
DRAM Precharge Time
Row Address Setup and Hold Time
Data Valid to Write Enable Time
Read Access Time

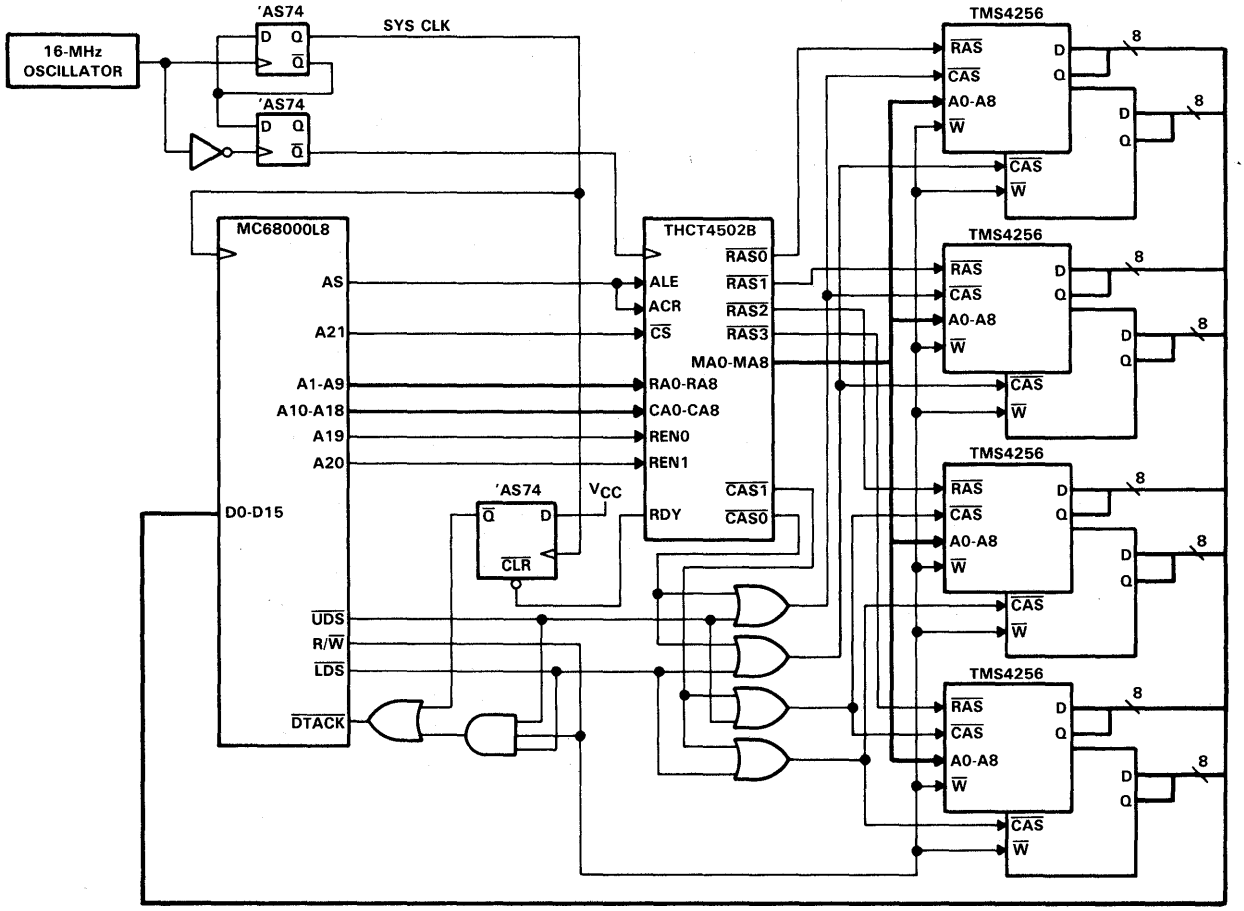


Figure 2-13. THCT4502B/MC68000L8 Interface Block Diagram

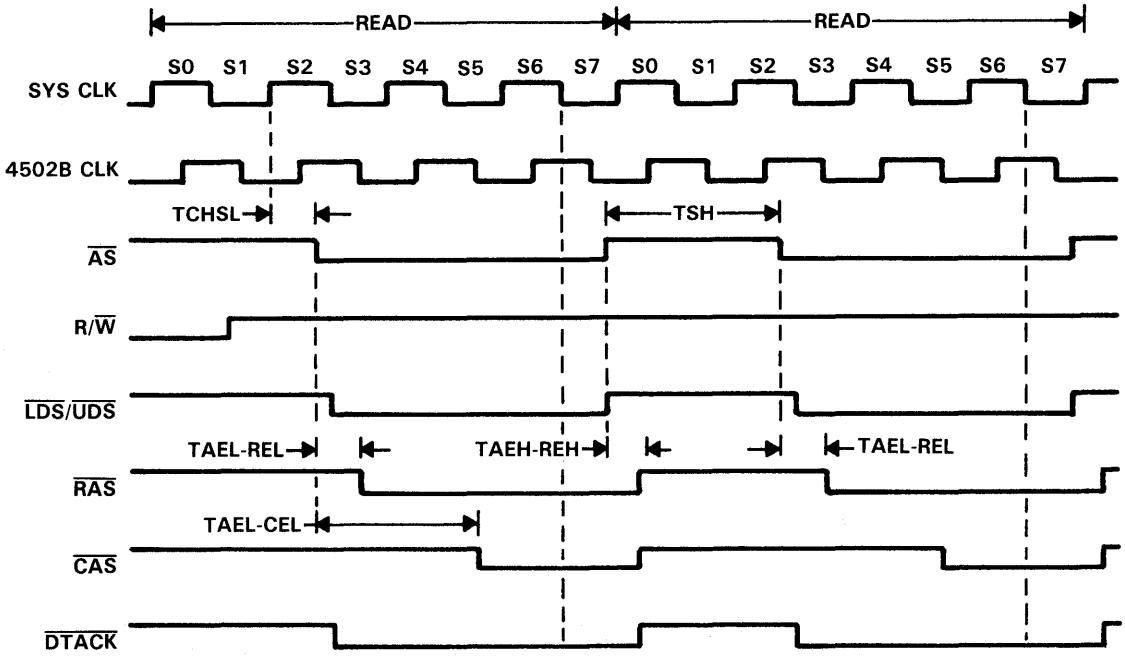


Figure 2-14. THCT4502B/MC68000L8 Read Cycle Timing Diagram

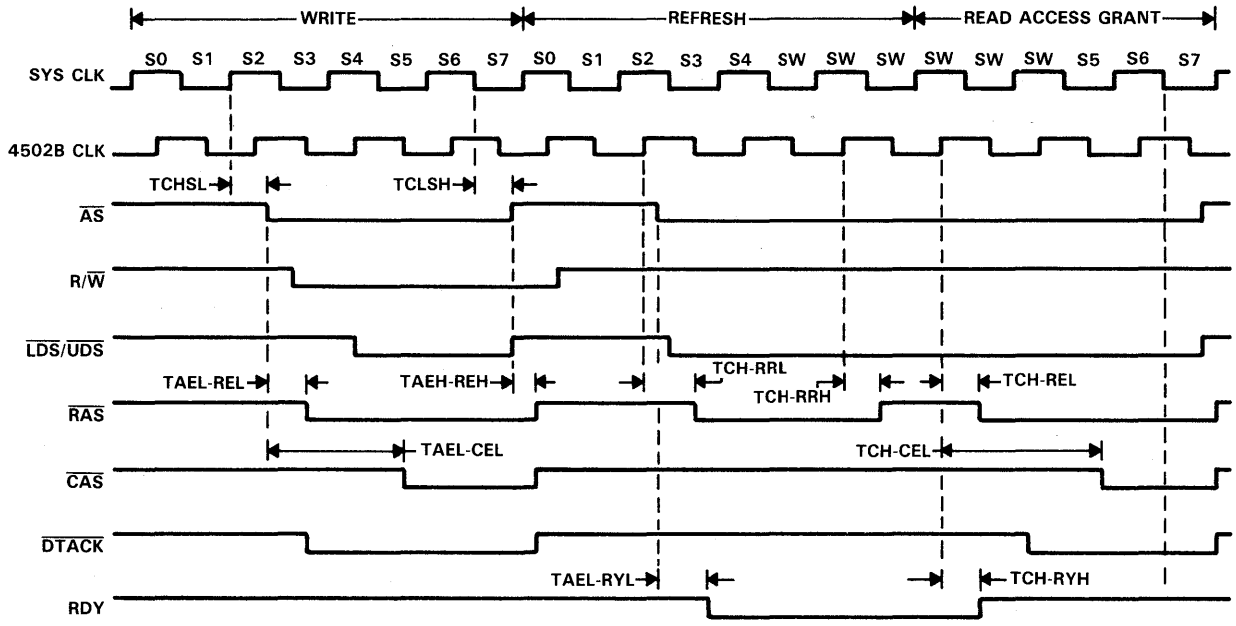


Figure 2-15. THCT4502B/MC68000L8 Write Access, Refresh, and Read Access Timing Diagram

2.4.2 ALE-to-Clock Relationship

When using the THCT4502B, the high-to-low transition of ALE should not occur between 15 ns before and 15 ns after the falling edge of the clock signal. This condition guarantees the proper selection between refresh and access cycles.

When connecting the Address Strobe (AS) of the MC68000 processor directly to ALE, ensure that the following condition is met.

$$15 < 0.5T - t_{CHSL}$$
$$15 < 0.5(125) - 60$$
$$15 < 2.5$$

At 8 MHz, this condition cannot be guaranteed. Therefore a circuit is required to shift the input phase of the THCT4502B clock signal by 90 degrees. As shown in Figure 2-13, this circuit can be built using standard 'AS74 D-type flip-flops. With the THCT4502B CLK shifted by 90 degrees, the new equation becomes:

$$15 < 0.5T + 0.25T - t_{CHSL}$$
$$15 < 0.5(125) + 0.25(125) - 60$$
$$15 < 33.75$$

It should be noted that all of the following equations take into account the 90 degree phase shift. At lower clock frequencies, such as 6 MHz, the AS signal can be directly connected to the THCT4502B and the phase shift circuits are not required.

2.4.3 DRAM Refresh Time

The refresh clock frequency is controlled by the strap input pins (TWST, FS1, and FS0) on the THCT4502B. Table 2-3 shows the strap configuration for the THCT4502B. At 8 MHz, with no wait states, setting TWST low, FS1 high, and FS0 high yields a refresh rate of 11.375 μs/row. The TMS4256/4257 requires that each of the 256 rows be refreshed at least once every 4 ms. With a refresh rate of 11.375 μs/row, the time required to refresh all 256 rows will be 2.9 ms. This easily satisfies the 4-ms refresh requirement.

Table 2-3. Refresh Clock Frequency Input Pin Strap Configuration

STRAP INPUT MODES			WAIT STATES FOR MEMORY ACCESS	REFRESH RATE	MINIMUM CLOCK FREQUENCY (MHz)	REFRESH FREQUENCY (kHz)	CLOCK CYCLES FOR EACH REFRESH
TWST	FS1	FS0					
L	L	L†	0	EXTERNAL	–	REFREQ	4
L	L	H	0	EXTERNAL	–	REFREQ	3
L	H	L	0	CLK ÷ 61	3.904	64-95‡	3
L	H	H	0	CLK ÷ 91	5.824	64-88§	4
H	L	L	1	CLK ÷ 61	3.904	64-95‡	3
H	L	H	1	CLK ÷ 91	5.824	64-75‡	4
H	H	L	1	CLK ÷ 106	6.784	64-73‡	4
H	H	H	1	CLK ÷ 121	7.744	64-83¶	4

† This strap configuration resets the Refresh Timer Circuitry.
‡ Upper figure in refresh frequency is the frequency that is produced if the minimum clock frequency of the next select state is used.
§ Refresh frequency if clock frequency is 8 MHz.
¶ Refresh frequency if clock frequency is 10 MHz.

2.4.4 DRAM Precharge Time

The precharge time is the time required between access cycles to allow internal nodes on the DRAM to charge to their correct reference levels. This is specified on the DRAM data sheet as $t_{W(RH)}$ min. As with most DRAMs, there is a choice of performance ranges. For the TMS4256/4257, $t_{W(RH)}$ ranges from 100 ns on the −12 device to 120 ns on the −20 device.

When using the THCT4502B, there are three precharge conditions which can occur during normal operation. Each condition must be checked to be sure the precharge condition is met. The following equations check these three conditions.

1. Access-to-Access cycle

$$t_{W(RH)} < t_{SH} - t_{AEH\text{-}REH} - t_{t(REH)} + t_{AEL\text{-}REL}$$
$$t_{W(RH)} < 150 - 35 - 30 + 35$$
$$t_{W(RH)} < 120$$

2. Access-to-Refresh cycle

$$t_{W(RH)} < 1.5T + 0.25T + t_{CH\text{-}RRL} - t_{CLSH} - t_{AEH\text{-}REH} - t_{t(REH)}$$
$$t_{W(RH)} < 1.5(125) + 0.25(125) + 50 - 70 - 35 - 30$$
$$t_{W(RH)} \text{ h } 133.75$$

3. Refresh-to-Access cycle

$$t_{W(RH)} < T - t_{CH\text{-}RRH} - t_{t(REH)} + t_{CH\text{-}REL}$$
$$t_{W(RH)} < 125 - 30 - 30 + 45$$
$$t_{W(RH)} < 110$$

When the listed equations are correct, the THCT4502B guarantees the precharge condition for either the −12 or −15 TMS4256/4257 DRAMs.

2.4.5 Row Address Setup and Hold Time

To meet the row address setup-time requirement, the address must be present at the RA0-RA8 and CA0-CA8 inputs to the THCT4502B for at least 10 ns ($t_{AV\text{-}AEL}$) before ALE goes low. The row address setup time from the MC68000L8 is defined by the t_{AVSL} specification. At 8 MHz, t_{AVSL} is 30 ns minimum. This meets the THCT4502B specification. The row address setup time to the DRAM must also be satisfied. For the TMS4256/4257, $t_{su(RA)}$ is specified as 0-ns minimum. The following equation applies:

$$0 \text{ ns} < t_{AVSL} + t_{AEL\text{-}REL} - t_{RAV\text{-}MAV}$$
$$0 \text{ ns} < 30 + 35 - 42$$
$$0 \text{ ns} < 23$$

When the equation is correct, the THCT4502B guarantees the row address setup time to the DRAM. The row address hold time required by the TMS4256/4257 is 15 ns. This specification is guaranteed by the THCT4502B. From the data sheet, $t_{REL\text{-}MAX}$ is specified as 20 ns min.

2.4.6 Data Valid to Write Enable Setup Time

Data can be written into DRAM by two different methods. Depending upon the mode of operation, the falling edge of $\overline{CAS}$ or the the falling edge of $\overline{W}$ will strobe the data into memory. When $\overline{W}$ goes low prior to $\overline{CAS}$ going low, data out will remain in the high-impedance state for the entire cycle. This permits common input/output operation. This type of cycle is referred to as an early write cycle. When $\overline{W}$ goes low after $\overline{CAS}$ goes low, the type of cycle is referred to as delayed-write or read-modify-write cycle. To avoid bus contention, this operation requires a buffer between the Q outputs and the microprocessor.

The circuit shown in Figure 2-13 generates an early write cycle. Therefore, data valid to write enable needs to be referenced to the falling edge of $\overline{CAS}$. The TMS4256/4257 requirement for an early write cycle is $t_{su(WCL)}$, which is 0 ns minimum. The following equation applies:

$$0\text{ ns} < t_{CHSL} + t_{AEL\text{-}CEL} - 0.5T - t_{CLDO}$$
$$0\text{ ns} < 60 + 115 - 0.5(125) - 70$$
$$0\text{ ns} < 42.5$$

When the equation is correct, the MC68000/THCT4502B combination guarantees that data will be valid before $\overline{CAS}$ goes low.

2.4.7 Read Access Time from CAS

When the microprocessor tries to read data from memory, the Read-Access-Time guarantees that data is available. When using the THCT4502B, there are two possible access situations. The most common is the normal access cycle. Another possible access situation is the access-grant cycle. The access-grant cycle occurs when an access cycle immediately follows a refresh cycle.

For the TMS4256/4257, access from $\overline{CAS}$ is specified as $t_{a(C)}$. When using the TMS4256/4257, three speed types are available for selection. The three speed types are as follows:

Speed type -12 $t_{a(CA)} = 60$ ns
Speed type -15 $t_{a(CA)} = 75$ ns
Speed type -20 $t_{a(CA)} = 100$ ns

The following equations apply to the circuit shown in Figure 2-14.

1. Normal Access Cycles

$$t_{a(C)} < 2.5T - t_{CHSL} - t_{AEL\text{-}CEL} - t_{t(CEL)} - t_{p(OR)} - t_{DICL}$$
$$t_{a(C)} < 2.5(125) - 60 - 115 - 20 - 15 - 15$$
$$t_{a(C)} < 87.5$$

2. Access Grant Cycles

$$t_{a(C)} < 2.5T - 0.25T - t_{CH\text{-}CEL} - t_{t(CEL)} - t_{p(OR)} - t_{DICL}$$
$$t_{a(C)} < 2.5(125) - 0.25(125) - 140 - 20 - 15 - 15$$
$$t_{a(C)} < 91.25$$

As shown by the equations, the only speed type that does not meet the access time requirement is the -20 device. The -12 and -15 devices both meet $t_{a(C)}$.

2.4.8 Other Considerations

The $\overline{DTACK}$ input on the MC68000L8 informs the microprocessor that data is available. Wait states are inserted by holding $\overline{DTACK}$ high. This process for the access-grant cycle is illustrated in Figure 2-15. If an access request occurs during a refresh cycle, the THCT4502B completes the refresh cycle, then finishes the access request. In this situation, the $\overline{DTACK}$ signal is held high until data is available. The AS74 flip-flop shown in Figure 2-13 is used to time the $\overline{DTACK}$ signal in relationship to the falling edge of S6.

On normal accesses, the RDY signal is high allowing either $\overline{UDS}$, $\overline{LDS}$ or R/$\overline{W}$ to force $\overline{DTACK}$ low. During write cycles, R/$\overline{W}$ will force $\overline{DTACK}$ low. During read cycles, $\overline{UDS}$ and/or $\overline{LDS}$ will force $\overline{DTACK}$ low. During access-grant cycles, the low RDY signal holds $\overline{DTACK}$ high until it is released.

2.4.9 Summary

This application report provides an example of how to interface the THCT4502B with the MC68000L8. The major design criteria has been calculated and checked against typical DRAM specifications. When using processor speeds lower than 8 MHz, the interface is simplified further because it is not necessary to shift the THCT4502B input clock frequencies. Additional design ideas can be obtained from an Applications Brief "TMS4500B/MC68000 INTERFACE", Texas Instruments publication SMCA008.

2.5 Programmer and Software Manufacturers Addresses†

2.5.1 Programmer Manufacturers Addresses

ECI Semiconductor
975 Comstock St.
Santa Clara, CA 95054
(408) 727-6562

DATA I/O
10525 Willows Rd. NE
Redmond, WA 98073-9746
(206) 881-6444

DIGITAL MEDIA
11770 Warner Ave. Suite 225
Fountain Valley, CA 92708
(714) 751-1373

Kontron Electronics
1230 Charleston Rd.
Mountain View, CA 94039-7230
(415) 965-7020

Stag Micro Systems
528-5 Weddell Drive
Sunnyvale, CA 94089
(408) 745-1991

Storey Systems
3201 N. Hwy 67, Suite E
Mesquite, TX 75150
(214) 270-4135

Structured Design
988 Bryant Way
Sunnyvale, CA 94087
(408) 988-0725

Sunrise Electronics
524 S. Vermont Avenue
Glendora, CA 91740
(818) 914-1926

Valley Data Sciences
2426 Charleston Rd.
Mountain View, CA 94043
(415) 968-2900

Varix
1210 Campbell Rd. Suite 100
Richardson, TX 75081
(214) 437-0777

Digelec
1602 Lawerence Ave. Suite 113
Ocean, NJ 07712
(201) 493-2420

2.5.2 Software Manufacturer Addresses

Assisted Technologies Division (CUPL)
Personal CAD Systems
1290 Parkmoor Avenue
San Jose, CA 95126
(408) 971-1300

DATA I/O (ABEL)
10525 Willows Rd. NE
Redmond, WA 98073-9746
(206) 881-6444

†Texas Instruments does not endorse or warrant the suppliers referenced.

3 Cache Memory Systems

3.1 Introduction

As the typical operating speeds of processors have increased to provide for the ever increasing need for computing power, the necessity of developing a memory hierarchy (the incorporation of two or more memory technologies in the same system) has become apparent. One of these memory technologies is selected on the basis of fast access time (with associated high cost per bit) to allow minimum system cycle time. The other technologies are chosen with the lowest possible cost per bit relative to speed in order to achieve the maximum system memory capacity. In a system with a multiple level hierarchy, the speed-to-cost relationship depends upon the frequency of access and the total memory requirement at that level. By proper use of this hierarchy through coordination of hardware, system software, and in some cases user software, the overall memory system will reflect the characteristics that approximate the fast access time of the fast memory technology and the low cost per bit of the low cost memory technology. Large computer systems have made use of this memory optimization technique to maintain very large data bases and high throughput (see Figure 3-1). Many smaller processor systems use this technique to allow mass storage of data, where a tape or a disk is the low-cost memory and Random Access Memory (RAM) is the fast memory technology.

Because of the increase in processor speeds, memory hierarchy is now extending to the RAM memory used in microcomputer systems. Typically, Dynamic RAM (DRAM) is used as the bulk or main memory and High-Speed Static RAM (HSS) serves as the fast-access memory. This HSS RAM is usually 1K to 8K words deep and serves as a fast buffer memory between the processor and the main memory. This small fast buffer memory is called "cache" memory because it is the storage location for a carefully selected portion of the data from the main memory. The addresses for that portion of memory currently in the buffer memory is saved in the cache tag RAM (a small memory that is used to store the addresses of the data that has been mapped to cache).

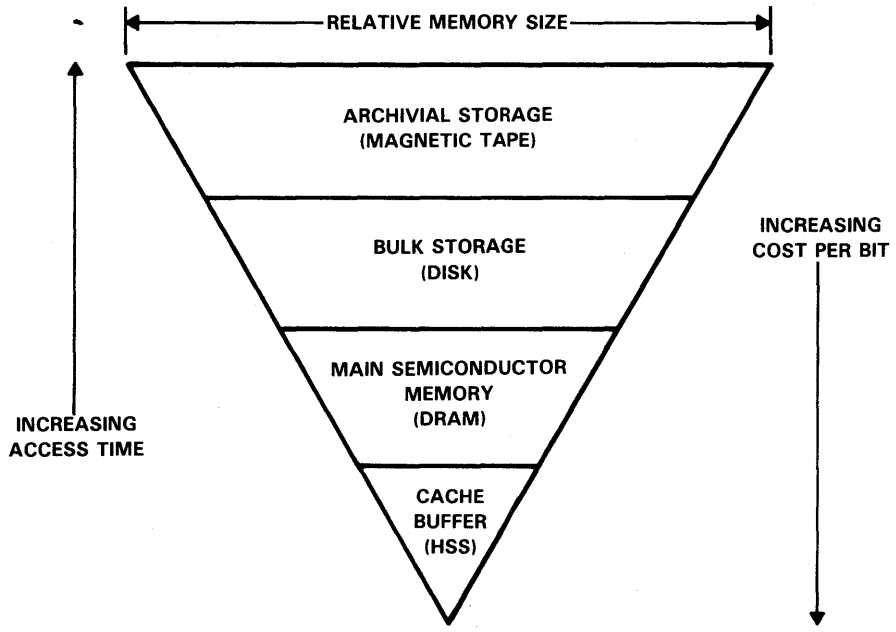


Figure 3-1. Memory Size vs Access Time and Cost Per Bit

3.2 Memory Systems with Cache

When the processor accesses main memory, the processor address is compared to the addresses currently present in the cache tag RAM. When a match occurs, the required data is resident in the cache and the access is called a "hit" and is completed in the cycle time of the fast memory. When a match does not occur (a "miss"), the main memory is accessed and the processor must be delayed to allow for the slower access cycle of the main memory. Whether a hit has occurred is determined by the cache-tag RAM. Figure 3-2 shows the relative placement of the processor, main memory, cache, and cache-tag RAM within a system.

Since there must be comparisons made between the current processor address and the addresses in the cache, the cache-tag RAM must have a very fast access time to prevent the degradation of processor accesses even when a match occurs. Previously, the memory used for the cache-tag RAM was the same as that used for the cache, which (because of added delays through comparison logic) meant that the full benefits of the cache were not realized.

The Cache Address Comparators were designed to reduce this cache access degradation to a minimum by incorporating the matching logic on-chip. This provides match-recognition times that are compatible to the access time of the cache-buffer memory.

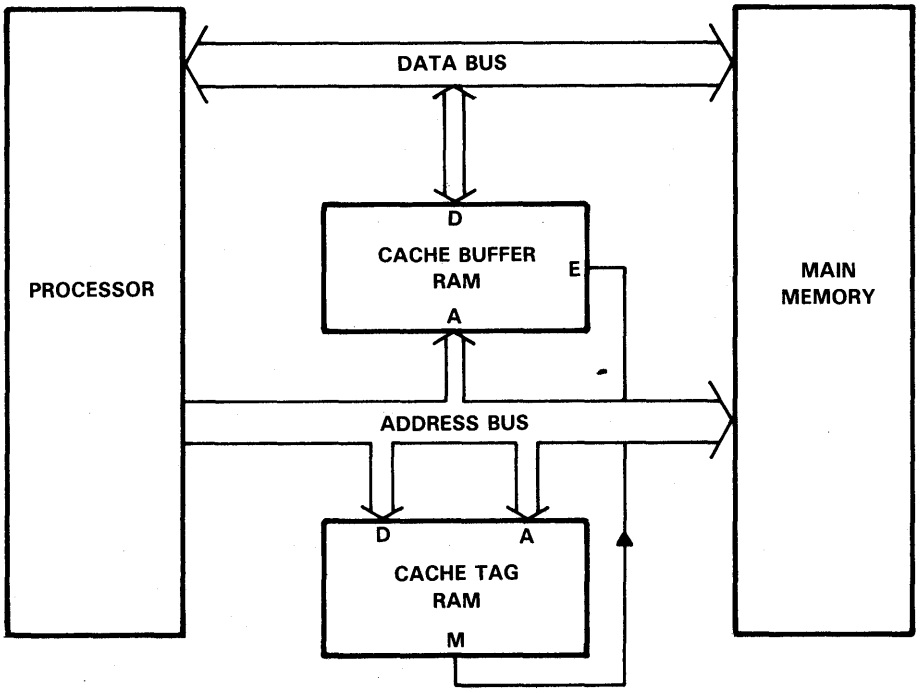
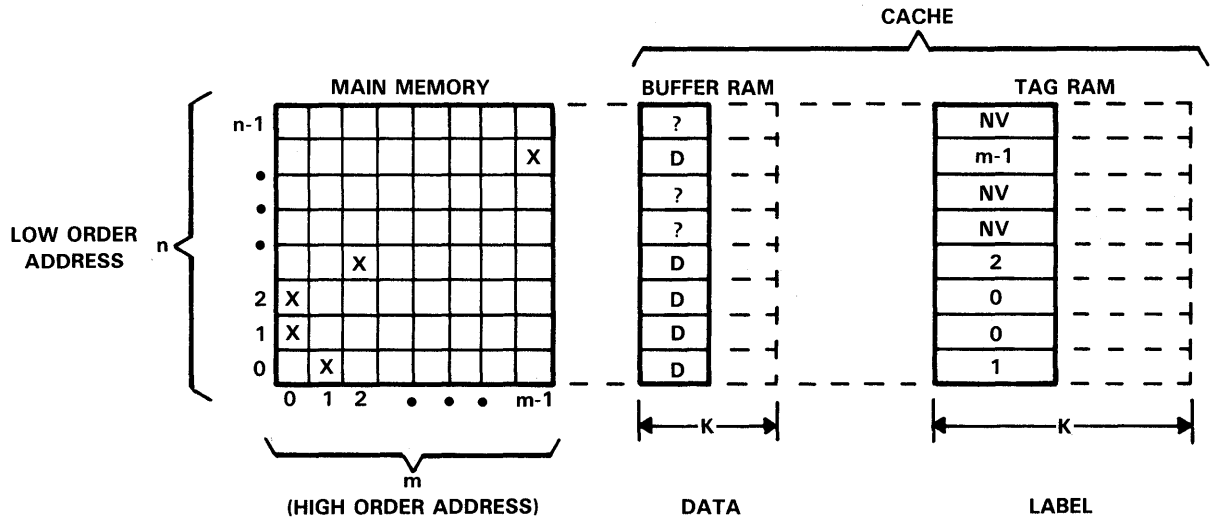


Figure 3-2. Typical Memory System with Cache

3.3 Cache Memory Systems Using 'ACT2151 and 'ACT2152

3.3.1 Set-Associative Cache Address Matching

The 'ACT2151 and 'ACT2152 implement the set-associative type of cache address matching. This algorithm may be more clearly understood by considering main memory as an (m) by (n) array of blocks and the cache is an (n) by (k) array (see Figure 3-3). Each block is composed of (x) words, and transfers between main memory and cache memory always move all (x) words in that block. Corresponding to every block in the buffer RAM is a tag address specifying which block of main memory is currently resident in the buffer RAM at that location. The set-associative algorithm maps each modulo (n) group of (m) blocks into the corresponding (n) row of the cache. The low order address lines of the processor covering the sets (n) select a row of the cache buffer and the corresponding row in the tag RAM. The data is stored in the cache buffer and the high-order address specifying the block (m) is saved in the tag RAM. The high-order address then becomes the tag.



K = Number of BUFFER/TAG groups for multiple cache systems
X = Blocks moved to cache
D = Valid data from main memory
? = Areas of cache that have not been loaded from main memory
NV = Code to indicate non-valid label
0, 1, 2, m-1 = Labels from high order address specifying the block moved from main memory.

Figure 3-3. Set-Associative Cache Address Matching

3.3.2 Cycle Time Improvement

There are several algorithms used to determine which areas of main memory should be resident in cache and which should be replaced (first-in, first-out; least recently used; or random). Since programs typically have the property of locality (over short periods of time most accesses are to a small group of memory addresses), these replacement algorithms can make the cache have the majority of processor accesses resulting in hits. The hit ratio (number of hits × 100%/number of memory accesses) runs 90% and higher in systems with well coordinated memory to cache mapping routines. As the block size (x) increases, the replacement mapping algorithm options have greater impact on the cache performance.

When running at maximum frequency, many microprocessors are operating with memory access times of 100 ns or less. After allowing for address buffering, decoding, and propagation delays through data buffers, the maximum access time that can be tolerated is 60 ns or less before processor throughput is affected. For large memory systems, DRAM can be used to achieve a cost effective memory.

However, these cannot meet a 60-ns access requirement. If the actual system throughput for a system with cache and one without cache are compared, the advantages of cache become obvious.

For comparison of the two architectures, assume that a processor is implemented in which 30% of the active cycle involve main memory (the other 70% used for instruction decoding and internal operations). Also assume that the processor cycles at 125 ns with a required memory access time of 60 ns. If the memory is not ready, the cycle time is extended by 125-ns increments till satisfied. This processor using 120-ns DRAMs would require one delay increment on main memory accesses and 200-ns DRAMs would require two delay increments. The average cycle time can be calculated for each memory speed as follows:

Average Cycle Time = [(INT) × (CYC)] + [(MEM) × (CYC + DEL)]

where INT = percent of time doing internal operations
CYC = processor cycle time
MEM = percent of time doing memory accesses
DEL = number of delay increments × 100 ns

For a processor using 120-ns DRAMs:

Average Cycle Time = [(70%) × (125 ns)] + [(30%) × (125 + 125)]
Average Cycle Time = 163 ns

For a processor using 200-ns DRAMs:

Average Cycle Time = [(70%) × (125 ns)] + [(30%) × (125 + 250)]
Average Cycle Time = 200 ns

For the same system with cache memory assume a 90% hit ratio with 60-ns cache and 120-ns DRAM:

Average Cycle Time = [INT × CYC] + [MEM × [(HIT × CAC) + (MIS × (CYC + DEL))]]

where INT = percent of time doing internal operations
CYC = processor cycle time
MEM = percent of time doing memory accesses
DEL = number of delay increments × 100 ns
HIT = percent of memory accesses hit cache
MIS = percent of memory accesses miss cache
CAC = cache memory access cycle time

Average Cycle Time = [70% × 125] + [30% × [(90% × 125) + (10% × 125 + 125))]]
Average Cycle Time = 129 ns

This value represents a 20% improvement with 120-ns devices over the non-cache implementation with 120-ns devices and 35% using 200-ns devices. This performance improvement can be further demonstrated for those systems using custom or bit-slice processors where the memory cycle time as well as access time is of concern. For this example, consider a processor with a cycle time of 50 ns and main memory cycle time of 100 ns (use the same access ratios as in the previous example):

Average Cycle Time = [(70%) × (50)] + [(30%) × (100)] = 65 ns
(Without Cache)

Average Cycle Time = [70% × 50] + [30% × [(90% × 50) + (10% × 100)]
(With Cache) = 52 ns

This represents a 20% decrease in average cycle time for the processor using 50-ns cache memory. If the main memory was rated at a cycle time of 200 ns, either using slow main memory or due to allocation of alternate cycles for some other activity (multiprocessors, direct memory access, display refresh, etc.), the cache would still give an average cycle time of 55 ns. This is an improvement of 63% over the 95 ns average cycle time for a non-cache system.

3.3.3 Cache Memory Configurations

Figures 3-4, 3-5, and 3-6 illustrate applications for the 'ACT2151 and the 'ACT2152 in cache memory systems. Figure 3-4 shows a cache-memory configuration that has a 512M-byte main memory with a block size of 4 32-bit words. In this particular application, a cache containing 1024 four-word blocks was chosen thus defining the main (n) × (m) array as being 1024 sets of 32,728 four word blocks. The 128M-word memory requires an address bus of 27 lines. The least significant bits (A2-A3) are used as a word select for one of the four words in each block. The next least significant address lines (A4-A13) are used as the set select inputs to the cache buffer RAM and the cache tag RAM. The remaining high order address lines (A14-A28) form the label or tag which is stored and compared by the tag RAM.

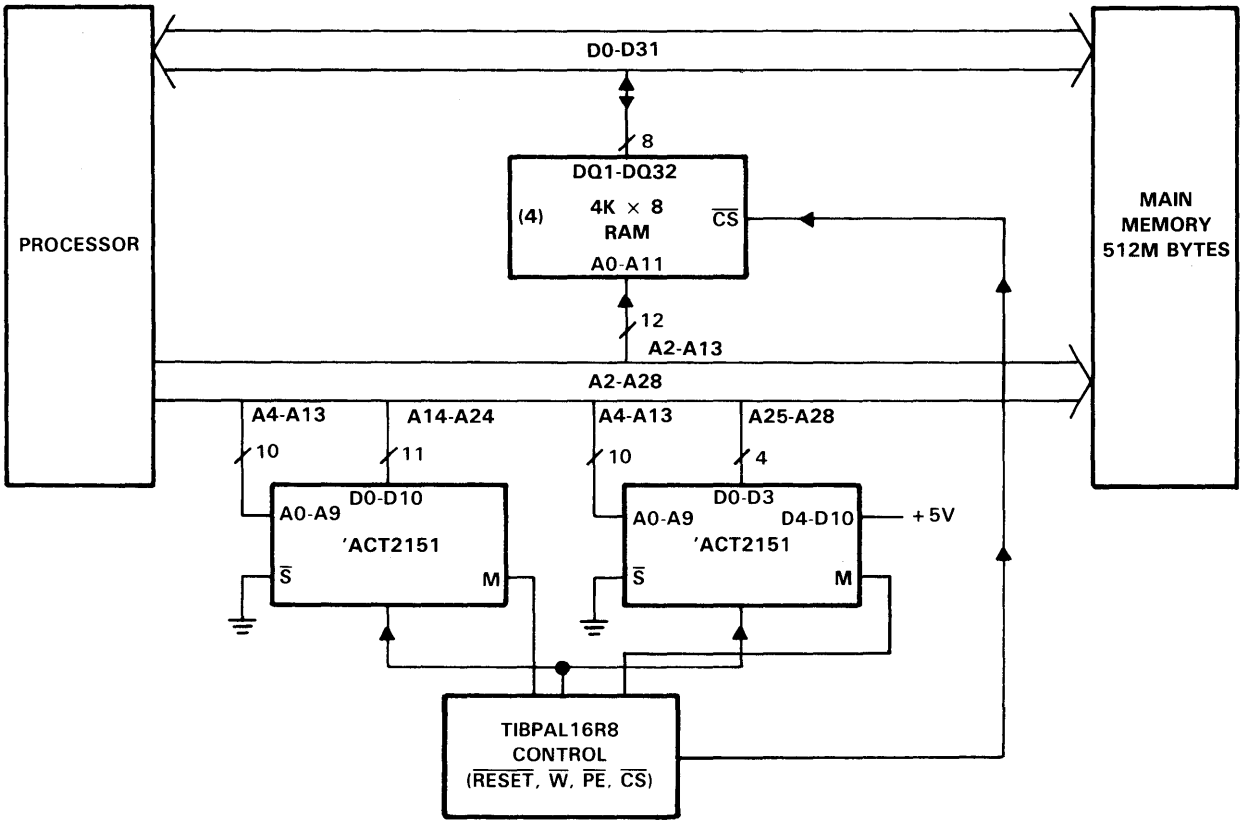


Figure 3-4. Cache Memory Configuration (Block Size = 4)

Since the label in this example is composed of 15 address lines, two 'ACT2151 devices are used to expand the tag. The 15 address lines are the data inputs to the tag RAM. The other data inputs are tied to 5 V so that, after Reset, invalid data cannot force a match. The match output of the two 'ACT2151 devices are combined to form the enable for the cache data buffer. If the contents of either 'ACT2151 do not contain a match, the cache is not enabled. These signals are also used by the control circuits to inform the system that the address is not present in the cache so that main memory might be accessed. The control circuit also resets the cache upon power-up. This is accomplished by taking the $\overline{\text{RESET}}$ input of the 'ACT2151 low. After reset, no matches will occur at any locations until that location has been written.

In the application shown in Figure 3-5, the expansion of the cache RAM is carried out in both depth (more sets) and width (wider tag). The block size was chosen as one such that the 4K cache now represents 4096 blocks of one word each. The high-order addresses are still used as the label to the tag RAM. A13 is used to select between two 'ACT2152 pairs. Each pair contains labels for 2048 of the cache-memory blocks. Address lines A2 thru A12 are used as the set-address inputs. If the chip select ($\overline{\text{CS}}$) is at a logic high (deselected), the 'ACT2152 match output (M) is high. An AND function can be used to enable the cache data buffers and also notify the control circuit if access needs to be made into the main memory. The logic for this system illustrates that the upper pair are compared for the first 2048 blocks within cache and the lower pair are compared for the second depending on the state of address A13.

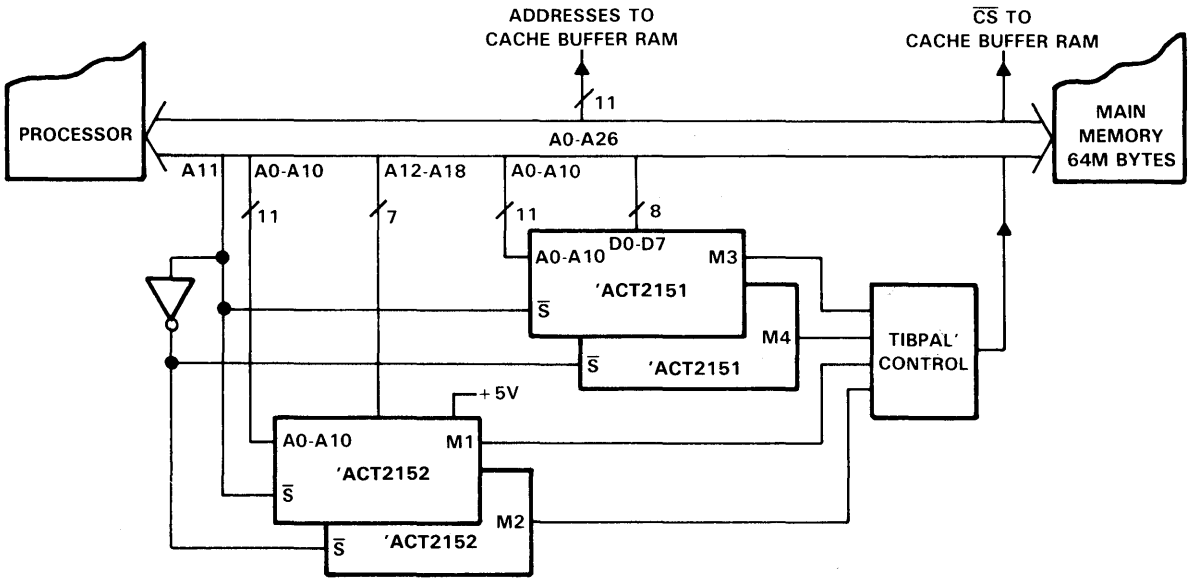


Figure 3-5. Cache Memory Configuration (Block Size = 1)

A dual cache structure (K = 2) is shown in Figure 3-6. The M-word memory is divided into 4096 sets of 256 four-word blocks. In this example, A0 and A1 are used to select which one of the four words within a block are accessed. A2-A12 select which of the 2048 block labels are to be compared. Addresses A14-A21 form the eight-bit label for the block. Address A13 is used by the cache control logic in conjunction with the possible processor status lines as chip select inputs. The match outputs from the two 'ACT2152 devices, A1 and A2, are NANDed to form an active-low enable to the cache data buffers and to serve as a request to the control logic. The match outputs from B1 and B2 also are NANDed to perform a similar function for cache RAM B. If no match is found in cache RAM A or B, the control logic will initiate an access from main memory. The purpose of the dual-cache architecture is to allow for rapid switching between multiple tasks or programs since the processor can have access to one cache while the controller moves data between main memory and the other cache. The dual or multiple cache approach also yields more replacement options than the single cache architecture. When an access results in a miss in the single cache system, the data in cache is replaced by the current data even though the old data may still be useful. By using independent caches, the control can determine which data is most expendable and replace that block while the other caches keep their potentially useful data.

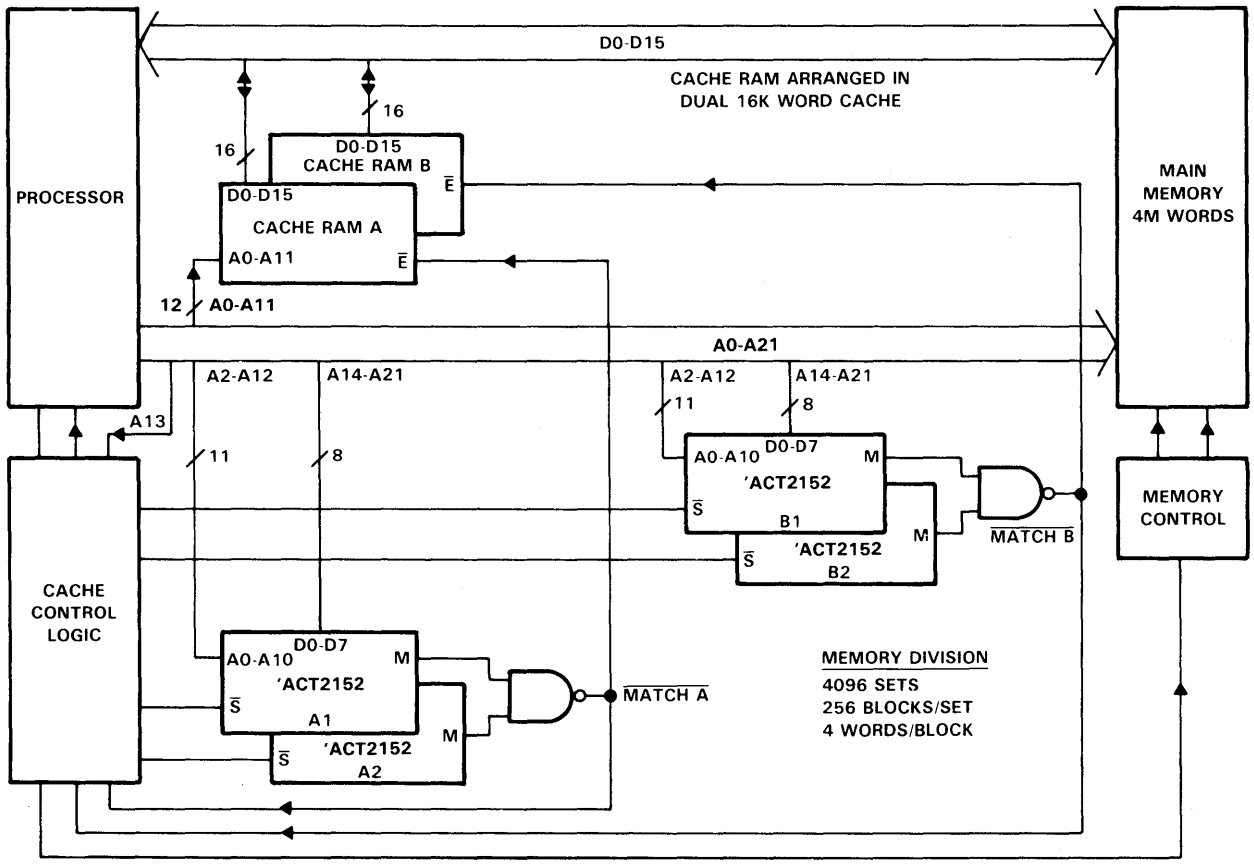


Figure 3-6. Cache Memory Configuration, Dual Cache (K = 2)

3.3.4 Summary

Cache-memory architecture can enhance the throughput of many microprocessor systems. This allows large low-cost memory to perform like a high-speed RAM. The 'ACT2151 and 'ACT2152 reduce the tag memory implementation cost and complexity and provides label comparison times comparable to the access times of high-speed memories. These additional benefits make high-performance microprocessor designs that can use the same techniques of optimizing cost, memory size, and throughput that had previously been available only in larger computer applications.

3.4 Article Reprints

The following three articles are being reprinted in this report for your convenience. The articles are "Caches Keep Main Memories From Slowing Down Fast CPUs", "Cache-Memory Functions Surface on VLSI Chip", and "Match Cache Architecture to Computer System".

Until main-memory speeds catch up to CPU processing speeds, cache memories can be called upon to keep overall throughput rates up—especially as main-memory sizes grow.

Caches keep main memories from slowing down fast CPUs

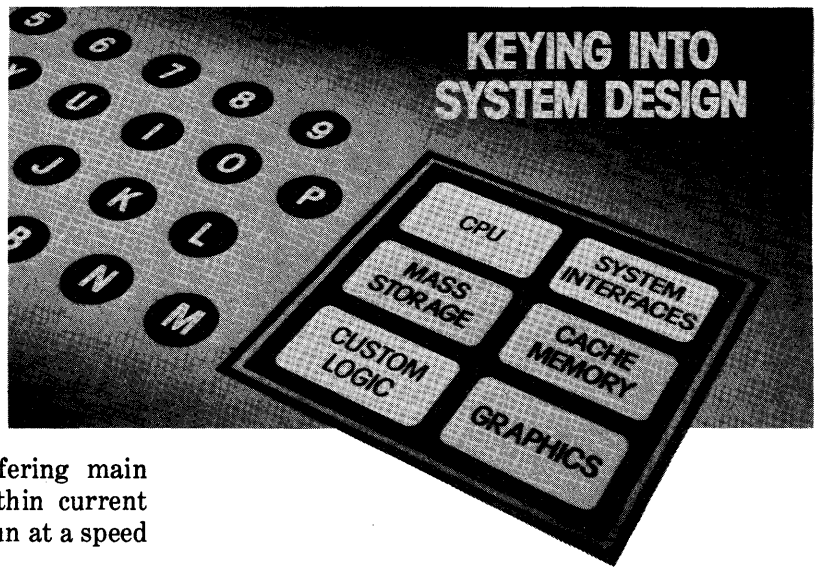
This article begins a series on cache-memory systems long employed in mainframes and high-end minicomputers, and just now ready to enter microcomputers as large, but slow dynamic RAMs become available. The strategy of Texas Instruments is sketched by Richard N. Gossen, manager of advanced development, in this issue (p. 32).

Whenever a speed mismatch occurs between main-memory bulk storage and a fast-processing CPU, a cache memory can provide the interface to take full advantage of the CPU's processing speed. From the main memory, the cache memory extracts and temporarily stores enough data to satisfy immediate CPU needs. Writing from the main memory is at the main-memory's slow speed, but reading to the CPU is at the CPU's high speed. As the word "cache" implies, the memory's operation is hidden from (or rather, transparent to) the user.

Figure 1a exemplifies an 8-Mbyte main-and-cache-memory system in a typical large computer (see "Caches Needed as Main Memories Grow"). The main memory alone can attain a cycle time of about 400 ns, but with error-detection-and-correction circuitry added to the system, a cycle time of about 500 ns is more likely. On the other hand, an ECL- or TTL-based CPU can be ten times faster with about a 50-ns cycle time. Buffering main memory with a 50-ns cache—well within current technology—enables the computer to run at a speed close to the CPU's maximum speed.

Figure 1b shows a more detailed block diagram of a typical cache memory. It represents a set-associative cache system with 2-kbytes of data storage capacity in a single-set configuration and it serves a 16-bit microprocessor system with a 22-bit address bus. The basic storage elements are two RAM arrays: one, a 1024 × 16-bit-word unit for data storage; and the other, a 1024 × 13-bit-word unit for address, or tag, storage. Addresses that arrive via the CPU data bus (A_0 to A_{21}) are compared with those held in the tag RAM. If they match, the desired data are located in the cache's data RAM and the main memory can be bypassed.

When first turned on, or should the computer system malfunction or be shut down momentarily, the cache would then probably contain improper or erroneous data. Thus, a most important function not



Cliff Rhodes, Static-RAM Design Manager
Texas Instruments Inc.
4000 Greenbriar Dr., Stafford, TX 77477

specifically shown in the block diagram is a special control-logic section to perform initialization operations for loading the cache with valid initial data. Particularly important is the proper loading of the tag RAM to prevent false matches.

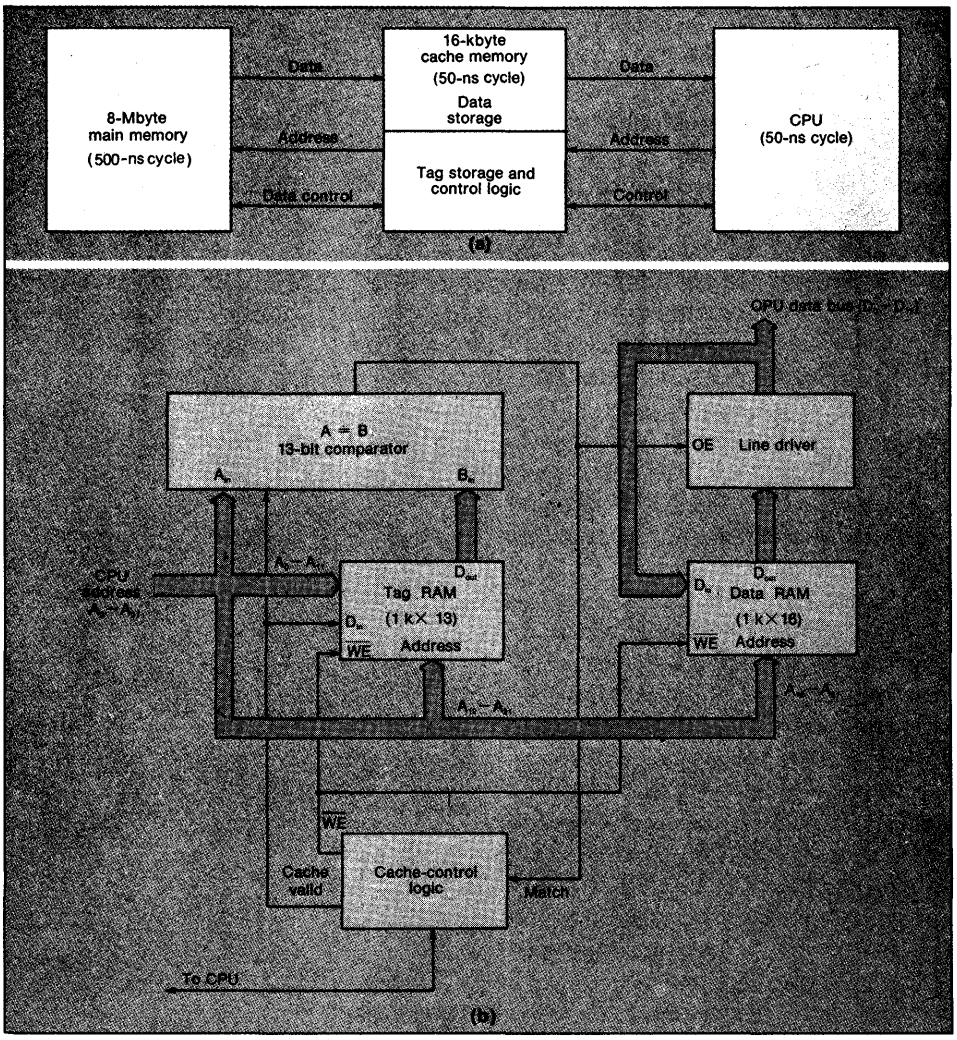
Traditionally, cache memories have been constructed with static-RAM bipolar-semiconductor technology, but recent improvements have given NMOS static RAMs an edge by requiring less power at the same speeds as bipolars. The CPUs, of course, are built from high-speed ECL or Schottky TTL, whereas main memories usually are composed of dynamic RAMs that are about an order of magnitude slower than the CPU.

Cache concept based on probability

No matter what the cache is made of or how it is configured, its operation is based on "property-of-locality" probability principles, which experience has shown to have the following characteristics: First, over short time periods, most CPU memory accesses are made to adjacent, small groups of locations; therefore, even a small cache, storing carefully selected data, will have data the CPU needs most of the time. Second, data stored in the cache and recently used will likely be reused shortly thereafter. Finally, data adjacent to data that have been recently used will most likely be used next.

Usually, then, several adjacent words are transferred from main memory into the cache: The immediate need may be for just one of the words, but eventually the subsequent words are likely to be required. This procedure reduces repeated accesses to main memory ("misses"), and increases the probability of finding the data in the cache, or "hits."

To demonstrate the effectiveness of a cache, consider a typical system, where 20% of all CPU operations are memory accesses (misses), the CPU cycle-



1. In a computer system with a main memory, whose cycle time is a relatively slow 500 ns, a fast 50-ns cache memory is interposed to match the CPU's 50-ns cycle time (a). In more detail, a set-associative cache system has address tag words stored in one RAM, while the data words are stored in a separate RAM (b).

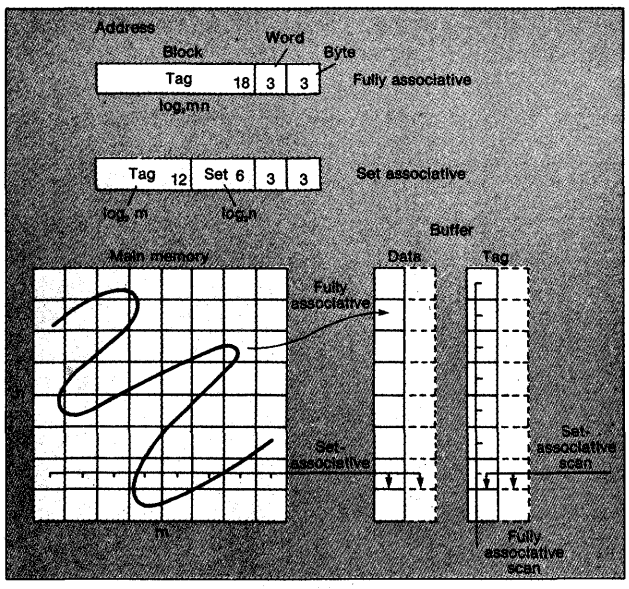
time is 50 ns, and main-memory cycle-time is 500 ns. Accordingly, average machine-cycle time is

$$\frac{20 \times 500 + 80 \times 50}{100} = 140 \text{ ns.}$$

However, with a 90%-efficient, 50-ns cache, the average cycle time is

$$\frac{2 \times 500 + 18 \times 50 + 80 \times 50}{100} = 59 \text{ ns.}$$

Note that with an effective 90% hit ratio, the CPU is forced to access the main memory on just 2% of all machine cycles (10% of 20% of the cycles). Thus, most memory accesses are handled through the cache, and the average machine cycle-time is cut to almost a third. However, these calculations are simplified, because only a monoprogram instruction stream is considered. Indeed, properly designed cache memories routinely achieve considerably better performance in mainframes and minicomputers with actual, more complex programs.



2. A fully associative mapping algorithm allows any one of the m × n main-memory blocks to be placed into any one of the cache-memory blocks.

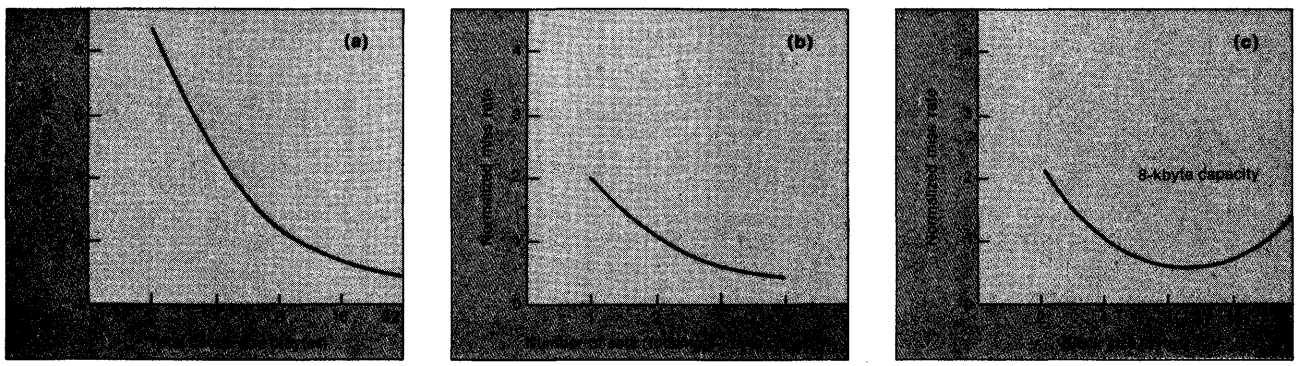
A cache memory's data-storage section can be implemented fairly simply with standard, high-speed static RAMs. However, the address, or tag-storage, section must do more than just store—it must also compare addresses on the CPU bus with those it stores. This is best done with an associative-addressing technique.

Data access by association

Data in an associative or content-addressable memory are not accessed merely by a location, or address, code as in conventional memories, but are found according to some property or "value" of the data. Instead of an address word, a so-called search key, or descriptor, is presented to the cache, which represents particular values of all or some of the bits of a stored word. When it is compared together with a "lock"—the so-called tag bits—with all the words stored in the cache, the search key ferrets out all associated words. If the key has few attributes—is therefore said to be "loose"—many words can match and be accessed.

Though simple in concept, the associative-search procedure is very complex in execution. The two most common mapping algorithms that associate a set of data in main memory—called a "block"—with a corresponding block in the cache are designated "fully associative mapping" and "minimal set-associative mapping."

In fully associative mapping (Fig. 2), any one of the m × n blocks in main memory can be placed in any one of the cache blocks, which then has a tag address associated with it that specifies from which main-memory block it came. (One of the tag bits—a control bit—checks the validity of the block, and



3. The larger the cache, the smaller the number of misses (a). Splitting the cache into several independent sets further reduces the miss ratio (b); however, in both these cases the improvement rate tapers off sharply beyond some specific point. And when block-size and block-quantity are traded off against each other (c), miss rates are minimized sharply at some particular optimum size/number relationship.

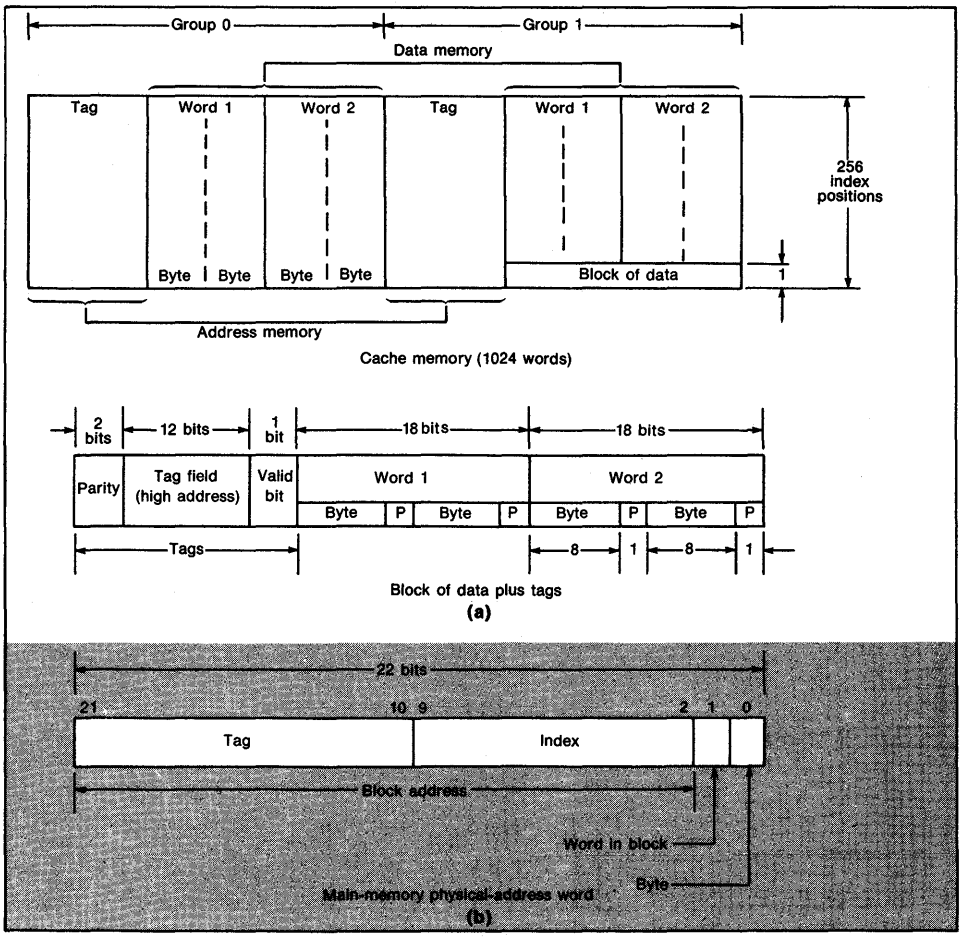
the low-order address bits can define bytes and data-transfer units, or words.) Each address generated by the CPU must be compared with all of the tags, and the field of address tags must span all main-memory m × n blocks, regardless of the cache's capacity. In this case, the cache acts as a linear array.

In set-associative mapping, a selection from an n row of m blocks is placed into the corresponding row of the cache. Then only those bits covering dimension m become the tag of the set n. Thus, for each CPU address only the tags in that row must be compared. However, for the minimum effective set—having a dimension of two—a linear array must swap blocks frequently because the cache cannot hold more than one block from a row at any time.

Whether fully associative or set-associative mapping, if data-block addresses in the cache tag-store match those on the CPU address-bus, the data are made available from the cache. And if no match, or a miss, occurs, the CPU is delayed while the needed data are fetched from main memory. In this process, the entire block containing the data sought is transferred into the cache.

Of course, the larger the cache, the smaller the number of misses on each CPU cycle. However, for a normalized miss-rate-vs-cache-size plot that assumes a fixed number of blocks in the cache and a main-memory size from 2 to 4 Mbytes, the rate of improvement in miss rate diminishes rapidly above 8 kbytes of cache size (Fig. 3a). Thus, the cost of a very large cache is not paid back in higher performance, when optimum size is exceeded.

However, further improvement can be obtained by breaking the cache capacity into a number of sets —defined as the number of parallel, independent caches in a system. For the same 2-to-4-Mbyte main memory and with the total cache size fixed at 8 kbytes, two separate caches of 4 kbytes each offer better performance than a single 8-kbyte implementation, because another quick data-replacement trial is available whenever a miss occurs the first time (Fig. 3b). With a single set, a miss forces accessing the slow main memory, which replaces all the data in the cache, even though the replaced data may soon be required again in the program. However, again performance increase slows significantly above an



4. The cache for the PDP 11/70 is organized into two 256 blocks of data totaling 1024 words (a). Every block has a tag field representative of the physical address of the word in the main memory (b).

optimum size—in this case, above between two and three independent sets for the 8-kbyte cache capacity.

Naturally, block size and the number of blocks also affect miss rate. However, in a constant-capacity cache, trading the number of blocks against their size seems to raise a conflict: Large blocks accommodate more adjacent data and thus would tend to reduce the miss rate. But a higher number of smaller blocks also would help reduce misses by providing more data choices. Figure 3c shows a minimum miss rate at about 8 bytes per block, or 1000 blocks, for a total 8-kbyte cache capacity. For block sizes between 2 and 16 bytes, an 8-, 16-, or 32-kbyte cache offers the same normalized miss-rate performance, although larger the total capacity, the smaller the absolute miss-rate.

To complete this general overall description of cache operation, one additional important function must be examined—the data-replacement algorithm. Again, any time a cache memory records a miss, a new block containing the required data must

Caches needed as main memories grow

Although advanced microprocessor-based systems are beginning to see the dawn of the cache-memory era, large systems like the IBM 360 class and large minicomputers like the DEC PDP-11 series have been "caching-in" for years. On a cost-effective basis, a cache system offers higher system speed for the cost of just a small quantity of fast memory plus its associated logic.

The resulting speed depends on the size and organization of the cache, not the size of main memory, and no programming changes follow when a cache system is used. Nevertheless, it is the increased main-memory size that will fuel the growth of cache-type systems in the upcoming high-performance microcomputers and microprocessor systems.

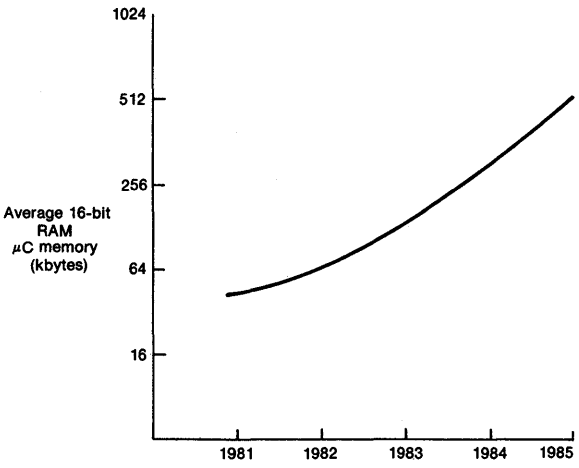
The high-density RAM chips that will significantly increase the main-memory sizes that microprocessors and microcomputers will be called upon to support are relatively slow dynamic units (see table). By 1983, as the 64-kbit dynamic RAM becomes a cost-effective chip, the average 16-bit processor will be operating with 128 kbytes of memory. And by 1985, with the expected maturity of the 256-kbit dynamic RAM, ½-Mbyte memories will be commonplace in 16-bit processor systems (see curve). Already, TI's 16-bit 99000 processors can address up to 16 Mbytes with the addition of a memory-management unit. Systems of this size, like mainframes and large minicomputers, will demand cache memories to enhance performance.

While physical memory size encourages the growth of cache systems, improved microprocessor performance also contributes to wider cache use. Processor speeds will certainly increase, necessitating fast cache memories to enable CPUs to make full use of their improved capabilities. The accompanying table projects the expected speed performance in 1982 of several of the best known processors.

1982 projected 16-bit μC performance

Processor	Maximum clock rate	Fastest supportable memory access
99000	6 MHz	90 ns
8001A	6 MHz	215 ns
68000	8 MHz	225 ns
8086-?	8 MHz	280 ns

For example, TI's 99000, with a maximum clock-rate of 6 MHz, is expected to support memory access times of 90 ns. This meshes perfectly with the new generations of NMOS static RAMs, whose access times are now in the 30-to-50 ns range. A 99000 will almost certainly benefit from a cache-memory system when it supports main memories in the megabyte range. Indeed, the Zilog Z8001A, the Motorola M68000,



and the Intel 8086-2 are somewhat slower than the 99000 in memory-access times; therefore, they should most definitely benefit from a cache for high-performance applications.

In such microprocessor systems, dynamic RAM used as the main memory will always remain the limiting factor to improved system performance because it is slower than a microprocessor. Even a dynamic RAM with access time as fast as 150 ns slows considerably when operated in a 1-Mbyte memory system using error-detection-and-correction circuitry. The best performance of such a RAM is in the range of 400 to 500 ns, minimum. If a processor is forced to interface at this slow speed, severe performance penalties result in the system.

be fetched from main memory to replace the block already in the cache. But which block should be removed to make way for the new information?

Clearly, replacement should be based on some type of index of value for maximum effectiveness, not randomly performed. An index of value can be based on the chronology of the data, such as FIFO (first-in, first-out), or the frequency of use, LRU (least recently used), or a combination of the two. Of these, LRU is one of the most popular techniques. It is based on the theory that if information has been often and recently referenced, it is likely to be referenced again in the near future.

LRU offers some advantage over the FIFO algorithm. Even though FIFO eliminates the possibility of loading data and immediately removing it, FIFO has a serious disadvantage: Even when a block of data is frequently and continually used, eventually it becomes the oldest and is removed, although experience shows it likely will be needed again, soon. In addition, FIFO can introduce some unusual side effects.

The associative cache in the PDP-11

In an actual cache system, say the PDP-11/70, a 1024-word (2048-byte) memory is organized as an associative cache in two groups, or sets—each group containing 256 blocks of data and each block containing two words divided into two bytes (Fig. 4a). Every block also has a tag field to represent the physical address in main memory, where the original copy of the data-block resides.

Data from main memory can be stored in the cache in an index position determined by its main-memory physical address (Fig. 4b). An 8-bit index field (bits 2 to 9) of the main memory's 22-bit physical-address word determines which of the 256 cache-memory-array blocks will contain the data (either in group 0 or group 1 as determined by the hit or miss conditions). And the lowest two bits (bits 1 and 2) select word-1 or word-2 and byte locations in the block. But only the high-address field (bits 10 to 21) —the tag field—is stored in the cache.

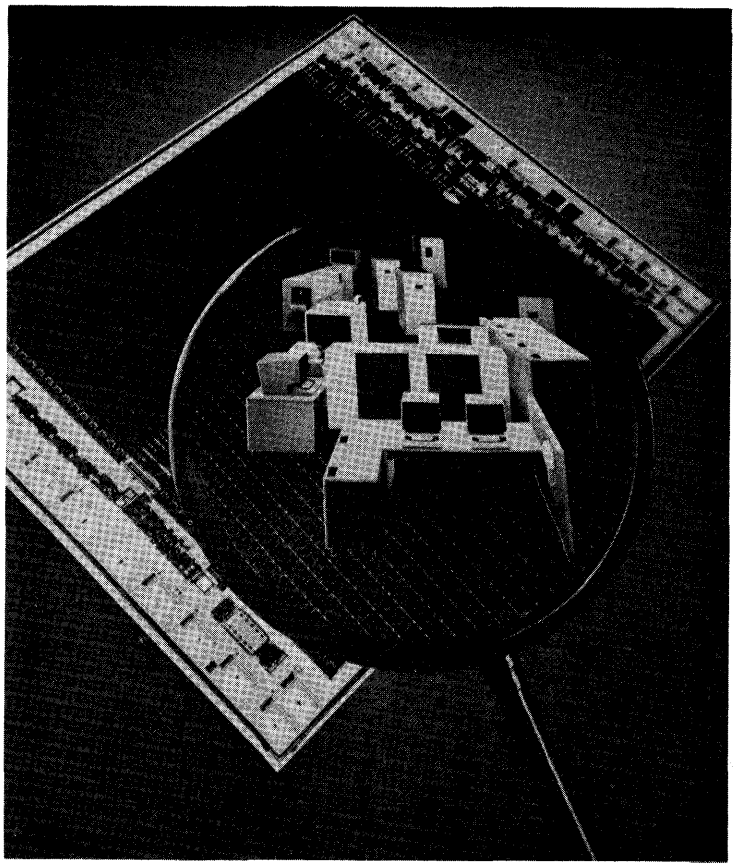
Data are always sought in the cache first. If the information is not present—a miss—a two-word block of data is transferred (written) from main to cache memory. In a typical program, writes to the cache occur just 10% and reads from the cache, 90% of the time. Read hits average 80 to 95% of all memory operations in a typical program.□

Design

A cache-tag store and comparator on a single chip will reduce parts count, save space, and also greatly simplify cache systems in upcoming minicomputers and microcomputers having large memories.

Cache-memory functions surface on VLSI chip

This is the second article in a series on cache-memory systems. The first article, which covers the basic philosophy of cache systems, appeared in the Jan. 21 issue (p. 179). The overall approach was sketched by Richard N. Gossen, manager of advanced memory development at Texas Instruments, in the same issue (p. 32).



Given the growth now occurring, and in the offing, in the main-memory size of minicomputer and microcomputer systems, cache memories will be needed to take full advantage of their CPUs' speed. The TMS2150 cache-address and comparator IC represents a major step in simplifying the cache designer's task, as it handles most of the so-called tag functions—cache-address storage and comparison.

A cache memory is a small, fast buffer memory interposed between a fast CPU and a relatively slow main memory, like a dynamic RAM. In this way, with anticipated and frequently used information prestored in the cache, the CPU can obtain most of the data and addresses it needs at a speed comparable to its own. By proper design, the number of information accesses to the large but slow main memory can be reduced to a minimum. With a special memory-mapping technique, a small number of cache storage locations can represent large blocks of backing-memory information.

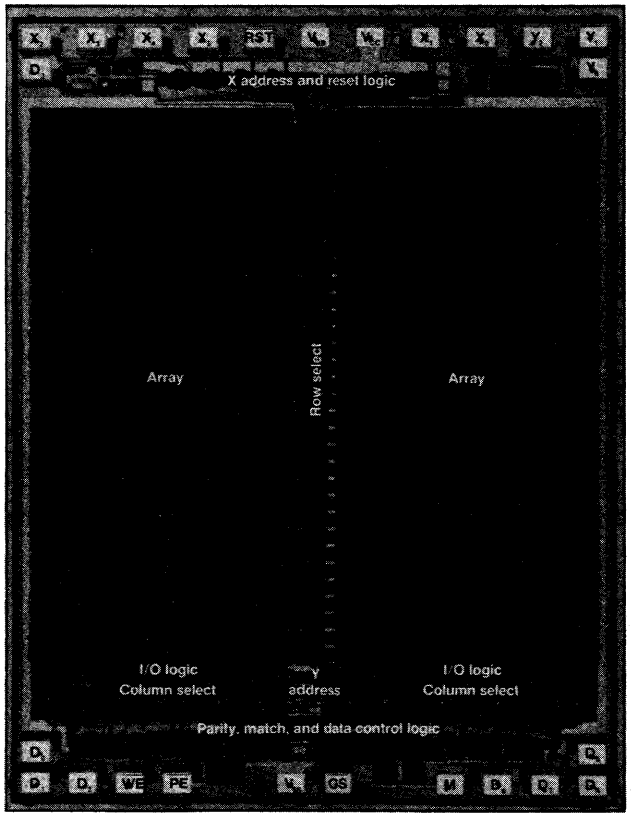
The cache, a fast static RAM, is divided into two sections—the tag store for the cache addresses; and the data store for numerical, program, or other types of data. Cache memories, however, require more than mere storage. Just as important is high-speed data comparison to check a portion of the CPU address field against the tag addresses previously stored in the static RAM. This operation determines whether the data addressed by the CPU resides in the cache.

The 2150 (Fig. 1) stores the cache tags (or addresses) in a 512-word × 9-bit static RAM, and also contains a 9-bit comparator. In addition, it generates and checks parity. The RAM's high speed, of course, matches or exceeds that of most available

Clifford C. Rhodes, Static RAM Design Manager
Jino Chun, Memory Design Engineer
Troy Herndon, Memory Design Engineer
Texas Instruments Inc.
P.O. Box 1443, Houston, TX 77001

microprocessors, and the 9-bit comparator circuit, which is integrated into the chip's memory-sensing amplifiers, is about 50% faster than bipolar comparators currently found in cache systems.

Housed in a 24-pin, 300-mil ceramic DIP, the 2150 works over an ambient temperature range of 0° to 70°C. Operation is from a single +5-V power supply, and the chip interfaces directly with both TTL and MOS logic circuits. Because of the ceramic package, power dissipation can go as high as 660 mW; typical dissipation, however, is 400 mW. To simplify cache-system design, the 2150 works fully statically—no clock or synchronizing signals needed—and it is easily expandable to fit any size processor bus or memory system (see "VLSI Built with Proven Techniques").



1. The TMS2150 cache-tag store and comparator, a single VLSI chip housed in a ceramic 24-pin DIP, occupies 24,600-mil² of silicon and is fabricated with proven 4.5-μm design rules and NMOS technology.

The use of the 2150 makes for a minimum chip-count cache system, especially in conjunction with the companion TMS2149 1-k-×-4-bit static RAMs to store the data. The 2150 alone replaces 14 chips in conventional systems. Still, even the simplest cache-control circuit with the 2150 requires several TTL devices for buffering and control and some fast RAMs (like the 2149s) for storage.

In the block diagram of the 2150 (Fig. 2), the tag static-memory array of 64 rows by 72 columns is organized into the 512 words of 9 bits each, for a total of 4608 bits.

Initializing this memory is simple: Merely pulsing the $\overline{\text{Reset}}$ terminal low to clear all 512 memory locations forces the chip's Match output terminal high; and the reset pulse can be as short as 35 ns. Initializing a conventional cache memory, however, is much more complex. It requires a set of sequential operations that is time-consuming and demands far more hardware than the 2150's asynchronous single pulse.

A read cycle is enabled when the chip-select ($\overline{\text{CS}}$) input is driven low while the write-control input ($\overline{\text{W}}$) is held high (Fig. 3a). During this cycle, nine input-address bits (A_0-A_8) select a 9-bit word in the memory array for comparison with eight input-data bits (D_0-D_7) and an internally generated parity bit. Upon a valid match, the Match output terminal goes high. However, if the parity check indicates an error in the internal-memory data, the parity-error output ($\overline{\text{PE}}$) and the Match output go low. The $\overline{\text{PE}}$ output is an open-collector type, allowing simple $\overline{\text{OR}}$-tie connections to other devices.

For a write cycle, both $\overline{\text{CS}}$ and $\overline{\text{W}}$ must be driven low. Then, the data on the D_0-D_7 terminals, plus an even parity bit from the internal parity generator are written into the memory-array location addressed (or rather tagged) by A_0-A_8. A parity error can be forced by holding the $\overline{\text{PE}}$ terminal low, which is very useful for testing.

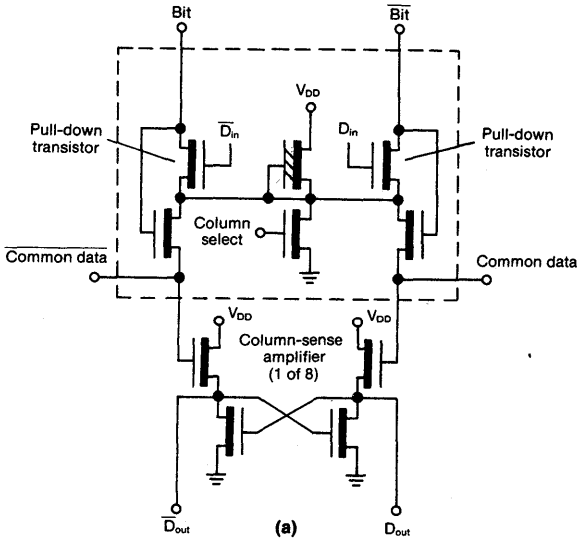
The 512-×-9-bit tag memory-array structure permits the system to be expanded in building-block fashion for either wider or deeper tag stores. Patterned after bit-slice techniques, the 2150 can be considered an 8-bit-slice cache-address and comparator; accordingly, a 16-bit word could be divided into two 8-bit segments and operated on in parallel by two cache systems, speeding up performance in comparison with serial operation.

A key speed specification of a cache-address comparator is the delay time, t_d (A), needed for the signal to go from the address input to the match outputs. Generally, this specification is a worst-case delay path in a cache-memory system. The 2150 is available with four delay versions—maximums of 45, 55, 70, and 90 ns—to meet a variety of cache-memory

VLSI built with proven techniques

Occupying just 24,600 mil² of silicon, the TMS2150 cache-address store and comparator is fabricated with conservative, 4.5-μm design rules and 2.5-μm NMOS polysilicon gate lengths. Many of the 2150's circuit techniques were first proven on the TMS2147H and TMS2149 4-k low-power, fast static RAMs. One such circuit, a distributed column-sensing amplifier (Fig. a), significantly improves the speed-power product over that of previous high-speed MOS designs.

During read cycles, tag data stored in the 2150's on-board memory are not directly accessible; instead, they are compared with the input data and checked for parity. Since this parity check must be performed at high speed, the sense amplifiers must reach valid



(a)

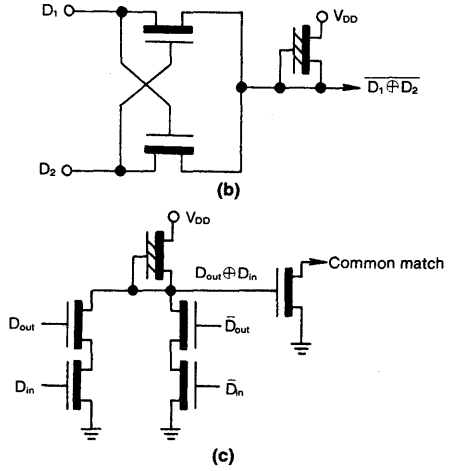
logic levels very quickly. That is beyond conventional differential amplifiers, but does not faze the distributed, column-sensing amplifier in the 2150. One method employed in the sense amplifier to help achieve the required speed is to isolate the bit lines (BIT and $\overline{BIT}$) fully from the data lines (D_{out} and $\overline{D}_{out}$) thus reducing the amplifier loading, which would hold speed down.

A cross-coupled pair of FETs driven by data-line source-followers acts as the main sense amplifier. This circuit provides fast level shifting, high gain, and excellent performance over the total expected range of semiconductor processing variations and operating temperatures.

Reading occurs when a column is selected by a high-level signal to the circuit's column-select input. This input activates the column amplifier, which differentially drives the pair of common-data lines to the sensed state, which is then quickly transformed into sharp, clean logic levels by the sense amplifier.

To write data into the on-board memory, the pair of data-in signal lines (D_{in} and $\overline{D}_{in}$) must activate the gates of the bit-line pull-down transistors. With the desired column selected, forcing a data-in line to the V_{DD} voltage level pulls the associated bit line low to write the data in. At the same time, the complementary Data In line is held low, which permits the complementary bit line to rise to V_{DD} via the bit-line bias circuitry (not shown).

Because the bit lines are fully isolated from data-line loading, they can be driven efficiently by the



chip's small memory-cell transistors. Also, the sense-amplifier circuit has a precisely controlled differential-voltage gain. Moreover, the dynamic requirements for the column-decoding circuitry (also not shown) are light, because only one small transistor activates the column selection. And since the transistor's source terminal is at ground potential, a relatively low column-selection voltage is sufficient to activate the column.

In addition, the Exclusive-NOR gates were specially designed to minimize chip real estate. To simplify the layout, the parity circuit's Exclusive-NOR gate (Fig. b) requires just a single-polarity input signal, minimizing the needed interconnection area. Similarly, the comparator's Exclusive-NOR circuit (Fig. c) uses a common match line for the 9-bit comparator circuit to hold down the interconnection area.

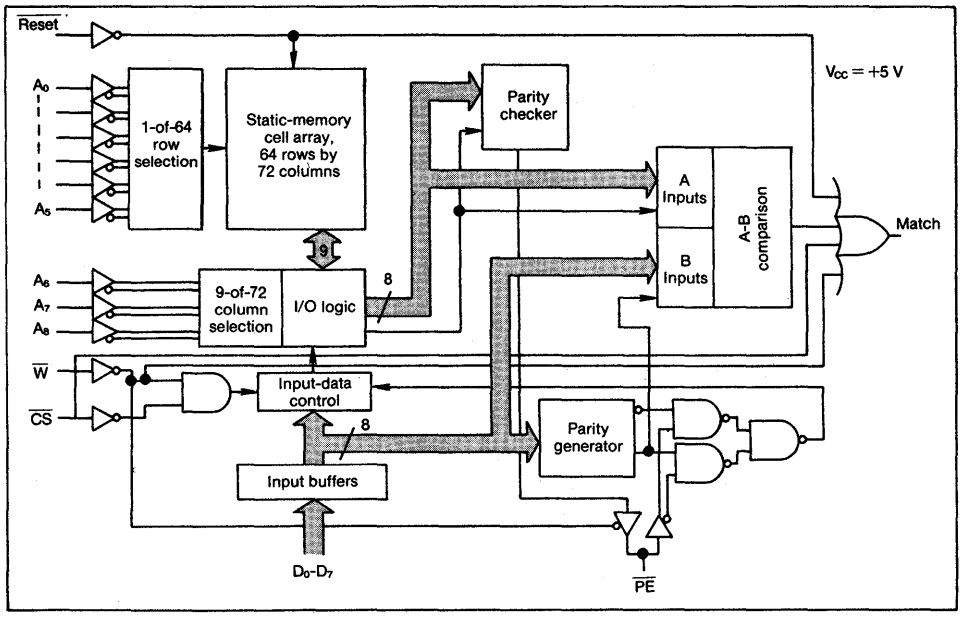
system speeds (Fig. 3a shows a unit whose t_d (A) is about 30 ns).

The fastest specified version (45 ns) is about 20% faster than the fastest conventional bipolar RAM and external TTL comparator circuit. In addition to being fast, the address-to-match signal time of the 2150 is relatively stable over the operating temperature range, increasing just 20% from its 25°C value with the Match output driving a 30-pF load (Fig. 4). In addition, t_d (A) changes little with supply-voltage variations.

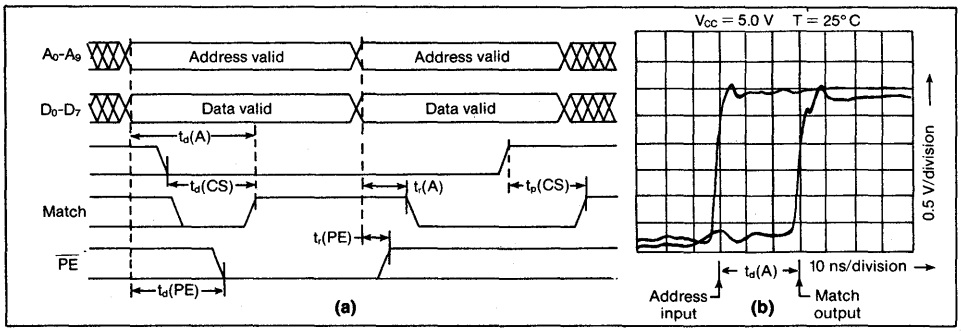
Other important timing parameters include the chip-select-to-match delay time, t_d (CS), which is about 25 ns maximum in the fastest 2150 version, and the input-address-to-PE delay time, t_d (PE), which is specified at a maximum of 55 ns for the same version.

Applying the 2150

With the tag-store and comparator circuitry on a single chip, the 2150, of course, is a large saver of circuit-board space. One or more 2150s together with several 2149 static RAMs can be placed on a single board, rather than the 1½ to 2 boards usually



2. The 2150 contains a 64-row-×-72-column static-RAM memory array organized as 512 words of 9 bits each. The RAM stores tag-address data for the cache system, and the rest of the chip provides the logic for comparing the stored tag address with the address on the data-bus line for validity, and providing or checking the data's parity.



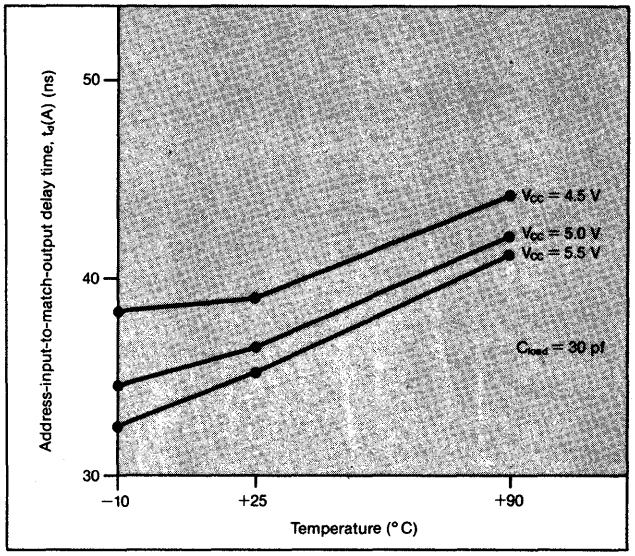
3. The timing cycles for 2150 start with a $\overline{CS}$-low signal. After comparison and matching, either a Match high or a parity error ($\overline{PE}$) low is obtained (a). The worst-case time delay between the address input and match output, t_d (A), is a key 2150 specification, and is available as one of four maximum values. A unit actually measured has a typical 30-ns t_d (A) delay (b).

required. Moreover, a single-board cache memory virtually eliminates the delay times from the capacitances introduced by buffers, conductor traces, board connectors, and backplane wiring.

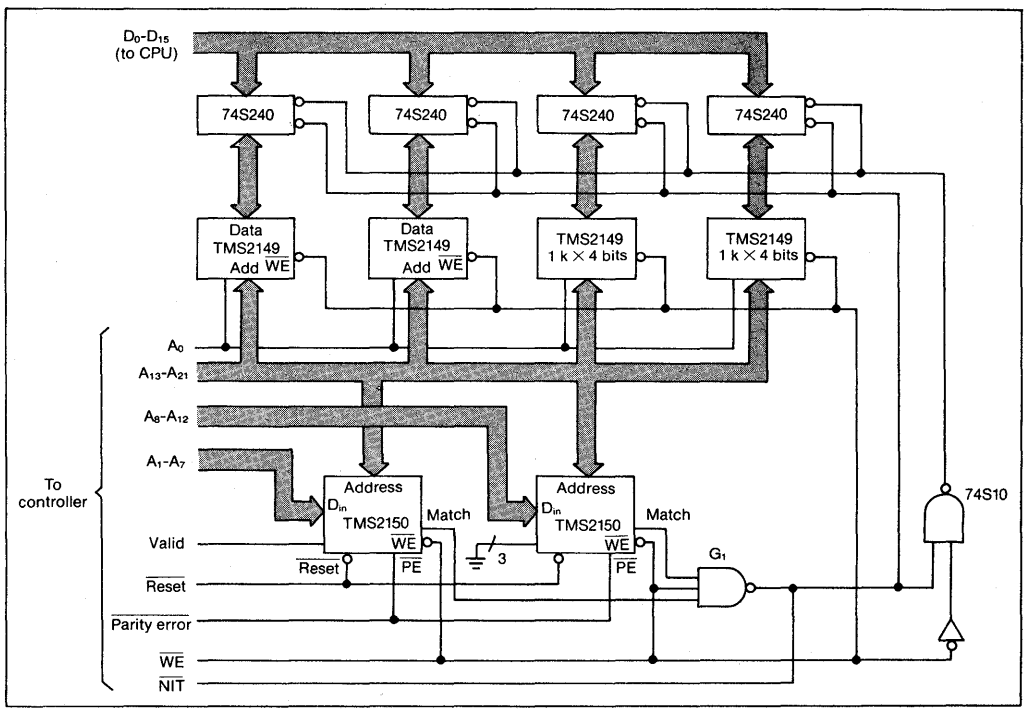
For example, the board can contain a single-set, 2-kbyte cache memory for 16-bit words (Fig. 5). Two 2150s serve for tag storage and comparison and four 2149s hold the 16-bit data words. Taken together with some TTL packages—four 74S240 octal buffers and one 74S10—they make an 11-package tag and data-storage system that requires only a controller (not shown) to support a two-way interleaved backing dynamic memory. This cache board, which operates at a total delay from address input to valid-data output of less than 80 ns, can be applied to almost any 16-bit minicomputer or microprocessor system having a 22-bit address field.

Acting as 8-bit-slice devices, the two 2150s split the processor address bus into two sections when comparing and matching addresses. When an address match is verified by both chips, the Match outputs—gated through G_1—supply an enable signal to the 74S240s configured as bidirectional buffers. In that way, address-input data can move from the 2149 static RAMs to the processor data bus.

When the write-enable ($\overline{WE}$) line is pulled low, data are entered into the 2149 RAMs from the processor, while the tag addresses of the data are entered into the 2150's internal tag-store RAM. □



4. The delay between the address input and the match output, $t_d(A)$, of the 2150 is relatively insensitive to variations in temperature or V_{CC}.



5. The 2150 readily lends itself to building-block implementations of cache-memory systems, as in this 2-kbyte single-set cache that employs two 2150s for a 16-bit processor with a 22-bit address field. In addition, the cache system requires four TMS2149 static RAMs, four 74S240 bidirectionally connected buffers, and a 74S10 gate chip. Thus the cache circuit comprises 11 chips. Not shown is the hit/miss and controller circuit that a cache also requires.

Performance could be improved by fitting cache-memory hardware to the system software or fine-tuning the software to the cache hardware.

Match cache architecture to the computer system

The following article is the third in a series on cache-memory systems. The previous article covered the details of a particular cache tag-store and comparator IC (Feb. 18, 1982, p. 159). The first article covered the basic cache philosophy (Jan. 21, 1981, p. 179). Texas Instruments' overall approach was sketched by Richard N. Gossen, manager of Advanced Memory Development, in the Jan. 21 issue (p. 32).

Cache-system architecture can take many forms, each with its own performance advantages and disadvantages and differing degrees of economy. But for optimum performance, the architecture of a cache-memory system should be matched specifically to the architecture of the overall computer system. Moreover, the cache's hardware and operational logic should be fitted to both the statistical and structural properties of the computer system's software and be highly transparent to it.

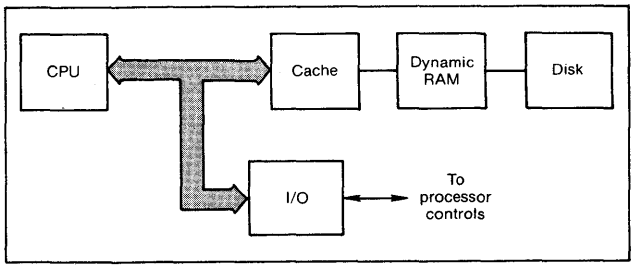
Of course, existing software can also be tuned to fit a particular cache hardware and its functional properties. A properly configured and finely tuned cache software-hardware system can approach well over 80% of the throughput that a very expensive all-high-speed memory could deliver. What's more, a cache system can do it with a slow, low-cost bulk memory plus a small amount of additional high-speed hardware for the cache.

Very high speed memory is expensive; therefore, the typical computer system cannot afford too much. On the other hand, bulk memory offers a large amount of low-cost capacity, but it is slow. A cache-memory system can combine the advantages of the two types of memories economically: small, but fast memory in the cache with large but slow memory for bulk storage.

Jerry VanAken, Computer Systems Engineer
Texas Instruments Inc.
8600 Commerce Park, Houston, TX 77036

A well-designed cache-memory system can manage to keep the few most-likely-to-be-accessed data in the cache for quick reference, while the bulk memory serves as a backup on those occasions when the processor references data not contained in the cache. When successfully implemented, this approach yields an almost transparent, economical memory system with the capacity of the bulk storage, but with the quick response of the cache.

The memory system of an inexpensive microcomputer system today is likely to be made up of a relatively low-cost disk and dynamic RAMs. Such a two-level memory hierarchy is well-suited to the needs of the less expensive popular microprocessor, whose minimum memory cycle times are on the order of 500 ns. Some of the newer high-performance microprocessors, however, have much shorter cycle times. An example is the 24-MHz 16-bit TMS99000, whose minimum memory cycle is 167 ns. For such a processor to access a block of relatively slow dynamic RAM, it must be slowed down by adding wait states to each memory cycle. This can be avoided by adding a third level to the memory hierarchy in the form of a high-speed cache of relatively simple design. The performance of such a processor can be improved dramatically, and the cache itself will represent only an incremental cost to the total



1. Even a simple single-microprocessor system can benefit from a cache system. With it, the number of accesses that must be made to disks and slow main memories like dynamic RAMs can be reduced substantially.

system—especially with the help of support chips such as the TMS2150 cache address comparator to keep the chip count low.

Although the per-bit cost of disk and semiconductor memory has decreased dramatically in recent years, microcomputers remain expensive largely because each decrease in the per-bit cost of memory devices is countered by a proportionate increase in the size of the average memory system. However, adding a cache to a memory system can produce a more than proportionate yield on the user's investment in his memory system—in the form of more memory-access cycles per second per dollar.

But even though the addition of a cache to a conventional, centralized high-performance microcomputer can greatly increase memory effectiveness, even greater improvements are possible with a distributed-intelligence architecture. The centralized-processor arrangement shown in Fig. 1 is based on the economics of past years, when the processor part of the system represented a much larger part of the overall system cost than it does today. Now, the situation is different: to have a single $20 microprocessor control $1000 worth of memory no longer makes economic sense. Low-cost independent microprocessors with local I/O arrangements in a distributed-intelligence system not only makes more economic sense, but can provide substantially better overall performance by using more of the available memory bandwidth (Figs. 2 through 5).

Moreover, the amount of memory bandwidth available can be effectively improved through the addition of caches; hence, the distributed-intelligence processing system can benefit from properly applied caching even more than the centralized-processor system of Fig. 1. In the simplest distributed system (Fig. 2), a central, global, bulk-storage memory can serve many independent microprocessors along a common bus. As the number of processors on the bus increases, the system throughput at first increases proportionately. But the bus gradually saturates—its bandwidth capability can handle no more data (Fig. 3). Adding more processors soon does not improve overall throughput: Since access to the memory for all data and instructions, as well as messages between the processors, is via the bus, the bus quickly becomes very busy. As more processors are added, contention for the bus mounts, and delays become longer.

Distribute memory too

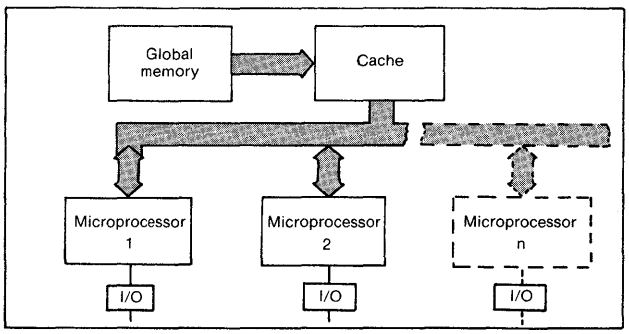
Moving some of the memory to the local sites of execution (Fig. 4)—in a so-called function-to-function architecture (FFA)[1]—will help alleviate bus-contention problems by locally storing most of the instructions and data needed for the special functions performed at that site. In this way, the global memory need contain just the shared data and those instructions needed for overall coordination, which together with interprocessor messages now have more "room" on the bus. This arrangement then allows still more processors to be handled with a given bus bandwidth.

Depending on the effort expended in organizing the software and the amount of local memory, the traffic on the bus can be cut way down—perhaps to as little as 10% of that of a nondistributed memory architecture. But to allow a high-performance microprocessor to operate at full speed, this local memory should be the fast, static-RAM type, which unfortunately is expensive and, in practice, limited to small capacities. However, configuring this small memory into a cache system would help matters since its capacity, though small, will be filled continually with current data (in a properly designed system). The small cache capacity would be as effective as a much larger static-RAM block mapped into a fixed set of memory addresses.

Moreover, with more of the bus bandwidth made available, entire blocks of data can be moved with each global-memory access. Block transfers from the global memory can have much the same advantage as moving blocks from a disk: Following the initial access time, the overhead time for each additional data word in the block is merely incremental.

For example, the global memory is likely to be made up of several dynamic RAMs, which support paged-mode operation. In this mode, only one row address is needed for a subsequent series of column addresses, which decreases the amount of overhead time per access. Or, the global-memory circuit may access not one, but several words in parallel, and then feed these to the system one-by-one at the maximum transfer rate of the bus. (Recent bus interfaces, such as the proposed IEEE-P896 standard, have been designed to support such efficient block transfers.)

Block transfers are particularly beneficial to



2. A distributed-intelligence, or function-to-function, system, having several local microprocessors instead of one centralized CPU, not only makes more economic sense, but can provide substantially better overall throughput.

caches because of their locality property, which characterizes the memory-access patterns of all programs. Basically, if a program accesses one word in a data or instruction block, it is likely to access other words in the same block subsequently. By reading the entire block into the cache at once, there's a good chance that the cache will be able to satisfy a greater number of additional access requests from the local processor without requiring more trips to the global memory for data.

Where to locate the cache

After the decision has been made to go with block transfers of data to the cache in the distributed-intelligence system, the next thing to determine is the location of the cache. In Fig. 2, the cache is located on the global-memory board. This cache location can decrease the time on the bus for memory accesses; accordingly, the bus is available for more data transfers. While this cache location decreases the overhead time per memory-data bus transfer, the number of such bus transfers remains the same as without the cache.

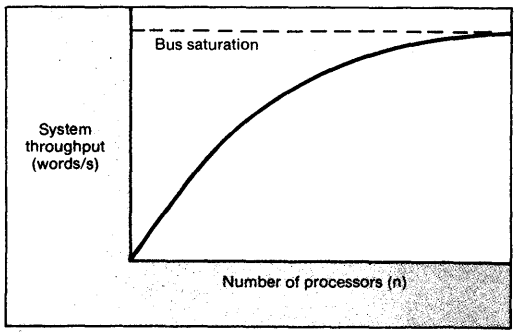
However, if the cache memory is located as in Fig. 4, the number of transfers over the bus is reduced. A block of data is accessed from main memory just once, but locally the same data can be used many times over without having to go back to the main memory via the bus. The bus bandwidth, thus freed up, now allows an increase in the amount of data transferred in each block. In addition, the cache block can be made large enough to achieve a desired hit ratio.

But if the processors run very long, uninterrupted programs and need just a few global-memory accesses, a simple FFA configuration (Fig. 4 without a cache, but with ordinary local memory) could be the most economical approach. With such programs, each processor executes the same on-board routines and accesses the same data locations repeatedly. Then the global memory need handle just messages between processors and system-wide instructions and data.

With the traffic on the bus reduced substantially, more processors can be added to the bus. And global-to-local-memory transfers could proceed via a direct-memory-access (DMA) system, which of course would be initialized under software control. Transfers via a local cache system (Fig. 4 with cache systems), however, would carry out the transfers automatically, transparent to the software.

Almost transparent

When a distributed system is implemented with local caches, the software is virtually unaware of the split between the local and global memories:



3. When a multiprocessor system shares a single bus, throughput rises proportionately at first with the number of processors, but tapers off as the bus's data-handling capacity saturates.

Hardware—the cache's control logic—maps the contents of each local cache into the global memory, and the cache is largely transparent to the software. (By comparison, in a DMA arrangement, the software would have to be totally involved in the local-global memory split.) Accordingly, with distributed cache systems, existing software (such as Pascal) for centralized-CPU systems can be used, with but minor modifications, for a higher-performance distributed-intelligence system.

Clearly, the distributed cache-system approach is general-purpose: All data and instructions can be mapped into the global memory as in the centralized system, and the local caches will then (almost) transparently remap the information for local use. In other words, the distributed system with local cache can best serve a general-purpose processing environment, where the specific functions of the individual processors cannot be predicted in advance, and thus where the contents of the local memories cannot be fixed at the time the system hardware is configured.

On the other hand, with a fairly fixed and predictable installation, perhaps with Fortran software, or in a plant-process-control application, where the data and software needed on each processor (board) are firmly established, the FFA approach could be used in place of local caches. But to achieve even greater performance, a distributed memory system can combine two or more of the approaches described. For example, cache memories can be employed for both the local and global memories—a combination of Figs. 2 and 4. The local cache decreases the number of references to the global memory, and the global cache decreases the average length of the bus cycles when global accesses do become necessary. Or, in a variation of the distributed-processor system of Fig. 2c, local memory and local cache can be used on the same processor. In Fig. 2d, the addition of a cache improves the performance of a large but slow local memory composed of 64-kbyte (or larger) dynamic RAMs.

Although, in general, employing local caches in a distributed system (like the one in Fig. 2c) will remove local instruction and data traffic from the bus and speed throughput substantially, trying to make the caches appear totally transparent can introduce interference problems with messages between processors. In the distributed system of Fig. 2c, for example, messages between processors should not be accessed through the local caches because this class of data is not held there: The caches contain only local current instructions and data from the global memory. As a result, in Fig. 3, when microprocessor 1 writes a message to microprocessor 2, the data should pass via a particular location in the global memory that acts as a message buffer. In the process, the local cache on the microprocessor-1 board must be prevented from intercepting the message. And when microprocessor 2 tries to read the message, its cache also must be removed from the message path. Otherwise, microprocessor 2 will encounter stale data in cache memory, and not the new message from microprocessor 1, which had just been deposited into the global-memory.

A write-through (as opposed to nonwrite through or write-back) caching policy can ensure microprocessor 1 writes its message to global memory, but additional steps are needed to ensure that microprocessor 2 reads the message from global memory without interference from its cache. Software recognition of the special status of interprocessor messages can easily solve this potential problem. But this approach constitutes a lack of total transparency for the caches.

One approach passes all interprocessor messages through the microprocessor's I/O space. Since I/O data are not cached, this strategy automatically avoids the interference problem. Interprocessor messages pass through the I/O space, bypassing the caches altogether. So if the software is initially written to handle the interprocessor messages via the I/O space, then when caches are introduced, they will automatically be transparent to the caches.

Alternately, a particular set of addresses in the global-memory space can be dedicated to message passing. The control logic in each processor-board's cache could incorporate a comparison circuit that recognized the message-space addresses and allows access to the message area in global memory to bypass the cache.

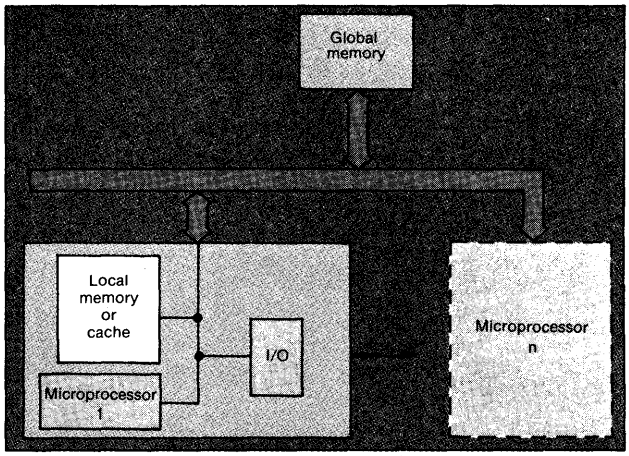
System data should also bypass the caches and be taken by a processor directly from the master copy in the main memory: Such data are constantly being updated by the other processors. If passed via the local caches, such data would invariably be perceived as stale because of the constant updating. If allocated to specific and exclusive global-memory addresses, these system data can be handled like the interprocessor messages to bypass the caches.

Similarly, each processor should have exclusive access to its own private instruction and data segments in the global memory. In this way, the data are "protected," with some support from bus hardware, from a processor that may go "berserk" and corrupt the instruction and data segments of the other processors in the system.

Clearly, with caches in distributed-intelligence systems, the memory accesses must be organized rationally to optimize throughput, avoid inefficient data thrashing and, most important of all, avoid using data belonging to other processors. Of course, with a single cache (as in Figs. 1 and 2), it is rather difficult to mix up the data, since the cache should always contain updated master versions of the corresponding blocks of memory (which is never changed without knowledge of the cache). In a multiprocessor system like Fig. 4, however, each local cache is supposed to keep a separate, accurate copy of some portion of the global memory—as it pertains to its own processor. But a mixup is possible because of the multiplicity of processors.

The point of all this is that it is exceedingly difficult to make the cache memories in a distributed system totally transparent to software, while simultaneously ensuring that each processor is provided with a coherent, updated version of the contents of global memory. Methods have been proposed for accomplishing just this, but they tend to be expensive in terms of the hardware required, and are therefore beyond the reach of the typical microprocessor-based system.

In some systems, the entire contents of the cache may have to be "flushed," if for any reason its

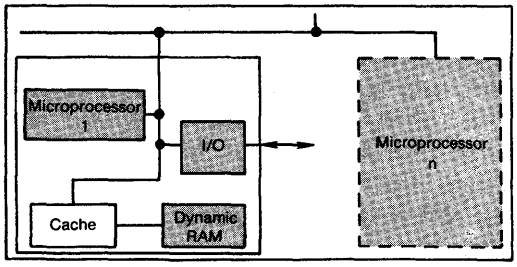


4. A cache, located at the slow global memory as in Fig. 2, can reduce the number of required slow accesses to main memory, thus leaving the bus more time for other activities. But locating part of the memory at each processor—in a distributed architecture—is even better.

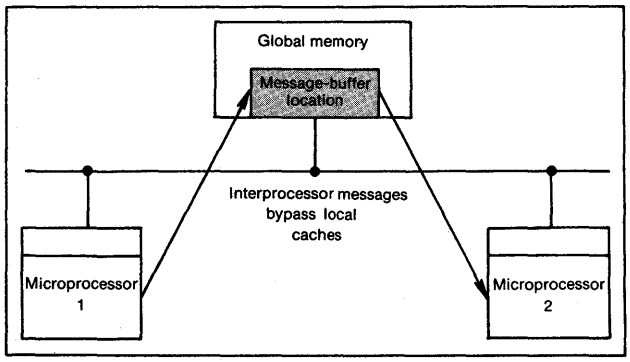
contents have become invalid. For example, a DMA device may alter the contents of main memory, invalidating the contents of the cache. Also, consider the case of a processor attached to a memory mapper (like a 74LS610), which translates the logical addresses output from the processor into the physical addresses used to access the memory. A cache will usually be attached directly to the processor to avoid lengthening the cache access time with the propagation delay through the mapper circuitry. However, this means that the cache contents are mapped into the logical rather than the physical address space. Consequently, when the map file is altered, this makes the cache contents invalid since the mapping of the logical into the physical address space is no longer the same.

For applications where the cache contents must frequently be flushed, a cache-reset function is essential. Without the ability to flush the cache instantaneously, the system would be forced to clear each cache block, one by one.

But flushing the entire cache when just a small portion of its contents needs updating is wasteful. Naturally, a more complex reloading arrangement can be designed to provide higher caching efficiency where only part of the cache data must be replaced frequently, but only at the expense of increased overhead in logic and software. Selectively dumping only the affected areas of the cache capacity—such as when a DMA operation partially alters the contents of the global memory—instead of a total reset, would be more efficient, but also much more expensive. It would also occupy more board space.



5. The local memory can be in "ordinary" RAM form (as in function-to-function architecture) or in cache form. Or, both a cache and a local dynamic RAM can be used.



6. Interprocessor messages should bypass caches to avoid interference. This can be accomplished with the software or special overhead hardware, or by employing the microprocessor's I/O space to carry the messages.

Complexity requires board space. With the trend to smallness in electronic packaging, sometimes compromises must be made. CPU speed is generally compromised when going from a multiboard minicomputer design to a design that just barely crams a CPU onto a single board. To put substantial memory onto the board as well usually requires going to a slower microcomputer design, which puts the CPU into a chip, and leaves board room for the memory. However, the speed lost because of these compromises can be partially recovered by incorporating a cache on the board (or even on the microprocessor chip). The cache effectively raises the individual processor's memory access speed when it is used with a slow on-board dynamic RAM (Fig. 5).

The TMS9995 microprocessor is a precache step in this direction: It contains a modest chunk (256 bytes) of high-speed random-access memory, which is mapped into a fixed area of the processor's address space. Thus, it does not qualify strictly as a cache. However, it can be used to the same effect by loading the RAM—under explicit software control—with currently needed data and instructions.

The next step would be to put a cache on the processor board (or chip) with as much memory as the board space allows, all of which would be almost transparent to the system software. Equally important, a cache could make the most of the limited amount of memory that now can be put onto a board with the microprocessor and I/O.

Despite the transparency, a programmer who is aware of the cache's capability can fine-tune the software to maximize its efficiency. On the other hand, overly refined software for one hardware system can produce poor efficiency on another, while remaining transportable in the sense that it executes without error. But as software can be adapted to maximize the efficiency of cache configuration and minimize its limitations, so can cache-system hardware be designed to best fit very extensive existing software. It is a two-way street.□

References

1. "Functional Architecture Threatens Central CPU," ELECTRONIC DESIGN, Sept. 3, 1981, p. 141-156.

4 Error Detection and Correction (EDAC)

4.1 Use of an Error Detection and Correction (EDAC) Device

4.1.1 Introduction

The DRAM technology of today (i.e., 256K/M) has enabled system designers to use much larger memory sizes than ever before. However, as with most advances in technology, this has brought a new problem. For system memory sizes larger than 1/2 million bits, it is generally considered that error detection and correction is required to guarantee system reliability without a tradeoff in performance. Although present methods of parity checking will identify errors, they are not able to correct them. And not correcting these errors can be costly. For example, in personal computers when parity errors are encountered, the system has to be reset to eliminate the problem. This system reset destroys any data stored in RAM and it must be reentered. Obviously this is unacceptable to your customers. To eliminate this problem, TI has produced a cost effective Error Detection and Correction (EDAC) device.

4.1.2 Error Types and Sources in Dynamic Memories

Two kinds of errors occur in memory devices; soft and/or hard errors. A hard error is a physical failure of the memory device (e.g., an internal short or an open lead). This type of error causes the memory location to always be either a high or a low. A soft error is a random occurrence of a memory location change from a high level to low level. These errors may be caused by system noise, alpha particle radiation, or power surges.

In spite of design techniques used by memory chip manufactures to reduce these errors, they are still a source of major concern in your system. Table 4-1 indicates that as the density of memory chips increase their probability of errors also increase. Therefore, your data integrity decreases in larger memory arrays.

Table 4-1. Chip Densities vs Soft-Error Rates

CHIP DENSITY BITS/CHIP	TYPICAL SOFT-ERROR RATE (% PER 1000 HOURS)
64K	0.10 – 0.20
256K	0.15 – 0.30
1M	0.20 – 0.35

4.1.3 Solutions to Boost System Reliability

There are several alternatives available that will either decrease or eliminate these errors in your system. One method used to determine data integrity is the incorporation of parity checking. This can be accomplished by using an SN74ALS29833 Parity Bus Transceiver. To identify an error, the data word and the generated parity are compared by performing an exclusive-OR operation. If several bits in the data word are in error or the parity has changed, the exclusive-OR output would be low. While data integrity can be determined using this method, it is unable to correct errors.

To obtain the desired level of quality, some type of error-correction scheme must be incorporated. An EDAC chip provides the simple solution to the problem, while dramatically extending the system Mean Time Between Failures (MTBF). This is accomplished by detecting and correcting single bit errors and detecting double bit errors. See Table 4-2.

Table 4-2. System MTBF Increases with an EDAC

	MTBF†	
	Without EDAC	**With EDAC**
CORRECTABLE SOFT ERROR (SINGLE BIT)	7 Months	>200 Years

†Based on 16M-Bit memory system using 256K DRAMs with a 0.30% per 1000 hour soft error rate.

When you include the other system variables causing errors (power surges, noisy systems, etc.), your memory system MTBF, without an EDAC could be reduced to several days. These types of memory-cell errors can be corrected using an EDAC.

4.1.4 EDAC Operation

When data is written to memory, the TI SN74AS632 (32-Bit EDAC) generates parity check bits. Each check bit is generated by performing a specific parity check on the 32-bit data word. For example, CB0 is obtained by comparing specific bits of the 32-bit word with those corresponding to an "X" in the Hamming Code Parity Algorithm (see Table 4-3). CB0 will be at a high level if the total number of highs corresponding to these locations is an odd number. CB0 will be at a low level if this number is even. This procedure is repeated 7 times to obtain the 7 check bits, CB0-CB6 of the Hamming Code. Check bits CB0-CB2 are used to determine odd parity. Check bits CB3-CB6 are used for even parity.

Table 4-3. Hamming Code Parity Algorithm

CHECK WORD BIT	32-BIT DATA WORD																													
	31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
CB0	X		X	X		X					X		X	X	X			X			X		X	X	X	X		X				X
CB1				X		X		X		X		X		X	X	X				X		X		X		X		X		X	X	X
CB2	X		X			X	X		X			X	X			X	X		X			X	X		X			X	X			X
CB3			X	X	X				X	X	X				X	X			X	X	X				X	X	X			X	X
CB4	X	X							X	X	X	X	X	X			X	X							X	X	X	X	X	X		
CB5	X	X	X	X	X	X	X	X									X	X	X	X	X	X	X	X								
CB6	X	X	X	X	X	X	X	X																	X	X	X	X	X	X	X	X

The seven check bits are parity bits derived from the matrix of data bits as indicated by "X" for each bit.

These check bits are stored along with the data in your systems main memory. This additional memory requirement is the only overhead involved with the use of an EDAC. Figure 4-1 shows a typical system using an EDAC and illustrates this overhead.

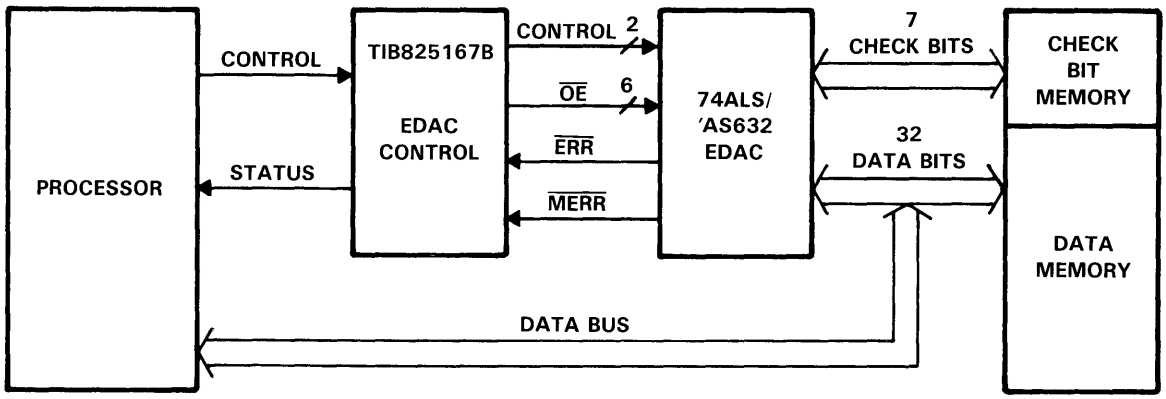


Figure 4-1. Typical 'AS632 System

During a read cycle, the data and check bits are read from memory, any of which may be invalid. New check bits are computed from the stored data bits. To determine the validity of the data, the new and old check bits are exclusive-ORed producing a 7-bit syndrome code. When decoded, these syndrome bits describe the condition of the data word: free of errors, having a single bit error, or having multiple errors. See Table 4-4. Any single error in the 32-bit word can be corrected. Both single and double bit errors are indicated to the processor via single and double bit error flags.

There are two additional options for implementing EDAC into your system; detect only and correct always. Of these two, correct always is the easiest to implement. The EDAC always corrects single-bit errors and writes this corrected word onto the system data bus or into memory.

Because days can elapse between errors, correction can be done only when needed. The detect-only option increases your system performance during a read cycle by allowing data to be written directly to the system processor. If a single or double bit error occurs, the EDAC will flag the processor. This enables the processor to enter a wait cycle until the word is corrected. This method of implementation does not use the error correction portion of the EDAC until the processor determines what action to take in the event of an error.

Another method of ensuring data integrity in your system is to use an EDAC unit during memory refresh. The EDAC will "clean" every memory location of errors during the mandatory refresh cycles. This process is known as memory scrubbing. The data can then be checked again during a memory-access cycle. By checking the data twice, the time between corrections is reduced. Therefore, the probability of multibit errors in your system declines.

Table 4-4. 'AS632 Syndrome Decoding

SYNDROME BITS							ERROR
6	5	4	3	2	1	0	
L	L	L	L	L	L	L	unc
L	L	L	L	L	L	H	2-bit
L	L	L	L	L	H	L	2-bit
L	L	L	L	L	H	H	unc
L	L	L	L	H	L	L	2-bit
L	L	L	L	H	L	H	unc
L	L	L	L	H	H	L	unc
L	L	L	L	H	H	H	2-bit
L	L	L	H	L	L	L	2-bit
L	L	L	H	L	L	H	unc
L	L	L	H	L	H	L	DB31
L	L	L	H	L	H	H	2-bit
L	L	L	H	H	L	L	unc
L	L	L	H	H	L	H	2-bit
L	L	L	H	H	H	L	2-bit
L	L	L	H	H	H	H	DB30
L	L	H	L	L	L	L	2-bit
L	L	H	L	L	L	H	unc
L	L	H	L	L	H	L	DB29
L	L	H	L	L	H	H	2-bit
L	L	H	L	H	L	L	DB28
L	L	H	L	H	L	H	2-bit
L	L	H	L	H	H	L	2-bit
L	L	H	L	H	H	H	DB27
L	L	H	H	L	L	L	DB26
L	L	H	H	L	L	H	2-bit
L	L	H	H	L	H	L	2-bit
L	L	H	H	L	H	H	DB25
L	L	H	H	H	L	L	2-bit
L	L	H	H	H	L	H	DB24
L	L	H	H	H	H	L	unc
L	L	H	H	H	H	H	2-bit

SYNDROME BITS							ERROR
6	5	4	3	2	1	0	
L	H	L	L	L	L	L	2-bit
L	H	L	L	L	L	H	unc
L	H	L	L	L	H	L	DB7
L	H	L	L	L	H	H	2-bit
L	H	L	L	H	L	L	DB6
L	H	L	L	H	L	H	2-bit
L	H	L	L	H	H	L	2-bit
L	H	L	L	H	H	H	DB5
L	H	L	H	L	L	L	DB4
L	H	L	H	L	L	H	2-bit
L	H	L	H	L	H	L	2-bit
L	H	L	H	L	H	H	DB3
L	H	L	H	H	L	L	2-bit
L	H	L	H	H	L	H	DB2
L	H	L	H	H	H	L	unc
L	H	L	H	H	H	H	2-bit
L	H	H	L	L	L	L	DB0
L	H	H	L	L	L	H	2-bit
L	H	H	L	L	H	L	2-bit
L	H	H	L	L	H	H	unc
L	H	H	L	H	L	L	2-bit
L	H	H	L	H	L	H	DB1
L	H	H	L	H	H	L	unc
L	H	H	L	H	H	H	2-bit
L	H	H	H	L	L	L	2-bit
L	H	H	H	L	L	H	unc
L	H	H	H	L	H	L	unc
L	H	H	H	L	H	H	2-bit
L	H	H	H	H	L	L	unc
L	H	H	H	H	L	H	2-bit
L	H	H	H	H	H	L	2-bit
L	H	H	H	H	H	H	CB6

SYNDROME BITS							ERROR
6	5	4	3	2	1	0	
H	L	L	L	L	L	L	2-bit
H	L	L	L	L	L	H	unc
H	L	L	L	L	H	L	unc
H	L	L	L	L	H	H	2-bit
H	L	L	L	H	L	L	unc
H	L	L	L	H	L	H	2-bit
H	L	L	L	H	H	L	2-bit
H	L	L	L	H	H	H	unc
H	L	L	H	L	L	L	unc
H	L	L	H	L	L	H	2-bit
H	L	L	H	L	H	L	2-bit
H	L	L	H	L	H	H	DB15
H	L	L	H	H	L	L	2-bit
H	L	L	H	H	L	H	unc
H	L	L	H	H	H	L	DB14
H	L	L	H	H	H	H	2-bit
H	L	H	L	L	L	L	unc
H	L	H	L	L	L	H	2-bit
H	L	H	L	L	H	L	2-bit
H	L	H	L	L	H	H	DB13
H	L	H	L	H	L	L	2-bit
H	L	H	L	H	L	H	DB12
H	L	H	L	H	H	L	DB11
H	L	H	L	H	H	H	2-bit
H	L	H	H	L	L	L	2-bit
H	L	H	H	L	L	H	DB10
H	L	H	H	L	H	L	DB9
H	L	H	H	L	H	H	2-bit
H	L	H	H	H	L	L	DB8
H	L	H	H	H	L	H	2-bit
H	L	H	H	H	H	L	2-bit
H	L	H	H	H	H	H	CB5

SYNDROME BITS							ERROR
6	5	4	3	2	1	0	
H	H	L	L	L	L	L	unc
H	H	L	L	L	L	H	2-bit
H	H	L	L	L	H	L	2-bit
H	H	L	L	L	H	H	DB23
H	H	L	L	H	L	L	2-bit
H	H	L	L	H	L	H	DB22
H	H	L	L	H	H	L	DB21
H	H	L	L	H	H	H	2-bit
H	H	L	H	L	L	L	2-bit
H	H	L	H	L	L	H	DB20
H	H	L	H	L	H	L	DB19
H	H	L	H	L	H	H	2-bit
H	H	L	H	H	L	L	DB18
H	H	L	H	H	L	H	2-bit
H	H	L	H	H	H	L	2-bit
H	H	L	H	H	H	H	CB4
H	H	H	L	L	L	L	2-bit
H	H	H	L	L	L	H	DB16
H	H	H	L	L	H	L	unc
H	H	H	L	L	H	H	2-bit
H	H	H	L	H	L	L	DB17
H	H	H	L	H	L	H	2-bit
H	H	H	L	H	H	L	2-bit
H	H	H	L	H	H	H	CB3
H	H	H	H	L	L	L	unc
H	H	H	H	L	L	H	2-bit
H	H	H	H	L	H	L	2-bit
H	H	H	H	L	H	H	CB2
H	H	H	H	H	L	L	2-bit
H	H	H	H	H	L	H	CB1
H	H	H	H	H	H	L	CB0
H	H	H	H	H	H	H	none

CB X = error in check bit X
DB Y = error in data bit Y
2-bit = double-bit error
unc = uncorrectable multibit error

The circuit illustrated in Figure 4-2 is an example of a memory system that used scrubbing. This circuit consists of the TI SN74ALS6302, a 1M-DRAM Controller, the TMS4C1024, 1M DRAMs, the SN74AS632, a 32-bit EDAC, and control circuits.

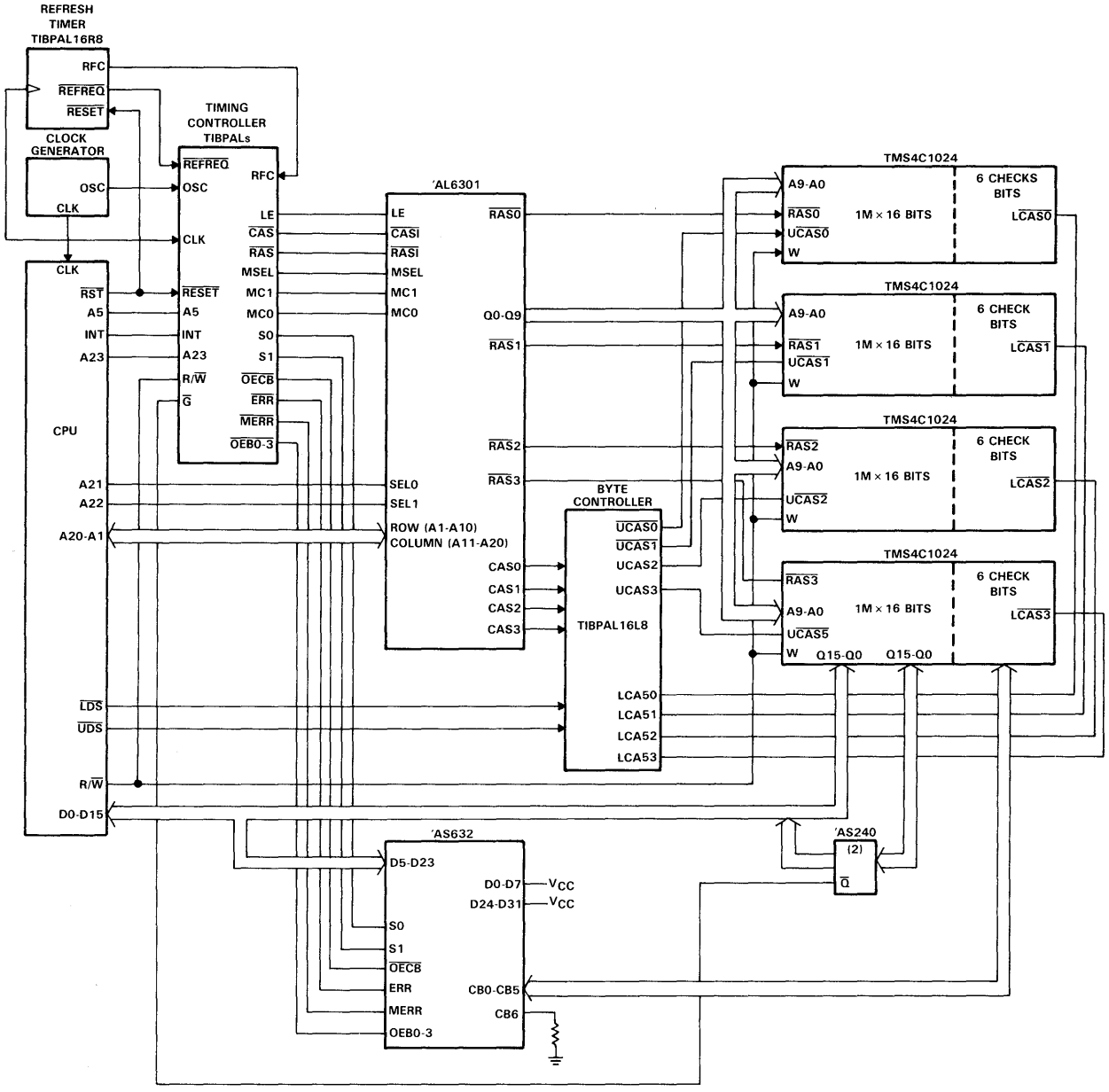


Figure 4-2. Memory Management Systems Using Scrubbing

4.1.5 Texas Instruments EDAC Family

Because of the increase in MTBF, the SN74AS632 can increase system reliability typically by well over 500-fold. The 'AS632 provides built-in diagnostics to assure reliable device operation. Byte-write capability is included to allow operation on 8-bit, 16-bit, or 32-bit word widths in 3-state bus applications. The 'AS632 provides the fastest correction time, 32 ns, and error-detection time, 25 ns, available today. The architecture of the 'AS632 is illustrated in Figure 4-3.

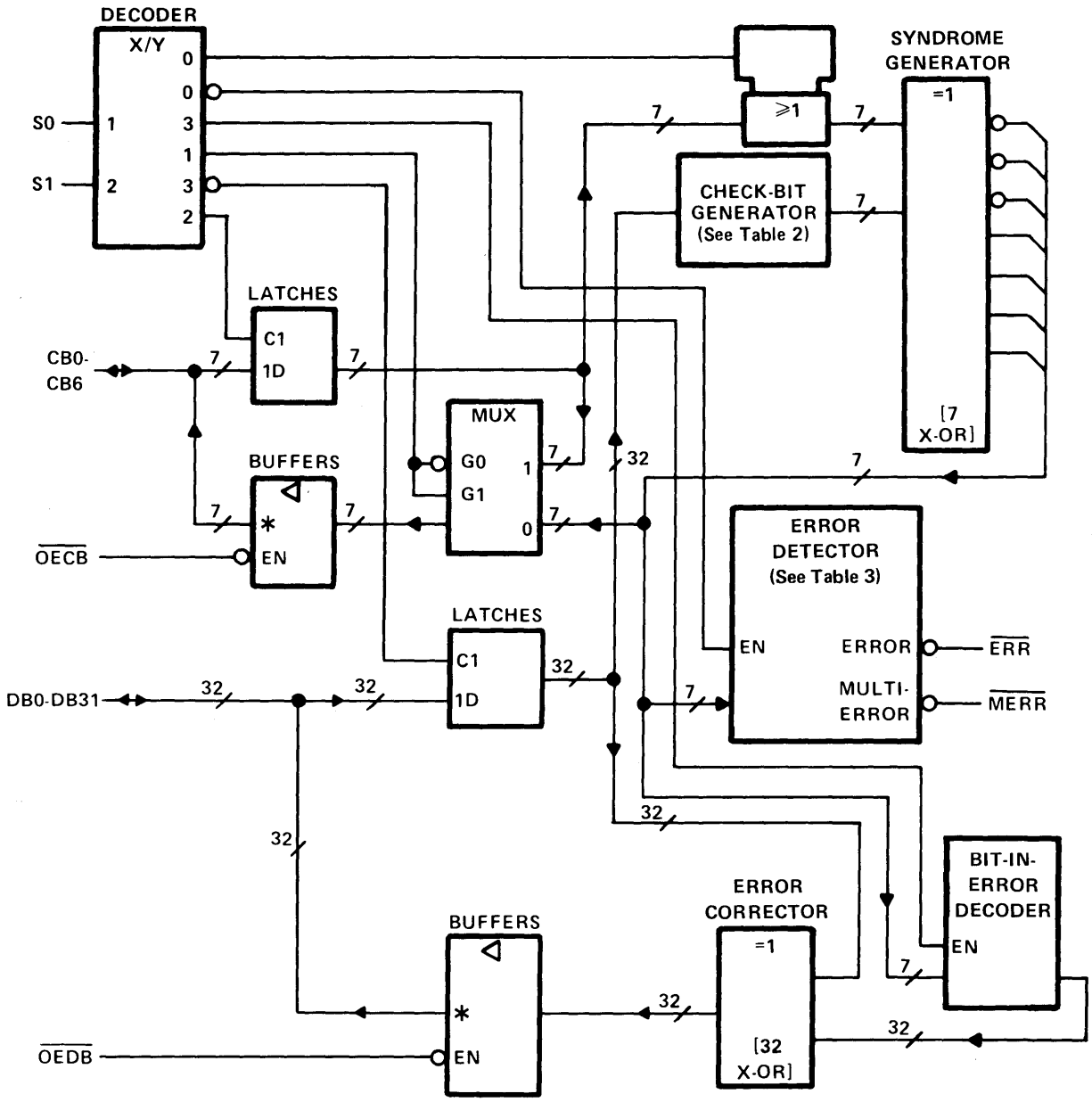


Figure 4-3. 'AS632 Logic Diagram

Along with the 'AS632 32-bit EDAC, TI has a complete family of high-performance EDAC products to fit your particular application. See Table 4-5.

Table 4-5. Texas Instruments Error Detection and Correction Devices

DIP PINS	DEVICE TYPE	DETECTION† TIME (MAX)	CORRECTION TIME (MAX)	FEATURES	AVAILABLE
40	ALS616	40	65	16-BIT, 3-STATE	NOW
40	AS616	25‡	32	SPEED ENHANCED ALS616	2Q87
28	LS630	30	65	16-BIT, NO BYTE-WRITE, 3-STATE	NOW
28	LS631	30	65	16-BIT, NO BYTE-WRITE, OPEN COLLECTOR	NOW
52	ALS632A	40	58	32-BIT, 3-STATE	NOW
52	ALS632B	30	37	SPEED ENHANCED ALS632A	NOW
52	AS632	25	32	FASTEST EDAC AVAILABLE	NOW
48	ALS634A	40	58	32-BIT, NO BYTE-WRITE, 3-STATE	NOW
48	AS634	25‡	32	SPEED ENHANCED ALS634	1Q87

†Single Bit Error
‡Design Goals

All of the products listed in Table 4-5 offer the following:

1. Built-in Diagnostic Capabilities
2. Modified Hamming Code Operation
3. Dependable Texas Instruments Quality and Reliability

4.1.6 Summary

Memory errors are becoming a very important concern to the system designer. To effectively ensure data integrity, a method of correcting data errors is necessary. An EDAC unit provides you with this essential function along with increasing system MTBF from days to years. The TI EDAC family offers you ease of implementation, high performance, and a device that is compatible with any microprocessor you might be using.

For more information on the TI family of EDAC devices, please contact your local TI Sales Representative or the Customer Response Center at 1-800-232-3200.

For your convenience, the TI documentation is listed below.

	TI Reference Number
Error Detection and Correction Application Reports:	
SN54/74LS630 or SN54/74LS631	SDLA003
SN54/74ALS632B, 'ALS633, 'ALS634A, 'ALS635	SDAS102
Data Sheets:	
SN54/74AS632	SDAS101
SN54/74ALS632B, 'ALS633, 'ALS634A, 'ALS635	SDAS105B
SN54/74ALS616, 'ALS617	SDAS047
SN54/74LS630, 'LS631 (TTL Data Book Vol. 2)	SDLD001
LSI Data Book	SDVD001

4.2 Error Detection and Correction Using 'ALS632B, 'ALS633, 'ALS634A, and 'ALS635

4.2.1 Introduction

With memory systems continuing to expand and the expectation of 256K-byte DRAMs in the near future, error detection and correction has become increasingly important. Generally, the larger the chip density, the greater the probability for device errors. It is easy to recognize this probability when one considers that a 32-bit × 64K-byte memory, using 64K-byte DRAMs, equals approximately 2.1 million bits of information. This expands to 8.4 million bits of information when using 256K-byte DRAMs. For memory sizes larger than 0.5 million bits, error detection and correction is required to guarantee high reliability.

The SN54/74ALS632B, SN54/74ALS633, SN54/74ALS634A, and SN54/74ALS635 provide a solution to these requirements in 32-bit machines. In addition, the 'ALS632B and 'ALS633 provide the necessary hardware to perform byte-write operations which are typically used in the more advanced systems. To ensure the integrity of the error detection and correction circuit, diagnostic capabilities have been provided in all four devices.

The 'ALS632B series devices are not limited to 32-bit systems. They can be implemented in 16- or 24-bit systems. In the case of 16-bit systems, the additional memory needed for holding the check bits can be reduced when compared to conventional 16-bit EDACs.

The pin functions are listed in Table 4-6. Mechanical data for the 'ALS632B, 'ALS633, 'ALS634A, and 'ALS635 is shown in Figure 4-4.

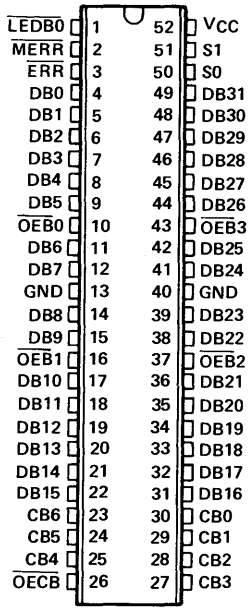
Table 4-6. Pin Function for 'ALS632B, 'ALS633, 'ALS634A, and 'ALS635

PIN NAME	DESCRIPTION
S1, S0	Selects the operating mode of the EDAC S1 S0 MODE OPERATION L L WRITE Input dataword and output checkword H L READ & FLAG Input dataword and output error flags H H CORRECT Latched input data and checkword/output corrected Data and error syndrome code L H DIAGNOSTIC Input various datawords against latched checkword/output valid error flags
DB0 through DB31	I/O port for entering or outputing data
$\overline{OEB0}$ through $\overline{OEB3}$ ('ALS632B, 'ALS633)	Three state control for the data I/O port. A high allows data to be entered, and low outputs the data. Each pin controls 8 data I/O ports (or one byte). $\overline{OEB0}$ controls DB0 through DB7, $\overline{OEB1}$ controls DB8 through DB15, $\overline{OEB2}$ controls DB16 through DB23, and $\overline{OEB3}$ controls DB24 through DB31.
$\overline{OEDB}$ (ALS634, ALS635)	Three state control for the data I/O port. When low allows data to outputed and a high allows data to be entered.
$\overline{LEDB0}$	Controls the dataword output latch. When low, the data output latch is transparent. When high, the latch stores whatever data was setup at its inputs when the last low to high transistion occured on the pin.
CS0 through CS6	I/O Port for entering or outputing the checkword. It is also used to output the syndrome error code during the error correction mode.
$\overline{OECS}$	Three state control for the checkword I/O port. A high allows data to be entered and a low allows either the checkword or syndrome code (depending on EDAC mode) to be outputed.
$\overline{ERR}$	Single error output flag, a low indicates at least a single bit error.
$\overline{MERR}$	Multiple error output flag, when low indicates two or more errors present

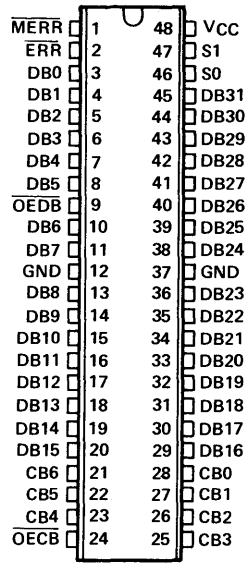
ceramic packages – side-braze (JD suffix)

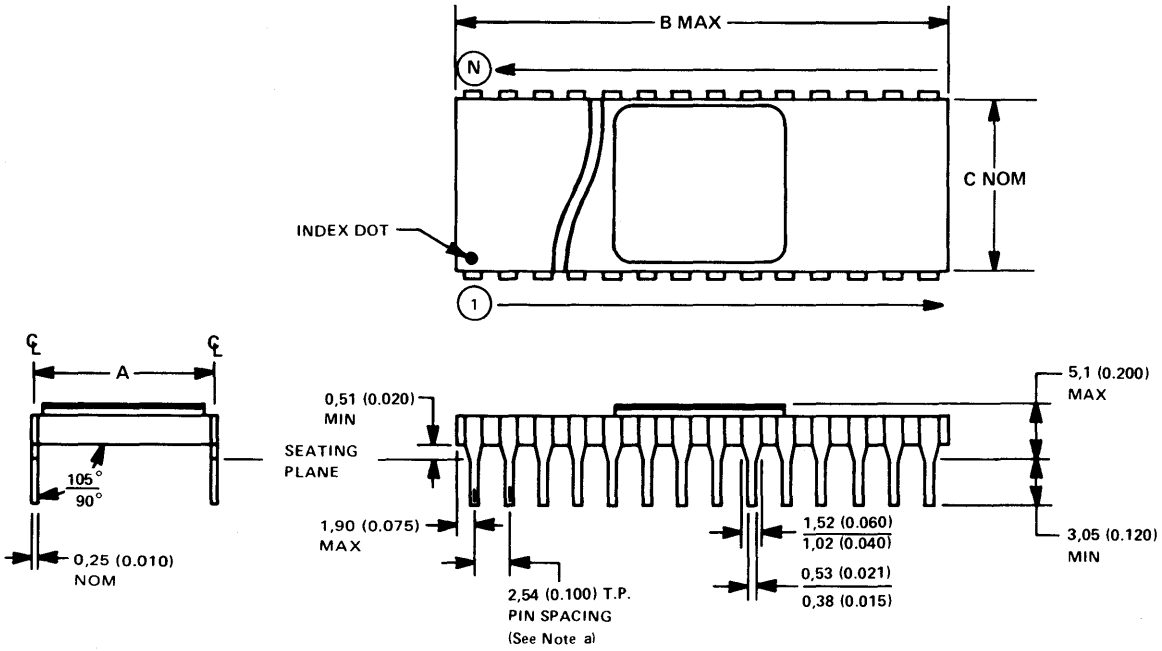
This is a hermetically sealed ceramic package with a metal cap and side-brazed tin-plated leads.

'ALS632B, 'ALS633 . . . JD PACKAGE
(TOP VIEW)



'ALS634A, 'ALS635 . . . JD PACKAGE
(TOP VIEW)





DIM \ PINS	48	52
A ± 0,25 (0.010)	15,24 (0.600)	15,24 (0.600)
B MAX	62,2 (2.45)	67,3 (2.65)
C NOM	15,0 (0.590)	15,0 (0.590)

ALL DIMENSIONS ARE IN MILLIMETERS AND PARENTHETICALLY IN INCHES

NOTE: a. Each pin centerline is located within 0,25 (0.010) of its true longitudinal position.

Figure 4-4. Mechanical Data

4.2.2 Operational Description

4.2.2.1 Write Mode

During a memory write cycle, the EDAC is required to generate a 7-bit check word to accompany the 32-bit data word before being written into memory. To place the 'ALS632B, 'ALS633, 'ALS634A and 'ALS635 in the write mode, take S1 and S0 low. Output-enable controls $\overline{OEB0}$ through $\overline{OEB3}$ for the 'ALS632B, 'ALS633 or $\overline{OEDB}$ for the 'ALS634A, 'ALS635 must be taken high before the data word can be applied. Output-enable control $\overline{OECS}$ must be taken low to pass the check word to the external bus.

The check word will be generated in not more than 30 ns after the data word has been applied. During the write mode, the 'ALS632B series EDACs can be made to appear transparent to memory, because typical write times of most DRAMs are much larger than the propagation delay of data to check word.

4.2.2.2 Read-Flag-Correct Operation

During a memory read cycle, the function of the 'ALS632B series EDACs is to compare the 32-bit data word against the 7-bit check word previously stored in memory. It will then flag and correct any single-bit error which may have occurred. Single-bit errors will be detected through the ERR flag and double-bit errors will be detected through the MERR flag. Figure 4-5 shows a typical timing diagram of the read-flag-correct operation.

When S0 is taken high, the EDAC will begin the internal correction process, although the error flags are enabled while in the read mode. For many applications, the simplest operation can be obtained by always executing the correction cycle, regardless if a single-bit error has occurred.

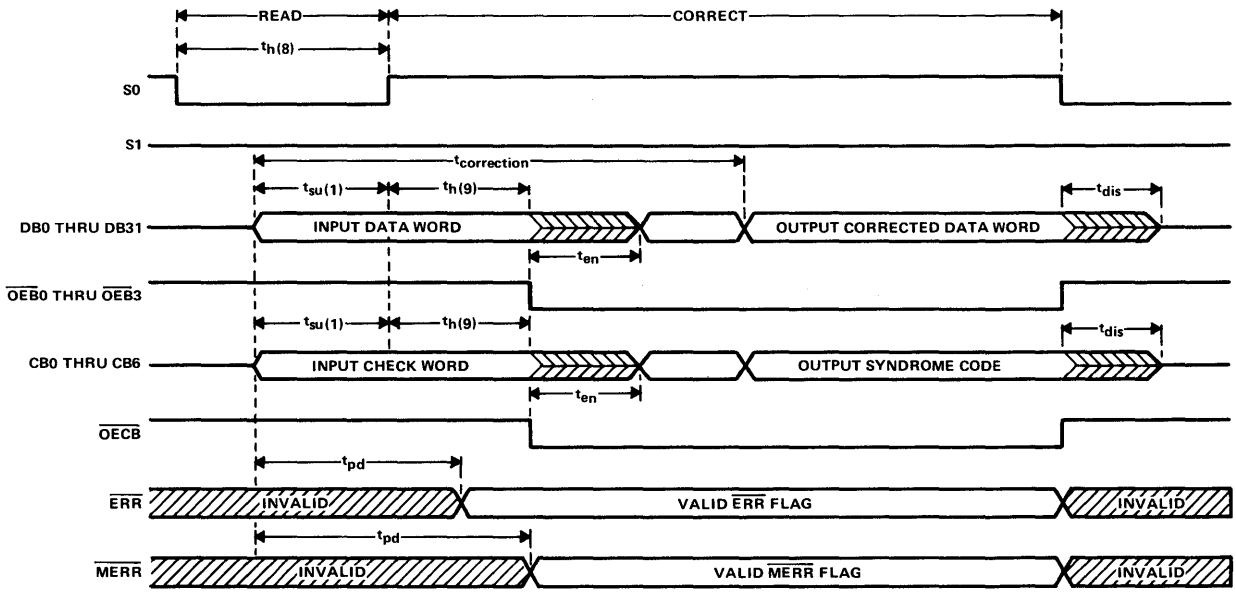


Figure 4-5. Read-Flag-Correct Timing Diagram

4.2.2.3 *Important Timing Considerations for Read-Flag-Correct Mode*

The most frequently asked question for an EDAC is how fast can a correction cycle be executed. Before S0 can be taken high, the data and check word must be set up for at least 5 ns. In addition, the data and check word must be held for at least 10 ns after S0 goes high. This ensures that the data and check word are saved in the EDAC input latches. After the hold time has been satisfied, the source which is driving the data bus can be placed in high impedance and the EDAC's output drivers can be enabled. This is accomplished by taking $\overline{OEB0}$ through $\overline{OEB3}$ ('ALS632B, 'ALS633) or $\overline{OEDB}$ ('ALS634A, 'ALS635) low.

If the minimum data setup time is used as a reference and the output drivers are enabled after the minimum data hold time, then correction will be accomplished in 37 ns or less.

4.2.2.4 *Read-Modify-Write Operations*

The 'ALS632B and 'ALS633 contain the necessary hardware to perform byte-write operations. The 'ALS634A and 'ALS635 are not capable of byte-write operations because they do not contain an output data latch or individual byte controls. When performing a read-modify-write function, perform the read-flag-correct cycle as previously discussed and shown in Figure 4-5. This ensures that corrected data is used at the start of the modify-write operation.

The corrected data is then latched into the output data latch by taking $\overline{LEDBO}$ from low to high. Upon completing this, modifying any byte or bytes is accomplished by taking the appropriate byte control $\overline{OEB0}$ through $\overline{OEB3}$ high. This allows the user to place the modified byte or bytes back onto the data bus while retaining the other byte or bytes. An example of a read-modify-write for byte 0 is shown in Figure 4-6.

Since the check word is no longer valid for the modified data word, a new one is generated by taking S0 and S1 low. After the appropriate propagation delay, the new check word will be available.

4.2.2.5 *Important Timing Considerations for Read-Modify Write Operations*

$\overline{LEDBO}$ should not be transitioned from low to high for 30 ns after S0 goes high. This ensures that corrected data is latched into the data output latches. However, $\overline{LEDBO}$ should be taken high before either S0 or S1 go low. Again, this is to ensure that the corrected data is stored into the data output latches. It is important that the new check word be available no later than 32 ns after S0 and S1 go low.

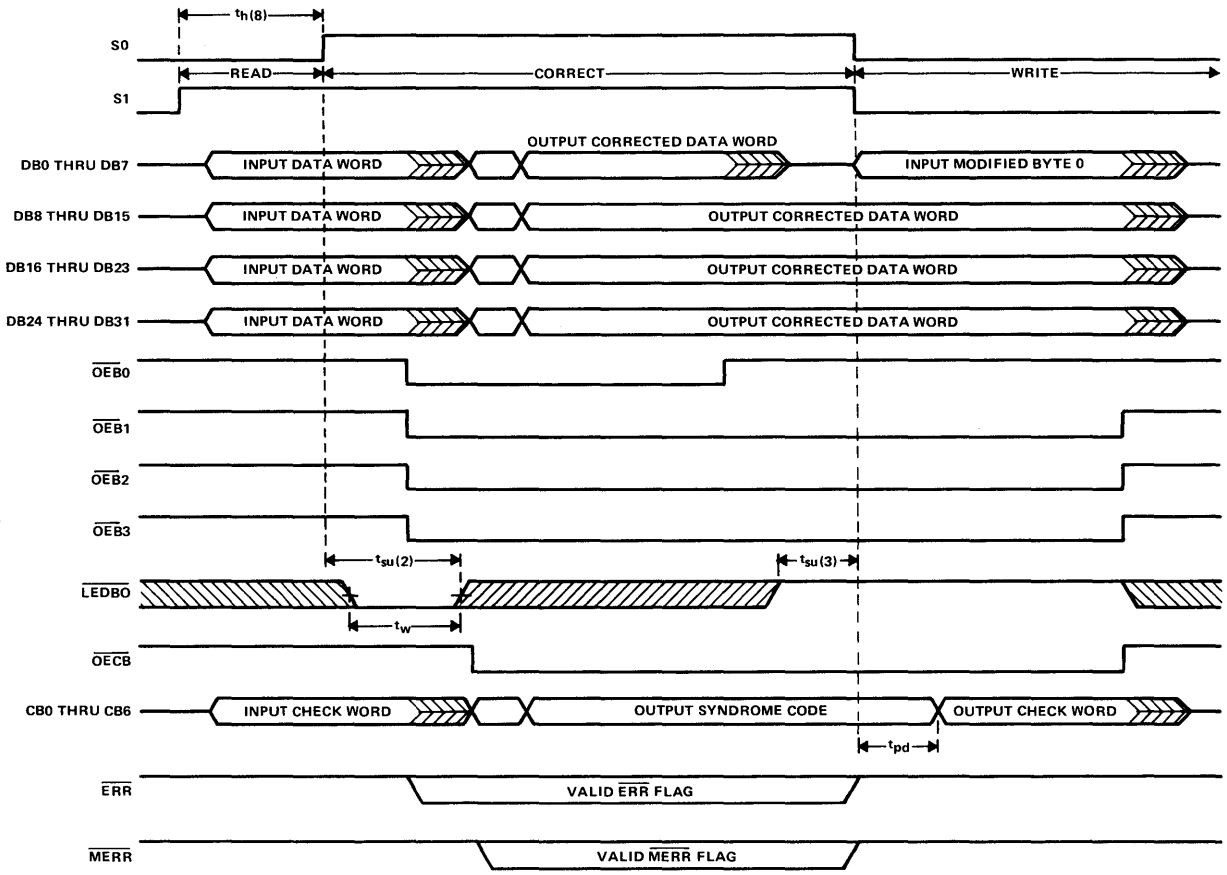


Figure 4-6. Read-Modify-Write Operation

4.2.2.6 Diagnostic Mode Operation

The purpose of the diagnostic mode is to provide the capability of detecting when the EDAC or memory is failing. There are several possible methods of using this feature. Figure 4-7 shows a typical timing diagram of some diagnostics which can be performed with these devices. Generally, the EDAC is first placed in the read mode (S0 = L, S1 = H) and a valid check word and data word are applied. A valid check word is one in which the associated data word is known. The EDAC is next placed into the diagnostic mode by taking S0 high and S1 low. This latches the valid check word into the input latches but leaves the data input latches transparent. To verify that the valid check word was properly latched, $\overline{OECS}$ can be taken low causing the valid check word to be placed back onto the bus. Since the data input latches remain transparent, this allows various diagnostic data words to be applied against the valid check word. A diagnostic data word is one in which either a single- or double-bit error exists. In either case, the error flags respond. The output data latch can be verified by taking $\overline{LEDBO}$ high and confirming the stored diagnostic data word is the same. This is possible because error correction is disabled while in the diagnostic mode (S0 = H, S1 = L). Taking S1 high and $\overline{LEDBO}$ low will verify that the EDAC will correct the data word. In addition, the error-syndrome code can be verified by taking $\overline{OECS}$ low. It should be noted that only the 'ALS632B and 'ALS633 are capable of this pass through verification of the diagnostic data word. The 'ALS634A and 'ALS635 do not have the output data latch required to perform this function.

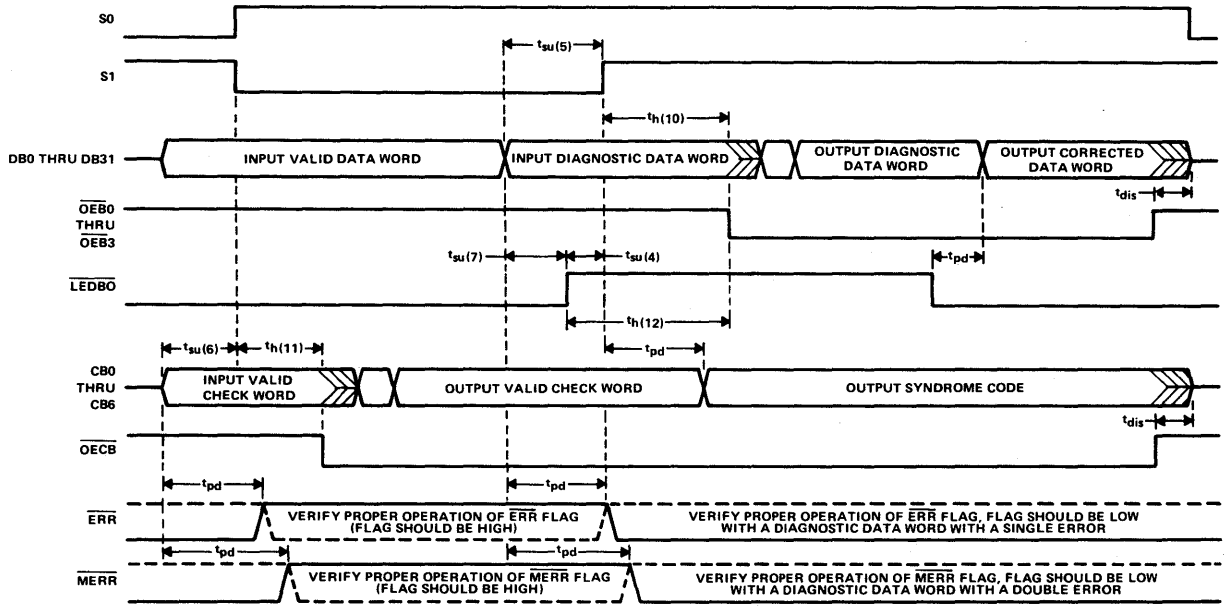


Figure 4-7. Diagnostic Mode Timing Diagram

4.2.2.7 16-Bit Systems Using the 'ALS632B Series EDACs

The 'ALS632B series EDACs can reduce the memory size required in 16-bit systems where conventional 16-bit EDACs (6 check bits, 16 data bits) are presently used. Figure 4-8 shows the typical system architecture for the 16-bit EDAC. In this system, 88 devices would be required for the 22-bit × 256K-byte memory array, assuming 64K-byte DRAMs are used. It is easy to see that 27.3%, or 24 devices, are required for storing the check bits. When using the 'ALS632B series EDACs, the memory required for the check bits can be reduced to 17.9%, or only 14 devices. This reduces the total number of DRAMs required by 10 devices. Figure 4-9 shows the architecture using the 32-bit EDAC. The four 'LS646s are used to group two 16-bit data words into one 32-bit data word. In addition, this type of system can be used in byte-write operations where the other system cannot.

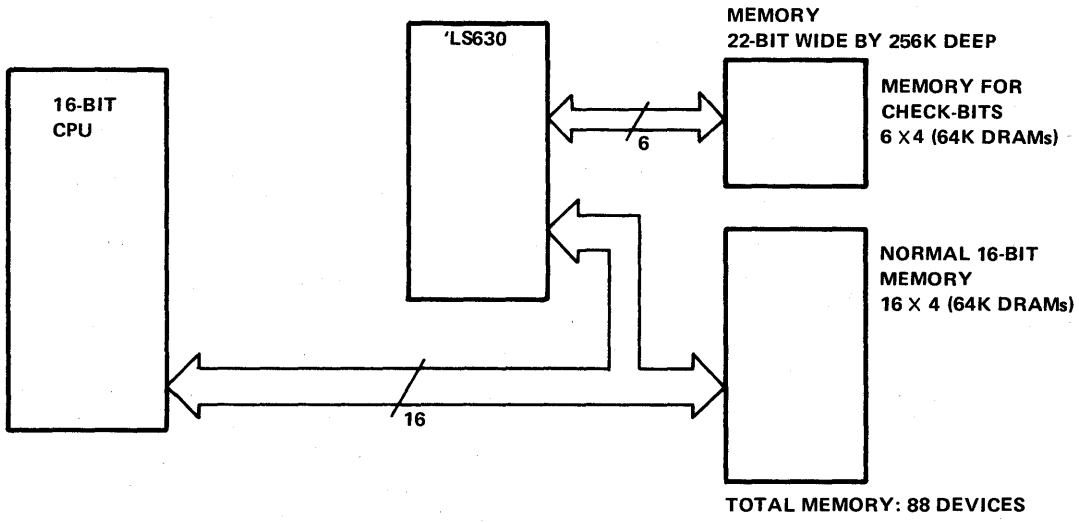


Figure 4-8. 16-Bit System Using Conventional 16-Bit EDAC

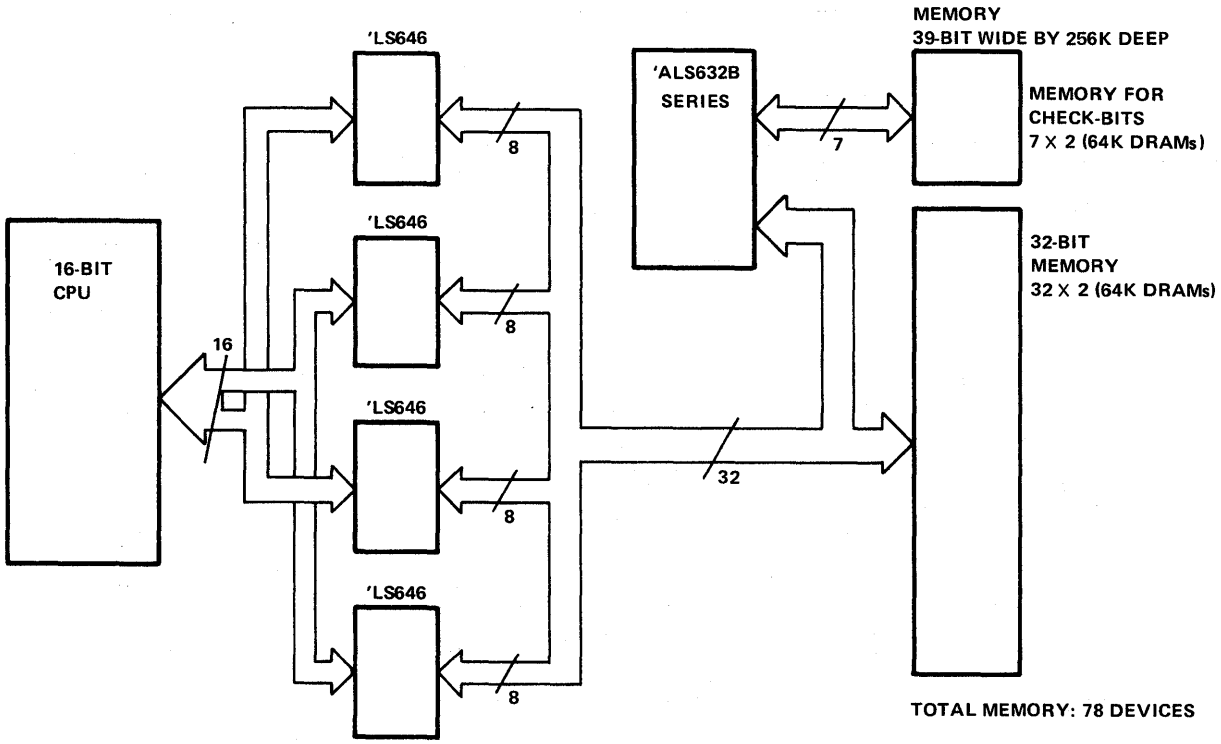


Figure 4-9. 16-Bit System Using 32-Bit EDAC

5 First-In First-Out Memories (FIFO)

5.1 High-Speed Bus Coupling Considerations - FIFO Memory Buffers

5.1.1 Introduction

High-speed First-In-First-Out (FIFO) memory buffers are becoming very important tools for those system design engineers looking for innovative ways to increase system performance. Texas Instruments (TI) brought you the first monolithic FIFO (SN74S225). But many of present day systems require more than the SN74S225 can provide. To meet the needs of those systems, TI has designed an enhanced family of IMPACT™ Bipolar and EPIC™ CMOS FIFO products. Table 5-1 lists some typical applications, key requirements, and the TI FIFO available to meet those needs.

Table 5-1. FIFO Applications

APPLICATION	KEY REQUIREMENTS	FIFO PRODUCTS
CPU Buffering	Data rate of processor Word width/depth Zero fall-through	'LS222/224/227/228 'ALS229A/232A/233A 'ALS2232/2233/2234
Peripheral I/O	Deep/fast Data-path synchronization Status flags	'ALS234/235/236 'ALS2232/2233/2234 TACT7202 TACT2202
Data Acquisition	High data rate	'ALS229A/232A/233A 'ALS234/235/236 'ALS2232/2233/2234
Data/Telecom	Low power/large depth Status flags	TACT7202 TACT2202

This report explains how a TI FIFO can help boost your system performance by maximizing data transfer rates, handling large data streams, or matching different transfer rates. It will also define FIFO architectures and the details of the design considerations needed.

A FIFO is a dual-port buffer memory that is organized in a manner that the first data entered into the memory is the first removed. One port is the input, where the data "producer" enters words into the buffer. The other port is the output, where the data "consumer" removes words. Data in the buffer cannot be randomly addressed like a RAM. A FIFO operates much like a line of people at a checkout counter.

There are two major architectures used in single-chip FIFO; toggle fall-through and zero fall-through.

5.1.2 Toggle Fall-Through Architecture

The toggle fall-through type of FIFO consists of an array of registers. Figure 5-1 illustrates this architecture for an M-word by N-bit FIFO. The output of each register is connected to the input of the following register in a chain-like fashion. Data is input to the first register and is removed from the last register. As each word is input into the FIFO, internal control logic toggles the word through the series of registers to the last one available. As each word is output from the FIFO, all the words are shifted down one register.

EPIC is a trademark of Texas Instruments Incorporated

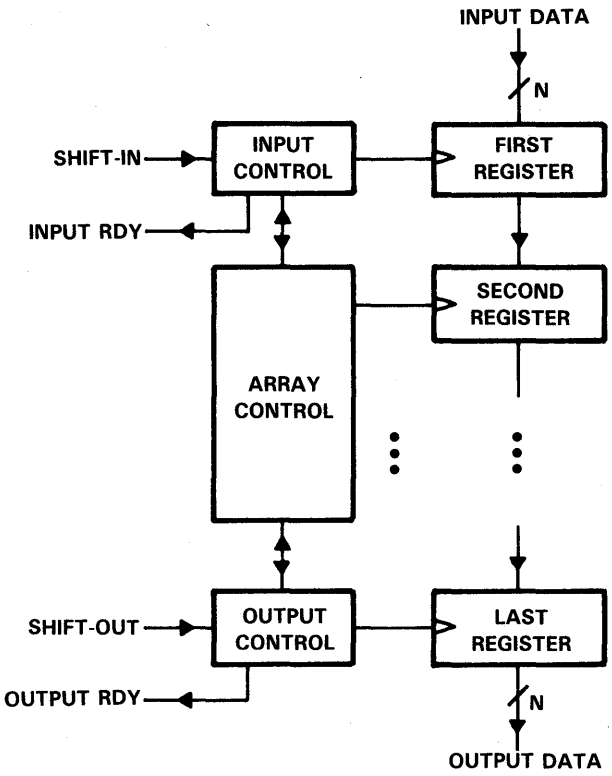


Figure 5-1. Toggle Fall-Through FIFO (M words by N bits)

A toggle fall-through FIFO is described by the number of words in depth, number of bits in width, maximum input and output clocking rates, and fall-through time. The fall-through time is the maximum delay required for a word to travel from the input to the output in an empty FIFO. The complement to this specification is the bubble-through time or the delay it takes for all words to ripple down one register after a word has been read from a full FIFO. However, these two specifications are roughly equivalent so only the greater value is included in the data sheet.

TI offers several toggle fall-through FIFO products. The SN74S225 is a 16 × 5, 10 MHz FIFO. It has 3-state outputs and is cascadable in depth. The SN74ALS234 is a 16 × 4, 30 MHz, cascadable FIFO with 3-state outputs. The SN74ALS235 is a 16 × 5, 25 MHz, cascadable FIFO. It has 3-state outputs and includes half-full and almost full/empty flags. The SN74ALS236 is a bi-state version of the SN74ALS234.

5.1.3 Zero Fall-Through Architecture

The zero fall-through type of FIFO consists of a dual-port RAM with read and write address pointers. Figure 5-2 illustrates this architecture for an M-word by N-bit FIFO. Data is input to the word addressed by the write pointer and data is output from the word addressed by the read pointer. Upon reset, both pointers are cleared to a value of zero. After each word is read or written, the respective pointer is incremented by one. Internal comparison logic is used to generate condition flags such as full and to prevent overrun and under-run (too much writing and too much reading of data).

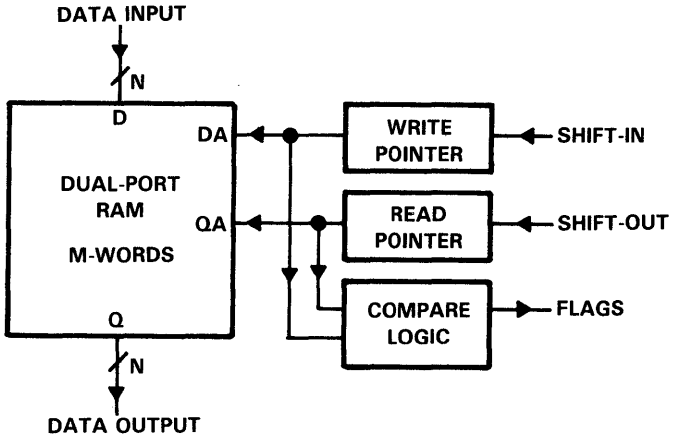


Figure 5-2. Zero Fall-Through FIFO (M words by N bits)

A zero fall-through FIFO is described in terms similar to the toggle products. However, the fall-through time now consists of the delay for incrementing the pointer and comparing the new pointer values. This time is roughly equivalent to the time between shift clocks for the input and the output, or "zero". In many applications zero fall-through FIFOs are preferred to toggle FIFOs for this reason.

TI offers several zero fall-through FIFO products. The SN74LS222 and SN74LS224 are 16 × 4, 10-MHz FIFOs with 3-state outputs. The SN74LS227 and SN74LS228 are open-collector versions of the SN74LS222 and SN74LS224. The SN74ALS229A is a 16 × 5, 30-MHz, 3-state FIFO. It has 4 flags: full, empty, full − 2, and empty + 2. The SN74ALS232A is a 16 × 4, 30-MHz, 3-state FIFO with full and empty flags. The SN74ALS233A is a version of the SN74ALS229A with full − 1 and empty + 1 flags instead of the full − 2 and empty + 2.

The SN74ALS2232 is a 64 × 8, 40-MHz FIFO with 3-state outputs and both full and empty flags. The SN74ALS2233 is a 64 × 9, 40-MHz FIFO with 3-state outputs and four flags: full, empty, almost full/empty, and half full. The SN74ALS22XX is a 64 × 9, 40-MHz, cascadable FIFO with 3-state outputs and both full and empty flags. The TACT7202 is a 1K × 9, 16-MHz cascadable FIFO with full and empty flags. The TACT2202 is a 1K × 8, 16-MHz FIFO with full and empty flags.

5.1.4 Buffering Design Considerations

A FIFO can be used as a buffer between two communication devices. In buffering applications where the delay from input to output is not critical (e.g., CPU to printer) either a toggle or zero fall-through FIFO can be used. In this instance, only the input and output clocking rates and the depth of the FIFO are critical. In buffering applications where the fall-through delay is important (e.g., bus interface) then the zero fall-through architecture should be used.

The rare case for FIFO operation is when the consumer is faster than the producer. A FIFO depth of one word would suffice. The other situation (see Figure 5-3) requires more words.

Figure 5-3. Buffering Application

The following equations can be used, to determine the depth needed. If the data producer writes a frame of (L) words at a rate of (X) MHz and the data consumer reads words at a rate of (Y) MHz, the resulting equations are:

1. $L \bullet 1/X + (\text{BUFFER DELAY}) = L \bullet 1/Y$
2. $\text{MAX (BUFFER DELAY)} = \text{DEPTH} \bullet 1/X$
3. $\text{MAX (L)} \bullet 1/X + \text{DEPTH} \bullet 1/X = \text{MAX (L)} \bullet 1/Y$
4. $\text{DEPTH} \bullet 1/X = \text{MAX (L)} \bullet (1/Y - 1/X)$
5. $\text{DEPTH} = \text{MAX (L)} \bullet (X/Y - 1)$

For example, with L = 100 words maximum, X = 8 MHz, and Y = 5 MHz, the necessary depth = 60 words. In this case, a 64-word FIFO would suffice.

The reverse of equation 5 gives the maximum length of a frame for a given FIFO depth:

6. $$\text{MAX (L)} = \text{DEPTH} \bullet \frac{1}{(X/Y - 1)}$$

For example, with depth = 64, X = 1 MHz, and Y = 0.8 MHz, the maximum length of a frame = 256 words.

5.1.5 Synchronization Design Considerations

In synchronizing applications, the data producer and consumer can operate continuously but asynchronously. The maximum throughput of the FIFO depends on both the clock rate at each port (F_{in} and F_{out}) and the fall-through time (T_F). (f_{max}) is derived from the maximum one-word time delay through the FIFO. The equations are:

7. $$\text{MAXIMUM DELAY} = \frac{1}{F_{in}} + T_F$$
8. $$\frac{1}{f_{max}} = \frac{1}{F_{in}} + T_F$$
9. $$f_{max} = \frac{1}{1/F_{in} + T_F}$$

For a toggle fall-through FIFO in these conditions f_{max} is considerably less than F_{in}. For example, if F_{in} = 30 MHz and T_F = 1000 ns, then f_{max} = 967 KHz. For a zero fall-through FIFO, f_{max} approaches F_{in}. For example, if F_{in} = 35 MHz and T_F = 40 ns, then f_{max} = 14.6 MHz. To ensure only a single-clock delay from the input and to output ports, the FIFO must be clocked at a rate less than f_{max}.

The FIFO may be operated at higher rates by working near the half-full condition. The number of words that can be written into an empty FIFO in the fall-through time (or read from a full FIFO) determines the margins from the empty condition for operating the FIFO at its maximum rate (F_{in}). The numbers correspond to:

10. $$\text{MARGIN} = \frac{F_{in}}{1/T_F}$$

For example, if F_{in} = 30 MHz and T_F = 1000 ns, then the margin = 30 words. For a 64-word FIFO this would mean that the FIFO could be operated at its maximum throughput rate (F_{in}) when it is between 30 words and 64 words full. Figure 5-4 shows the throughput curve for this type FIFO.

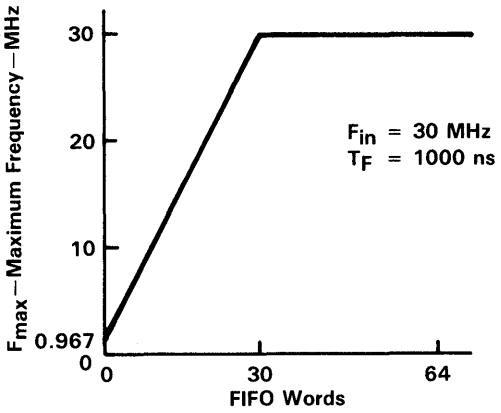


Figure 5-4. Throughput Curve for 64-Word, 30-MHz FIFO

In general, cascading N FIFOs in depth causes equations 9 and 10 to change to:

11. $$f_{max} = \frac{1}{1/(F_{in} + N \bullet T_F)}$$

12. $$MARGIN = \frac{F_{in}}{1/(N \bullet T_F)}$$

5.1.6 Summary

FIFOs are versatile building blocks for the design of data communication products. The need for buffering and/or synchronization of data can be met by selecting the appropriately-sized toggle or zero fall-through FIFO using the methods presented in this report. TI produces many different single-chip FIFO products for a wide range of applications. Contact your local TI representative to obtain individual FIFO data sheets for further information about a particular product.

6 BiCMOS

6.1 BiCMOS Memory Drivers Boost Performance

In current memory management systems, the replacement of discrete logic with single-chip solutions for DRAM control, Error Detection and Correction, and Cache Tag control has greatly improved memory access times. However, in large MOS memory applications the use of external drivers in conjunction with the memory management products can provide added drive to maintain maximum performance. These drivers must meet the requirements of high drive for high capacitive loads, high speed for maximum system throughput, and low power for system power constraints. The designer can now meet these needs with the TI 2000 series Bus Interface devices with improved performance and reliability. The devices offered in new BiCMOS, 'AS, and 'ALS technologies provide designers with the characteristics needed to drive the high capacitive loads in MOS memory and bus-intensive systems while reducing undershoot for reliable system performance.

6.1.1 Reducing Undershoot Problems

In order to maintain maximum system throughput, memory drivers require high-speed operation with very fast switching speeds. As a result, these switching speeds together with the high inductance and capacitance in bus intensive environments can create problems with output signal undershoot and overshoot. This undershoot and overshoot can cause system reliability problems such as false reads at the input to DRAMs. Commonly, these problems with undershoot and overshoot are controlled with an external series resistor, which increases package count and board space. The 2000 series devices provide on-chip 25-Ω series damping resistors on all outputs to reduce undershoot and overshoot without adding to board real estate. Figure 6-1 compares the initial undershoot of the 'AS640 and the 'AS2640 with on-chip series damping resistors. The 'AS2640 can reduce initial undershoot by 58% thus supplying a more reliable input to systems susceptible to undershoot problems.

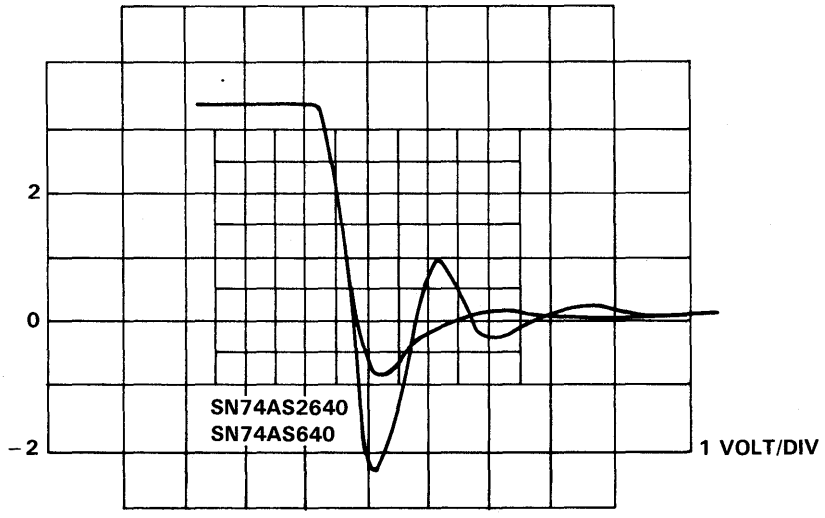


Figure 6-1. Effect of On-Chip Series Output Resistors

6.1.2 BiCMOS Drivers Match MOS Memory Needs

The 2000 series devices offered in the new TI BiCMOS technology provide the advantages of both bipolar and CMOS. BiCMOS combines 2-μM IMPACT^{TM} bipolar with 1.5-μM CMOS to provide the high drive and speeds of bipolar and the low power of CMOS. These interface devices have TTL input and output transistors with CMOS internal circuits. The output transistors supply 48/64 mA of drive current necessary for bus structures such as VME and MULTIBUS II, while the CMOS internal circuits provide low power during disabled or 3-state operation. As with all 2000 series devices, the BiCMOS parts have series damping resistors to reduce undershoot and overshoot.

The BiCMOS drivers can provide the drive and speed necessary in MOS memory applications with a power savings over bipolar devices. Figure 6-2 shows a 4-M word × 32-bit memory configuration consisting of a SN74ALS6301 Dynamic Memory Controller (DMC), a SN74BCT2828 Memory Driver and 4-M words of memory comprised of four banks of TMS4C1024 DRAMs. Each SN74ALS6301 can control up to 4M words of memory. The memory driver provides extra drive to maintain maximum performance in a 32-bit system. The 10-bit SN74BCT2828 gives a single-package reliable solution with up to 48 mA of output-drive current.

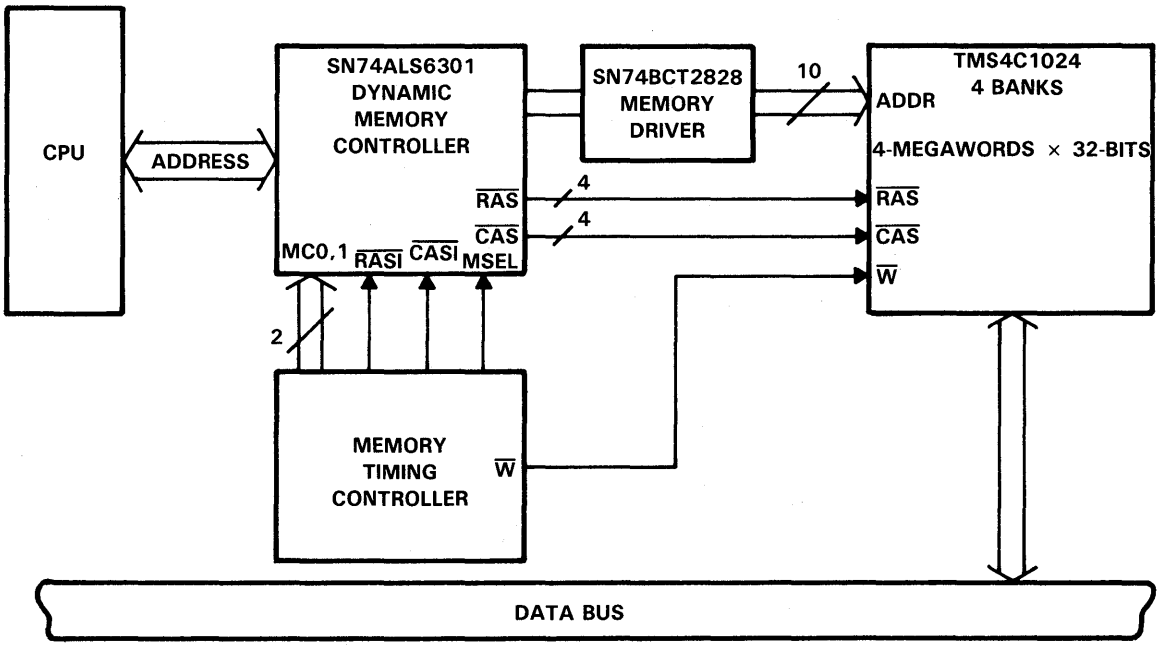


Figure 6-2. 4M Word × 32-Bit Memory System

6.1.3 BiCMOS Lowers Power by 50% or More

When comparing the performance of the SN74BCT2828 to the functionally equivalent AM29828, there is a considerable power reduction. As shown below, there is a 50% reduction in supply current while enabled. However, the real savings comes from the disabled operation. There is more than a 95% supply current reduction while disabled. Since the amount of time a driver is enabled varies with each system, power reduction will vary with the minimum being 50% improvement.

	AM29828	'BCT2828
I_{CC} enabled	80 mA	40 mA
I_{CC} disabled	80 mA	3 mA

In applications that involve multiple drivers the power savings is even more apparent. For example, if a system requires five drivers with only one enabled at any given time, the AM29828 would use almost 8 times more current then the SN74BCT2828.

	AM29828	'BCT2828
I_{CC} enabled	1 × 80 mA	1 × 40 mA
I_{CC} disabled	4 × 80 mA	4 × 3 mA
Total	400 mA	52 mA

Result = 87% power savings

6.1.4 Less Undershoot Means Higher Reliability

The use of the 2000 series BiCMOS drivers also provides the reduced undershoot to prevent false reads at the inputs to the DRAMs without the addition of external resistors. Figure 6-3 shows the improvement of initial undershoot of the SN74BCT2828 compared with the AM29828. The SN74BCT2828 undershoot is 40% less than the AM29828 providing a more reliable signal with the same package count.

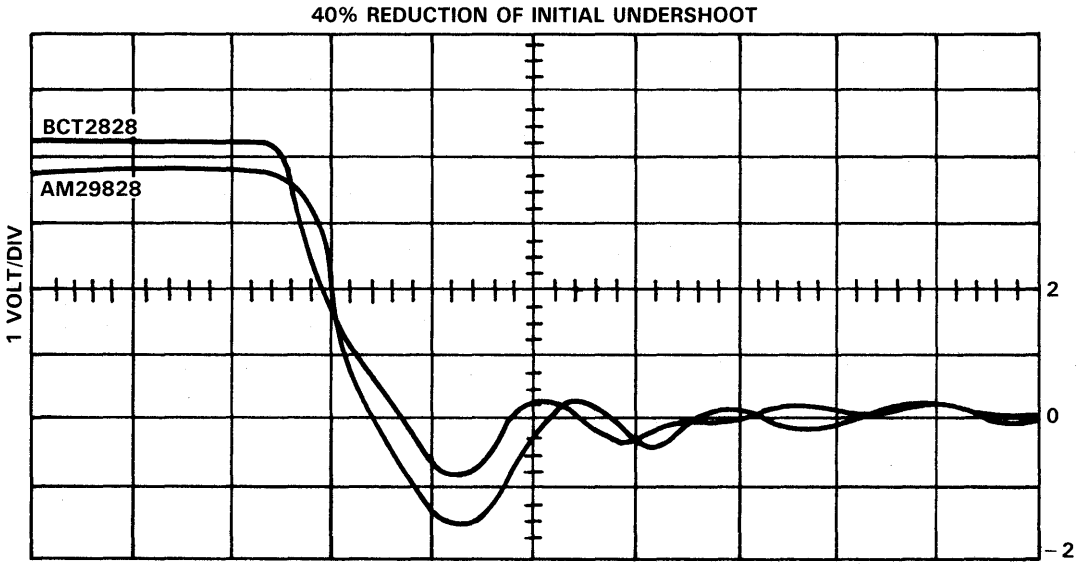


Figure 6-3. Initial Undershoot Comparison of 74BCT2828 vs AM2928

6.1.5 How Do I Get More Information

Each of the 2000 Bus Interface series devices provide the designer with reliable signals without increasing package count and board real estate. The high drive and speed complement Memory Management products for use in large memory and bus applications. The onset of BiCMOS also brings a tremendous power savings which can be appreciated in all designs. Below is a listing of the 2000 series offered. For more information on these Bus Interface and Memory Management products, contact your local Texas Instruments field sales representative or authorized distributor.

Device	Description	Output	I_{OL} (mA)
'BCT2240	Octal Buffer/Driver	Inverting	35
'BCT2241	Octal Buffer/Driver	True	35
'BCT2244	Octal Buffer/Driver	True	35
'BCT2540	Octal Buffer/Driver	Inverting	35
'BCT2541	Octal Buffer/Driver	True	35
'BCT2827	10-bit Buffer/Driver	True	12
'BCT2828	10-bit Buffer/Driver	Inverting	12
'ALS2240	Octal Buffer/Driver	Inverting	15
'ALS2242	Octal Transceiver	Inverting	30
'ALS2244	Octal Buffer/Driver	True	30
'ALS2540	Octal Buffer/Driver	Inverting	30
'ALS2541	Octal Buffer/Driver	True	30
'AS2620	Octal Transceiver	Inverting	35
'AS2623	Octal Transceiver	True	35
'AS2640	Octal Transceiver	Inverting	35
'AS2645	Octal Transceiver	True	35

6.2 BiCMOS Bus Interface

6.2.1 Abstract

Bipolar and CMOS processes have their individual advantages. The advantages of bipolar are speed and output drive current capability. The advantage of CMOS is significantly lower power consumption with continually improving speed performance. The merge of the two processes in order to use their individual advantages for optimal product development was therefore a predictable technology transition.

This portion of this report concerns the use of such a process, BiCMOS, and the advantages provided in bus-interface logic. Ultimately, the system advantage gained from the use of BiCMOS bus interface logic results in a 25% reduction of total system power.

6.2.2 Introduction

Bus-interface logic requires very high output-drive currents of 48/64 mA. These currents are required to drive high-capacitive loads and backplanes and to meet the required specifications of established standards. Advanced speed performance is also a necessity to allow the rapid transfer of information and to compliment the performance of other system components.

Excessive power consumption was the tradeoff that system designers were forced to accept to achieve the desired output-drive current and speed performance. In an average system, 30% of the total device supply current is required to support the bus interface logic. The use of BiCMOS bus-interface logic can reduce the required device supply current by more than 90%. This results in an overall system power savings of more than 25%.

6.2.3 Reduction of Supply Current Demand Without Sacrificing Performance

The combination of bipolar and CMOS components makes the power savings a reality without sacrificing required output drive current or speed performance. An examination of bus-interface logic in system operation reveals that the device is either enabled

(active) or disabled. Since in bus configurations only one device is active at any given time, the remaining devices tied to the bus are disabled. Therefore, for the majority of the time, devices tied to the bus are in a disabled mode. Further evaluation reveals that the currently available bipolar devices that are capable of meeting the specifications required for bus-interface logic require supply currents ranging from 75 mA to 160 mA per device depending upon the function.

BiCMOS requires approximately 10 mA maximum (I_{CCZ}) when disabled and further reduces the active supply current demand by approximately 50% compared to equivalent bipolar devices. Table 6-1 is a comparison of the SN74BCT29861 and AM29861 supply currents. Figure 6-4 illustrates typical switching performance and disabled supply current demand between the two devices.

Table 6-1. SN74BCT29861/AM29861 I_{CC} Comparison

	SN74BCT29861		
I_{CC}	Supply current	Enable	30 mA (Max)
	(V_{CC} = 5.5 V @ 70°C)	Disable	7 mA (Max)
	AM29861		
†I_{CC}	Supply current		150 mA (Max)
	(V_{CC} = 5.5 V @ 70°C)		

†Advanced Micro Devices Bipolar Microprocessor Logic and Interface 1985 Data Book. No breakout given for enable or disable I_{CC}.

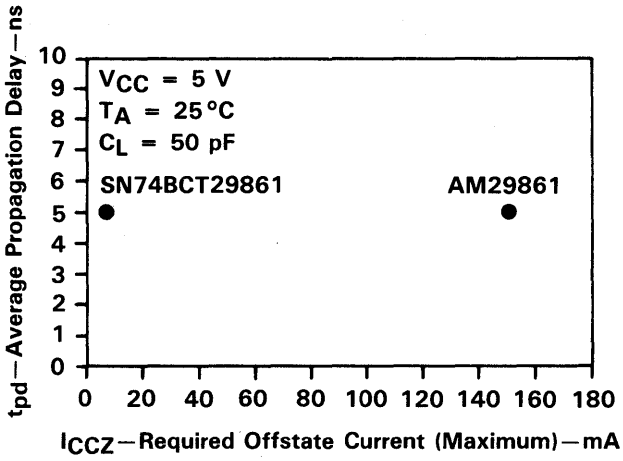


Figure 6-4. SN74BCT29861 and AM29861 Required Off-State Current vs Average Propagation Delay

To highlight the system power savings advantage exhibited by BiCMOS bus-interface products, see the conditions in Figure 6-5. Assuming a bus network contains a fanout of ten bus interface devices, Figure 6-5 illustrates that only one device is enabled, while the other nine are disabled.

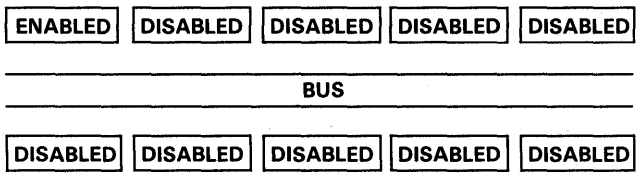


Figure 6-5. Bus Network

Assumption:

		Advanced Bipolar	BiCMOS
ICC	(Enable)	150 mA	30 mA
ICCZ	(Disable)	150 mA	10 mA
		Advanced Bipolar	
ICC	(Enable)	1 × 150 mA = 150 mA	
ICCZ	(Disable)	9 × 150 mA = 1350 mA	
ICC	(Total)	1500 mA	
		BiCMOS	
ICC	(Enable)	1 × 30 mA = 30 mA	
ICCZ	(Disable)	9 × 7 mA = 63 mA	
ICC	(Total)	93 mA	

Result: 94% power savings.

6.2.4 Combinational Bipolar and CMOS Optimal Process Solution

BiCMOS bus-interface logic is a TTL-to-TTL interface product that provides the optimal combination of speed performance, output drive, and low power. To achieve these characteristics, TI combines 2-μm bipolar IMPACT^{TM} (Implanted Advanced Composed Technology) process with 1.5-μm CMOS process is shown in Figure 6-6.

The bipolar process provides output transistors capable of supplying the required 48/64 mA. The transistors also use the smaller TTL voltage swings of -0.5 V- to -3.5 V as compared to their rail-to-rail or GND-to-V_{CC} voltage swings that are associated with CMOS transistors. The smaller voltage swings associated with TTL outputs reduce the overall effect of transient voltage noise on the ground pins. Excessive noise spikes can be detrimental to reliable system performance due to output glitching, loss of stored data, increase of system noise, etc.

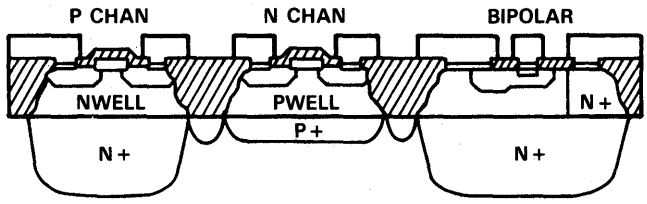


Figure 6-6. BiCMOS Process

The following equation is a simple method of calculating the induced voltage on the ground and V_{CC} pins due to transient currents caused by switching capacitive loads.

$$V_L(t) = -L_P C_L \frac{d^2V_O(t)}{dt^2}$$

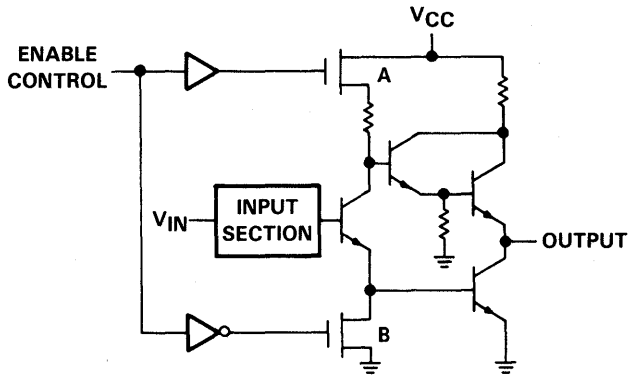
Where:

$V_{L(t)}$ = Voltage transient
L_P = Package inductance
C_L = Load capacitance
$d^2V_O(t)$ = Change in the slope of the transition edge
dt = Transition edge time

Since L_P, C_L, and dt are the same, the amount of voltage level transition swing is the only difference between the bipolar and CMOS transistors. Since CMOS transistors require a wider voltage swing, it becomes apparent that a CMOS output transistor will produce a larger amount of voltage noise that, if excessive, could cause system reliability problems.

The CMOS process provides a disable circuit that consumes considerably less current than a pure bipolar circuit. Figure 6-7 illustrates how the CMOS components combine with the bipolar components to interrupt the flow of supply current during the disable mode or three state. The remaining internal components are also fabricated from CMOS which further reduces the required amount of supply current.

Both the bipolar and CMOS processes provide the capability to adequately meet the advanced speed performance required for bus interface.



- DURING OPERATION: A SHORTED, B OPEN
- DURING THREE-STATE: A OPEN, B SHORTED

Figure 6-7. BiCMOS Three-State Gate Schematic

6.2.5 Variety of Functional Options in Two-Package Configurations

Additional design support for bus-interface logic is the availability of popular functions in multiple variation such as true or inverting outputs and synchronous or asynchronous operation. BiCMOS will be offered with two pinout options; 1) The traditional pinout for pin-to-pin compatibility with existing bipolar devices. 2) Flow through architecture with center power pins to further reduce the voltage noise associated with multiple output switching.

As indicated by the $V_L(t)$ equation, the amount of switching noise can be reduced through a decrease in the package inductance.

The functional options that will be available in BiCMOS are as follows:

FUNCTION	DESCRIPTION
'240 Series	Octal Buffers/Drivers
'245 Series	Octal Transceivers
'373 Series	Octal Latches
'543 Series	Octal Registered Transceivers
'646 Series	Octal Registered Transceivers
'2000 Series	Memory Drivers
'29818/819	Pipeline Registers
'29820 Series	8-10 Bit Buffers and Registers
'29830 Series	Bidirectional Parity Transceivers
'29840 Series	8-10 Bit Latches and D-Latches
'29850 Series	Bidirectional Parity Transceivers with Latches
'29860 Series	9-10 Bit Transceivers

6.2.6 Summary

BiCMOS bus-interface logic is a TTL-to-TTL product that provides a 95% reduction in standby current demand. This results in a 25% total system power savings without sacrificing high output drive or speed performance.