SDC-93-584

SDC
SOLENOIDAL DETECTOR NOTES

Top-Down Design of a Prototype VME Slave Interface

September 10, 1993

Elise Y. Tung
Lawrence Berkeley Laboratory

September 10, 1993

Top-Down Design of a Prototype VME Slave Interface

Elise Y. Tung
Lawrence Berkeley Laboratory

1. Abstract

This technical report discusses the design of a simple VMEbus slave interface board using state-of-the-art software tools. Since VMEbus-based circuit cards are widely used within SDC, the ability to quickly design an interface board that is functional, reliable and portable is extremely important. The unique feature of this design is that a top-down design methodology was employed, and many industry standard tools were used for design and simulation so that design flaws could be located and corrected early in the design process and technology obsolescence could be prevented. The main goal of this design was to experiment with new software design tools and create a path for future designs, so the interface board was simplified significantly compared to a full VMEbus slave implementation. The slave interface design will be enhanced in the near future.

2. Introduction

The idea of top-down design is to start working from the top-level module, and then divide this module into a series of functional blocks and interfaces between each functional block. Each functional block can then be divided into even smaller modules depending on how complex the bottom level functional blocks are desired to be. By establishing a hierarchical structure, the logic of the design is much more simplified, thus design correctness is easily obtained.

Mentor Graphics Corporation suite of EDA (Electronic Design Automation) tools and IEEE Standard 1076-1987 VHDL (VHSIC Hardware Description Language) were used in the design of the VMEbus Slave Interface Board. By using industry standard tools as a common design database, the design can be easily maintained throughout its life cycle. Moreover, having each module written in VHDL code prevents parts obsolescence and made the design portable to any system. Employing Mentor Graphics Corporation's simulation tools (Quicksim II) and Logic Modeling Corporation's VME Simulation model also enables complete simulation of the design at both the system and component level.

3. Design Specification

The design specifications for the VMEbus slave interface board are as follows:
a) A24 (23:1) Address Bus line and D16 (15:0) Data Bus line
   Slave I/O data transfers using D08(EO), D16
   Supports READ, WRITE, and RMW operations

b) Upper 5 bits on Address Bus are board address specified from the Master board, a 5-bit dip-switch is the slave board's selectable board address
c) 13 chip select fields, 1-7 active low, 8-13 active high
d) interface to memory

In addition to the hardware specifications, LED readouts are added for testing. These readouts include LED indicators for Board Select (bs) and one of the general purpose chip selects and several HexDisplays to show data on the data bus lines.

4. Circuit Description

4.1. Operational Overview

The design of the slave interface controller is based on "**THE VMEbus SPECIFICATION** -- conforms to: ANSI/IEEE STD1014-1987 IEC 821 and 297." The IEEE1014-87 standard specifies a high-performance backplane bus for use in microcomputer systems that employ single or multiple microprocessors so that data can be transferred between functional modules. A master is a functional module that initiates data transfer bus (DTB) cycles, while a slave is a functional module that detects data transfer bus (DTB) cycles and, when those cycles specify its participation, transfer data between itself and the master.

The VMEbus slave interface controller responds to signals on the VMEbus, generates on-board control signals correspondingly, and finally issues signals to the master on the VMEbus to fulfill the handshaking protocol.

There are six modules in the interface system:
1) Address Modifier(AM) Decoding Module (am_dcd)
2) Board Address Decoding Module (brd_dcd)
3) Chip Select Control Module (cs_ctl)
4) Read/Write Interface Module (mem_st_ctl2)
5) Transceiver Control Module (xvr_ctl)
6) Data Acknowledgment Module (dtack_ctl)

Whenever there is a bus cycle initiated by the master, the VME slave interface will first check if the data transfer (short, standard or extended) is supported by the interface. The interface will then check that the board address on the address bus matches the interface board address. After passing this check, the interface will decode the addresses on the address bus to enable one of the chip selects. Once data transfer has been accepted by the slave, the slave interface will assert the Data Acknowledgement (DTACK) signal to acknowledge the master. The master will finish the cycle by deasserting the data strobe signal(s). When the slave detects this change on the data strobe, it will deassert DTACK to finish the bus cycle and fulfill the handshake mechanism.

An overall view of the interface board is shown in figure 1. The block diagram of the interface controller is shown in figure 2.

4.2 Address Modifier(AM) Decoding Module (am_dcd)

The VMEbus implementation residing on the board is capable of operating with the following Address Modifier (AM) codes:

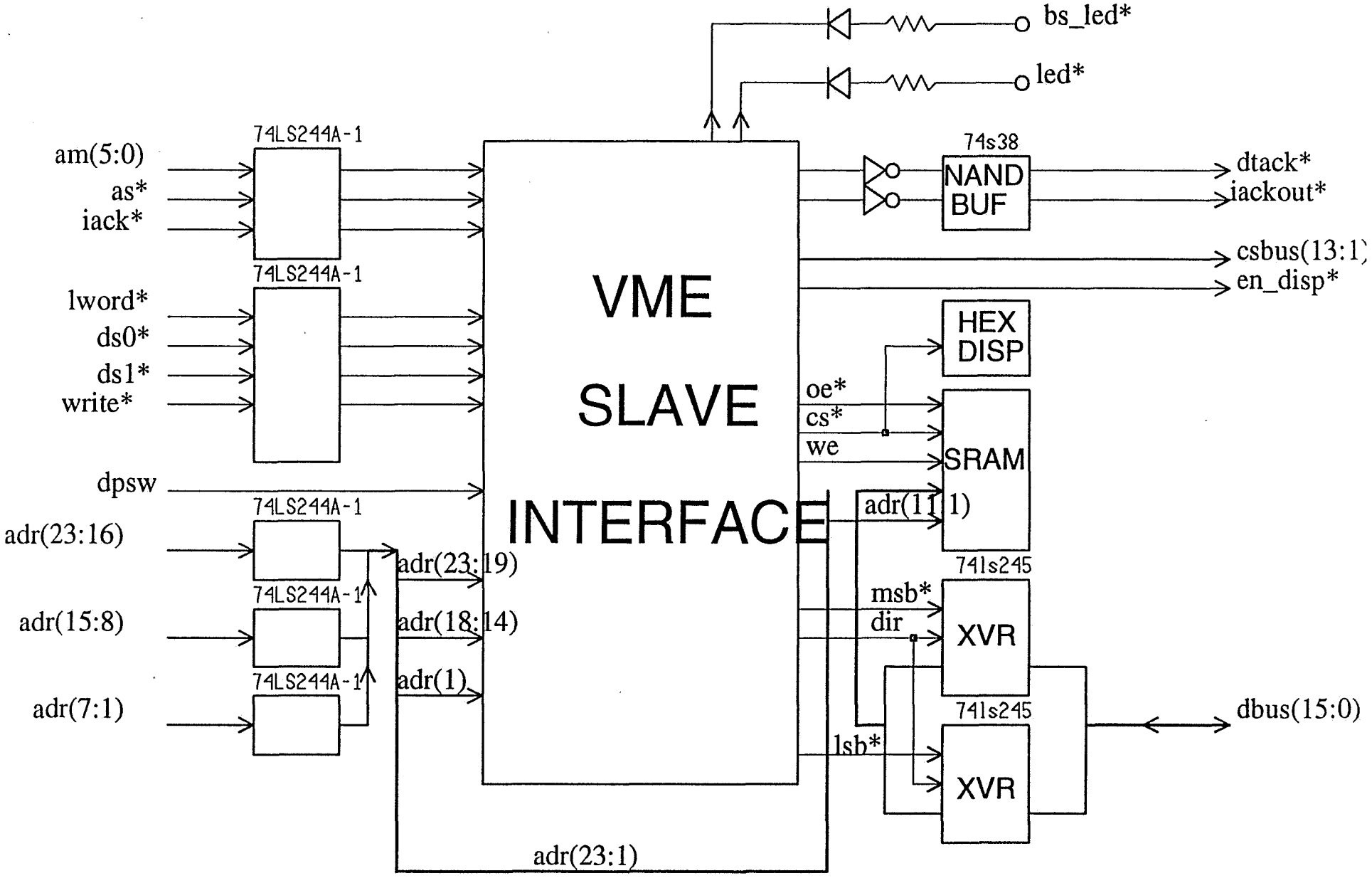


Figure 1. An overall view of the interface board

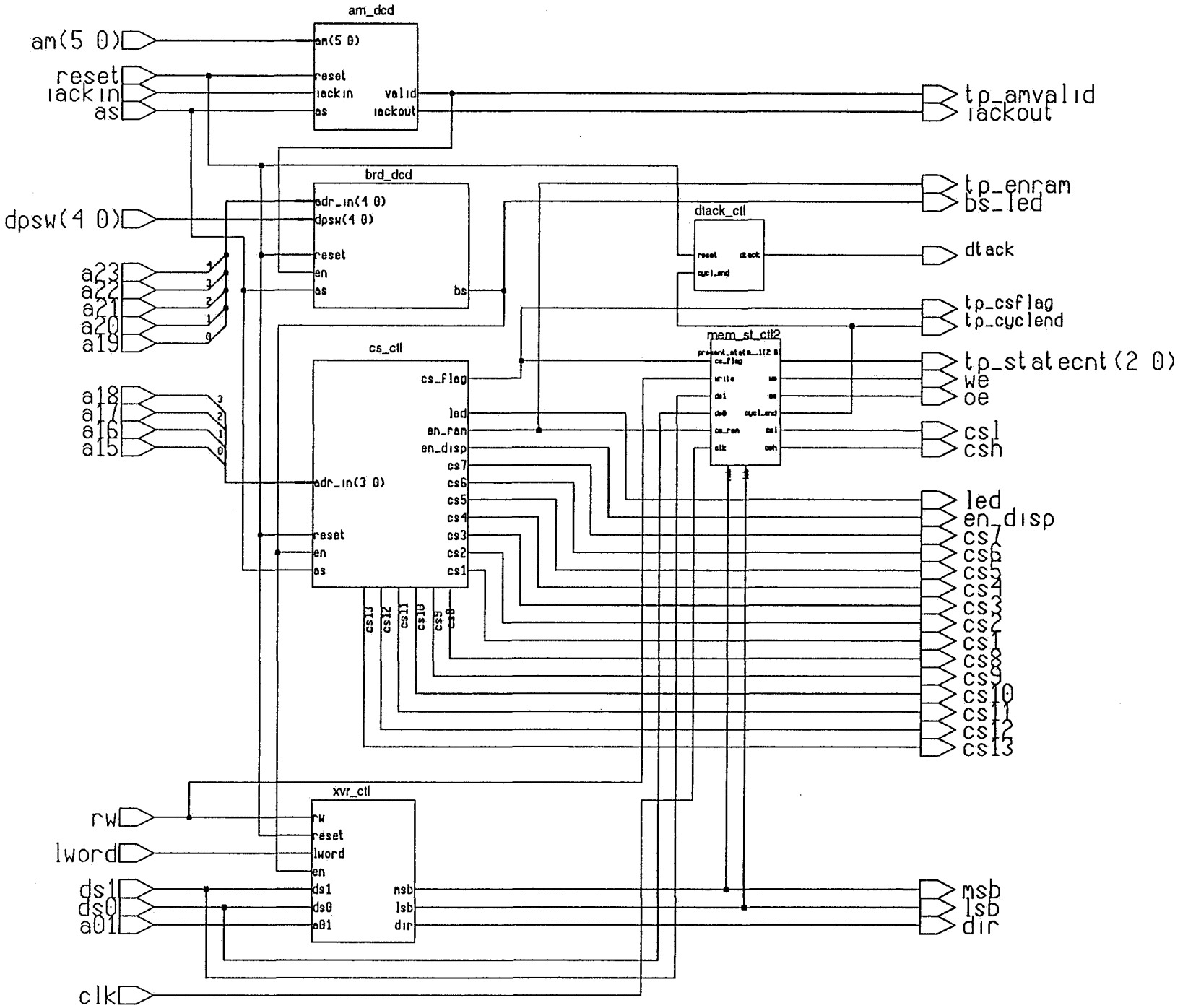


Figure 2. Block diagram of the interface controller

Table 1. Supported Address Modifier codes

AM Codes (AM5-AM0)	Function
3E (HEX)	Standard Supervisory Program Access
3D (HEX)	Standard Supervisory Data Access
3A (HEX)	Standard Nonpriviledged Program Access
39 (HEX)	Standard Nonpriviledged Data Access

The am_dcd module checks whether the data transfer (short, standard or extended) is supported by the interface. The design only supports standard data transfer so the AM codes can only be 3E, 3D, 3A, 39. If the AM codes are valid, the am_dcd module will send a "valid" signal to Board Address Decoding Module to proceed. If the AM codes are not supported by the interface, it will deassert the "valid" signal so that the board won't respond to the master's request. See the appendix for the VHDL code.

4.3 Board Address Decoding Module (brd_dcd)

The brd_dcd module ensures the board address on the address bus matches the slave board's base address. Address bits 19-23 on the address bus are the board address. The slave's board address is set by 5 bits on a dip switch. If the addresses match, the brd_dcd module will send a Board Select (bs) signal to the Chip Select Control Module to enable its function. See the appendix for the VHDL code.

4.4 Chip Select Control Module (cs_ctl)

The cs_ctl module will start decoding the address bus once the enable signal from brd_dcd is received. Address bit 15-18 are used for decoding purposes. Table 2 shows the decoding table.

Table 2. decoding table

cs \ bit	18	17	16	15	active
\en_disp	1	1	1	1	L
\en_ram	1	1	1	0	L
cs13	1	1	0	1	H
cs12	1	1	0	0	H
cs11	1	0	1	1	H
cs10	1	0	1	0	H
cs9	1	0	0	1	H
cs8	1	0	0	0	H
\cs7	0	1	1	1	L
\cs6	0	1	1	0	L
\cs5	0	1	0	1	L
\cs4	0	1	0	0	L
\cs3	0	0	1	1	L
\cs2	0	0	1	0	L
\cs1	0	0	0	1	L
\led	0	0	0	0	L

En_ram enables the SRAM Interface Module for read and write cycles to the SRAM. Other chip select signals are for future board enhancements and testing. See the appendix for the VHDL code.

4.5 READ/WRITE Interface Module (mem_st_ctl2)

The mem_st_ctl2 module manages the VMEbus accesses for the following normal slave I/O transfers: READ, WRITE, READ-MODIFY-WRITE.

The mem_st_ctl2 module uses an external clock (originally assumed to be at 33ns clock period when writing VHDL codes) to set the control signals of the SRAM -- Chip Select (CS), Write Enable (WE), Output Enable (OE). The three signals are set during different clock periods using the timing information of IDT 6116SA55 so that they will meet all the setup and hold times to ensure reading the right data from or writing the right data to the SRAM. Both read and write cycles take three clock cycles to finish.

A VHDL implementation of a Moore State Machine is used to control the CS, WE and OE signals. There is one reset state, three states for the READ cycle and three states for the WRITE cycle:

RESET state:
When either data strobes (ds0, ds1) is high, the state machine goes to state 0. OE, CS are disabled, and WE is set to read. Cycl_end flag is unset.
READ cycle:
Since Tacs(Chip Select Time) is at most 50 ns, it takes 2 clock cycles to finish the READ(state 1, state 2). Cycl_end flag is set in state 3 to finish the cycle.
WRITE cycle:
Since Tcw(Chip Select to End of Write) is about 40 ns, it takes 2 clock cycles to finish the write(state 4, state 5). Cycl_end flag is set in state 6 to finish the cycle.

The mem_st_ctl2 module also takes care of the bus events which do not require access to the SRAM. In these cases the state machine will jump from reset state(state0) directly to the end cycle state(state6) which will set the Cycl_end flag.

The block diagram for the Moore State Machine used in the READ/WRITE interface module is shown in figure 3. See the appendix for the VHDL code.

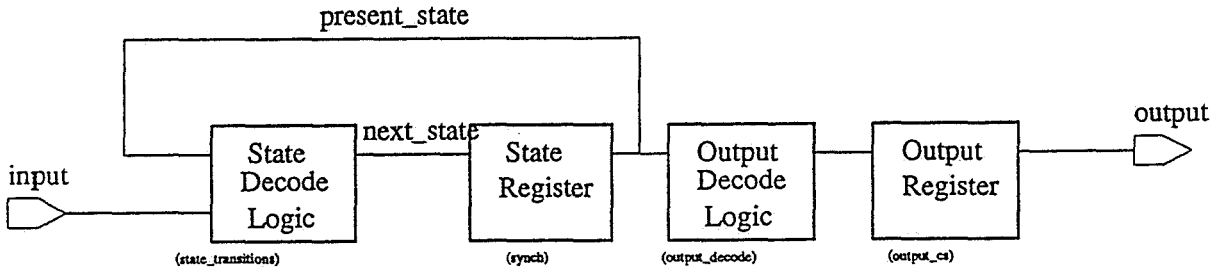


Figure 3. Block diagram for the Moore State Machine in R/W interface module

4.6 Transceiver Control Module (xvr_ctl)

The transceiver control module is used to control which byte, the most significant byte (msb) or the least significant byte(lsb), to choose when receiving from or sending data to the data bus. The choice is made based on the data strobes(ds1, ds0), the address bit 1(a01) and lword. (See table 3)

Table 3. control table of signals msb, lsb

ds1	ds0	a01	lword	msb	lsb
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	1	0
0	0	0	0	0	0
0	0	1	1	0	0

Note: 1) ds1, ds0, lword, msb and lsb are all active low.
2) Other values of ds1, ds0, a01, lword will have both msb and lsb disabled. (msb = '1', lsb = '1')

This design supports single-byte and double-byte access only. Unaligned data access is not supported.
Single-byte access: BYTE(0), BYTE(1), BYTE(2), BYTE(3)
Double-byte access: BYTE(0-1), BYTE(2-3)

See the appendix for the VHDL code.

4.7 Data Acknowledgement module(dtack_ctl)

Since bus error control is not supported in this design, it makes the dtack_ctl module very simple to implement. Whenever the cycl_end flag is set by the READ/WRITE interface module, the dtack_ctl module will enable the Data Acknowledgement (DTACK) signal as long as the reset signal from the front panel is deasserted. This is to acknowledge that the slave has responded to the bus cycle initiated by the master so cycle can be finished. See the appendix for the VHDL code.

5. Hardware Description

The slave interface controller was targeted into a CrossPoint FPGA (CP20420-155CPGA). Besides , there are two IDTSA55 CMOS SRAMs on the board. Each SRAM can hold 16K (2K x 8 bit) of data. Bits 1 to 11 on the Address Bus are connected to each SRAM's address bits. The 8 bit data lines of each SRAM are connected to the D16 Data Bus , one SRAM connected to bits 8-15 and the other connected to bits 0-7.

There are buffers for all the signals on the VME bus. All the drivers and receivers are chosen from the VME bus specification's suggestion list.

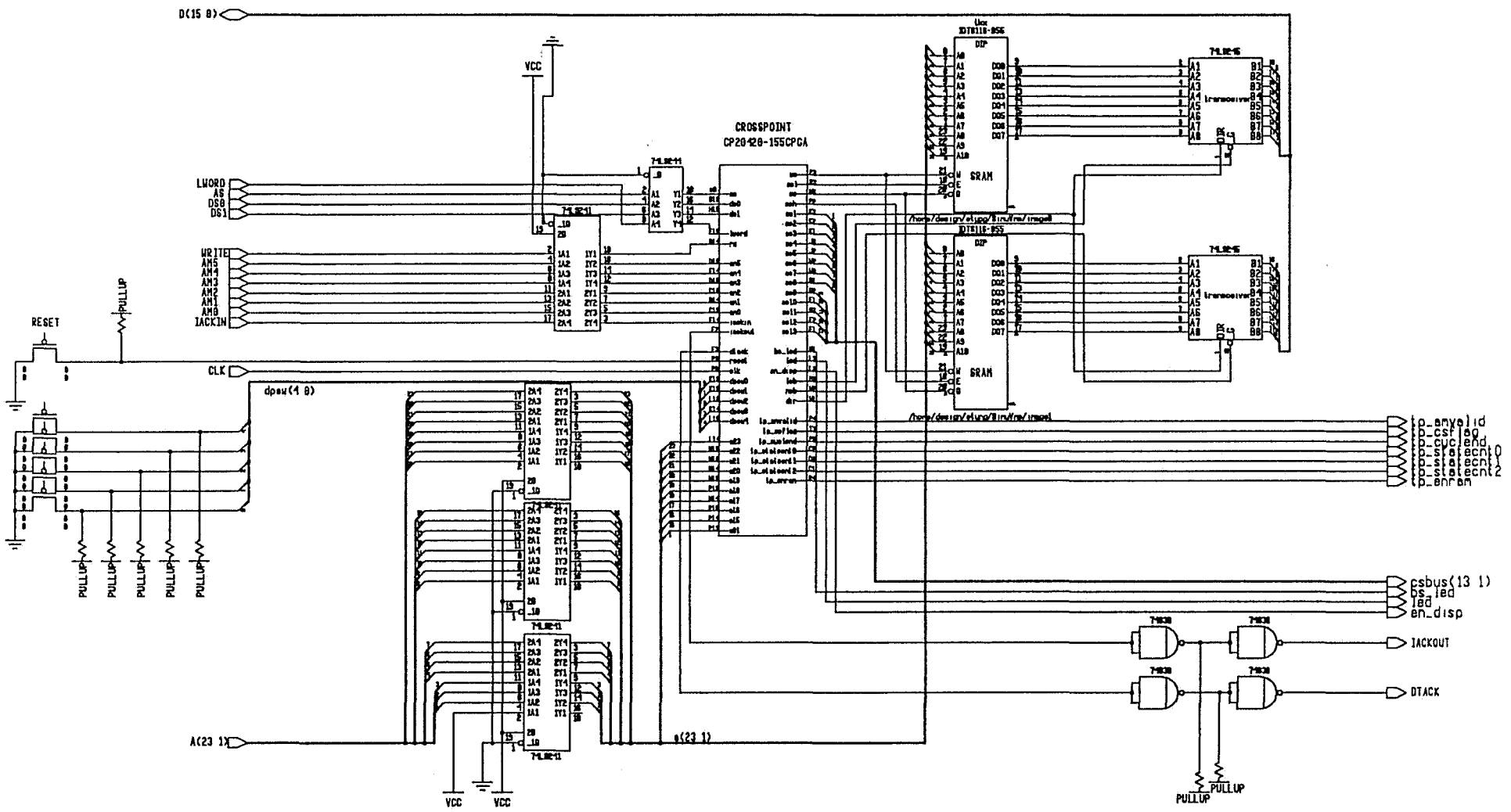


Figure 4. Connections between control chip and registers

74LS245:	transceiving the lines D00-15
74LS244:	driving or receiving the lines A01-A31, AM0-AM5, IACKIN, LWORD, WRITE
74S38:	driving the lines DTACK, IACKOUT

There is one RESET switch on the board's front panel, which will generate a reset signal(active low) when pressed. This resets the signals of the entire board. There is also a 5-bit dip switch on the board to select the board's base address.

A schematic of the hardware on board is shown in figure 4.

6. Using Software in the Design Process

VHDL implementation is used to model each functional block. Since it is much easier dealing with the logic of the functional block instead of the gate-level implementation, writing VHDL codes has saved a lot of time and effort. For example, the cs_ctl module is a circuit of over seventy gates but it only takes less than a hundred lines of VHDL codes to describe the logic of this module. After using the Mentor Graphics Corporation (MGC) QuicksimII tool to simulate the compiled VHDL code, the logic for that functional block is guaranteed to work, only leaving the timing aspect of the functional block to be inspected. The compiled VHDL codes are synthesized into the general library gates by MGC's Autologic tool and are then optimized using CrossPoint's technology file. All the timing and area requirements are taken care of in the process of optimization.

In order to check if the slave interface design is working precisely according to the VMEbus specification, Logic Modeling Corporation (LMC)'s VME SimuBus is used as a master model in the simulation to generate the bus cycle. Also, LMC's SRAM model is used as the memory device in the simulation. So before the design is turned into a CrossPoint FPGA chip, a comprehensive simulation has been done to ensure that it works.

The simulation diagram is shown in figure 5. Figure 6 shows the traces generated in the simulation and figure 7 is a closer view of the trace diagram.

7. Performance:

All single-byte accesses and double-byte accesses are tested to be fully functional. Using the internal clock at 20ns clock period for the READ/WRITE interface module, the time from when the first Data Strobe falls low to when the Data Acknowledgment falls low is about 90 ns. Since it takes 3 clock cycles in the READ/WRITE interface module, the time taken from when the master generates a bus cycle to when the slave finishes address decoding is only around 30 ns.

Having designed this basic interface, many other more complicated VME interfaces can be easily derived from it. Since VHDL is an industry standard and is technology independent, the design is portable to all systems and can be developed into a gate-level implementation using different technologies. Further enhancement can be easily made by slightly modifying the VHDL codes for the specific module.

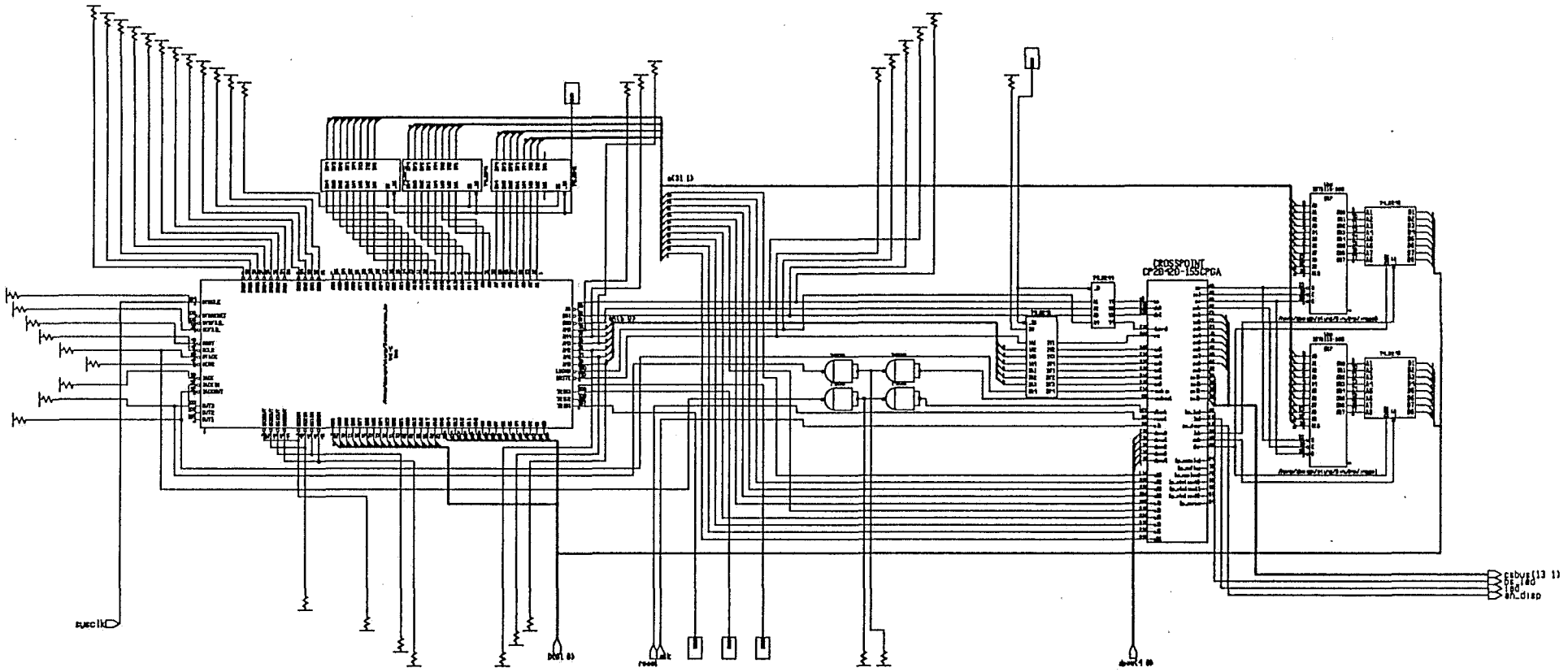


Figure 5. SIMI[illegible]ION DIAGRAM

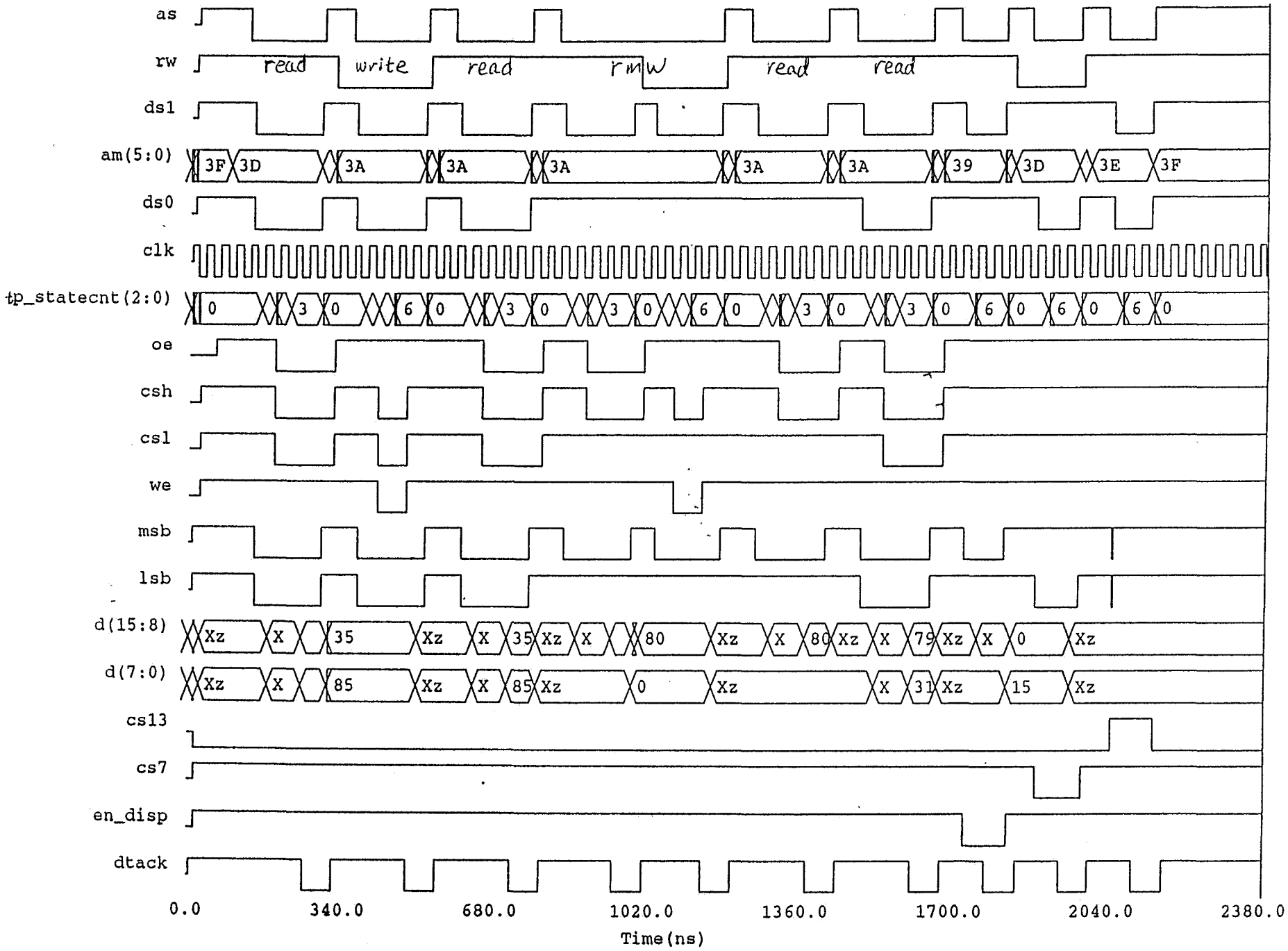
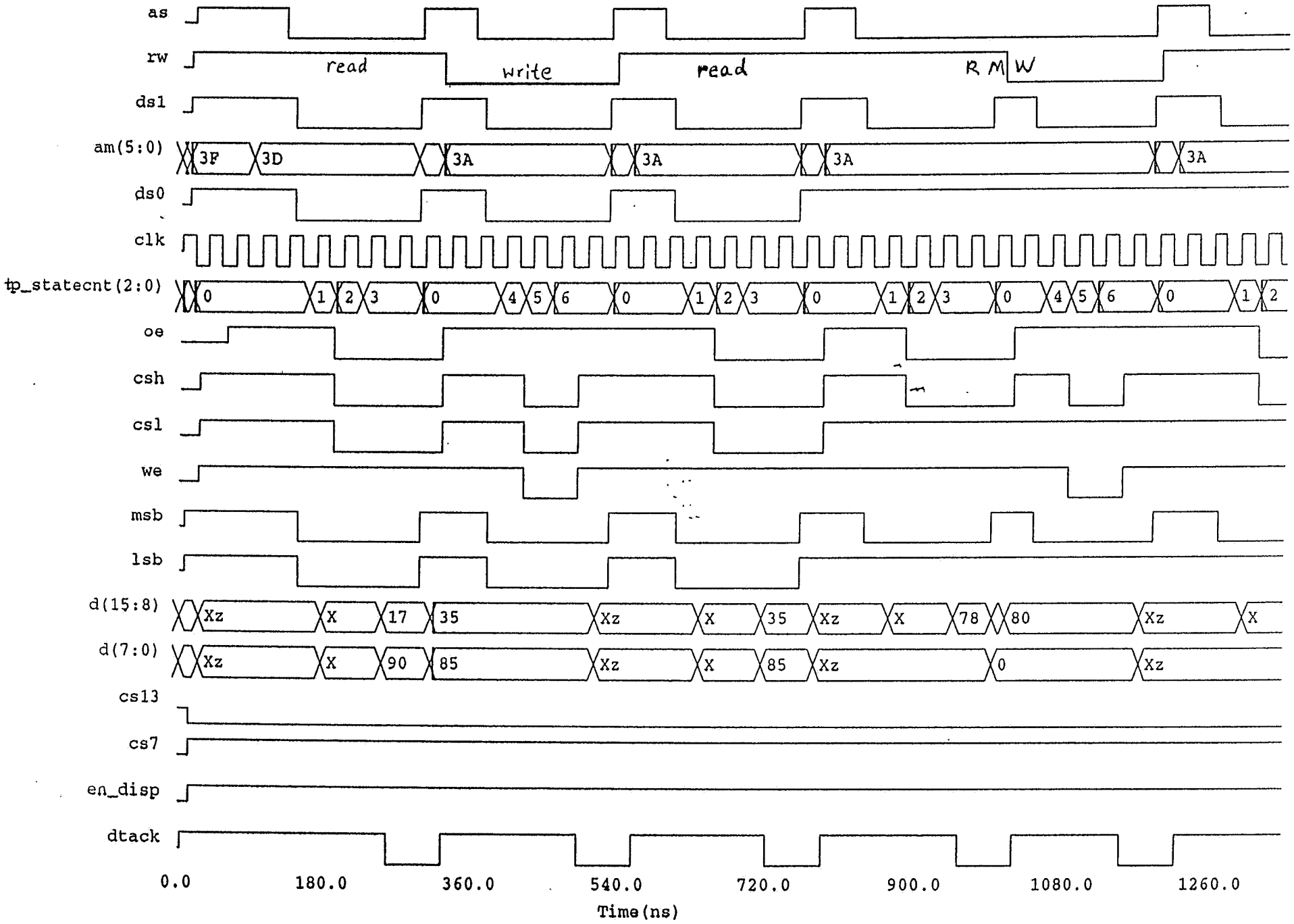

as
rw
read
write
read
rmw
read
read
ds1
am(5:0)
3F
3D
3A
3A
3A
3A
3A
39
3D
3E
3F
ds0
clk
tp_statecnt(2:0)
oe
csh
csl
we
msb
lsb
d(15:8)
d(7:0)
cs13
cs7
en_disp
dtack
0.0
340.0
680.0
1020.0
1360.0
1700.0
2040.0
2380.0
Time(ns)

Figure 6


as
rw
read
write
read
R M W
ds1
am(5:0)
3F
3D
3A
3A
3A
3A
ds0
clk
tp_statecnt(2:0)
0 1 2 3 0 4 5 6 0 1 2 3 0 1 2 3 0 4 5 6 0 1 2
oe
csh
csl
we
msb
lsb
d(15:8)
Xz X 17 35 Xz X 35 Xz X 78 80 Xz X
d(7:0)
Xz X 90 85 Xz X 85 Xz 0 Xz
cs13
cs7
en_disp
dtack
0.0
180.0
360.0
540.0
720.0
900.0
1080.0
1260.0
Time (ns)

Figure

```
                        cycl_end_tmp <= '0';
                WHEN s5 => we_tmp <= '0';        -- idle
                        cs <= '0';
                        oe_tmp <= '1';
                        cycl_end_tmp <= '0';
                WHEN s6 => we_tmp <= '1';        -- End of Write
                        cs <= '1';
                        oe_tmp <= '1';
                        cycl_end_tmp <= '1';
        END CASE;
    END PROCESS output_decode;

    output_cs: PROCESS (clk)
    BEGIN
        IF (clk'LAST_VALUE = '1' AND clk = '0') THEN
                csh <= cs OR msb;
                csl <= cs OR lsb;
                we <= we_tmp;
                oe <= oe_tmp;
                cycl_end <= cycl_end_tmp;
        END IF;
    END PROCESS output_cs;

END behave;
```

Acknowledgment

I would like to acknowledge in particular Stephen Wunduke of the SDC group, Lawrence Berkeley Laboratory, for his time, teaching, and advice. I would also like to thank other members of the SDC group in Lawrence Berkeley Laboratory, without whose help this project could not have been completed: Scott Dow, Richard Jared, Frederic Kral, Michael Levi, Robert Minor, Phil Smith. Finally, I would like to thank the University of California at Berkeley Engineering Cooperative Program for the opportunity to participate in the SDC project.

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 1993
-- Revision: May 14 1993
--
--
-- Module Name: am_dcd
--
--
-- Module Description: This module is an AM decoder.  It is used to check if the AM
--                     signals on the VME bus are set to do standard data transfer
--                     (excluding block transfer).  If AM signals are set correctly,
--                     signal "valid" will be set (active high); else "valid" will be
--                     unset.
--
--
-- Module Inputs:
--
--      am => AM(5 to 0) on Dbus
--      reset => active low, asynchronous reset signal
--      as => address strobe
--      iackin => interrupt acknowledgement in signal, will reset process
--
-- Module Outputs:
--
--      valid => indicate if AM signals are set to do standard data transfer(D24)
--      iackout => interrupt acknowledgement out signal
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY am_dcd IS
        PORT (  reset   : IN qsim_state;                        -- active low
                am      : IN qsim_state_vector(5 downto 0);
                as      : IN qsim_state;                        -- active low
                iackin  : IN qsim_state;                        -- active low
                iackout : OUT qsim_state;                       -- active low
                valid   : Out qsim_state);                      -- active high
END am_dcd;

ARCHITECTURE behave OF am_dcd IS
        SIGNAL mst_clr: qsim_state;
        SIGNAL valid1: qsim_state;
BEGIN
        iackout <= iackin;
        mst_clr <= iackin AND reset;  -- either iackin or reset will reset the system

        decode: PROCESS(mst_clr, as)
        BEGIN
                IF (mst_clr = '0') THEN   -- reset
                        valid1 <= '0';
                ELSIF (as'EVENT AND as = '0' AND as'LAST_VALUE = '1') THEN
                        CASE am IS
                                -- standard transfer
                                WHEN "111110" | "111101" | "111010" | "111001"=>
                                        valid1 <= '1';
                                WHEN OTHERS => valid1 <= '0';
                        END CASE;
                END IF;
        END PROCESS decode;

        -- use a 2nd process so that "AM" is only decoded on falling edge of "as" and
        -- valid will keep unset after disasserting reset, even though "as" is asserted
        decode2: PROCESS(as, valid1)
        BEGIN
                IF (as = '1') THEN
                        valid <= '0';
                ELSE
                        valid <= valid1;
                END IF;
        END PROCESS decode2;
END behave;
```

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 1993
-- Revision: May 14 1993
--
--
-- Module Name: dtack_ctl
--
--
-- Module Description: This module is to control the dtack(data transfer
--                     acknowledgement) signal on the VME bus.  "dtack" will be
--                     asserted or dissasserted depending on the "cycl_end" signal
--                     from mem_st_ctl module
--
--
-- Module Inputs:
--
--      reset => active low, asynchronous reset signal
--      cycl_end => active high, used to set dtack signal
--
-- Module Outputs:
--
--      dtack => active low
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY dtack_ctl IS
        PORT (reset     : IN qsim_state;    -- active low
              cycl_end  : IN qsim_state;    -- active high
              dtack     : OUT qsim_state);  -- active low
END dtack_ctl;

ARCHITECTURE behave OF dtack_ctl IS
BEGIN
        ack_proc: PROCESS(reset,cycl_end)
        BEGIN
                IF (reset = '0') THEN  -- reset asserted, or bs disserted
                        dtack <= '1';
                ELSE                          -- dtack disserted when cycl_end disserted
                        dtack <= NOT cycl_end;  -- dtack active low
                                                -- cycl_end active high
                END IF;
        END PROCESS ack_proc;
END behave;
```

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 1993
-- Revision: May 7 1993
--
--
-- Module Name: mem_st_ctl2
--
--
-- Module Description: This module is a Moore machine to control read/write from a
--                     memory device.  The states are based on the timing requirement
--                     of read/write of IDT SRAM 6116SA55.  The clock period is
--                     assumed to be 30 ns.
--
-- Module Inputs:
--
--      ds0, ds1 => data strobe 0, 1; active low
--      write => read/write; high--read; low--write
--      cs_ram => enable signal from the cs_ctl module indicating the master is
--                trying to do a r/w from the SRAM.  active low
--      cs_flag => signal from the cs_ctl module.  When it is set, it indicates
--                 that the master chose other chip select signals, not cs_ram.
--      clk => 30 ns period.  Used for state transistion in read/write operation
--      lsb => control signal for least significant byte
--      msb => control signal for most significant byte
-- Module Outputs:
--
--      csl => chip select of SRAM0, active low
--      csh => chip select of SRAM1, active low
--      oe => output enable of SRAM, active low
--      we => write enable of SRAM, active low
--      cycl_end => signal to dtack_ctl module to indicate cycle ends, active high
--
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY mem_st_ctl2 IS
        PORT (ds1      : IN qsim_state;      -- active low
              ds0      : IN qsim_state;      -- active low
              write    : IN qsim_state;      -- active low
              cs_ram   : IN qsim_state;      -- active low
              clk      : IN qsim_state;
              cs_flag  : IN qsim_state;      -- active high
              msb      : IN qsim_state;      -- active low
              lsb      : IN qsim_state;      -- active low
              csl      : OUT qsim_state;     -- active low
              csh      : OUT qsim_state;     -- active low
              oe       : OUT qsim_state;     -- active low
              we       : OUT qsim_state;     -- low: write; high: read
              cycl_end: OUT qsim_state);     -- active high
END mem_st_ctl2;

ARCHITECTURE behave OF mem_st_ctl2 IS
        TYPE states IS (s0, s1, s2, s3, s4, s5, s6);
        SIGNAL reset: qsim_state;
        SIGNAL present_state : states := s0;   -- reset state
        SIGNAL next_state: states;
        SIGNAL cs: qsim_state;          -- tmp chip select of SRAM
        SIGNAL we_tmp, oe_tmp, cycl_end_tmp: qsim_state;  -- tmp signals
BEGIN
        reset <= ds1 NAND ds0;  -- either one of the data strobe drops low will start
                                -- the operation; both of them have to go high to stop
                                -- the operation.

        synch: PROCESS (clk, reset)
        BEGIN
                IF (reset = '0') THEN
                        present_state <= s0;
                -- clock period assumed to be 30ns
                ELSIF (clk'LAST_VALUE = '1' AND clk'EVENT AND clk = '0') THEN
                -- assume clock or cs_ram are disabled once cycle ends
                        IF (cs_flag = '1') THEN       -- no need to invoke cycle, so
                                present_state <= s6; -- only set cycl_end
                        ELSIF (cs_ram = '0') THEN
                                present_state <= next_state;
                        ELSE
                                present_state <= s0;
                        END IF;
                END IF;
        END PROCESS synch;

        state_transitions: PROCESS (present_state, write)
        BEGIN
                CASE present_state IS
                        WHEN s0 => CASE write IS
                                        WHEN '1' => next_state <= s1;       -- read
                                        WHEN others => next_state <= s4;    -- write
                                   END CASE;
                        WHEN s1 => next_state <= s2;
                        WHEN s2 => next_state <= s3;
                        WHEN s3 => next_state <= s3;
                        WHEN s4 => next_state <= s5;
                        WHEN s5 => next_state <= s6;
                        WHEN s6 => next_state <= s6;
                END CASE;
        END PROCESS state_transitions;

        output_decode: PROCESS (present_state)
        BEGIN
                CASE present_state IS
                        WHEN s0 => we_tmp <= '1';              -- reset state
                                   oe_tmp <= '1';
                                   cs <= '1';
                                   cycl_end_tmp <= '0';

                        WHEN s1 => cs <= '0';      -- Tacs = 50ns, takes 2 clock cycles
                                   we_tmp <= '1'; -- Toe_tmp = 40 ns, takes 2 clock
                                   oe_tmp <= '0'; -- cycles
                                   cycl_end_tmp <= '0';
                        WHEN s2 => oe_tmp <= '0';        -- idle
                                   cs <= '0';
                                   we_tmp <= '1';
                                   cycl_end_tmp <= '0';
                        WHEN s3 => oe_tmp <= '0';        -- End of Read
                                   cs <= '0';
                                   we_tmp <= '1';
                                   cycl_end_tmp <= '1';

                        WHEN s4 => we_tmp <= '0';        -- Tcw = 40ns, chip select
                                   cs <= '0';            -- to End of Wri
                                   oe_tmp <= '1';
```

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 1993
-- Revision: May 10 1993
--
--
-- Module Name: brd_dcd
--
--
-- Module Description: This module is a board decoder.  It is used to check if the
--                     board addr a(23:19) is the same as the preset address on the
--                     dip switch.  "bs" (active low) will be set if they are the
--                     same, otherwise unset.
--
--
-- Module Inputs:
--
--      adr_in => bits 23-19 on the address bus
--      reset => asynchronous reset signal (active low)
--      as => address strobe (active low)
--      dpsw => dip switch to preset the board address
--      en => output from am_dcd module to validate a standard data transfer
--            (active high)
--
-- Module Outputs:
--
--      bs => set if it is the correct board address (active low)
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY brd_dcd IS
        PORT (reset:    IN qsim_state;                      -- active low
              as:       IN qsim_state;                      -- active low
              dpsw:     IN qsim_state_vector(4 downto 0);
              adr_in:   IN qsim_state_vector(4 downto 0);
              en:       IN qsim_state;                      -- active high
              bs:       OUT qsim_state);                    -- active low
END brd_dcd;

ARCHITECTURE behave OF brd_dcd IS
        SIGNAL bs1: qsim_state;
BEGIN

        decode: PROCESS(reset, as)
        BEGIN
                IF (reset = '0') THEN
                        bs1 <= '1';
                ELSIF (as'EVENT AND as'LAST_VALUE = '1' AND as = '0') THEN
                        IF (dpsw = adr_in) THEN
                                bs1 <= '0';                 --active low
                        ELSE
                                bs1 <= '1';
                        END IF;
                END IF;
        END PROCESS decode;

        -- use two processes so that the board address is only checked on falling
        -- edge of "as" and "bs" will keep unset after disasserting reset, even
        -- though "as" is asserted
        decode2: PROCESS(as, bs1, en)
        BEGIN
                IF (as = '1') THEN
                        bs <= '1';
                ELSIF (en = '1') THEN  -- make sure it is valid data transfer
                        bs <= bs1;
                ELSE
                        bs <= '1';
                END IF;
        END PROCESS decode2;

END behave;
```

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 1993
-- Revision: May 12 1993
--
--
-- Module Name: cs_ctl
--
--
-- Module Description: This module is a chip select controller.  It takes the address
--                     bits a(18:15) and enables a specific chip select signal
--                     according to the cs_table
--
-- Module Inputs:
--
--      adr_in => bits 18-15 on the address bus
--      reset => asynchronous reset signal (active low)
--      as => address strobe (active low)
--      en => output from bs_dcd module to validate board address (active low)
--
-- Module Outputs:
--      led => enable signal for a led
--      cs1-13 => 13 chip select signals
--      cs_flag => signal to flag if the enabled signal is not en_ram
--      en_ram => signal to enable an SRAM
--      en_disp => signal to enable a hex displayer
--
-- cs_table:
-- Bits:        adr(    18      17      16      15)     lsig bit#
-- ============================================================
-- \en_disp             1       1       1       1       15
-- \en_ram              1       1       1       0       14
-- cs13                 1       1       0       1       13
-- cs12                 1       1       0       0       12
-- cs11                 1       0       1       1       11
-- cs10                 1       0       1       0       10
-- cs9                  1       0       0       1       09
-- cs8                  1       0       0       0       08
-- \cs7                 0       1       1       1       07
-- \cs6                 0       1       1       0       06
-- \cs5                 0       1       0       1       05
-- \cs4                 0       1       0       0       04
-- \cs3                 0       0       1       1       03
-- \cs2                 0       0       1       0       02
-- \cs1                 0       0       0       1       01
-- \led                 0       0       0       0       00
-- ============================================================
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY cs_ctl IS
                PORT (  reset           : IN qsim_state;
                        adr_in          : IN qsim_state_vector(3 downto 0);
                        as              : IN qsim_state;
                        en              : IN qsim_state;                --active low
                        led             : OUT qsim_state;               --active low
                        cs1             : OUT qsim_state;               --active low
                        cs2             : OUT qsim_state;               --active low
                        cs3             : OUT qsim_state;               --active low
                        cs4             : OUT qsim_state;               --active low
                        cs5             : OUT qsim_state;               --active low
                        cs6             : OUT qsim_state;               --active low
                        cs7             : OUT qsim_state;               --active low
                        cs8             : OUT qsim_state;               --active high
                        cs9             : OUT qsim_state;               --active high
                        cs10            : OUT qsim_state;               --active high
                        cs11            : OUT qsim_state;               --active high
                        cs12            : OUT qsim_state;               --active high
                        cs13            : OUT qsim_state;               --active high
                        cs_flag         : OUT qsim_state;               --active high
                        en_ram          : OUT qsim_state;               --active low
                        en_disp         : OUT qsim_state);              --active low

END cs_ctl;

ARCHITECTURE behave OF cs_ctl IS
        signal lsig:    qsim_state_vector( 15 downto 0); -- grouping together the
                                                         -- 16 output control signals
        signal sig:     qsim_state_vector( 15 downto 0); -- final output of the 16
                                                         -- controls signals
BEGIN
        decode:PROCESS(reset, as)
        BEGIN
                IF (reset = '0') THEN
                        lsig <= "1100000011111111";
                ELSIF (as'EVENT AND as'LAST_VALUE = '1' AND as = '0') THEN
                        CASE adr_in IS
                                WHEN "0000" => lsig <= "1100000011111110";
                                WHEN "0001" => lsig <= "1100000011111101";
                                WHEN "0010" => lsig <= "1100000011111011";
                                WHEN "0011" => lsig <= "1100000011110111";
                                WHEN "0100" => lsig <= "1100000011101111";
                                WHEN "0101" => lsig <= "1100000011011111";
                                WHEN "0110" => lsig <= "1100000010111111";
                                WHEN "0111" => lsig <= "1100000001111111";
                                WHEN "1000" => lsig <= "1100000111111111";
                                WHEN "1001" => lsig <= "1100001011111111";
                                WHEN "1010" => lsig <= "1100010011111111";
                                WHEN "1011" => lsig <= "1100100011111111";
                                WHEN "1100" => lsig <= "1101000011111111";
                                WHEN "1101" => lsig <= "1110000011111111";
                                WHEN "1110" => lsig <= "1000000011111111";
                                WHEN others => lsig <= "0100000011111111";
                        END CASE;
                END IF;
        END PROCESS decode;

        decode2: PROCESS(as,en,lsig)
        BEGIN
                IF (as = '1') THEN
                        sig <= "1100000011111111";
                ELSIF (en = '0') THEN   -- active low
                        sig <= lsig;
                ELSE
                        sig <= "1100000011111111";
                END IF;
        END PROCESS decode2;

        led             <= sig(0);
        cs1             <= sig(1);
        cs2             <= sig(2);
```

```
        cs3             <= sig(3);
        cs4             <= sig(4);
        cs5             <= sig(5);
        cs6             <= sig(6);
        cs7             <= sig(7);
        cs8             <= sig(8);
        cs9             <= sig(9);
        cs10            <= sig(10);
        cs11            <= sig(11);
        cs12            <= sig(12);
        cs13            <= sig(13);
        en_ram          <= sig(14);
        en_disp         <= sig(15);

        cs_flag <= (NOT sig(15)) OR sig(13) OR sig(12) OR sig(11) OR
                   sig(10) OR sig(9) OR sig(8) OR (NOT sig(7)) OR (NOT sig(6)) OR
                   (NOT sig(5)) OR (NOT sig(4)) OR (NOT sig(3)) OR (NOT sig(2)) OR
                   (NOT sig(1)) OR (NOT sig(0));
END behave;
```

```
-- ********************************************************************************
--
-- Copyright University of California - Berkeley (LBL) 1993
-- All rights reserved
--
--
-- Author: Elise Tung
-- Date Created: April 24 1993
-- Revision: July 29 1993
--
--
-- Module Name: xvr_ctl
--
--
-- Module Description: This is a transceiver controller module.  Based on ds0, ds1,
--                     a01 and lword signals, the controller can tell if it is single
--                     byte access or double byte access.  Then the controller will
--                     enable the corresponding transceivers.
--
-- Module Inputs:
--
--      reset => global reset disables both transceivers
--      en => signal from brd_dcd module's board select (bs)
--      lword => long word signal from VME bus
--      ds0 => Data Strobe from VME bus
--      ds1 => Data Strobe from VME bus
--      a01 => address bit 1 on VME bus
--      rw => read/write signal, used to determine the direction of the transceivers
--
-- Module Outputs:
--
--      dir => direction of the transceivers
--      msb => enable signal for the most significant byte
--      lsb => enable signal for the least significant byte
-- ********************************************************************************

LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;

ENTITY xvr_ctl IS
        PORT(reset:     IN qsim_state;  -- active low
             en:        IN qsim_state;  -- connect to bs, active low
             lword:     IN qsim_state;
             ds0:       IN qsim_state;
             ds1:       IN qsim_state;
             a01:       IN qsim_state;
             rw:        IN qsim_state;  -- high: read; low: write
             dir:       OUT qsim_state; -- high: buffer A to B; low: buffer B to A
             msb:       OUT qsim_state; -- active low
             lsb:       OUT qsim_state);-- active low
END xvr_ctl;

ARCHITECTURE behave OF xvr_ctl IS
        signal ctl:     qsim_state_vector(3 downto 0);
BEGIN
        ctl <= (ds1, ds0, a01, lword);

        decode: PROCESS(reset, ctl, rw, en)
        BEGIN
                IF (reset = '0') THEN
                        msb <= '1';
                        lsb <= '1';
                ELSIF (en = '0') THEN
                        CASE ctl IS
                                WHEN "0101" | "0111" => msb <= '0';
                                                        lsb <= '1';
                                WHEN "1001" | "1011" => msb <= '1';
                                                        lsb <= '0';
                                WHEN "0001" | "0011" => msb <= '0';
                                                        lsb <= '0';
                                WHEN others => msb <= '1';
                                               lsb <= '1';
                        END CASE;

                        IF (rw = '1') THEN          -- read
                                dir <= '1';             -- buffer A to B
                        ELSE                        -- write
                                dir <= '0';             -- buffer B to A
                        END IF;
                ELSE
                        msb <= '1';
                        lsb <= '1';
                END IF;
        END PROCESS decode;
END behave;
```