AN1769/D
REV. 1.1
1/1999

PowerPC™

Application Note

Designing a Minimal PowerPC™ System

Gary Milliorn
Motorola RISC Applications
risc10@email.sps.mot.com

This application note describes how to design a small, high-speed Motorola PowerPC-processor based system. In this document, the terms ‘60x’ and ‘7xx’ are used to denote a 32-bit microprocessor from the PowerPC™ architecture family that conforms to the bus interface of the PowerPC 603e™, PowerPC 604e™, or PowerPC 750™ microprocessors, respectively. MPC60x and MPC7xx processors implement the PowerPC architecture as it is specified for 32-bit addressing, which provides 32-bit effective (logical) addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits (single-precision and double-precision).

This document contains the following topics:

MOTOROLA

To locate any published errata or updates for this document, refer to the website at http://www.mot.com/SPS/PowerPC/.

Part 1 Introduction

To keep the design simple, only the most basic features necessary to run a debugger program are included. These features are as follows:

- A PowerPC processor (this includes the MPC603e, MPC603ev, MPC604, MPE603e, MPE603ev, MPE604, MPC740 and MPC750)
- Flash ROM storage (start-up code)
- Read/write memory (downloaded code, program variables)
- Serial I/O channel (communication)
- Memory and I/O controller
- Power, clocks and reset

While this application note is general in focus, it will also occasionally diverge in order to describe the implementation details of an actual board, known as “Excimer,” which implements the basic techniques described in this application note. The details of Excimer provide a base upon which you can build a design, with the general sections describing ways to support customization.

1.1 Design Philosophy

The PowerPC high-performance family (MPC60x and MPC7xx) bus interface may at first appear intimidating due to the presence of split address/data bus tenuring, bus snooping, multiprocessing support, cache coherency support, and other advanced features. Such features can be used to obtain additional performance for high-performance systems, but for the purposes of a small, high-speed embedded controller (particularly one with only one bus master), many of these complications can be avoided.

Since the processor does not contain an internal memory controller or I/O interface, that role has traditionally fallen to the Motorola MPC106 memory/PCI/cache controller. For a small board such as outlined here, the MPC106 is much more than is minimally needed. Indeed, complexity can sometimes reduce performance. Cache coherency instructions use valuable bus cycles, and allowing access by external masters (such as cache) requires delaying memory cycles in case the external device claims the cycle.

Instead, for this design, a programmable ASIC is used to provide the necessary controls for a block of RAM, ROM and access to I/O. The controller is not programmable by software but is instead pre-configured in hardware, and memory access cycles are tuned to provide only the necessary signals.

With these restrictions and goals, the typical block diagram may resemble that shown in Figure 1.

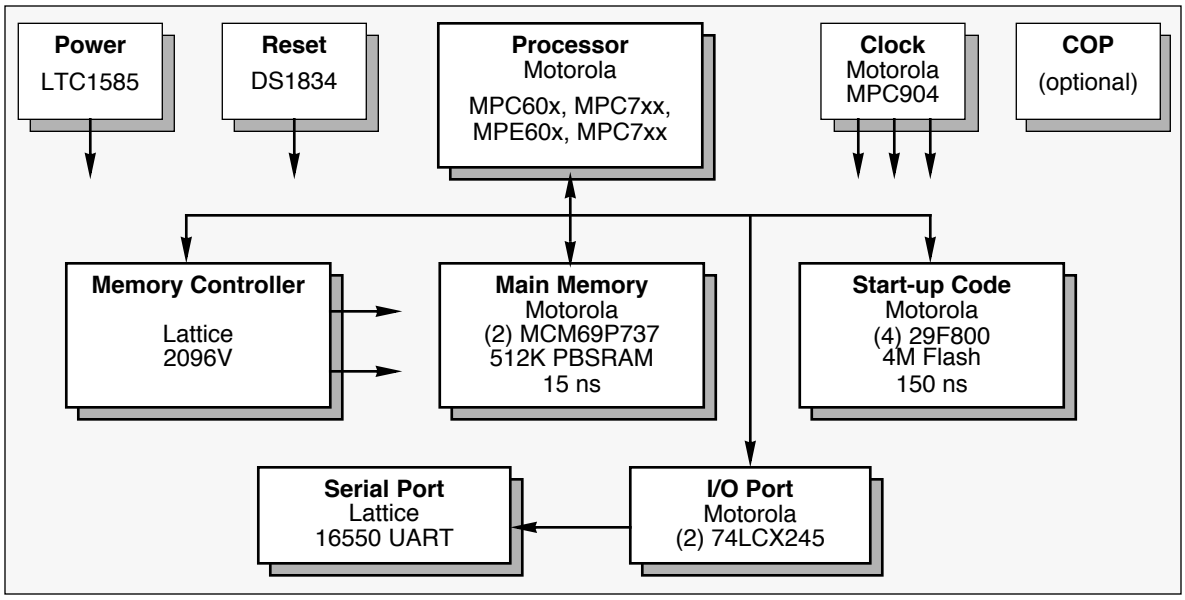


Figure 1. Typical Minimal System Block Diagram

1.2 Conventions

Various conventions used in this document are as follows:

SIGNAL	Active-high external signal (pin).
$\overline{\text{SIGNAL}}$	Active-low external signal (pin).
signal	Active-high internal signal (net); used when describing the memory controller.
signal_L	Active-low internal signal (net); used when describing the memory controller ('_L' indicates active low for internal signals).
name()	A block of HDL code implementing a function.

Occasionally, a name may have both forms; for example, the **$\overline{\text{TS}}$** hardware signal may be detected in a fragment of HDL code which refers to it as "ts_L."

Part 2 Processor Design

The processor may be any member of the MPC60x or MPC7xx family. All such devices offer 64-bit bus modes, and the MPC603x parts also offer a 32-bit bus interface which can make the memory design even simpler at a cost of speed. Since only the MPC603x parts support this mode, it will not be used in this design but it should be kept in mind where the number of parts or cost is even more important than speed.

Since all high-performance PowerPC processors use very similar bus interfaces, the choice of processor may be based on cost and performance issues and not on the interface costs. For a simple system with only one bus master, many signals on the processor bus may be ignored or wired to the desired state; refer to Table 1 for more information.

Table 1. PowerPC Bus Signal Connections[1]

Signal	Treatment
AP(0–3), $\overline{\text{APE}}$, $\overline{\text{BR}}$, $\overline{\text{CKSTP\_OUT}}$, $\overline{\text{CI}}$, CLKOUT, CSE(0–1), DP(0–7), $\overline{\text{DPE}}$, HALTED, $\overline{\text{QREQ}}$, $\overline{\text{RSRV}}$, TC(0–2), TMS, TDI, TDO, VOLTDETGND, $\overline{\text{WT}}$	Unused, leave unconnected.
$\overline{\text{CKSTP\_IN}}$, $\overline{\text{DBWO}}$, $\overline{\text{DBDIS}}$, DRVMOD1, RUN, $\overline{\text{SHD}}$, $\overline{\text{SRESET}}$, TBEN, $\overline{\text{TLBISYNC}}$, $\overline{\text{XATS}}$	Unused, pullup or connect to VDD (+3.3V).
L1_TSTCLK, $\overline{\text{LSSD\_MODE}}$	Unused, connect directly to VDD (+3.3V).
$\overline{\text{ABB}}$, $\overline{\text{ARTRY}}$, $\overline{\text{DBB}}$, $\overline{\text{GBL}}$, L2_TSTCLK,TCK	Unused, connect to 10K pullup to VDD (+3.3V).
$\overline{\text{BG}}$, $\overline{\text{DBG}}$, DRVMOD0, L2_INT	Connect to ground.
$\overline{\text{DRTRY}}$	Connect to $\overline{\text{HRESET}}$
$\overline{\text{QACK}}$	Connect to 1K pulldown to ground.
PLL_CFG(0–3)	Connect to VDD (+3.3V) or ground to configure CPU speed.
$\overline{\text{INT}}$, $\overline{\text{MCP}}$, $\overline{\text{SMI}}$	Connect to VDD pullup (+3.3V) and/or interrupt controller.
VDD, OVDD	Connect to appropriate voltage level.
AVDD, L2AVDD	Connect to PLL filter as shown in hardware reference manual.
$\overline{\text{HRESET}}$[2]	Connect to reset controller.
TRST[2]	Connect to reset controller, or connect to 1K pulldown (GND).
SYSCLK[2]	Connect to clock generator.
A(0–31), $\overline{\text{AACK}}$, $\overline{\text{TA}}$, $\overline{\text{TEA}}$, $\overline{\text{TBST}}$, $\overline{\text{TS}}$, TSIZ(0–2), TT(0–4)[2]	Connect to memory controller.
DH(0–31), DL(0–31)[2]	Connect to memory devices.

[1] This table combines MPC603x, MPC604x, and MPC750 processors. Not all of these signals are present on every device.

[2] These signals are the only ones that need consideration in a minimal system design. All others are assigned fixed values and can be safely ignored thereafter.

The MPC750 has additional signals for interfacing with a back-side L2 cache (**L2ADDR[0–16]**, **L2DATA[0–63]**, **L2DP[0–7]**, **$\overline{\text{L2CE}}$**, **$\overline{\text{L2WE}}$**, **L2CLK_OUTA**, **L2CLK_OUTB**, **L2SYNC_OUT**, **L2SYNC_IN**, and **L2ZZ**). Because the L2 interface is completely separate from the system bus, it does not affect the design of a minimal system in any way. Refer to the *MPC750 RISC Microprocessor Hardware Specifications*, or the *MPC750 Processor/Cache Module Hardware Manual*, for further details.

Part 3 Memory System Design

The one fairly complicated portion of a minimal system is the interface to the processor data bus. Unlike CISC processors, RISC processors do not typically perform data (re)alignment, so the data from each external device must be placed on the proper data lane. To attempt to use an 8-bit memory device to supply instructions or data to a 64-bit data bus, 8-bidirectional latching transceivers must be used to move the byte to the correct byte lane on the 64-bit bus, as shown in Figure 2.

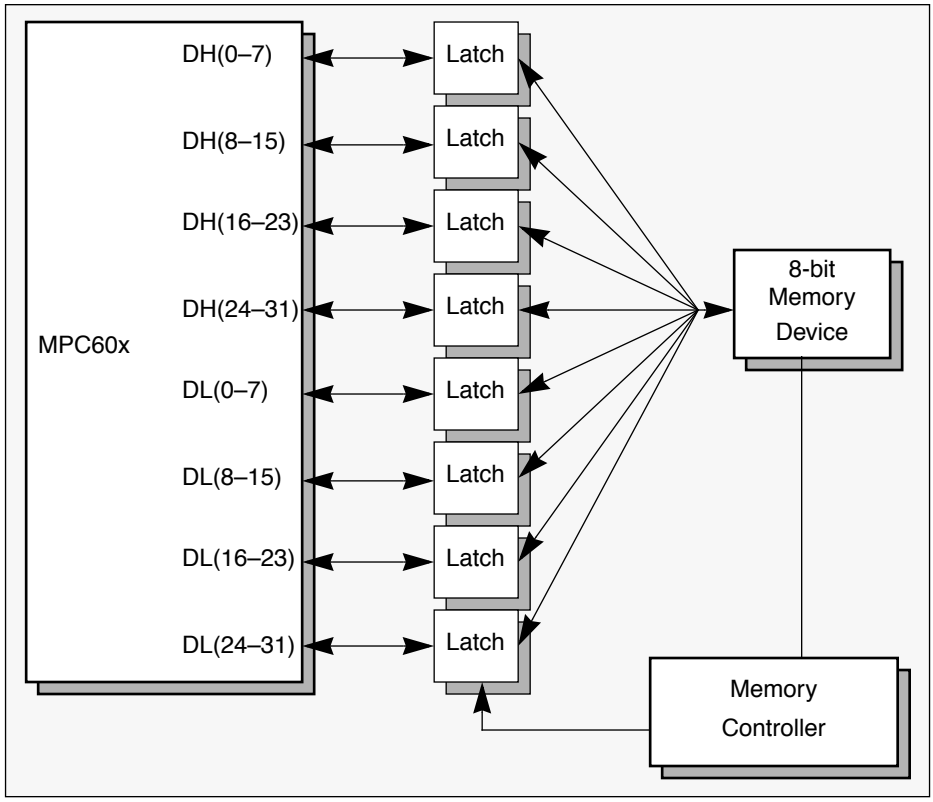


Figure 2. Byte Lane Redirection

Because the processor expects from one to eight bytes on each (non-burst) transfer, the memory controller must generate from one to eight memory cycles to the 8-bit memory by generating the address(es), latching the resulting data, and then presenting it to the processor with the $\overline{\text{TA}}$ signal. This process must be reversed when writing to memory. This can take significant amounts of logic, and is the approach taken by the MPC106 ROM interface, for example.

For this minimal system, we will instead take the approach that all memory is 64-bits wide. By using 32-bit pipelined-burst SRAM for the main memory and 16-bit Flash EPROM for start-up code, only 6 components will be needed, the controlling logic will be simple and inexpensive, and as a bonus the SRAM will allow very fast memory access speeds. The use of SRAM for main memory has become more attractive as speed and size increases and price falls. Currently SRAM devices at 66 MHz are commodity components due to their use as L2 caches in PCs; even 100-MHz parts are not terribly expensive. The minimal system block diagram is shown in Figure 3.

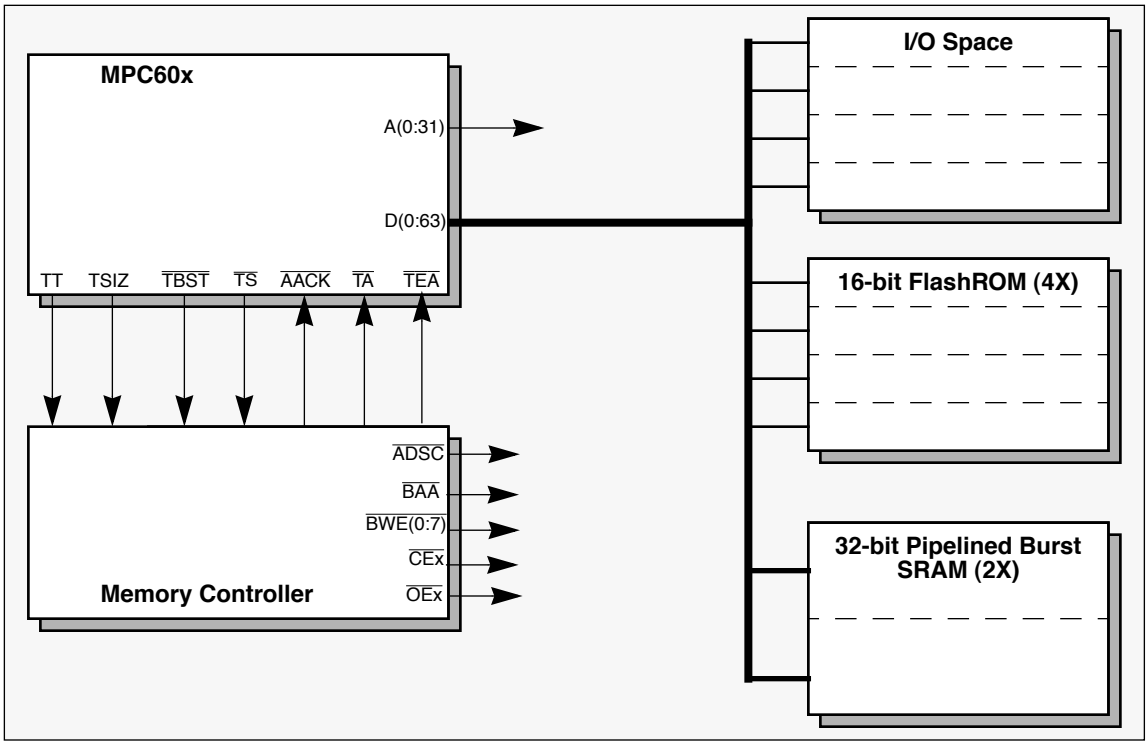


Figure 3. Minimal System Memory Architecture

The system address map is shown in Table 2.

Table 2. Excimer Address Map

Devices	Address		Burstable?	Access Cycles
	Start	End		
RAM	0x0000_0000	0x3FFF_FFFF	Y	3-1-1-1
Fast I/O	0x4000_0000	0x7FFF_FFFF	N	4
Slow I/O	0x8000_0000	0xBFFF_FFFF	N	12
Flash	0xC000_0000	0xFFFF_FFFF	N	6

The only challenging design problem faced is the handling of burst transfers. The MPC60x family can operate with caches disabled, thus preventing burst transfers, but this typically exacts a terrible penalty in performance that makes the additional effort at handling them well worthwhile. The first step in designing the memory controller (abbreviated MC in code) is to determine the types of controls that will be needed among the proposed memory devices—Flash EPROMs, SRAM and general I/O.

3.1 SRAM Memory Controls

The SRAM interface is centered around the controls necessary for a typical pipelined burst SRAM memory, as used on the MPC750 back-side cache or various other PC system's L2 cards. Flow-through SRAM memories could also be used, but the timing for write operations would change. Since this is a simple memory controller, it will be architected for only one type of SRAM. Most SRAM devices have numerous controls which are not needed, leaving us with the following:

$\overline{\text{A(n–0)}}$	Memory address, including LSB for burst transfers and critical-word first. Addresses 0 and 1 are the LSBs and are used for burst transfer addresses.
$\overline{\text{ADSC}}$	Latches address for single-beat or burst transfers
$\overline{\text{ADV}}$	Increments address for burst transfers
$\overline{\text{BWE(a-d)}}$	Active-low byte-write enables; if not asserted, the cycle is a burst read.
$\overline{\text{G}}$	Active-low output enable; asserted for all read operations.
$\overline{\text{SE1}}$	Active-low chip enable; asserted for all operations.

The memory controller must generate these signals for all SRAM transfers, whether single-beat or burst transfers. **$\overline{\text{ADSC}}$** and **$\overline{\text{SE1}}$** start the cycle by latching the address into the SRAM; these must be provided by the memory controller at the same time. Since **$\overline{\text{SE1}}$** is asserted one clock after **$\overline{\text{TS}}$** if the address matches an SRAM space, the memory controller also asserts **$\overline{\text{ADSC}}$** for memory cycles. The **$\overline{\text{BWE}}$** signals corresponding to the size of the transfer must be asserted if the cycle is a write cycle; otherwise, **$\overline{\text{G}}$** must be asserted to read in data (all byte lanes are driven and the processor selects the data from whichever byte lane is needed).

The remaining signal is **$\overline{\text{ADV}}$**, which must be asserted for three clock cycles if a burst transfer is selected; otherwise, it remains high. Although the data rate could be throttled with **$\overline{\text{ADV}}$** or **$\overline{\text{G}}$**, this is not necessary for the processor, so, to simplify the design, only fast SRAMs will be accommodated.

The remaining portion of the SRAM controller to specify is the initial access time. Most SRAMs available today can decode an address within 10 ns from the address strobe (**$\overline{\text{ADSC}}$**), so there is no need to delay before beginning a transfer.

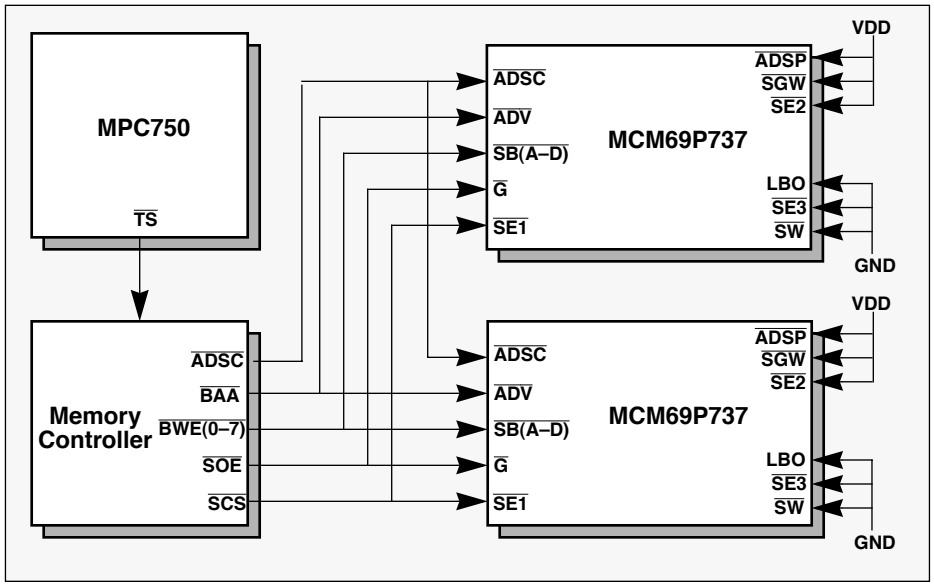


Figure 4. Pipelined Burst SRAM Memory Connections

Note that since we do not allow overlapped address and data tenures, we do not know whether the next transfer is to the same SRAM page or not, so we cannot stream data (that is, 3-1-1-1/1-1-1-1/... cycles). This requires much more logic and is left as an exercise for the reader.

A final issue which must be handled is terminating an access. Burst SRAM's operate by streaming data into or out of the chip on each clock edge after an initial setup sequence (**ADSC**), until instructed to stop. While read operations can be ignored by forcing **G** high, write operations cannot be similarly controlled, so instead a "de-select" cycle must be performed after each access. This is done by asserting **ADSC** without no chip select asserted; when deselected, the SRAM will stop reading or writing data.

3.2 Flash Memory Controls

Flash memory devices use traditional **OE**, **CS** and **WE** signals to perform single-beat read and write cycles (burst transfers are not permitted[1]), whether the data width of the device is 8-bits or 16-bits. Since the PowerPC bus does not care if data is placed on ignored byte lanes during read cycles, it will be acceptable to use **OE** in common for all flash ROMs.

Write cycles require more care. Requiring the processor to perform 64-bit writes is unacceptable because it is difficult (that is, requires the floating-point unit on the MPC devices) or impossible (on the non-floating-point MPE devices) to do a 64-bit single-beat bus transfer. Thus, flash devices must be fully-qualified with byte-enables during write cycles.

Using standard flash devices will require the following control signals:

BWE(0–7)	Active-low byte-write enables; if not asserted, the cycle is a read.
FOE	Active-low output enable; asserted for all read operations.
FCS	Active-low chip enable; asserted for all operations.

This gives a flash memory architecture as shown in Figure 5.

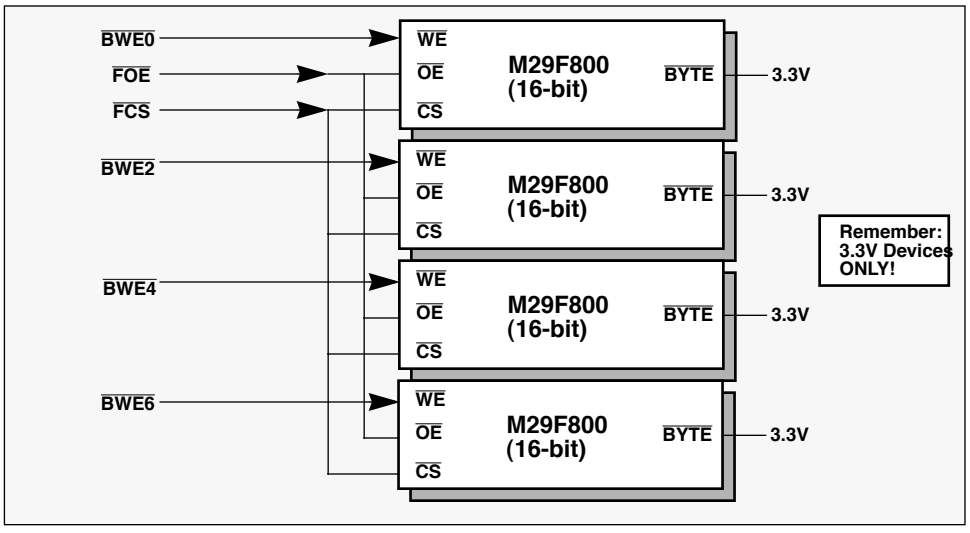


Figure 5. Flash Memory Connections

[1]This implies that the ROM space is non-cacheable; since Flash ROM is so much slower than SRAM, critical code should be copied to SRAM, so this may not be considered a big performance limitation.

Note that 16-bit devices have been used. This helps reduce the number of components, at a cost of restricting writes to 16, 32 or 64 bits in size. If 8-bit writes are required, then either 8 8-bit devices must be used, or devices which have multiple byte enables.

A further restriction on flash memory is that reading is slow (from 60 to 200 ns), and writing is even slower (as much as 15 ms). The memory controller delays the assertion of **$\overline{\text{TA}}$** for a fixed number of cycles on any access to match the read time; this handles the read access properly and gives sufficient time for the flash device to begin the program operation (the data does not need to be held throughout a write cycle).

Software must insure that a proper amount of time has elapsed after a write before another read or write occurs. This can be done with a simple timing loop, using I/O to check the **RDY/$\overline{\text{BSY}}$** signals, or the use of flash devices which can be queried by reading special addresses.

3.3 I/O Controls

Most simple I/O devices such as real-time clocks, serial ports, and other unique interfaces have fairly simple I/O controls—a chip select, an output enable, and a write control. The I/O controller can then be modeled very closely on the flash ROM controller; both have simple controls and both are relatively slow.

One difference between flash ROM and I/O is that most I/O devices are 8 or perhaps 16 bits, not 64, so I/O devices must be attached to particular byte lanes. The I/O controller responds to any size write, so data may be placed on any byte lane (software is responsible for positioning and retrieving the data properly). A fairly easy modification to the controller allows different I/O times for each address decoded, allowing fast and slow I/O devices to be mixed.

Note that the write strobes/direction control are the same byte write enables that have been described before. This reuse will allow a reduction of the size and complexity of the controller, but it also means that the software must generate the correct address when performing writes to I/O devices greater than 8 bits wide. For example, in Figure 6, a 16-bit I/O device is attached to D(0–15) and uses **$\overline{\text{BWE1}}$**, so to access the controller, software must issue 16- or 32-bit writes aligned with D0. The controller will assert **$\overline{\text{BWE1}}$** (all others are ignored).

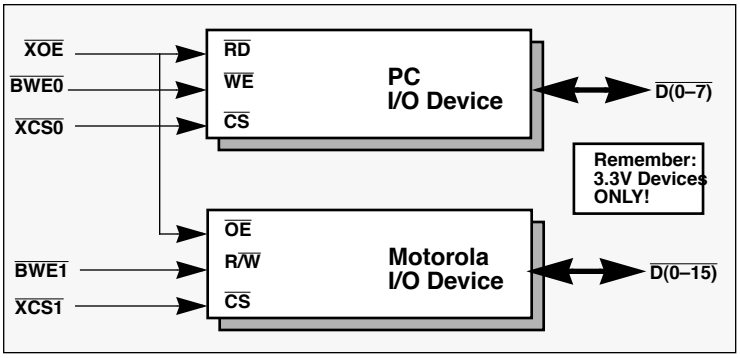


Figure 6. I/O Connections

The I/O controller supports both Motorola and PC control signals. In addition, these devices attach to the 3.3-V PowerPC data bus, so they must not drive over 3.3V. An easy solution to this is to add a 3.3-V buffer between the high-speed memory path and the I/O devices, which has a side benefit of allowing faster memory operation due to reduced capacitive loading.

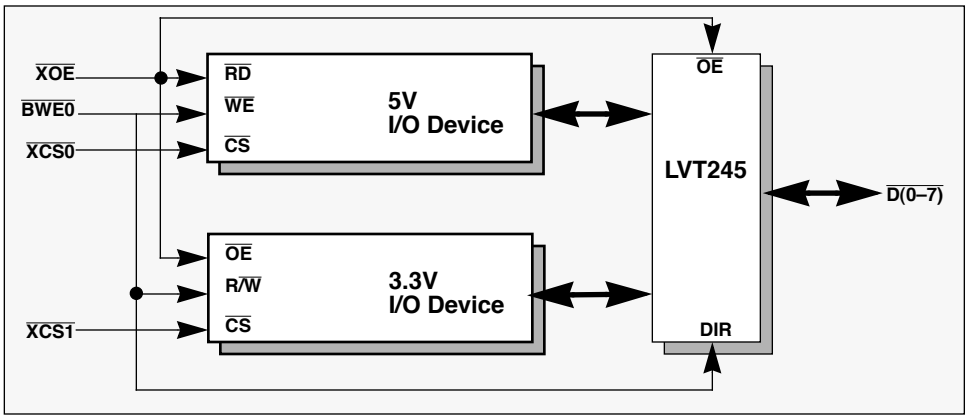


Figure 7. Buffered I/O Connections

3.4 Collected Controls

The previous sections have provided a general overview of the memory controller; this section provides the details. Table 1 shows most of the signals that are directly connected to the memory or I/O or are wired to some particular state. The controls needed for burst SRAM and flash ROM share common byte-write enables; the memory controller signals are listed in Table 3.

Table 3. Memory Controller Signal Handling

Signal	Treatment	Applies To
$\overline{\text{TS}}$	Examined for start of a cycle	All
TT(0–4), TSIZ(0–2), $\overline{\text{TBST}}$	Examined for type of cycle	All
A(0–1)	Examined for cycle destination (RAM, ROM, I/O)	All
A(29–31)	Examined for byte lane enables and burst transfer	All
$\overline{\text{AACK}}$	Asserted on final memory transfer	All
$\overline{\text{TA}}$	Asserted per-beat on each memory transfer	All
$\overline{\text{TEA}}$	Asserted on each unsupported memory transfer	All
$\overline{\text{BWE}}$(0–7)	Asserted on writes on individual byte lane(s)	SRAM, ROM
$\overline{\text{SCS}}$	Asserted on all SRAM accesses	SRAM
$\overline{\text{SOE}}$	Asserted on all SRAM read accesses	SRAM
$\overline{\text{ADSC}}$	Asserted on all burst SRAM accesses before the first cycle	SRAM
$\overline{\text{BAA}}$	Asserted on all burst SRAM accesses during cycles 2-4	SRAM
$\overline{\text{FCS}}$	Asserted on all Flash accesses	ROM
$\overline{\text{FOE}}$	Asserted on all Flash read accesses	ROM
$\overline{\text{XCS(0–1)}}$	Asserted on all I/O accesses	I/O
$\overline{\text{XOE}}$	Asserted on all I/O read accesses	I/O

All other signals are either wired to the necessary state or are unused as described in Table 1. For example, since the PowerPC bus is parked permanently, detecting **BG**, **ABB**, **DBG** and **DBB** are unnecessary. The memory controller interface then requires a total of 35 I/O signals, well within the capacity of any modern FPGA, leaving lots of I/O for additional functions.

3.5 Memory Controller Details

The remainder of Part 3, "Memory System Design," describes the internal operations of the memory controller as used on the Excimer reference board. The code is based upon synthesizable VHDL code, but could be easily adapted to Verilog, and any of the several IC-specific HDL variants that exist for Actel, Altera, Lattice, Xilinx et. al.

Figure 8 shows the internal architecture of the memory controller module.

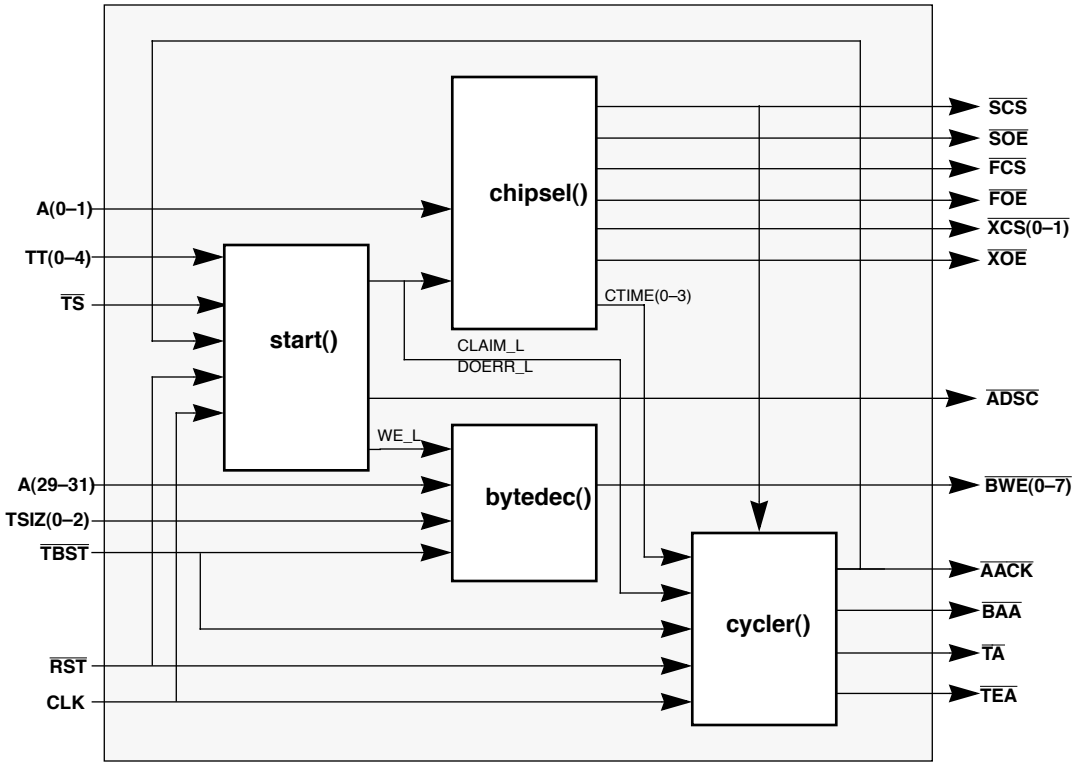


Figure 8. Memory Controller Architecture

3.5.1 Start Detection Module

Upon receiving a **TS**, the memory controller must examine the **TT(0–4)** signals to determine the type of cycle that will be performed. Of the 32 possible permutations, only those found in Table 4 are of interest:

Table 4. TT Encoding

TT0	TT1	TT2	TT3	TT4	Transaction	Memory Controller Action
0	0	0	1	0	Write-with-flush	Single-beat or burst write
0	0	1	1	0	Write-with-kill	Burst write
0	1	0	1	0	Read	Single-beat or burst read
0	1	1	1	0	Read-with-intent-to-modify	Burst read

All remaining **TT** codes are either address-only cycles (which are not needed), are caused by instructions not needed in single-processor environments (for example, the **eciwx**, **ecowx**, **dcbz**, **lwarx**, and **stcwx** instructions), or are reserved values. These simplifications are possible because there is no need to snoop the processor bus to maintain cache coherency.

While software should not generate such cycles, it is not reliable for a memory controller to simply ignore them. The memory controller, as the sole target of bus transactions, must terminate unacceptable bus cycles with **$\overline{\text{TEA}}$**; otherwise, the processor will wait forever for the (ignored) cycle to complete.

When any transfer begins, the "start()" module must either assert the "claim_l" or "doerr_l" signal to cause the appropriate actions to conclude the transfer cycle (which is handled in the bus state machine "cycler()"). The general architecture of "start()" is shown in Figure 9.

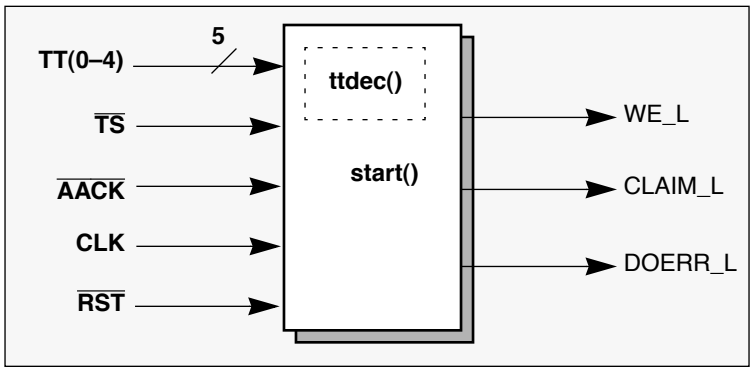


Figure 9. Start Detector Module

The **TT(0–4)** signals do not change during the address tenure, whether burst or single-beat, so the outputs remain valid until the memory controller asserts **$\overline{\text{AACK}}$**. The "start()" module provides the global CLAIM_L signal, used by other modules to detect whether a cycle is in-progress, or the DOERR_L signal, used to terminate unclaimed cycles, and a write signal (WE_L) to determine that the cycle is a write cycle. These signals are used exclusively by other modules, and remain valid until the memory controller completes the cycle by asserting **$\overline{\text{AACK}}$**.

The VHDL code for this module is:

```
-------------------------------------------------------------------------------------------
-- TTDEC.VHD
--
-- TTDEC() monitors the TT bus and determines whether the TT is of interest to
-- the MC or not.  If so, a signal is provided for start, and tt_we_L reflects
-- the read/write status.
--
-- NOTE: TTDEC must not be optimized or errors will occur when hierarchical
--       optimization is performed (TT1 and TT2 will be optimized away, making it
--       impossible to connect TTDEC to MC-- this is a bug in ViewSynthesis).
--       Recommended procedure is to dissolve TTDEC into it's parent level MC
```

```
--       before optimization.  ViewSynthesis doesn't seem to care about the NC
--       input pins at that level.
--
-- Copyright 1998, by Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.1
-- Date:     6/10/98
-- Notes:
--           All logic is active low when appended with a "_L".
--           Passed speedwave check 6/16/98.
-------------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


-------------------------------------------------------------------------------------------------
-- TTDEC
-------------------------------------------------------------------------------------------------
ENTITY TTDEC is
    PORT( tt                : in     std_logic_vector( 0 to 4 );   -- current transfer type.
          tt_take           : buffer std_logic;                    -- asserted when TT matches types.
          tt_we_L           : buffer std_logic;                    -- asserted when cycle is write.
          monitor           : buffer std_logic                     -- ViewSynthesis bug -- not useful.
        );

end; --PORT DEFINITION AND ENTITY


-------------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF TTDEC is

SIGNAL wflush, wkill, read, rwim         : std_logic;

BEGIN

    -- Detect only the following TT types.  "tt_take" will be asserted for all cycles we will claim.
    --
    wflush  <= '1' WHEN (tt(0) = '0' and tt(1) = '0' and tt(2) = '0' and tt(3) = '1' and tt(4) = '0')
            ELSE '0';
    wkill   <= '1' WHEN (tt(0) = '0' and tt(1) = '0' and tt(2) = '1' and tt(3) = '1' and tt(4) = '0')
            ELSE '0';
    read    <= '1' WHEN (tt(0) = '0' and tt(1) = '1' and tt(2) = '0' and tt(3) = '1' and tt(4) = '0')
            ELSE '0';
    rwim    <= '1' WHEN (tt(0) = '0' and tt(1) = '1' and tt(2) = '1' and tt(3) = '1' and tt(4) = '0')
            ELSE '0';

    tt_take  <= (wflush  or  wkill  or  read  or  rwim);
    tt_we_L  <= not (wflush  or  wkill);


    -- Needed due to ViewSynthesis bug: optimizes tt1 and tt2 away, then complains about their absence.
    monitor  <= read;

END BEHAVIOR;

-------------------------------------------------------------------------------------------------
-- START.VHD
--
-- START() is the portion of the memory controller which decodes incoming
--   transfers and decides whether they should be claimed by the controller
--   or terminated with an error condition.
--
-- Copyright 1998, by Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.3
-- Date:     9/23/98
-- Notes:
--           All logic is active low when appended with a "_L".
--           Passed speedwave check 6/16/98.
--           Moved ADSC* assertion to state machine.
-------------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


--=============================================================================================
-- START
---------------------------------------------------------------------------------------------
ENTITY START is
    PORT( tt_take            : in     std_logic;          -- asserted if good TT selection.
          tt_we_L            : in     std_logic;          -- asserted if good TT is write.
          ts_L               : in     std_logic;          -- transfer start strobe.
          aack_L             : in     std_logic;          -- asserted on transfer complete.
          clk                : in     std_logic;          -- bus clock.
          rst_L              : in     std_logic;          -- system reset.
          claim_L            : buffer std_logic;          -- asserted when cycle is claimed.
          doerr_L            : buffer std_logic;          -- asserted when cycle not claimed.
          we_L               : buffer std_logic          -- byte lane write selects.
        );

end; --PORT DEFINITION AND ENTITY


---------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF START is

BEGIN

   -- Derive a D flop to maintain selected status.  The register must be globally clocked to fit
   -- well in the Lattice 2xxx FPGA architecture, where clocks and resets are global (or expensive).

   monitor : PROCESS( clk, rst_L )
   BEGIN
    IF (rst_L = '0') THEN
        we_L     <= '1';
        claim_L  <= '1';
        doerr_L  <= '1';

    ELSIF (clk'EVENT  and  clk = '1') THEN
        IF (     (ts_L = '0'     and  tt_take = '1')        -- TS and something we want.
            or   (claim_L = '0'  and  aack_L = '1')) THEN   -- claimed, but not AACK'd
            claim_L <= '0';
            we_L    <= tt_we_L;
        ELSE                                                -- else AACK or no-claim
            claim_L <= '1';
            we_L    <= '1';
        END IF;

        IF (  (ts_L = '0'     and  tt_take = '0')           -- TS and something we dont' want.
           or (doerr_L = '0'  and  aack_L = '1')) THEN      -- errored, but not AACK'd
           doerr_L <= '0';
        ELSE                                                -- else AACK or claim
         doerr_L <= '1';
        END IF;
     END IF;
   END PROCESS;

END BEHAVIOR;
---------------------------------------------------------------------------------------------
```

3.5.2 Byte Write Enable

The next group of signals to generate are the byte lane write enables **$\overline{\text{BWE}}$(0–7)**. These signals are generated by using the transfer size signals **TSIZ(0–2)** along with the lower address bus signals **A(29–31)** to determine which byte lanes should be active.

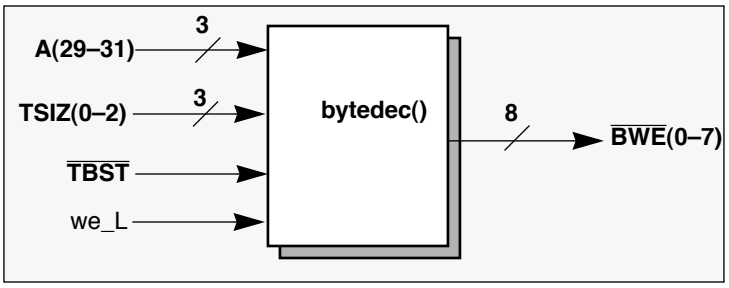


Figure 10. Byte Write Enable Module

Note that the "bytedec()" module examines the decoded write status (WE_L) but not CLAIM, so the byte lane enables are asserted for all write cycles regardless of the activity of the CLAIM signal. This is acceptable as long as the corresponding chip-select signals disable the attached memory and I/O devices, which is true for the devices used.

The VHDL code for the "bytedec()" module is lengthy but straightforward. The values are directly derived from the data alignment tables in the processor user manuals, for example Table 8-3 and Table 8-4 of the *MPC750 RISC Microprocessor User's Manual*. Burst transfers enable all byte lanes, while all other transfers enable only the byte lanes based upon the address and transfer size.

```
-------------------------------------------------------------------------------------------------
-- BYTEDEC.VHD
--
-- BYTEDEC() is the portion of the MC which provides
--   individual byte write enabled for each byte lane, depending upon
--   the size and address of the transfer.  If the cycle is a read cycle,
--   no outputs are asserted at all.
--
-- Copyright 1998, by Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.1
-- Date:     6/10/98
-- Notes:
--           All logic is active low when appended with a "_L".
--           Passed speedwave check 6/10/98.
-------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-------------------------------------------------------------------------------------------------
ENTITY BYTEDEC is
    PORT(    a           : in     std_logic_vector( 29 to 31 ); -- stable 60X bus address
             tsiz        : in     std_logic_vector( 0 to 2 );   -- current transfer size.
             tbst_L      : in     std_logic;                    -- asserted if transfer is burst.
             we_L        : in     std_logic;                    -- asserted if transfer is write.
             bwe_L       : buffer std_logic_vector( 0 to 7 )    -- byte lane write selects.
        );

end; --PORT DEFINITION AND ENTITY

-------------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF BYTEDEC is

SIGNAL be_L : std_logic_vector( 0 to 7 );   -- byte lane enables (read or write).

BEGIN

   -- Convert transfer size and address into byte lane enables.  Write masking
   -- occurs later.

   be_L(0) <= '0'   WHEN (  (tsiz = "001"  and  a = "000")    -- byte
                         or (tsiz = "010"  and  a = "000")    -- half-word
                         or (tsiz = "100"  and  a = "000")    -- word
                         or (tsiz = "000"  and  a = "000")    -- double-word
                         or (tsiz = "011"  and  a = "000")    -- three-byte
```

```
                    or (tbst_L = '0')                   -- burst
                    )
                ELSE '1';

be_L(1) <= '0'  WHEN (  (tsiz = "001"  and  a = "001")    -- byte
                    or (tsiz = "010"  and  a = "000")    -- half-word
                    or (tsiz = "100"  and  a = "000")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "000")    -- three-byte
                    or (tsiz = "011"  and  a = "001")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(2) <= '0'  WHEN (  (tsiz = "001"  and  a = "010")    -- byte
                    or (tsiz = "010"  and  a = "010")    -- half-word
                    or (tsiz = "100"  and  a = "000")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "000")    -- three-byte
                    or (tsiz = "011"  and  a = "001")    -- three-byte
                    or (tsiz = "011"  and  a = "010")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(3) <= '0'  WHEN (  (tsiz = "001"  and  a = "011")    -- byte
                    or (tsiz = "010"  and  a = "010")    -- half-word
                    or (tsiz = "100"  and  a = "000")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "001")    -- three-byte
                    or (tsiz = "011"  and  a = "010")    -- three-byte
                    or (tsiz = "011"  and  a = "011")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(4) <= '0'  WHEN (  (tsiz = "001"  and  a = "100")    -- byte
                    or (tsiz = "010"  and  a = "100")    -- half-word
                    or (tsiz = "100"  and  a = "100")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "010")    -- three-byte
                    or (tsiz = "011"  and  a = "011")    -- three-byte
                    or (tsiz = "011"  and  a = "100")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(5) <= '0'  WHEN (  (tsiz = "001"  and  a = "101")    -- byte
                    or (tsiz = "010"  and  a = "100")    -- half-word
                    or (tsiz = "100"  and  a = "100")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "011")    -- three-byte
                    or (tsiz = "011"  and  a = "100")    -- three-byte
                    or (tsiz = "011"  and  a = "101")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(6) <= '0'  WHEN (  (tsiz = "001"  and  a = "110")    -- byte
                    or (tsiz = "010"  and  a = "110")    -- half-word
                    or (tsiz = "100"  and  a = "100")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "100")    -- three-byte
                    or (tsiz = "011"  and  a = "101")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';

be_L(7) <= '0'  WHEN (  (tsiz = "001"  and  a = "111")    -- byte
                    or (tsiz = "010"  and  a = "110")    -- half-word
                    or (tsiz = "100"  and  a = "100")    -- word
                    or (tsiz = "000"  and  a = "000")    -- double-word
                    or (tsiz = "011"  and  a = "101")    -- three-byte
                    or (tbst_L = '0')                    -- burst
                    )
                ELSE '1';


-- Now mask the byte lanes with the write signal.

bwe_L(0) <= (be_L(0)  or  we_L);
bwe_L(1) <= (be_L(1)  or  we_L);
bwe_L(2) <= (be_L(2)  or  we_L);
```

```
    bwe_L(3) <= (be_L(3)  or  we_L);
    bwe_L(4) <= (be_L(4)  or  we_L);
    bwe_L(5) <= (be_L(5)  or  we_L);
    bwe_L(6) <= (be_L(6)  or  we_L);
    bwe_L(7) <= (be_L(7)  or  we_L);

END BEHAVIOR;
-------------------------------------------------------------------------------------------
```

The three-byte cycles arise from the need to handle misaligned transfers by breaking them into two separate cycles; refer to the *MPC603e and EC603e RISC Microprocessor User's Manual* or the *MPC750 RISC Microprocessor User's Manual* for details on this process. These cycles do not occur if unaligned transfers do not occur. Since many C compilers do not generate such code, the lines for three-byte handling can be deleted to simplify the controller and reduce gate count.

3.5.3 Chip Select

The chip-select module, shown in Figure 11, generates the four chip-select signals and selects the proper time delay for accesses to memory (this information is used by the "cycler()" module).

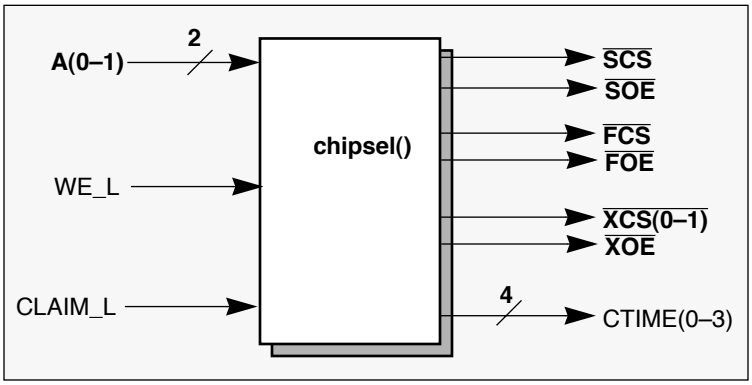


Figure 11. Chip Select Module

Table 5 shows the chip-select actions based upon the address.

Table 5. Chip Select Encodings

A(0–1)	I/O Area	Time (66 MHz)	Bus Clock Count	Timer Value
00	High-speed SRAM array	N/A	N/A	N/A
01	High-speed I/O	60 ns	4	1
10	Slow-speed I/O	180 ns	12	9
11	Flash boot ROM	80 ns	6	3

The timer values in Table 5 have a constant overhead of three subtracted from the expected timer values. This constant overhead is due to the start delay, the final $\overline{\text{TA}}$ assertion, and one clock needed to detect a zero-count on the timer. So for best performance, the actual timer values are offset by (-3). In examining chipsel(), the SRAM chip select is found to be fairly straightforward; the other chip selects differ in that a 4-bit timer value is generated to add delay to the assertion of $\overline{\text{TA}}$.

The code for this module is:

```
-------------------------------------------------------------------------------------------------
-- CHIPSEL.VHD
--
-- CHIPSEL() is the portion of the MC which decodes addresses
--   and provides corresponding chip select outputs, along with a clock
--   timer value which determines the rate of memory accesses.
--
-- Copyright 1998, by Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.2
-- Date:     6/10/98
-- Notes:
--           All logic is active low when appended with a "_L".
--           Passed speedwave check 6/16/98.
-------------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


-------------------------------------------------------------------------------------------------
-- CHIPSEL
-------------------------------------------------------------------------------------------------
ENTITY CHIPSEL is
    PORT( a                 : in     std_logic_vector( 0 to 1 );   -- stable 60X bus address
        claim_L             : in     std_logic;                    -- asserted for active cycles.
        we_L                : in     std_logic;                    -- asserted for write cycles.
        scs_L, soe_L        : buffer std_logic;                    -- SRAM chip-selects & enables.
        fcs_L, foe_L        : buffer std_logic;                    -- Flash chip-selects & enables.
        xcs_L               : buffer std_logic_vector( 0 to 1 );   -- I/O chip selects.
        xoe_L               : buffer std_logic;                    -- I/O output enable.
        ctime               : buffer std_logic_vector( 3 downto 0 )-- 4-bit time value.
        );

end; --PORT DEFINITION AND ENTITY


-------------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF CHIPSEL is
BEGIN

   -- Assert chip select if cycle is claimed and corresponding address is presented.
   scs_L <= '0'     WHEN (a = "00"  and  claim_L = '0')
                    ELSE '1';
   xcs_L(0) <= '0'  WHEN (a = "01"  and  claim_L = '0')
                    ELSE '1';
   xcs_L(1) <= '0'  WHEN (a = "10"  and  claim_L = '0')
                    ELSE '1';
   fcs_L <= '0'     WHEN (a = "11"  and  claim_L = '0')
                    ELSE '1';

   -- Assert corresponding output enables (OE_L) if the cycle is claimed and is not
   -- a write cycle.
   soe_L <= '0'     WHEN (   a = "00"  and  claim_L = '0'  and  we_L = '1')
                    ELSE '1';
   xoe_L <= '0'     WHEN (  (a = "01"  and  claim_L = '0'  and  we_L = '1')
                         or (a = "10"  and  claim_L = '0'  and  we_L = '1'))
                    ELSE '1';
   foe_L <= '0'     WHEN (   a = "11"  and  claim_L = '0'  and  we_L = '1')
                    ELSE '1';


    -- Provide corresponding timer value.  Note that SRAM is not timer controlled, so any
    -- value may be used.  All of these values should be changed if the bus frequency is
    -- changed.  If the clock rate is increased, the system may fail.  If lowered, clock
    -- cycles will be wasted.

    -- Note: there is a three clock overhead in the setup and termination of timed cycles
    -- (one on entry, one during AACK/TA*, and one exiting when the timer is zero).  Therefore,
    -- timing constants are offset by (-3).

    SET_TIMER : PROCESS( fcs_L, xcs_L(0) )
    BEGIN
    IF (fcs_L = '0') THEN
         ctime <= "0011";              -- Flash:     80 ns @ 15ns clocks (66 MHz) =  6 -3 => 3 clocks.
```

```
    ELSIF (xcs_L(0) = '0') THEN
       ctime <= "1001";              -- Slow I/O: 180 ns @ 15ns clocks (66 MHz) = 12 -3 => 9 clocks.
    ELSE
       ctime <= "0001";              -- Fast I/O:  60 ns @ 15ns clocks (66 MHz) =  4 -3 => 1 clocks.
  END IF;
  END PROCESS SET_TIMER;

END BEHAVIOR;
-------------------------------------------------------------------------------------------------
```

The chipsel() module is asynchronous because it relies on the synchronous signal, claim_L, and relies on the stability of the address bus (a) and write select (we_L) signals. These latter two signals are guaranteed to be stable until **$\overline{\text{TA}}$** is asserted because the cycler() module also delays the assertion of **$\overline{\text{AACK}}$** until the last **$\overline{\text{TA}}$**.

The chip-select module may be easily adapted to different device speeds, and for different I/O maps (within PowerPC architecture limitations). It may also be modified to provide access to internal register files or to increase the number of chip selects, within limitations of the FPGA chosen.

3.5.4 Cycler State Machine

The cycler() state machine module controls the remainder of any transaction claimed by the memory controller. For optimal performance, one of four flows are selected. The flows are as follows:

- SRAM single beat transfer
- SRAM burst transfer
- Programmed-length transfer (I/O and Flash)
- Error transactions

The first two optimize speed for the SRAM accesses, which are typically the majority of code and data accesses; the latter are handled in a more programmed method. Fortunately, the streamlined nature of burst transfers keeps the cycler() module from becoming too complicated.

Cycler() finishes any non-error transaction by asserting **$\overline{\text{AACK}}$** and **$\overline{\text{TA}}$** (one to four times, based upon the type of cycle). When **$\overline{\text{AACK}}$** is generated, the cycle has been completed and a new one can begin at the next clock cycle. Due to the pipelining nature of the SRAM, it actually takes 5 beats to do a read cycle, but one of those clock cycles has already been provided before cycler() can leave the IDLE state by the synchronous start() detector. Figure 12 shows the end-cycle module.

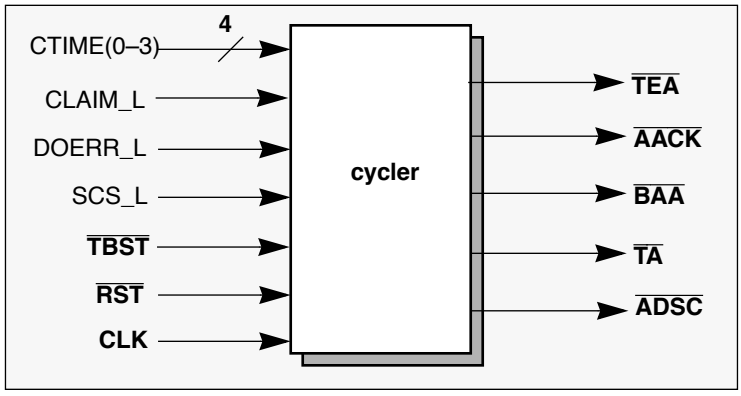


Figure 12. End-Cycle Module

As the VHDL code for the cycler() is generated by a state-machine CAD program, the code is uncommented and somewhat difficult to follow; refer instead to Figure 13 for details.

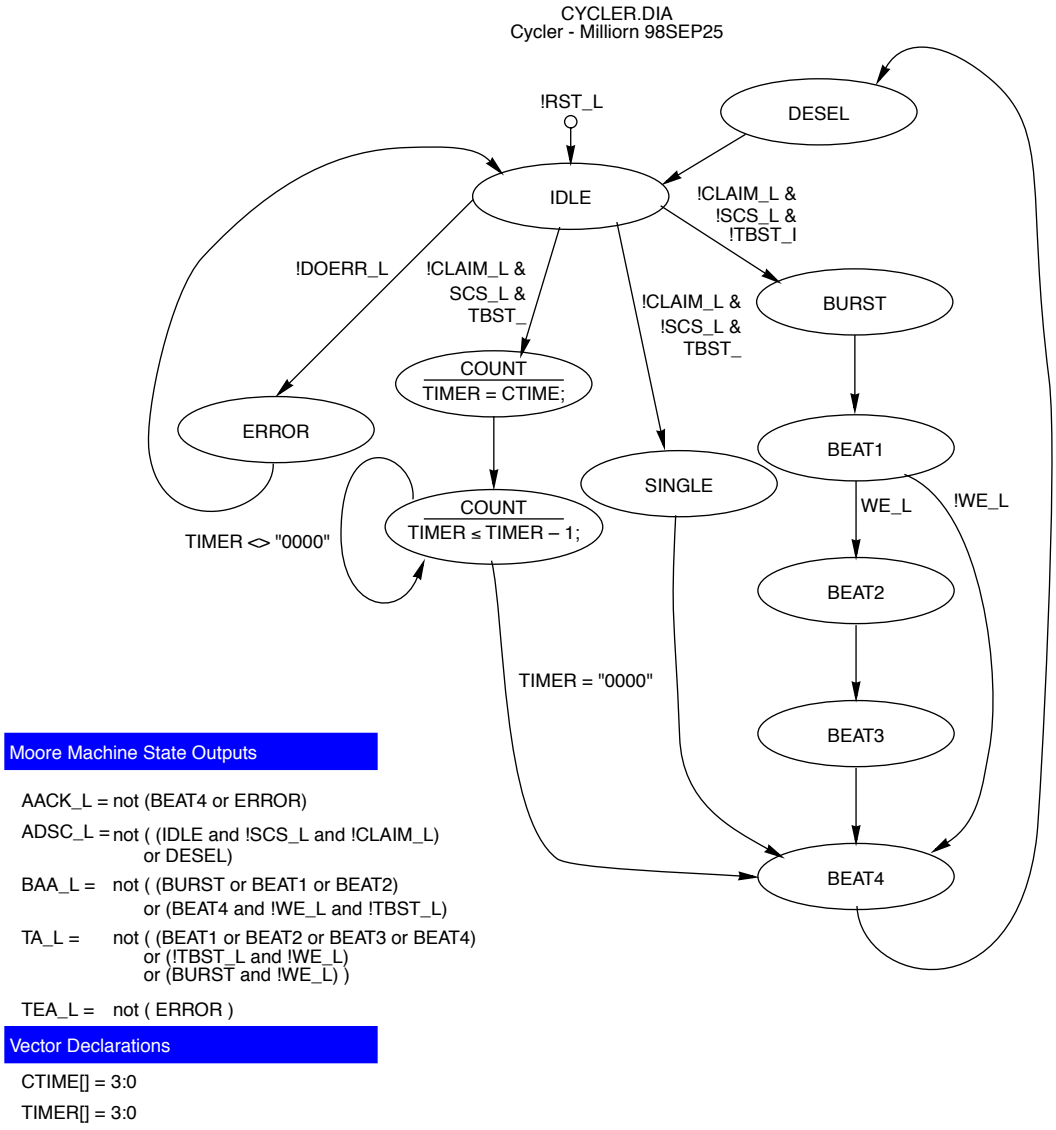


Figure 13. Cycler() State Flow

The state machine switches from the IDLE state to the BEAT1 state on detection of any claimed burst cycle (TBST_L and CLAIM_L asserted, which is only allowed for SRAM). This begins a four-beat burst transfer with **TA** low for four clock cycles (the state machine clocks at the bus frequency, and so proceeds from BEAT1 to BEAT4 automatically). In states BEAT1 through BEAT3, the **BAA** signal is asserted to cause the burst SRAM devices to increment the address. This produces an SRAM access rate of 2-1-1-1 (excluding the **TS**).

Alternately, if CLAIM_L is asserted but not **TBST**, and the cycle is for the SRAM (SCS_L asserted), then this is a single-beat access to SRAM. While this could have been handled by the timer (say by presetting it to 0001), the overhead of checking the timer costs additional cycles. By detecting SRAM single-beats separately, fast access to SRAM is guaranteed (two clocks).

Otherwise, the cycle is either an error or a single-beat access to Flash or I/O. In the latter cases, only one clock of **TA** is needed, but a lengthy delay may be needed to give the peripheral device time to complete the access. For such devices, within the cycler() state-machine is an internal timer which is continually re-loaded while in the IDLE state; in any other state, it counts downward. When the timer reaches zero and the

state is in COUNT, the state machine switches to the BEAT4 state to terminate the cycle with **TA** and **AACK**.

For any cycles which cannot be handled by the memory controller, DOERR_L will be asserted. This is caused either by address-only cycles or specialized data transfer instructions (**lwarx**, etc.); for such cycles, the state machine will assert **TEA** and **AACK**. The behavior of PowerPC processors does not specify what happens when **TEA** is asserted during address-only cycles; however, since this minimal system environment disallows such cycles, the resulting behavior is allowable (either the cycles are silently ignored and processing resumes, or the processor takes an exception).

In all these cases, **AACK** is not asserted until the last (or only) **TA** is asserted, releasing the address tenure as well as the data tenure. The re-assertion delay inherent before **TS** can be asserted guarantees a one-clock cycle recovery time on the data bus.

The VHDL code for this module is:

```
-------------------------------------------------------------------------------------------------
--  D:\USR\GMILLI~1\MC\CYCLER\CYCLER.VHD
--  VHDL code created by Visual Software Solution's StateCAD Version 3.2
--  Thu Sep 24 16:16:39 1998

-- This VHDL code (for use with Workview Office) was generated using:
--  one-hot state assignment with boolean code format.
--  Minimization is enabled,  implied else is enabled,
--  and outputs are manually optimized.

--LIBRARY LAT_VHD;
--USE LAT_VHD.VHD_PKG.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY synth;
USE synth.vhdlsynth.all;

ENTITY SHELL_CYCLER IS
    PORT (CLK,CLAIM_L,CTIME0,CTIME1,CTIME2,CTIME3,DOERR_L,RST_L,SCS_L,TBST_L,
        WE_L: IN std_logic;
        AACK_L,ADSC_L,BAA_L,TA_L,TEA_L : OUT std_logic);

    SIGNAL TIMER0,TIMER1,TIMER2,TIMER3: std_logic;
END;

ARCHITECTURE BEHAVIOR OF SHELL_CYCLER IS
    -- State variables for machine sreg
    SIGNAL BEAT1, next_BEAT1, BEAT2, next_BEAT2, BEAT3, next_BEAT3, BEAT4,
        next_BEAT4, BURST, next_BURST, CLOCK, next_CLOCK, COUNT, next_COUNT, DESEL,
        next_DESEL, ERROR, next_ERROR, IDLE, next_IDLE, SINGLE, next_SINGLE :
        std_logic;
    SIGNAL next_TIMER0,next_TIMER1,next_TIMER2,next_TIMER3 : std_logic;
    SIGNAL TIMER : std_logic_vector (3 DOWNTO 0);

    ATTRIBUTE PERMEABILITY OF BEHAVIOR: ARCHITECTURE IS TRUE;
BEGIN
    PROCESS (CLK, RST_L, next_BEAT1, next_BEAT2, next_BEAT3, next_BEAT4,
        next_BURST, next_CLOCK, next_COUNT, next_DESEL, next_ERROR, next_IDLE,
        next_SINGLE, next_TIMER3, next_TIMER2, next_TIMER1, next_TIMER0)
    BEGIN
        IF ( RST_L='0' ) THEN
            BEAT1 <= '0';
            BEAT2 <= '0';
            BEAT3 <= '0';
            BEAT4 <= '0';
            BURST <= '0';
            CLOCK <= '0';
            COUNT <= '0';
            DESEL <= '0';
            ERROR <= '0';
            IDLE <= '1';
            SINGLE <= '0';
            TIMER3 <= '0';
            TIMER2 <= '0';
            TIMER1 <= '0';
            TIMER0 <= '0';
        ELSIF CLK='1' AND CLK'event THEN
            BEAT1 <= next_BEAT1;
```

```
        BEAT2 <= next_BEAT2;
        BEAT3 <= next_BEAT3;
        BEAT4 <= next_BEAT4;
        BURST <= next_BURST;
        CLOCK <= next_CLOCK;
        COUNT <= next_COUNT;
        DESEL <= next_DESEL;
        ERROR <= next_ERROR;
        IDLE <= next_IDLE;
        SINGLE <= next_SINGLE;
        TIMER3 <= next_TIMER3;
        TIMER2 <= next_TIMER2;
        TIMER1 <= next_TIMER1;
        TIMER0 <= next_TIMER0;
    END IF;
END PROCESS;

PROCESS (BEAT1,BEAT2,BEAT3,BEAT4,BURST,CLAIM_L,CLOCK,COUNT,CTIME0,CTIME1,
    CTIME2,CTIME3,DESEL,DOERR_L,ERROR,IDLE,SCS_L,SINGLE,TBST_L,TIMER0,TIMER1,
    TIMER2,TIMER3,WE_L,TIMER)
BEGIN

    IF ((  (BURST='1'))) THEN next_BEAT1<='1';
    ELSE next_BEAT1<='0';
    END IF;

    IF (( WE_L='1' AND  (BEAT1='1'))) THEN next_BEAT2<='1';
    ELSE next_BEAT2<='0';
    END IF;

    IF ((  (BEAT2='1'))) THEN next_BEAT3<='1';
    ELSE next_BEAT3<='0';
    END IF;

    IF (( WE_L='0' AND  (BEAT1='1')) OR (  (BEAT3='1')) OR ( TIMER0='0' AND
        TIMER1='0' AND TIMER2='0' AND TIMER3='0' AND  (CLOCK='1')) OR (  (SINGLE='1')
        )) THEN next_BEAT4<='1';
    ELSE next_BEAT4<='0';
    END IF;

    IF (( DOERR_L='1' AND TBST_L='0' AND CLAIM_L='0' AND SCS_L='0' AND
        (IDLE='1'))) THEN next_BURST<='1';
    ELSE next_BURST<='0';
    END IF;

    IF (( TIMER0='1' AND  (CLOCK='1')) OR ( TIMER1='1' AND  (CLOCK='1')) OR (
        TIMER2='1' AND  (CLOCK='1')) OR ( TIMER3='1' AND  (CLOCK='1')) OR (
        (COUNT='1'))) THEN next_CLOCK<='1';
    ELSE next_CLOCK<='0';
    END IF;

    IF (( DOERR_L='1' AND SCS_L='1' AND CLAIM_L='0' AND TBST_L='1' AND
        (IDLE='1'))) THEN next_COUNT<='1';
    ELSE next_COUNT<='0';
    END IF;

    IF ((  (BEAT4='1'))) THEN next_DESEL<='1';
    ELSE next_DESEL<='0';
    END IF;

    IF (( DOERR_L='0' AND  (IDLE='1'))) THEN next_ERROR<='1';
    ELSE next_ERROR<='0';
    END IF;

    IF ((  (DESEL='1')) OR (  (ERROR='1')) OR ( DOERR_L='1' AND SCS_L='1' AND
        TBST_L='0' AND  (IDLE='1')) OR ( DOERR_L='1' AND CLAIM_L='1' AND  (IDLE='1'))
        ) THEN next_IDLE<='1';
    ELSE next_IDLE<='0';
    END IF;

    IF (( DOERR_L='1' AND CLAIM_L='0' AND SCS_L='0' AND TBST_L='1' AND
        (IDLE='1'))) THEN next_SINGLE<='1';
    ELSE next_SINGLE<='0';
    END IF;


    TIMER<= ((  ( BEAT1& BEAT1& BEAT1& BEAT1)) AND  ((  ( WE_L& WE_L& WE_L&
        WE_L)) ) AND  ( ("0000") ) ) OR  ((  ( BEAT1& BEAT1& BEAT1& BEAT1)) AND  ((
        ( NOT WE_L& NOT WE_L& NOT WE_L& NOT WE_L)) ) AND  ( ("0000") ) ) OR  ((  (
        BEAT2& BEAT2& BEAT2& BEAT2)) AND  ( ("1111") ) AND  ( ("0000") ) ) OR  ((  (
        BEAT3& BEAT3& BEAT3& BEAT3)) AND  ( ("1111") ) AND  ( ("0000") ) ) OR  ((  (
        BEAT4& BEAT4& BEAT4& BEAT4)) AND  ( ("1111") ) AND  ( ("0000") ) ) OR  ((  (
        BURST& BURST& BURST& BURST)) AND  ( ("1111") ) AND  ( ("0000") ) ) OR  ((  (
```

```
            CLOCK& CLOCK& CLOCK& CLOCK)) AND  ((  ( TIMER3& TIMER3& TIMER3& TIMER3)) OR (
              ( TIMER2& TIMER2& TIMER2& TIMER2)) OR (  ( TIMER1& TIMER1& TIMER1& TIMER1))
             OR (  ( TIMER0& TIMER0& TIMER0& TIMER0)) ) AND  ((  (TIMER3 &TIMER2 &TIMER1
            &TIMER0)) - ("0001") ) ) OR  ((  ( COUNT& COUNT& COUNT& COUNT)) AND  (
            ("1111") ) AND  ((  (TIMER3 &TIMER2 &TIMER1 &TIMER0)) - ("0001") ) ) OR  ((
            ( CLOCK& CLOCK& CLOCK& CLOCK)) AND  ((  ( NOT TIMER0& NOT TIMER0& NOT TIMER0&
             NOT TIMER0)AND  ( NOT TIMER1& NOT TIMER1& NOT TIMER1& NOT TIMER1)AND  ( NOT
            TIMER2& NOT TIMER2& NOT TIMER2& NOT TIMER2)AND  ( NOT TIMER3& NOT TIMER3& NOT
             TIMER3& NOT TIMER3)) ) AND  ( ("0000") ) ) OR  ((  ( IDLE& IDLE& IDLE& IDLE)
            ) AND  ((  ( DOERR_L& DOERR_L& DOERR_L& DOERR_L)AND  ( SCS_L& SCS_L& SCS_L&
            SCS_L)AND  ( NOT CLAIM_L& NOT CLAIM_L& NOT CLAIM_L& NOT CLAIM_L)AND  ( TBST_L
            & TBST_L& TBST_L& TBST_L)) ) AND  ((  (CTIME3 &CTIME2 &CTIME1 &CTIME0)) ) )
            OR  ((  ( DESEL& DESEL& DESEL& DESEL)) AND  ( ("1111") ) AND  ( ("0000") ) )
            OR  ((  ( ERROR& ERROR& ERROR& ERROR)) AND  ( ("1111") ) AND  ( ("0000") ) )
            OR  ((  ( IDLE& IDLE& IDLE& IDLE)) AND  ((  ( NOT DOERR_L& NOT DOERR_L& NOT
            DOERR_L& NOT DOERR_L)) ) AND  ( ("0000") ) ) OR  ((  ( IDLE& IDLE& IDLE& IDLE
            )) AND  ((  ( DOERR_L& DOERR_L& DOERR_L& DOERR_L)AND  ( NOT CLAIM_L& NOT
            CLAIM_L& NOT CLAIM_L& NOT CLAIM_L)AND  ( NOT SCS_L& NOT SCS_L& NOT SCS_L& NOT
             SCS_L)AND  ( TBST_L& TBST_L& TBST_L& TBST_L)) ) AND  ( ("0000") ) ) OR  ((
            ( IDLE& IDLE& IDLE& IDLE)) AND  ((  ( DOERR_L& DOERR_L& DOERR_L& DOERR_L)AND
             ( NOT TBST_L& NOT TBST_L& NOT TBST_L& NOT TBST_L)AND  ( NOT CLAIM_L& NOT
            CLAIM_L& NOT CLAIM_L& NOT CLAIM_L)AND  ( NOT SCS_L& NOT SCS_L& NOT SCS_L& NOT
             SCS_L)) ) AND  ( ("0000") ) ) OR  ((  ( IDLE& IDLE& IDLE& IDLE)) AND  ((  (
            DOERR_L& DOERR_L& DOERR_L& DOERR_L)AND  ( CLAIM_L& CLAIM_L& CLAIM_L& CLAIM_L)
            ) OR (  ( DOERR_L& DOERR_L& DOERR_L& DOERR_L)AND  ( SCS_L& SCS_L& SCS_L&
            SCS_L)AND  ( NOT TBST_L& NOT TBST_L& NOT TBST_L& NOT TBST_L)) ) AND  (
            ("0000") ) ) OR  ((  ( SINGLE& SINGLE& SINGLE& SINGLE)) AND  ( ("1111") ) AND
              ( ("0000") ) );

        next_TIMER3 <= TIMER(3);
        next_TIMER2 <= TIMER(2);
        next_TIMER1 <= TIMER(1);
        next_TIMER0 <= TIMER(0);
    END PROCESS;

    PROCESS (BEAT4,ERROR)
    BEGIN
        IF ((  (BEAT4='0')AND  (ERROR='0'))) THEN AACK_L<='1';
        ELSE AACK_L<='0';
        END IF;
    END PROCESS;

    PROCESS (CLAIM_L,DESEL,IDLE,SCS_L)
    BEGIN
        IF (( CLAIM_L='1' AND  (DESEL='0')) OR ( SCS_L='1' AND  (DESEL='0')) OR (
            (IDLE='0')AND  (DESEL='0'))) THEN ADSC_L<='1';
        ELSE ADSC_L<='0';
        END IF;
    END PROCESS;

    PROCESS (BEAT1,BEAT2,BEAT4,BURST,TBST_L,WE_L)
    BEGIN
        IF ((  (BURST='0')AND  (BEAT1='0')AND  (BEAT2='0')AND  (BEAT4='0')) OR (
            (BURST='0')AND  (BEAT1='0')AND  (BEAT2='0')AND WE_L='1' ) OR (  (BURST='0')
            AND  (BEAT1='0')AND  (BEAT2='0')AND TBST_L='1' )) THEN BAA_L<='1';
        ELSE BAA_L<='0';
        END IF;
    END PROCESS;

    PROCESS (BEAT1,BEAT2,BEAT3,BEAT4,BURST,TBST_L,WE_L)
    BEGIN
        IF ((  (BEAT1='0')AND  (BEAT2='0')AND  (BEAT3='0')AND  (BEAT4='0')AND
            TBST_L='1' AND  (BURST='0')) OR (  (BEAT1='0')AND  (BEAT2='0')AND  (BEAT3='0'
            )AND  (BEAT4='0')AND WE_L='1' )) THEN TA_L<='1';
        ELSE TA_L<='0';
        END IF;
    END PROCESS;

    PROCESS (ERROR)
    BEGIN
        IF ((  (ERROR='0'))) THEN TEA_L<='1';
        ELSE TEA_L<='0';
        END IF;
    END PROCESS;
END BEHAVIOR;

--LIBRARY LAT_VHD;
--USE LAT_VHD.VHD_PKG.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY synth;
```

```
USE synth.vhdlsynth.all;

ENTITY CYCLER IS
    PORT (CTIME : IN std_logic_vector (3 DOWNTO 0);
        CLK,CLAIM_L,DOERR_L,RST_L,SCS_L,TBST_L,WE_L: IN std_logic;
        AACK_L,ADSC_L,BAA_L,TA_L,TEA_L : OUT std_logic);
END;

ARCHITECTURE BEHAVIOR OF CYCLER IS
    COMPONENT SHELL_CYCLER
        PORT (CLK,CLAIM_L,CTIME0,CTIME1,CTIME2,CTIME3,DOERR_L,RST_L,SCS_L,TBST_L,
            WE_L: IN std_logic;
            AACK_L,ADSC_L,BAA_L,TA_L,TEA_L : OUT std_logic);
    END COMPONENT;
BEGIN
    SHELL1_CYCLER : SHELL_CYCLER PORT MAP (CLK=>CLK,CLAIM_L=>CLAIM_L,CTIME0=>
        CTIME(0),CTIME1=>CTIME(1),CTIME2=>CTIME(2),CTIME3=>CTIME(3),DOERR_L=>DOERR_L,
        RST_L=>RST_L,SCS_L=>SCS_L,TBST_L=>TBST_L,WE_L=>WE_L,AACK_L=>AACK_L,ADSC_L=>
        ADSC_L,BAA_L=>BAA_L,TA_L=>TA_L,TEA_L=>TEA_L);
END BEHAVIOR;

CONFIGURATION SHELL2_CYCLER OF CYCLER IS
    FOR BEHAVIOR END FOR;
END SHELL2_CYCLER;


-------------------------------------------------------------------------------------------------
```

The preceeding code was produced by a state machine compiler, so there are no comments and it is not very readable. The code uses a “one-hot” encoding (one register encodes each state), so each clock cycle the registers are reloaded with the encoded next state calculations in a typical Moore machine fashion. The remainder of the code computes the next state, and provides the encoded output. The code for calculating the timing value (TIME) looks complicated because all four bits are calculated in one statement.

3.5.5 Memory Controller Module

The final module is the memory controller itself, which simply interconnects the previous modules, and is shown previously in Figure 8.

The VHDL code for this module is:

```
-------------------------------------------------------------------------------------------------
-- MC.VHD
--
-- MC is an FPGA which implements a simple but fast MC for the PowerPC 60X/7XX
-- family of processors.  The controller is described in detail in Application Note AN17XX,
-- "A minimal PowerPC System Design".
--
-- Most of MC is just a top-level interconnect of lower-level modules:
--
--       start    : checks TT and asserts CLAIM or DOERR depending on whether the
--                  transfer will be handled or not.
--       ttdec    : uses TT bits to separate cycles into handled and non-handled types.
--       chipsel  : provides chip select and output enables for devices depending
--                  upon the current address.  Provides timing values for cycler to
--                  use. Speculatively asserts ADSC*.
--       bytedec  : provides byte-write enables for SRAM and Flash.
--       cycler   : handles timing of assertion of AACK* and TA*, or of AACK* and TEA*,
--                  depending on CLAIM or DOERR status.  Handles burst, single-beat
--                  with various timings.
--       int      : simple interrupt merge.
--
-- Copyright 1998, Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.3
-- Date:     6/21/98
-- Notes:
--         All logic is active low when appended with a "_L"
-------------------------------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
use ieee.std_logic_unsigned.all;


-------------------------------------------------------------------------------------------
-- MC
-------------------------------------------------------------------------------------------
ENTITY MC is
    PORT( clk, rst_L        : in     std_logic;                    -- general controls.

          a_high            : in     std_logic_vector( 0 to 1 );   -- upper 60X address
          a_low             : in     std_logic_vector( 29 to 31 ); -- lower 60X bus address
          ts_L              : in     std_logic;                    -- transfer start.
          tt                : in     std_logic_vector( 0 to 4 );   -- transfer type.
          tsiz              : in     std_logic_vector( 0 to 2 );   -- transfer size.
          tbst_L            : in     std_logic;                    -- asserted if transfer is burst.
          irq               : in     std_logic_vector( 0 to 3 );   -- interrupt inputs.
          altrst_L          : in     std_logic;                    -- alternate reset input.
          cophrst_L         : in     std_logic;                    -- COP port HRESET input.

          bwe_L             : buffer std_logic_vector( 0 to 7 );   -- byte lane write selects.
          scs_L, soe_L      : buffer std_logic;                   -- SRAM chip-selects & enable.
          fcs_L, foe_L      : buffer std_logic;                   -- Flash chip-selects & enable.
          xcs_L             : buffer std_logic_vector( 0 to 1 );   -- I/O chip selects.
          xoe_L             : buffer std_logic;                   -- I/O output enable.

          ta_L, tea_L       : out    std_logic;                   -- normal and error acks.
          aack_L            : out    std_logic;                   -- address acks.
          adsc_L            : out    std_logic;                   -- SRAM address latch.
          baa_L             : out    std_logic;                   -- SRAM burst address advance.

          int_L             : buffer std_logic;                   -- interrupt output.
          hreset_L          : buffer std_logic;                   -- CPU HRESET* output.
          mreset            : buffer std_logic;                   -- Misc active-high reset output.
          fcsled, scsled    : buffer std_logic;                   -- LED output drivers.
          xcsled            : buffer std_logic;                   -- "

          d                 : in     std_logic_vector( 0 to 7 );   -- data bus input.

          probe1            : buffer std_logic;                   -- internal monitors.
          monitor1          : buffer std_logic                    -- ViewSynthesis bug.
           );

end; --PORT DEFINITION AND ENTITY


-------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF MC is

   COMPONENT BYTEDEC
   PORT( a                  : in     std_logic_vector( 29 to 31 ); -- stable 60X bus address
         tsiz               : in     std_logic_vector( 0 to 2 );   -- current transfer size.
         tbst_L             : in     std_logic;                    -- asserted if transfer is burst.
         claim_L            : in     std_logic;                    -- asserted if transfer is claimed.
         we_L               : in     std_logic;                    -- asserted if transfer is write.
         bwe_L              : buffer std_logic_vector( 0 to 7 )    -- byte lane write selects.
        );
   END COMPONENT;

   COMPONENT CHIPSEL
   PORT( a                  : in     std_logic_vector( 0 to 1 );   -- stable 60X bus address
         claim_L            : in     std_logic;                    -- asserted for active cycles.
         we_L               : in     std_logic;                    -- asserted for write cycles.
         scs_L, soe_L       : buffer std_logic;                   -- SRAM chip-selects & enable.
         fcs_L, foe_L       : buffer std_logic;                   -- Flash chip-selects & enable.
         xcs_L              : buffer std_logic_vector( 0 to 1 );   -- I/O chip selects.
         xoe_L              : buffer std_logic;                   -- I/O output enable.
         ctime              : buffer std_logic_vector( 3 downto 0 )-- 4-bit time value.
        );
   END COMPONENT;

   COMPONENT INT
   PORT( irq                : in     std_logic_vector( 0 to 3 );   -- interupt inputs (var. polarity)
         int_L              : buffer std_logic                     -- interrupt output.
        );
   END COMPONENT;

   COMPONENT CYCLER
   PORT( CTIME              : IN std_logic_vector (3 DOWNTO 0);
         CLK,CLAIM_L,DOERR_L,
         RST_L,SCS_L,TBST_L : IN std_logic;
         AACK_L,ADSC_L,BAA_L,
         TA_L,TEA_L         : OUT std_logic
          );
```

```
    END COMPONENT;

    COMPONENT START
    PORT( tt_take            : in     std_logic;                    -- asserted if good TT selection.
          tt_we_L            : in     std_logic;                    -- asserted if good TT is write.
          ts_L               : in     std_logic;                    -- transfer start strobe.
          aack_L             : in     std_logic;                    -- asserted on transfer complete.
          clk                : in     std_logic;                    -- bus clock.
          rst_L              : in     std_logic;                    -- system reset.
          claim_L            : buffer std_logic;                    -- asserted when cycle is claimed.
          doerr_L            : buffer std_logic;                    -- asserted when cycle not claimed.
          we_L               : buffer std_logic                     -- byte lane write selects.
        );
    END COMPONENT;

    COMPONENT TTDEC
    PORT( tt                 : in     std_logic_vector( 0 to 4 );   -- current transfer type.
          tt_take            : buffer std_logic;                    -- asserted when TT matches types.
          tt_we_L            : buffer std_logic;                    -- asserted when cycle is write.
          monitor            : buffer std_logic                     -- unneeded, ViewSynthesis bug.
          );

    END COMPONENT;


    SIGNAL  tt_take          : std_logic;                           -- asserted for TT matches.
    SIGNAL  tt_we_L          : std_logic;                           -- asserted for TT match writes.
    SIGNAL  we_L             : std_logic;                           -- asserted for write cycles.
    SIGNAL  claim_L          : std_logic;                           -- asserted for cycles to process.
    SIGNAL  doerr_L          : std_logic;                           -- asserted for cycles to TEA*
    SIGNAL  ctime            : std_logic_vector( 3 downto 0 );      -- selected cycle time.
    SIGNAL  aack_internal_L  : std_logic;                           -- internal copy.


-------------------------------------------------------------------------------------------------
BEGIN

    TTDEC_1   : TTDEC PORT MAP (
                    tt => tt, tt_take => tt_take, tt_we_L => tt_we_L, monitor => monitor1
                );

    START_1   : START PORT MAP (
                    tt_take => tt_take, tt_we_L => tt_we_L, ts_L => ts_L, aack_L => aack_internal_L,
                    clk => clk, rst_L => rst_L,
                    claim_L => claim_L, doerr_L => doerr_L, we_L => we_L
                );

    CHIPSEL_1   : CHIPSEL PORT MAP (
                    a => a_high, claim_L => claim_L, we_L => we_L, scs_L => scs_L, soe_L => soe_L,
                    fcs_L => fcs_L, foe_L => foe_L, xcs_L => xcs_L, xoe_L => xoe_L,
                    ctime => ctime
                );

    BYTEDEC_1 : BYTEDEC PORT MAP (
                    a => a_low, tsiz => tsiz, tbst_L => tbst_L,
                    claim_L => claim_L, we_L => we_L,
                    bwe_L => bwe_L
                );

    CYCLER_1  : CYCLER PORT MAP (
                    CTIME => ctime, CLK => clk, CLAIM_L => claim_L,
                    DOERR_L => doerr_L, RST_L => rst_L, SCS_L => scs_L, TBST_L => tbst_L,
                    AACK_L => aack_internal_L, ADSC_L => adsc_L, BAA_L => baa_L,
                    TA_L => ta_L, TEA_L => tea_L
                );


    -- Copy internal aack to external aack, since VHDL is fussy about connecting OUT's to BUFFER's.
    aack_L <= '0'  WHEN (aack_internal_L = '0')
                   ELSE '1';

    -- The databus port is not currently used; add logic to use it to maintain its existance,
    -- otherwise errors will be generated for unused ports.
    probe1 <= '0'  WHEN (d = "11111111")
                   ELSE '1';


    ---------------------------------------------------------------------------------------------
    -- Sideband modules that are not part of the memory controller but are needed for the Excimer
    -- project include the interrupt controller, reset drivers and LED monitors.

    -- Extremely simple interrupt controller -- the databus is wired and ready to accept a more
    -- complicated version, if desired.
```

```
   INT_1     : INT PORT MAP (
                  irq => irq, int_L => int_L
               );


   ------------------------------------------------------------------------------------------------
   -- Assert HRESET to CPU when general reset is asserted or when COP resets it.
   -- The active high RESET is only asserted on the general reset, not by COP.

   hreset_L <= '0'   WHEN (altrst_L = '0'  or  cophrst_L = '0')
                     ELSE '1';
   mreset <= '1'     WHEN (hreset_L = '0')
                     ELSE '0';


   ------------------------------------------------------------------------------------------------
   -- Set the LED monitor outputs when any I/O action occurs.  While you could tie it to the
   -- chip selects, LEDs need some current so it is best to keep them isolated.

   fcsled <= '1' WHEN (fcs_L = '0')
                 ELSE '0';
   scsled <= '1' WHEN (scs_L = '0')
                 ELSE '0';
   xcsled <= '1' WHEN (xcs_L(0) = '0'  or  xcs_L(1) = '0')
                 ELSE '0';

END BEHAVIOR;
------------------------------------------------------------------------------------------------
```

3.6 Waveforms

This section shows several timing waveforms. Figure 14 shows single-beat access to SRAM, which is similar to I/O and Flash, except that no timer is used to keep the performance high. In this waveform, the data is available on the second clock after the **$\overline{\text{TS}}$** signal is asserted.

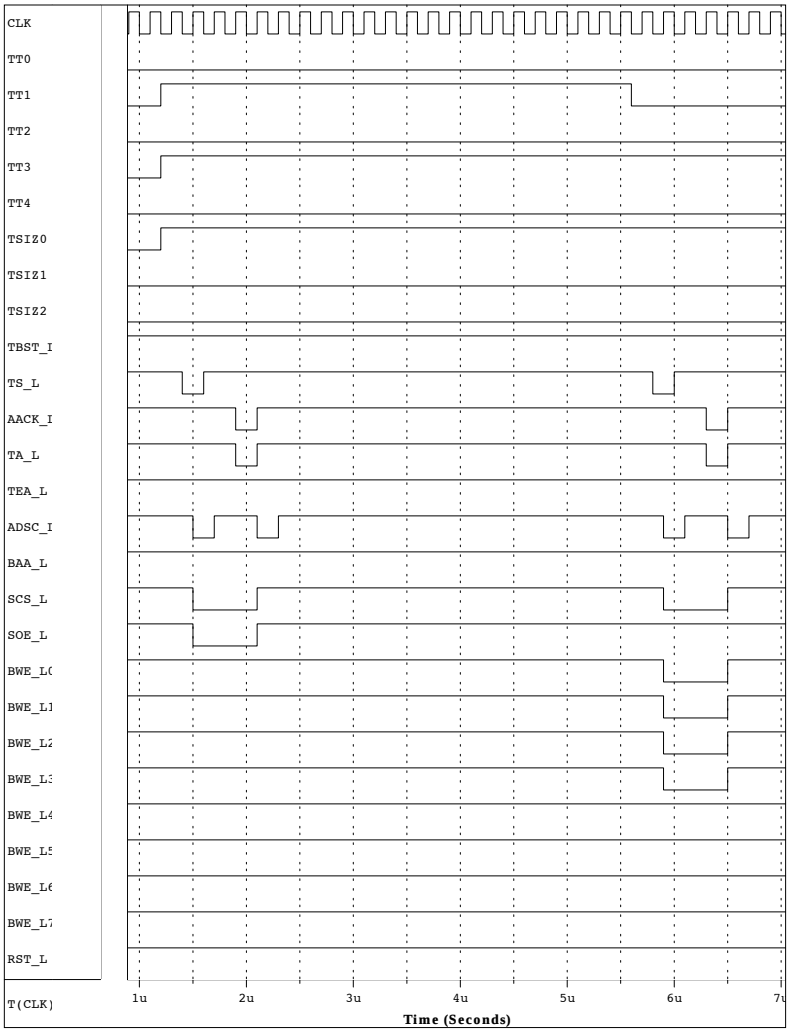


Figure 14. Pipelined Burst SRAM—Single-Beat Read/Write

Figure 15 shows pipelined burst SRAMs that need **ADSC** asserted to start, then TA asserted for a burst of four beats for the data. After the first beat, **BAA** is asserted to increment the address to the next location. At the end of each transfer, **ADSC** is strobed to deselect the SRAM.

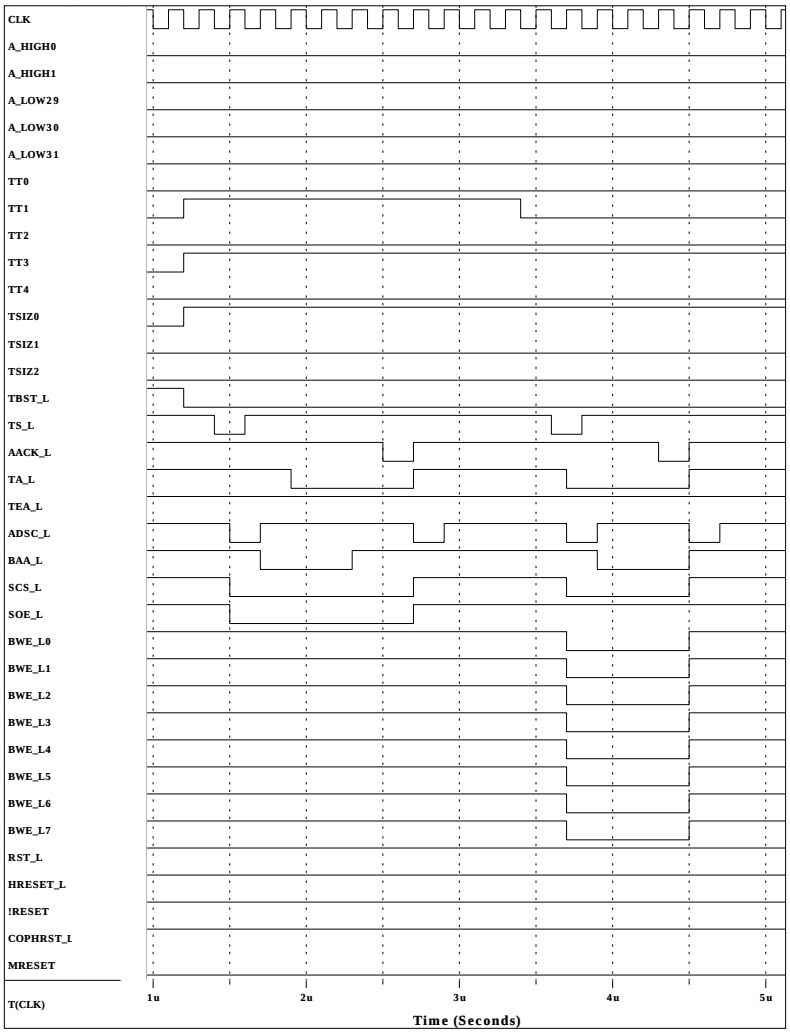


Figure 15. Pipelined Burst SRAM—Burst Read/Write

Figure 16 shows that the Flash access is controlled by timed values, in this case a value of 3 (as provided by the chipsel() module) which produces a 6-clock access time.

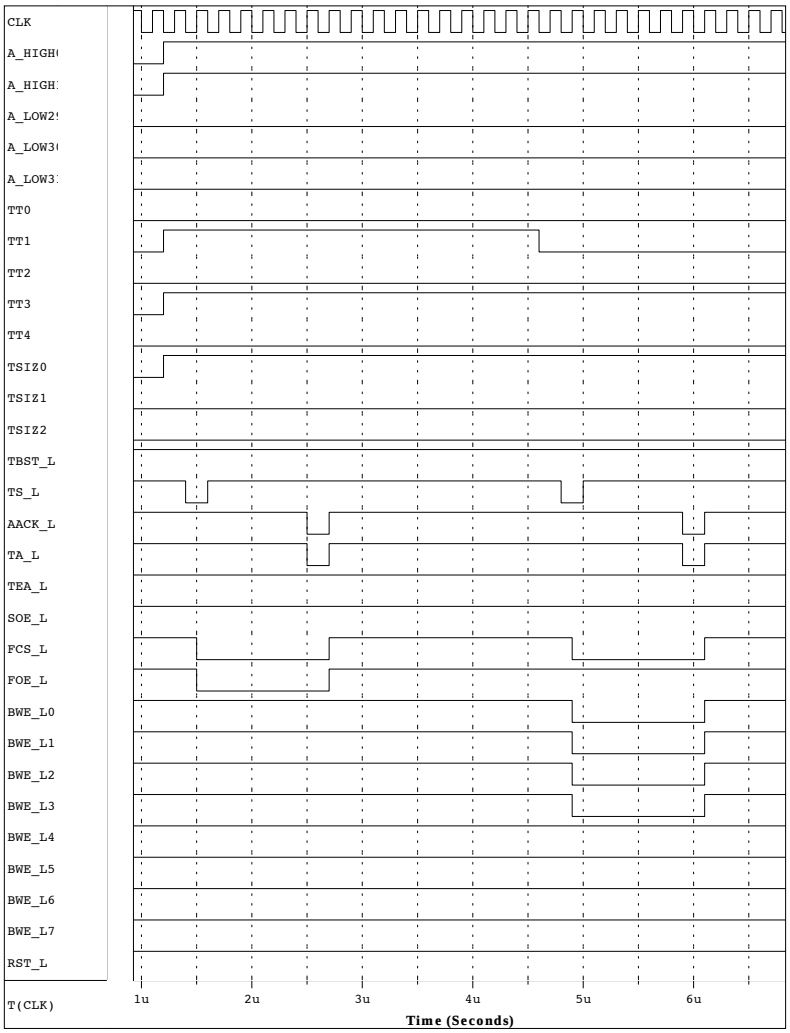


Figure 16. Flash ROM—Single-Beat Read/Write

Figure 17 shows two back-to-back accesses, one to the “slow” I/O space, and the second to the “fast” I/O space.

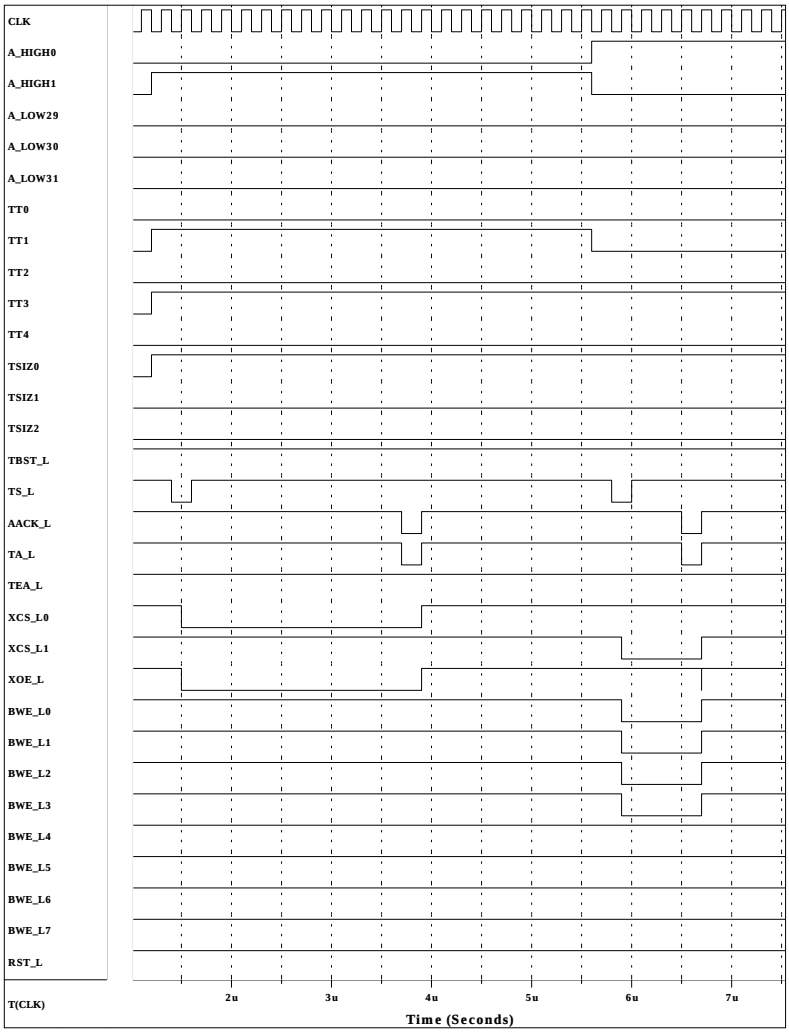


Figure 17. I/O ROM—Single-Beat Read/Write

3.7 Software

When writing software using this simple memory controller, be sure to consider the effects the restrictions have placed on the environment. For example, because the Flash and I/O areas do not support burst transfers, they cannot be made cacheable. If either the instruction or data cache is enabled on any PowerPC processor, burst transfers will always occur unless the memory management unit (via BATs or PTEs) is used to mark addresses as non-cacheable.

Part 4 Clock

Unlike some systems, the clock circuitry for a minimal system is quite simple. The processor, memory controller and two SRAM memories all need a separate bus clock (anywhere from 1 Hz to 100 MHz[1]) and have a 250 ps point-to-point skew allowance. The simplest way to do this is to connect a crystal oscillator device to all four loads as shown in . This is generally achievable with most clock oscillators.

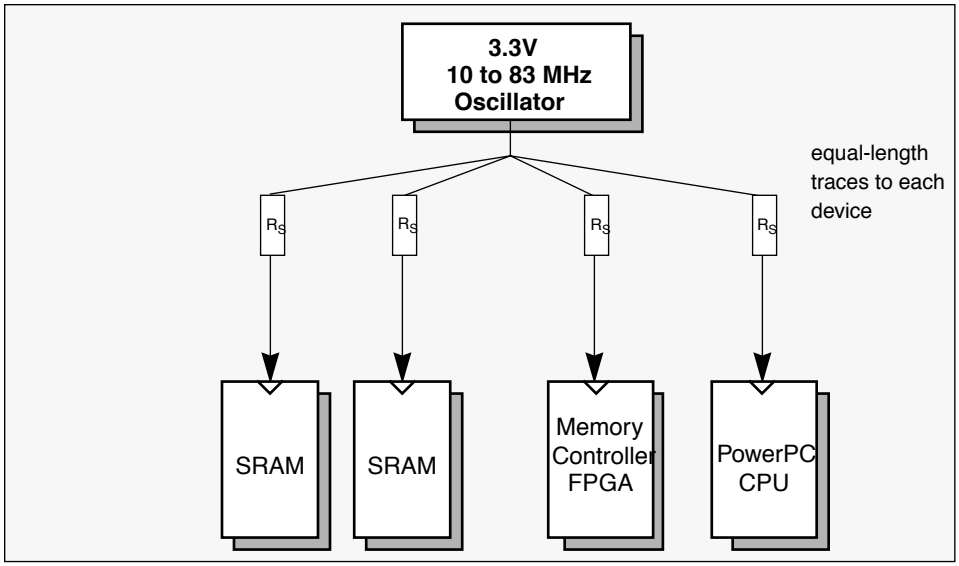


Figure 18. Simplest Clock Connection

The clock generator must have a very low output impedance in order to drive four loads from one output, and it may be unacceptable unless the clock traces can be kept very short (on the order of 3 cm or so).

If this is not possible, an alternative is to employ an inexpensive low-skew clock generator such as the Motorola MPC904 as shown in Figure 19. Using a crystal or an oscillator with this device, each component can have a dedicated clock signal. This can make the board routing much easier, and other devices in the Motorola MPC9xx family can provide other clocks that may be needed along with the primary system needs, increasing integration.

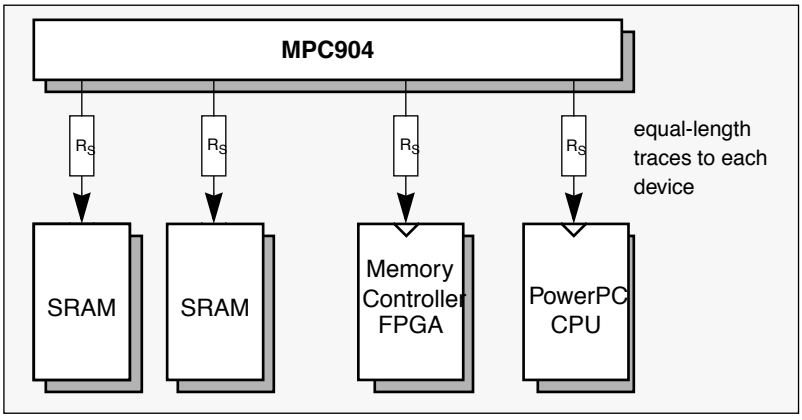


Figure 19. MPC904 Clock Connection

[1]Note: MPC604-class devices are not fully static and have minimum clock frequencies. MPC603- and MPC750-class devices are fully static. Refer to the respective hardware reference datasheets for details.

Part 5 Reset

In order to properly condition a PowerPC processor, the **HRESET** signal must be asserted whenever the system initially powers up and whenever the processor power supply (or supplies) fall below -5% of the nominal voltage described in the hardware specification. The JTAG **TRST** signal must be asserted at reset as well, to initialize the scan chain to a known state.

In addition, the initial power-up sequence requires that the **HRESET** signal be asserted for a minimum of 255 clocks in order to properly initialize the clock PLL and initialize hardware signals.

The simplest way to achieve all of these goals is to use one of many inexpensive devices available to drive the reset lines at the proper time. Called “reset controllers” or “system supervisory controllers”, these devices are typically very inexpensive (less than US$0.50), have small footprints (SOT23 to SO8), and are widely available from Texas Instruments, Maxim Semiconductor, and others. Figure 20 shows an example using these types of circuits.

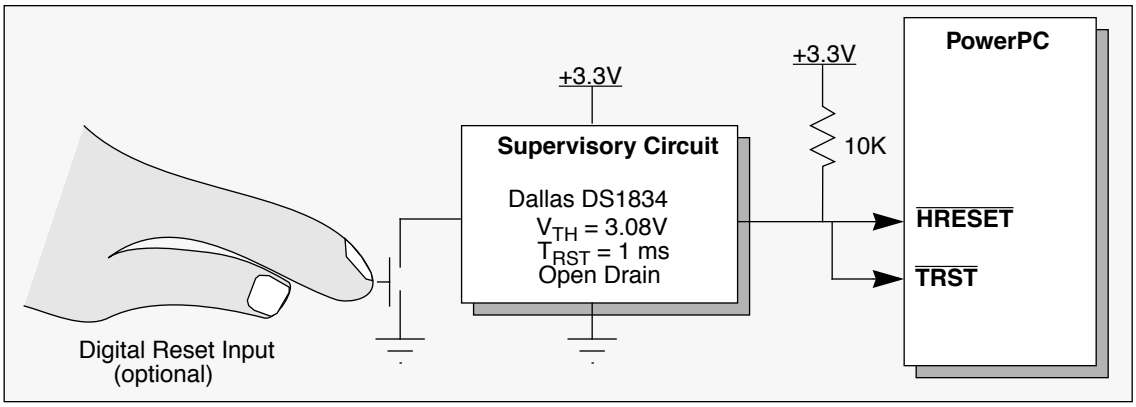


Figure 20. Reset Using Supervisory Controller

If the reliability of the power supply can be assured, or if the power supply provides a failure output, then the reset controller can be reduced to a simple R-C network, as shown in Figure 21.

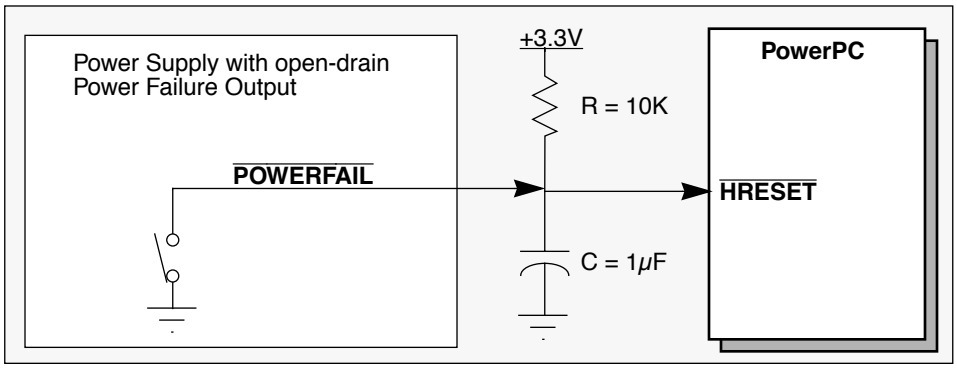


Figure 21. Simple Reset Controller

Because the drain on the **HRESET** signal is negligible, the simple equation t = RC suffices to calculate the necessary values. The above values shown give a 10 μs reset, which is sufficient for all bus speeds faster than 25 MHz.

Part 6 Power

In order to increase speeds without excessive heat loss, the newest, fastest PowerPC processors have cores which operate at low voltages. To remain compatible with external devices, the I/O cells have remained at 3.3V. This increases the complexity of a system somewhat by requiring multiple voltages levels.

Furthermore, as transistor counts rise in the processors, the static and transient current demands of the power supplies rise as well. Consequently, a well-designed, quiet and responsive power supply is a critical first step to a well-designed PowerPC-based system. There are many ways to derive power, ranging from batteries to radioisotope-thermocoupled generators. The two most popular methods are linear supplies and switching supplies, which are considered in further detail in sections 6.1 and 6.2.

6.1 Linear Regulators

Linear regulators operate by dissipating unwanted energy in the form of heat. With proper thermal management, linear regulators are very easy to design, inexpensive, and provide quiet, stable outputs. The disadvantages are the heat and the inability to generate higher voltages. Figure 22 shows an example of a linear 2.5-V power supply, similar to that used on the Excimer board.

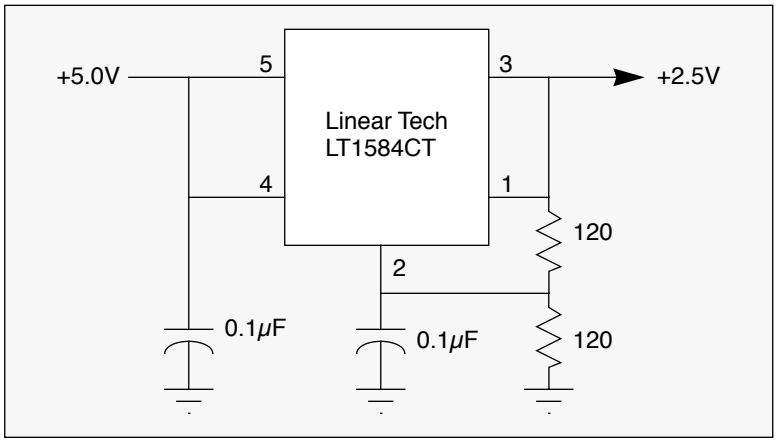
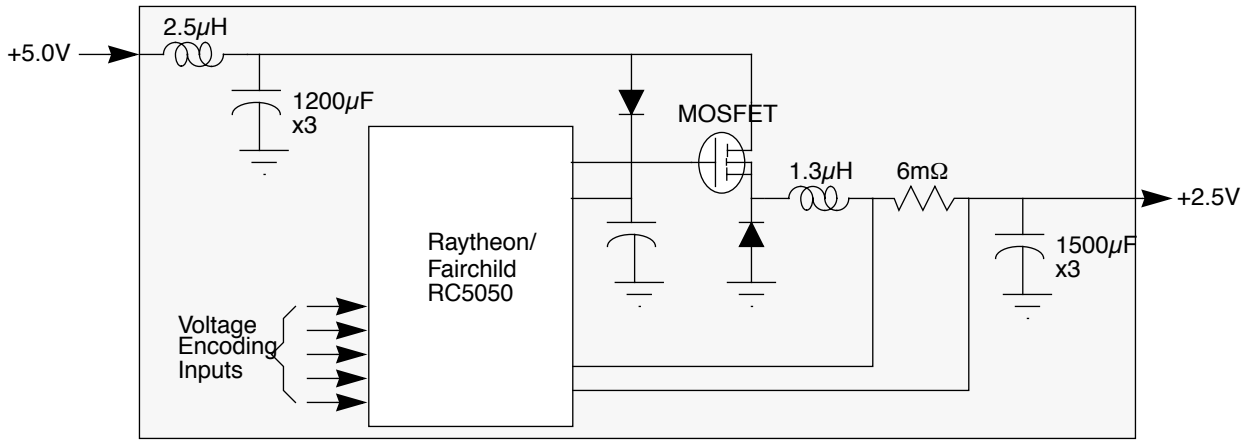


Figure 22. Excimer Linear Power Supply

6.2 Switchmode Regulators

An alternate method of providing other voltages is to use a switching power supply. These devices can efficiently convert high-voltage, low-current energy into low-voltage, high-current energy by storing it magnetically in an inductor. Switching power supplies require more complicated logic and careful design, but the rewards are that the efficiencies are high and that thermal dissipation is of little issue. Because switchers work basically by periodically dumping energy into a low-impedance load, clocking noise and transient effects can make for a noisy supply unless components are carefully selected. Figure 23 shows an example of a switching power supply.



NOTE: Not all details have been shown.

Figure 23. Simple Switching Power Supply

This switcher has a 5-bit digital input which allows the output voltage to be set in 0.1/0.05V increments in two ranges between 1.2V to 3.6V. This allows a single power supply to be easily programmed to meet current and future PowerPC processor requirements. In addition, the digital settings match those used on the PowerPC processor/cache module (interposer), which allows the processor to automatically select the desired voltage.

6.3 Power Supply Sequencing

Once consequence of multiple power supplies is that when power is initially applied, the voltage rails will ramp up at different rates depending upon the nature of the power supply, the type of load on each, and the manner in which the different voltages are derived. This can present a problem because the power supplies of a PowerPC processor have the following restrictions:

- VIN must not exceed OV_{DD} by more than 0.3V at any time including(requirement 1) during power-on reset.
- OV_{DD} must not exceed V_{DD}/AV_{DD} by more than 1.2V at any time including(requirement 2) during power-on reset.
- V_{DD}/AV_{DD} must not exceed OV_{DD} by more than 0.4V at any time including(requirement 3) during power-on reset.

On most PowerPC processors, the OV_{DD} (+3.3V I/O) load is typically less than 10% that of V_{DD} (+2.5V core) power, and the I/O cells are three-stated during reset, so a 3.3-V power supply may ramp up faster than the core voltage. Alternately, with more devices now operating at 3.3V, including the PCI bus, that power rail may be so loaded (from a system perspective) that the V_{DD} power will stabilize more quickly. Figure 24 shows an example of two possible power sequencing waveforms.

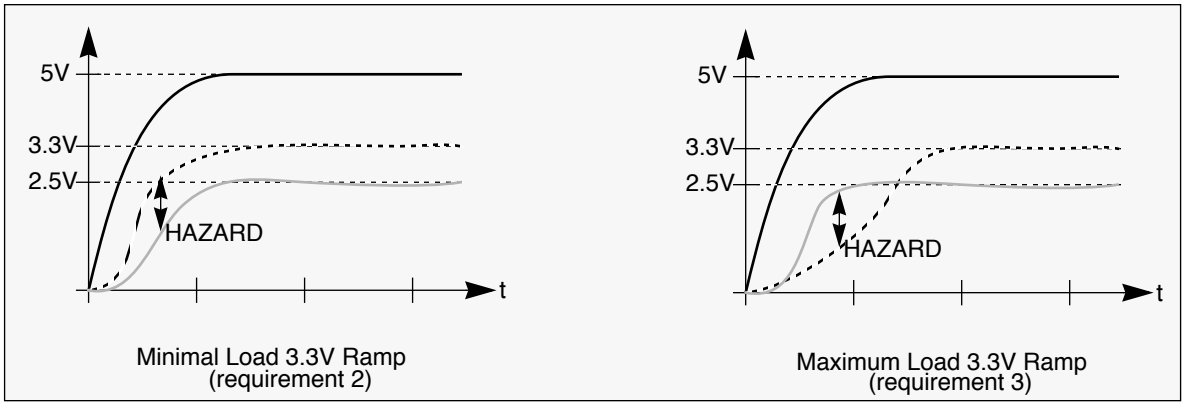


Figure 24. Power Supply Sequencing

It is virtually impossible to insure that all voltages ramp up to their steady state at an identical rate and at an identical time. In , either requirement 2 or requirement 3 will be violated depending only on the load on the 3.3-V power supply. Because such tracking is difficult to achieve, PowerPC processors may be subjected to a differential voltage between the V_{DD} and OV_{DD} power signals for up to 500 μs. If the power supplies cannot track within specified limits within this period, other means must be employed to correct the problem; otherwise, the long term reliability of the processor may be affected due to failure of internal protection circuitry.

One means of keeping two supplies synchronized is to use a so-called “bootstrap” diode between two power rails. An example is shown in .

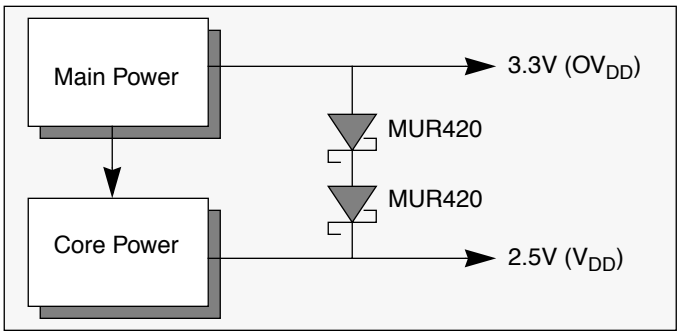


Figure 25. Bootstrap Diodes

The bootstrap diodes are selected such that a nominal V_{DD} will be sourced from the OV_{DD} power supply until the V_{DD} power supply becomes active. In the above example, a pair of MUR420 Schottky barrier diodes are connected in series; each has a forward voltage (V_F) of 0.6V at high currents, and so provides a 1.2V drop, maintaining the 2.5V power line at 2.1V.

Once the core power supply is stable at 2.5V, then the bootstrap diode(s) will be reverse biased and only a few nanoamperes of leakage current will flow.

NOTE: It is essential that the forward voltage be effective at the current levels needed by the processor; 1–3 amps or so depending on the PowerPC device. Many diodes have only a nominal V_F which falls off to nothing at high current; such devices are not acceptable.

6.4 Bypassing

A well-designed power supply will be quickly undermined if a poor bypassing system is used. Attention to bypassing is essential to eliminate poor ground-return paths through the PCB and to help quell transient noise and voltage drooping due to switching consideration.

High-frequency bypassing is provided by numerous 0.1 μF ceramic capacitors located near each power pin. Only surface mount devices may be used, and preferably in the smallest package possible (0805 or 0508—with power connections on the 'long' side). Each capacitor should have a direct via to the power or ground plane, with a short connection to the power pin.

On PowerPC devices in BGA packages, the solder pads connecting to power pins (balls) should be connected directly to a power or ground plane with a via. Since there are no pins, the bypass capacitors should surround the device on the bottom layer of the board. If placing components on the bottom of the board is not allowed, the next most preferable placement is to surround the part as close as possible to the BGA escape pattern.

In addition, a good design will include several "bulk" storage capacitors distributed around the PCB and connected to the V_{DD} and OV_{DD} power planes. These capacitors provide local energy storage for quick recharging of the smaller bypass capacitors, so the bulk capacitors should have a low equivalent series resistance (ESR) rating to ensure the quick response time necessary. Each bulk capacitor should be at least 100 μF, and there should be one device for every 20 high-frequency capacitors (more if they cannot be placed relatively close).

Part 7 Interrupts

The PowerPC processor has one standard interrupt signal ($\overline{\text{INT}}$) that can be connected to an external interrupt source if needed. This is in keeping with the RISC philosophy in which software manages (optional) highly complex details and hardware aims to be fast. As long as the interrupting device is level-sensitive, it can be wired directly to the processor's $\overline{\text{INT}}$ input (perhaps with an inverter, if necessary).

If extra interrupts are needed, the simplest manner is to merge all level-sensitive interrupts with a logic gate as shown in Figure 26.

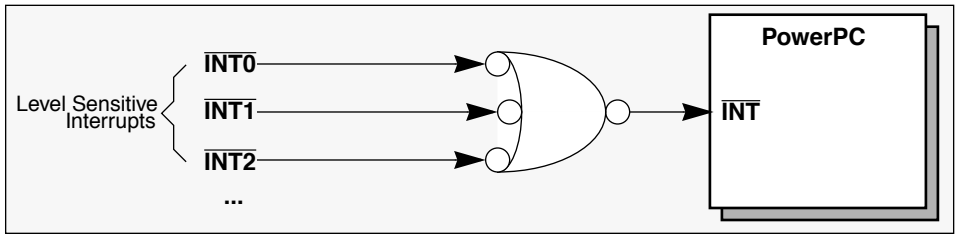


Figure 26. Simple Interrupt Merging

Software must poll all potential interrupting devices to determine which one (or more) has caused the interrupt and clear it. This approach does not allow any priority among interrupts, nor can any interrupt be masked unless the interrupting device provides a means to do so.

One way to quickly identify different interrupts is to assign them each an interrupt vector by reusing the special-purpose interrupts $\overline{\text{SMI}}$ and $\overline{\text{MCP}}$, as shown in Figure 27.

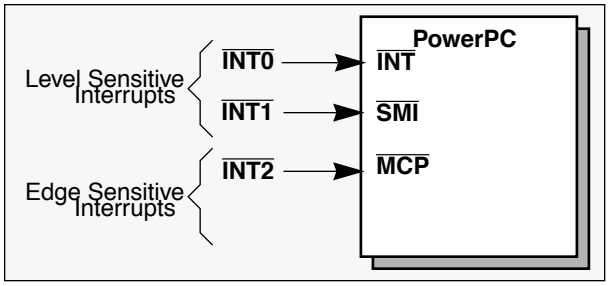


Figure 27. Interrupt Reuse

This approach does have several limitations for the **MCP** interrupt; in particular, the HID0[EMCP] bit and MSR[ME] enable bits must be properly set, and the interrupt remains edge-sensitive unless additional external hardware is used.

For systems needing a more traditional interrupt controller, many FPGA vendors offer IP cores which implement PC-style “8259” programmable interrupt controllers (PIC). There are sufficient resources in most FPGAs to include it with the memory controller by adding additional I/O controls, an 8-bit data bus, and **INT** output, and 1–-n interrupt inputs. Such an interrupt controller can include other advanced features such as edge-sensitive to level-sensitive conversion, and interrupt prioritizing and masking.

Excimer uses a very simple interrupt merging system, though provisions are in place to add programmable I/O to do interrupt masking.

The VHDL code for the Excimer interrupt controller is:

```
-------------------------------------------------------------------------------------------------
-- INT.VHD
--
-- INT() is a small interrupt controller for the Excimer project which
-- fits in some available gates of the Memory Controller (MC).
--
-- Copyright 1998, by Motorola Inc.
-- All rights reserved.
--
-- Author:   Gary Milliorn
-- Revision: 0.1
-- Date:     6/30/98
-- Notes:
--           All logic is active low when appended with a "_L".
--           Passed speedwave check 6/30/98.
-------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--=================================================================================================
-- INT
-------------------------------------------------------------------------------------------------
ENTITY INT is
    PORT( irq            : in     std_logic_vector( 0 to 3 );   -- interupt inputs (variable polarity)
          int_L          : buffer std_logic                     -- interrupt output.
        );

end; --PORT DEFINITION AND ENTITY

-------------------------------------------------------------------------------------------------
ARCHITECTURE BEHAVIOR OF INT is

BEGIN

    int_L <= '0' WHEN (  (irq(0) = '1')  or  (irq(1) = '1')        -- active high interrupts
                      or (irq(2) = '0')  or  (irq(3) = '0'))       -- active low interrupts.
                 ELSE '1';

END BEHAVIOR;
-------------------------------------------------------------------------------------------------
```

Part 8 COP

The common on-chip processor (COP) function of PowerPC processors allows a remote computer system (typically a PC with dedicated hardware and debugging software) to access and control the internal operations of the processor. While adding a COP connection to any PowerPC system adds little to no cost, it does add many benefits—breakpoints, watchpoints, register and memory examination/modification and other standard debugger features are possible through this interface.

The COP interface has a standard header for connection to the target system, based on the 0.025" square-post 0.100" centered header assembly (often called a “Berg” header). The connector typically has pin 14 removed as a connector key, as shown in Figure 28.

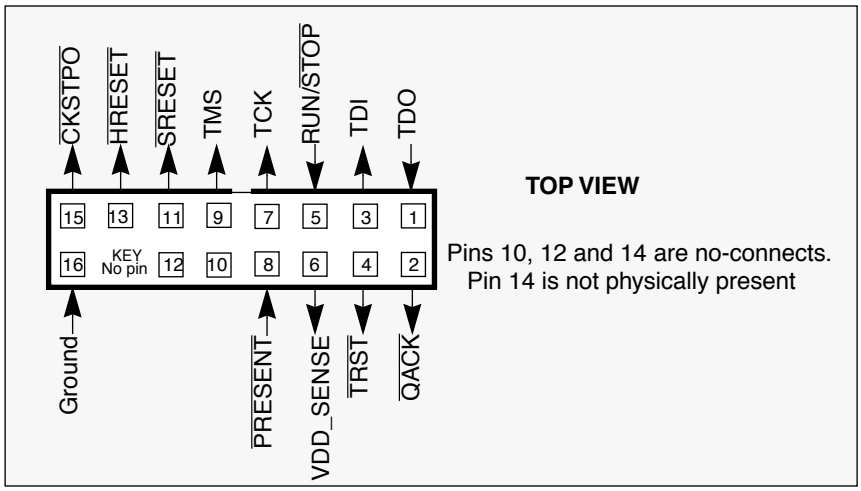


Figure 28. COP Connector Diagram

NOTE: There is no standardized way to number these headers; consequently, many different pin numbers have been observed on a variety of schematics. Some are numbered top-to-bottom then left-to-right, while others use left-to-right then top-to-bottom, while still others number the pins clockwise from pin one (as with an IC). Regardless of any local standardization, when adding a COP port to a system, insure that the signal placement follows that of Figure 28 when viewed from above the connector.

The COP interface connects primarily through the JTAG port of the processor, with some additional status monitoring signals. Table 6 shows the pin definitions.

Table 6. COP Pin Definitions

Pins	Signal	Connection	Applicable Processor	Special
1	**TDO**	**TDO**	All	See section 8.2.
2	**$\overline{\text{QACK}}$**	**$\overline{\text{QACK}}$**	603e, 603ev, 740, 750	
3	**TDI**	**TDI**	All	
4	**$\overline{\text{TRST}}$**	**$\overline{\text{TRST}}$**	All	Add 2K pulldown to ground. Must be merged with on-board **$\overline{\text{TRST}}$**, if any.
5	RUN/$\overline{\text{STOP}}$	**RUN**	604, 604e	Leave no-connect for all other processors.
6	VDD_SENSE	VDD	All	Add 2K pullup to VDD.

Table 6. COP Pin Definitions (Continued)

Pins	Signal	Connection	Applicable Processor	Special
7	**TCK**	**TCK**	All	
8	PRESENT	Optional	All	Add 10K pullup to VDD. May be used to separate JTAG scan chains; see section 8.2.
9	**TMS**	**TMS**	All	
10	N/A			
11	**$\overline{\text{SRESET}}$**	**$\overline{\text{SRESET}}$**	All	Merge with on-board **$\overline{\text{SRESET}}$**, if any.
12	N/A			
13	**$\overline{\text{HRESET}}$**	**$\overline{\text{HRESET}}$**	All	Merge with on-board **$\overline{\text{HRESET}}$**.
14	N/A		All	Key location; pin should be removed.
15	**$\overline{\text{CKSTPO}}$**	**$\overline{\text{CKSTPO}}$**	603e, 603ev, 740, 750	Add 10K pullup to VDD.
16	Ground	Digital Ground	All	

8.1 Merging Reset Signals

The COP port requires the ability to independently assert **$\overline{\text{HRESET}}$** or **$\overline{\text{TRST}}$** in order to fully control the processor. If the target system has independent reset sources, such as voltage monitors, watchdog timers, power supply failures, or push-button switches, then the COP reset signals must be merged into these signals with logic. It is not possible to just wire the reset signals together, damage to the COP system or the target system may occur.

The arrangement shown in Figure 29 allows the COP to independently assert **$\overline{\text{HRESET}}$** or **$\overline{\text{TRST}}$**, while insuring that the target can drive **$\overline{\text{HRESET}}$** as well. The pull-down resistor on **$\overline{\text{TRST}}$** insures that the JTAG scan chain is initialized during power-on if the COP is not attached; if it is, it is responsible for driving **$\overline{\text{TRST}}$** when needed.

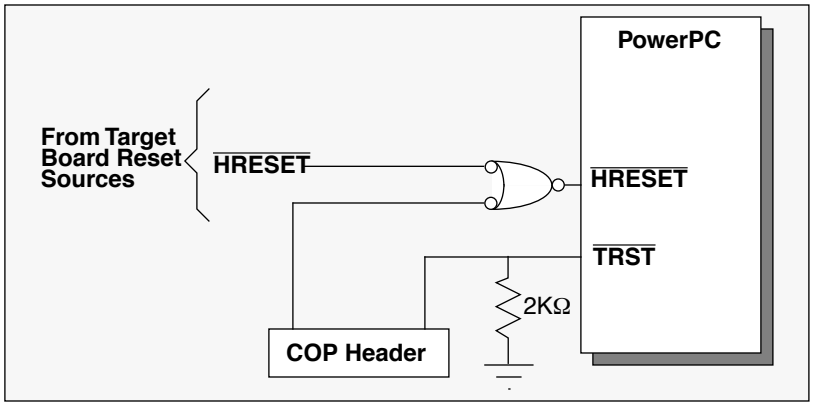


Figure 29. COP Reset Merging

8.2 Multiple Scan Chains

JTAG scan chains typically consist of numerous devices to perform in-circuit testing of printed circuit boards. Since some existing COP controller software may not be able to control the processor if any other device is present in the scan chain, it is often necessary to provide isolation for the PowerPC JTAG port.

Multiple scan chains is common on complex boards, so this is nothing new; however, for small systems it may be more desirable to provide an isolation capability that is only created when debugging is desired, and not while in mass production.

This isolation is shown in Figure 30 and can be done with logic, as in "Method 3", or manually with a removable jumper or zero-ohm resistor (or even an easily cut PCB trace).

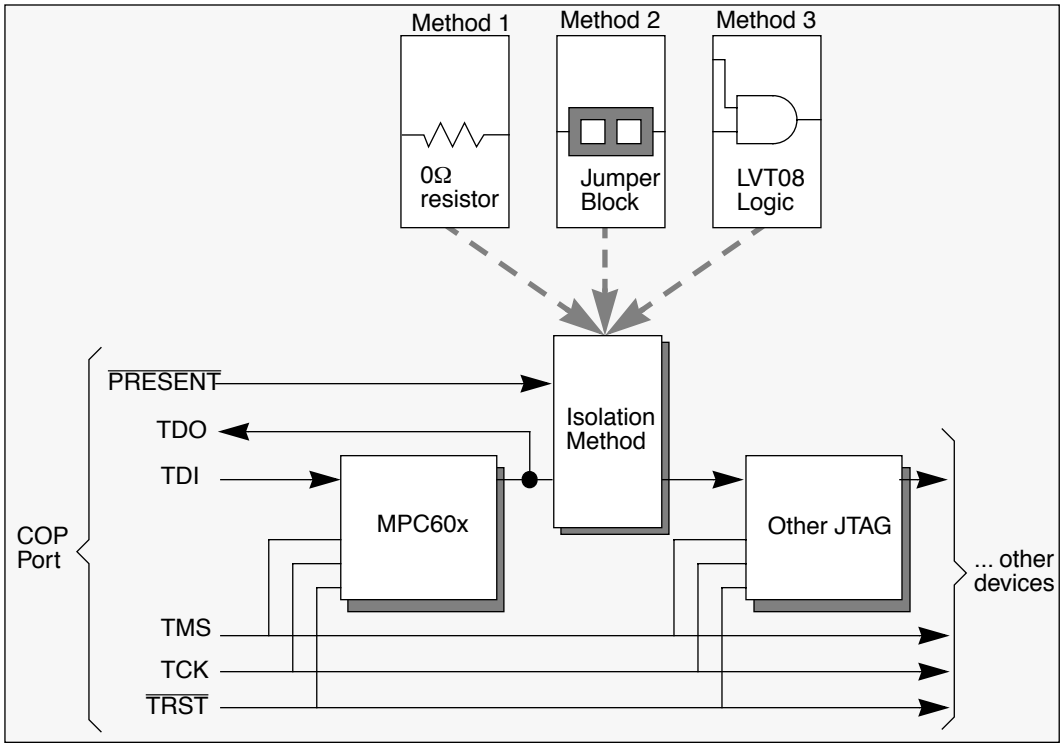


Figure 30. COP Isolation

As required by the IEEE 1189.1 (JTAG) standard, even though **TMS** and **TCK** will be active when COP commands are issued, the **TDI** chain for the rest of the system will float high, causing only IDLE commands to be issued to all other JTAG devices.

NOTE: Not all emulators assert the present signal. If "Method 3", the logic-controlled method, is used to separate the scan chain, insure that the chosen emulator will provide the $\overline{\text{PRESENT}}$ signal.

Part 9 Physical Layout

Figure 31 shows an example minimal system called Excimer; the size shown in an approximation of the actual size.

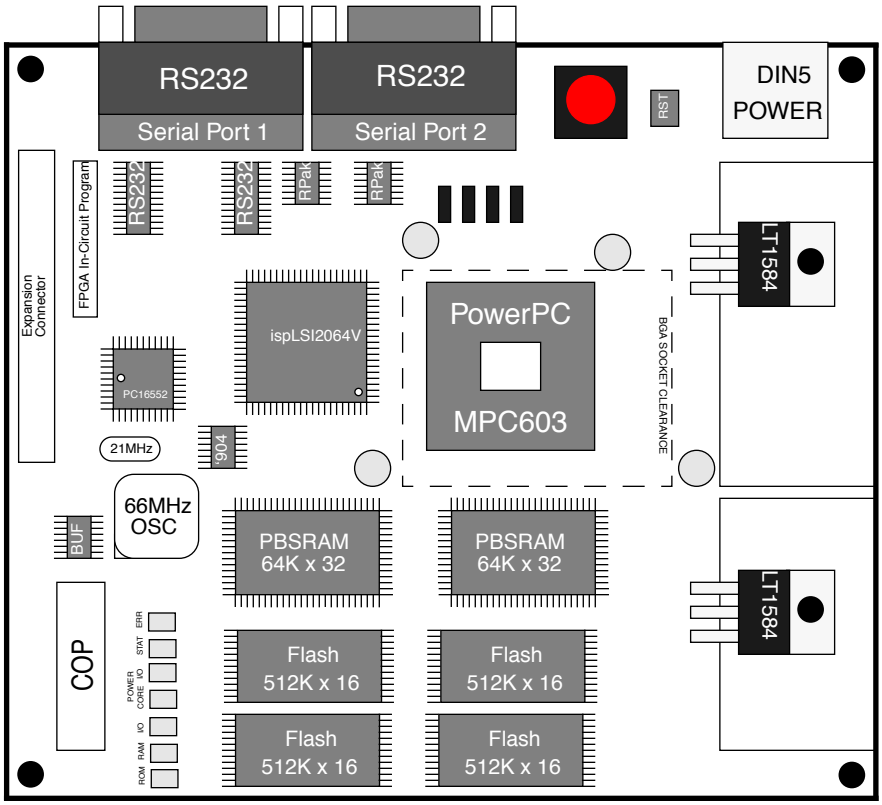


Figure 31. Excimer Minimal System Board

This design uses the standard 255-pin BGA pattern to allow any MPC603x or MPC604x device to be populated. An MPC750 design could be easily created by expanding the size for an additional two PBSRAM devices. A user-defined I/O area allows customer-specific interfaces to be attached. Miscellaneous discrete components are not shown.

Part 10 Conclusion

A PowerPC design can be easily implemented with a small amount of hardware by following the examples listed in this paper. The resulting system will exhibit fast memory access times and will allow benchmarking of various processors. If desired, the design can be enhanced with the following features:

- Stream accesses to the same page of SRAM
- Handle SRAM deselect in parallel with other accesses (even SRAM) to eliminate dead-time.
- Support burst flash memory
- Move cycle recognition into the state machine; this eliminates one clock latency on all memory cycles
- Allow address-only cycles

The possibilities are unlimited.

10.1 Reference Materials

Table 7 lists several documents which may be of use in learning to design a PowerPC system of any type.

Table 7. Reference Documentation

Document	Name	Why
MPC603EUM/AD Rev. 1	MPC603e and EC603e RISC Microprocessor User's Manual	Details on MPC603, MPC603e, and MPE603e interface.
MPC604EUM/AD	MPC604e RISC Microprocessor User's Manual	Details on MPC604/MPC604e interface.
MPC750UM/AD	MPC750 RISC Microprocessor User's Manual	Details on MPC750 and MPC740 interface. Details on back-side cache interface.
MPC106UM/AD	MPC106 PCI Bridge/Memory Controller User's Manual	Details on bus interface, and general information on memory controller design.
MPCPCMEC/D	Processor Cache Module Hardware Specifications	Details on power supply encoding and PCM socket (optional).

10.2 Resources

Table 8 lists many resources that are available to help understand and design PowerPC systems.

Table 8. Resources

What	Why	Where
Excimer Reference Design	Implementation of this application note; VHDL code file and schematics.	http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/index.html
Yellowknife X2, X4 Reference Designs	Examples of MPC60x systems and PCM modules.	http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/index.html
Application Notes	High speed design details	http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/index.html
PowerPMC750 Schematics	Example of interrupt controller.	http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/index.html

Motorola Literature Distribution Centers:

USA/EUROPE: Motorola Literature Distribution; P.O. Box 5405; Denver, Colorado 80217; Tel.: 1-800-441-2447 or 1-303-675-2140;
World Wide Web Address: http://ldc.nmd.com/

JAPAN: Nippon Motorola Ltd SPD, Strategic Planning Office 4-32-1, Nishi-Gotanda Shinagawa-ku, Tokyo 141, Japan Tel.: 81-3-5487-8488

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd Silicon Harbour Centre 2, Dai King Street Tai Po Industrial Estate Tai Po, New Territories, Hong Kong

Mfax™: RMFAX0@email.sps.mot.com; TOUCHTONE 1-602-244-6609; US & Canada ONLY (800) 774-1848;
World Wide Web Address: http://sps.motorola.com/mfax

INTERNET: http://motorola.com/sps

Technical Information: Motorola Inc. SPS Customer Support Center 1-800-521-6274; electronic mail address: crc@wmkmail.sps.mot.com.

Document Comments: FAX (512) 895-2638, Attn: RISC Applications Engineering.

World Wide Web Addresses: http://www.motorola.com/PowerPC/
http://www.motorola.com/netcomm/