Interfacing the Am29PLI60 to the Motorola Coldfire™ Processor

Application Note

July 2003

The following document refers to Spansion memory products that are now offered by both Advanced Micro Devices and Fujitsu. Although the document is marked with the name of the company that originally developed the specification, these products will be offered to customers of both AMD and Fujitsu.

Continuity of Specifications

There is no change to this document as a result of offering the device as a Spansion product. Any changes that have been made are the result of normal documentation improvements and are noted in the document revision summary, where supported. Future routine revisions will occur when appropriate, and changes will be noted in a revision summary.

Continuity of Ordering Part Numbers

AMD and Fujitsu continue to support existing part numbers beginning with "Am" and "MBM". To order these products, please use only the Ordering Part Numbers listed in this document.

For More Information

Please contact your local AMD or Fujitsu sales office for additional information about Spansion memory solutions.

Publication Number **22277** Revision **A** Amendment **0** Issue Date **November 1, 1998**

Interfacing the Am29PL160 to the Motorola Coldfire® Processor

Application Note

This application note describes a possible interface between the Motorola Coldfire® processor and the Am29PL160 page mode flash device. The design utilizes a wait state generator to assert a TA (Transfer Acknowledge) signal to terminate a bus read cycle.

Advantages of Page Mode Read Operations

The AM29PL160 Page Mode device allows for higher performance system operation by reducing the flash random access time from a typical 70 ns to lower than 25 ns for same page reads. By designing a "smart" interface between the processor and Flash system, same page reads can benefit from the reduced aggregate access times.

The Am29PL160 has an initial access time of 75 ns (at 100 pF loading), and subsequent accesses within the same page are at 25 ns (a page is defined as a memory region governed by Flash address bits A3-A19). Thus the initial read would require 1 or more wait states (depending on the bus frequency), and sequential access require fewer wait states (again, depending on frequency).

Knowing the bus frequency of the processor, the required number of wait states for initial and sequential read accesses can be calculated.

This document discusses a 33 MHz bus speed interface (with conservative timings), as bolded in Table 1.

Table 1. Possible Bus Timings

Frequency (MHz)	Period (ns)	Ideal Bus Read Cycle Timings	Conservative Bus Read Cycle Timings (10% margin)
16	62.5	1-0-0-0	1-0-0-0
25	40	2-0-0-0	2-0-0-0
33	**30.3**	**3-0-0-0**	**3-1-1-1**
40	25	3-0-0-0	4-1-1-1
66	15.2	5-2-2-2	6-3-3-3
90	11.1	7-3-3-3	8-3-3-3

Page Mode Interface Overview

The Page Mode controller (labeled PMC in Figure 1) consists of a single logic function block which has two functions: first, to compare incoming addresses governing Flash read requests, and second, to generate the appropriate number of wait states depending on the type of access (whether or not the access leaves the current page).

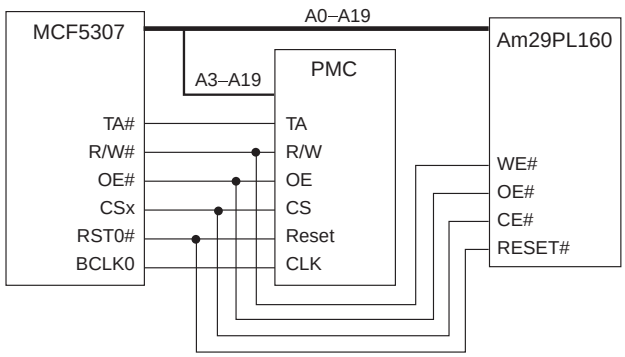

Figure 1. Page Mode Controller System Interface

The PMC interfaces between the processor and Flash device, with address and control signals being watched by the PMC as inputs. The only asserted output of the PMC is the TA signal, which indicates the completion of a read transfer cycle. The MCF5307 processor will hold the state of the bus until the TA signal is asserted, signaling the end of the read transfer cycle.

Page Mode Controller Operation

The function of the PMC is broken into two parts: first, it must detect and latch the incoming address pattern in order to determine if a read from Flash is occurring within the current page address. Since a page as defined as any memory address governed by A3 through A19 (A0 through A2 define a specific word within that page), it can detect a page transition whenever any of the address bits A3 through A19 change on subsequent read cycles. The Address Comparator sub-circuit of the PMC (see Figure 2) accomplishes this detection mechanism.

Publication# **22277** Rev: **A** Amendment/**0**
Issue Date: **November 1998**

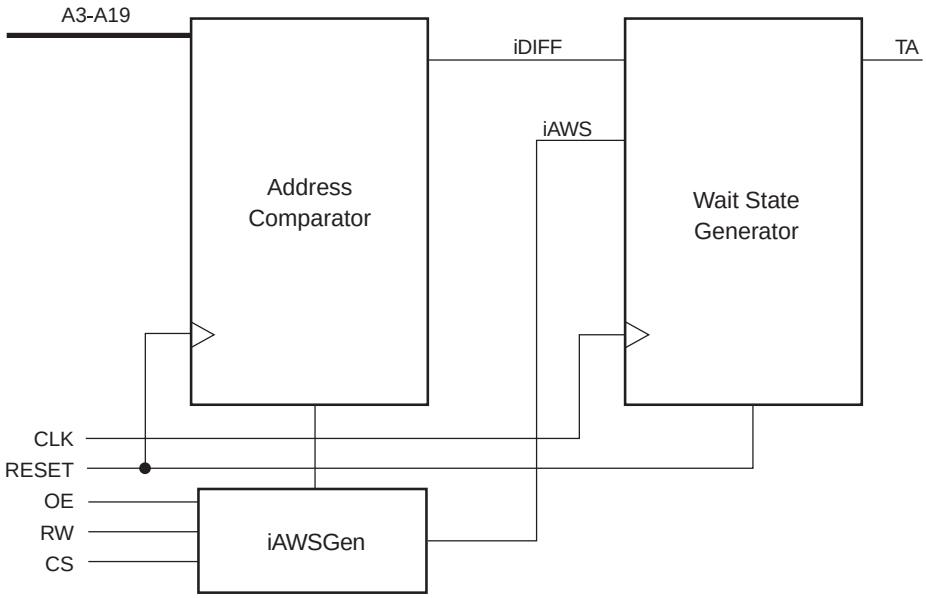

Figure 2. Page Mode Controller Block Diagram

Depending on whether or not a page transition is detected, the PMC must assert an appropriate number of wait states. Given the 3-1-1-1 read timings outlined in Table 1, if the PMC detects a page transition, it must assert at least 3 wait states in order to meet the 70 ns initial read access time of the PL160 device. If no transition was detected, it must assert only 1 wait state. The Wait State Generator sub-circuit of the PMC (see Figure 2) performs assertion of wait states.

The small block below the Address Comparator (labeled iAWSGen) is responsible for producing an internal signal named iAWS (which stands for AreWeSure?). This signal is a logical combination of the OE (Output Enable), RW (Read/Write), and CS (Chip Select) signals. The iAWS signal is asserted negative whenever the OE and CS signals are low, and the RW signal is high. This signal provides an internal enabling signal to the other logic functions so that they are only active when the Flash device is selected for reading.

Page Mode Controller VHDL Description

The entire PMC interface was designed and testing using VHDL, which provides a specification that is easily tested, simulated, and synthesized using software tools. By providing a VHDL representation of this circuit, it can be easily integrated into existing FPGA controllers or ASICs.

All VHDL code is displayed in fixed font format, with keywords boldfaced.

Address Comparator Design

The following text shows the VHDL code for the Address Comparator block.

```
-- Address Comparitor for PL interface to Coldfire Processors
-- Copyright AMD 1998

library ieee;
use ieee.std_logic_1164.all;

entity AddressComparitor is
   port( CLK, RESET: in std_logic;
         ADDIN: in std_logic_vector(16 downto 0);
         DIFF: out std_logic );
end AddressComparitor;

architecture Behavioral of AddressComparitor is
```

```
    -- PrevAddress stores the addres incoming on the last
    -- clock event

    signal PrevAddress: std_logic_vector(16 downto 0);

    begin
    ADDCOMP: process(CLK, RESET, ADDIN)

      begin

        if (RESET = '0') then
          PrevAddress <= "00000000000000000";  -- clear internal regs
        elsif rising_edge(CLK) then
          if (ADDIN = PrevAddress) then
              DIFF <= '1';       -- Don't assert DIFF if addresses match
          else
             DIFF <= '0';
          end if;

           PrevAddress <= ADDIN;  -- Save current address for next time
        end if;

      end process;

  end Behavioral;
  -- end code
```

The code describing the Address Comparator is very simple. The comparator accepts 3 incoming signals: a clock, a reset signal, and an incoming address (17 lines). The outgoing (or driven) signal is called DIFF, which is asserted low whenever the current address does not match the previous address. The PrevAddress signal is used to force a storage element for this entity (so the previous address is saved on the rising edge of each clock).

On each incoming clock, the current address and the previous address are compared. If they are the same (meaning that the system is accessing the same page), then the output signal DIFF is driver high, indicated a page hit. If the addresses differ in any bit, the DIFF signal is driven low indicating a page miss.

Note: *Since the MCF5307 will hold the state of the bus until assertion of the TA signal, it is not required to latch both the current state as well as the previous state of the bus. By only latching one set of addresses, the synthesized circuit is saved over 17 flip-flops.*

Whenever the RESET signal is asserted low, the state of the address latch will be cleared to all zeroes. This will force the first Flash read cycle to be properly constrained by the 3 wait state timings.

Wait State Generator Design

The Wait State Generator comprises of a simple state machine, and state decoding equations. It features a one-hot encoded design, which provides for a high speed interface in systems which may have high interconnect latencies by minimizing the complexity of the next-state decoding circuitry.

The bubble chart for the state machine design in shown in Figure 3. It consists of seven valid states: a waiting state, which is the default state, 4 transition states (to apply the wait states), and 2 assertion states where the TA signal is driven low.

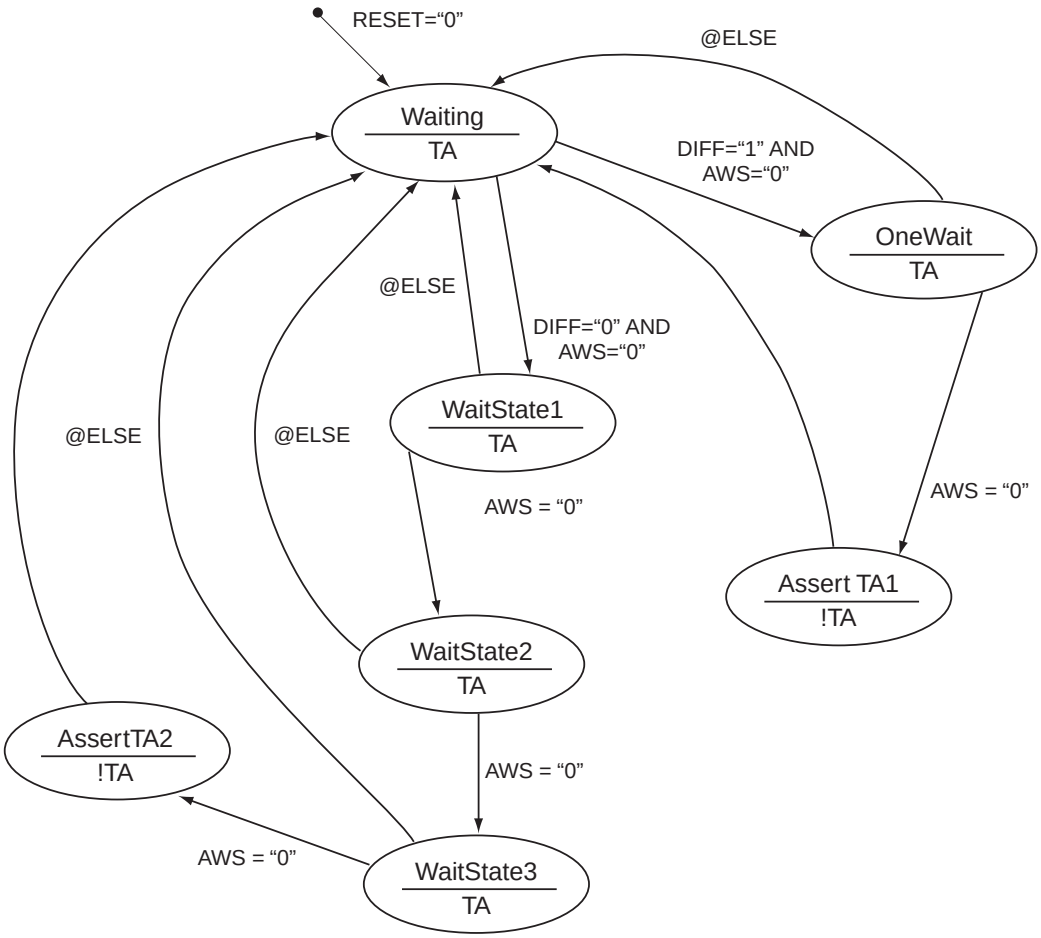

Figure 3. Wait State Generator Bubble Chart

From the waiting state, the Wait State Generator will either apply one or three wait states, depending on the logic level of the DIFF input signal. State transitions are overridden by the AWS input signal, so that the state machine will only switch states whenever the AWS signal is low (this is to prevent the machine from applying wait states while the page mode device is not being accessed by the system). The machine will reset to the waiting state whenever the RESET input is asserted low, or whenever the AWS signal is held high. This assures that the state machine is always in the waiting state whenever the Flash is accessed on non-sequential bus cycles (to prevent the state machine from applying the wrong number of wait states).

The following text show the VHDL representation of the Wait State Generator.

```
-- Wait State Generator for PL
interface to Coldfire Processors
-- Copyright AMD 1998

library ieee;
use ieee.std_logic_1164.all;

entity WaitStateGenerator is
   port( CLK, AWS, DIFF, RESET: in
std_logic;
         TA: out std_logic);
end WaitStateGenerator;

architecture Behavioral of Wait-
StateGenerator is

  -- State machine utilizes one-hot
encoding to
  -- minimize interconnect delays

  subtype states is
std_logic_vector(6 downto 0);
  constant AssertTA1: states :=
"0000001";
```

```
  constant AssertTA2: states :=
"0000010";
  constant OneWait: states :=
"0000100";
  constant Waiting: states :=
"0001000";
  constant WaitState1: states :=
"0010000";
  constant WaitState2: states :=
"0100000";
  constant WaitState3: states :=
"1000000";

  signal CurrentState, NextState:
states;

  begin

  -- First, set up SM transition
registers
  STREG: process( CLK )

    begin

      if rising_edge(CLK) then
         CurrentState <= NextState;
      end if;

    end process;

  -- Second, outline state transi-
tions
  STTRANS: process(CurrentState,
AWS, DIFF,RESET)

    begin

      if (RESET = '0' or
RESET'Event) then
         NextState <= Waiting;
         TA <= '1';
      end if;

    if CurrentState(3) = '1' then
-- Waiting

         TA <= '1';

         if (DIFF = '1' and AWS =
'0') then
           NextState <= OneWait;
        elsif (DIFF = '0' and AWS =
'0') then
           NextState <= WaitState1;
         else
           NextState <= Waiting;
        end if;

      end if;

    if CurrentState(2) = '1' then
-- OneWait

         TA <= '1';

         if (AWS = '0') then
           NextState <= AssertTA1;
        else
           NextState <= Waiting;
        end if;

      end if;

    if CurrentState(0) = '1' then
-- AssertTA1

         TA <= '0';

         NextState <= Waiting;

      end if;

    if CurrentState(4) = '1' then
-- WaitState1

         TA <= '1';

         if (AWS = '0') then
           NextState <= WaitState2;
        else
           NextState <= Waiting;
        end if;

      end if;

    if CurrentState(5) = '1' then
-- WaitState2

         TA <= '1';

         if (AWS = '0') then
           NextState <= WaitState3;
        else
           NextState <= Waiting;
        end if;

      end if;

    if CurrentState(6) = '1' then
-- WaitState3
```

```
        TA <= '1';

        if (AWS = '0') then
          NextState <= AssertTA2;
        else
          NextState <= Waiting;
        end if;

      end if;

    if CurrentState(1) = '1' then
-- AssertTA2
        TA <= '0';

        NextState <= Waiting;

      end if;

    end process;

end Behavioral;
-- end code
```

The Complete Page Mode Controller

All that remains is to instantiate the two main components of the Page Mode Controller into a single design entity. This entity contains a single Wait State Generator, a single Address Comparator, and a simple logic function to produce the internal iAWS signal.

The following text shows the VHDL code that represents the Page Mode Controller.

The VHDL description of the PMC takes a structural form, merely serving to connect the two components with internal signals. It also calculates the iAWS signal used to enable the two components.

```
-- Page Mode Controller VHDL Code
-- Copyright AMD 1998

library ieee;
use ieee.std_logic_1164.all;

use work.AddressComparitor;
use work.WaitStateGenerator;

entity PageModeController is
  port( pmcCLK, pmcRESET: in
std_logic;
        pmcCS, pmcRW, pmcOE: in
std_logic;
        pmcADDIN: in
std_logic_vector(16 downto 0);
        pmcTA: out std_logic);
end PageModeController;

architecture Structural of PageMo-
deController is

  component AddressComparitor
    port ( CLK, RESET: in
std_logic;
           ADDIN: in
std_logic_vector(16 downto 0);
           DIFF: out std_logic);
  end component;

  component WaitStateGenerator
    port (CLK, AWS, DIFF, RESET: in
std_logic;
          TA: out std_logic);
  end component;

  -- Local interconnect signals
  signal iAWS, iDIFF: std_logic;

begin

  -- Instantiate our low level
entities

  WSG: WaitStateGenerator port map
 ( CLK => pmcCLK,

 RESET => pmcRESET,

DIFF => iDIFF, AWS => iAWS,

TA => pmcTA );
  AC: AddressComparitor port map (
CLK => pmcCLK, RESET => pmcRESET,

DIFF => iDIFF,

ADDIN => pmcADDIN );

  PMCFUNC: process( pmcOE, pmcRW,
pmcCS, pmcCLK, pmcRESET, pmcADDIN)

  begin
```

```
      -- Calc the iAWS signal based
on input control signals

      iAWS <= (pmcOE or pmcCS or
(not pmcRW));
```

```
    end process;

end Structural;
-- end code
```

Simulation of the Page Mode Controller

Simulation of the PMC was performed using the VSS® tool from Synopsys. The input clock frequency was set to 33 MHz, and simulated bus cycles are applied to the PMC to test the TA output signal. The results of the simulation are shown in Figure 4.

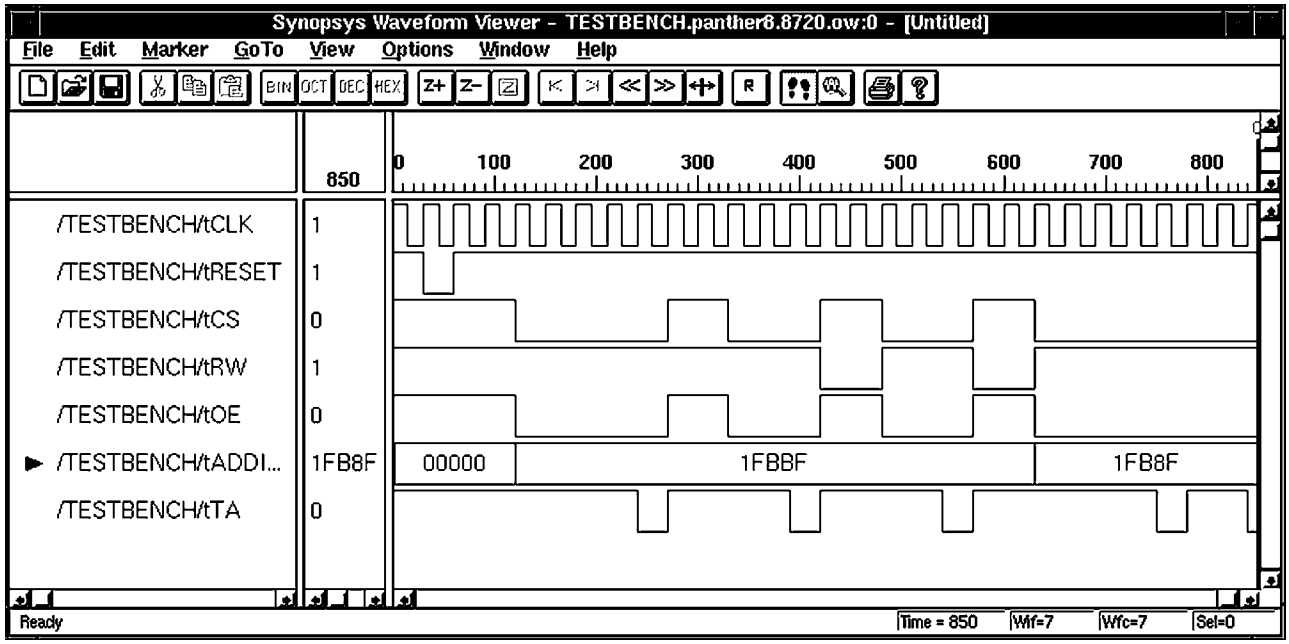

Figure 4. Simulation Results

The simulation results in Figure 4 show four separate events. The first valid Flash access (a valid flash access is described by the conditions CS low, RW high, OE low, and an address on the address lines) occurs at T1 (120 ns). The TA signal is then asserted 3 wait states (or 3 rising clock edges) after event T1. Event T2 (at time 330 ns) shows another Flash access of the same page (0x1FBBF). Since this access falls within the same page, the TA signal is asserted low after one wait state (or 1 rising clock edge) after event T2. Event T3 is the same as event T2, only it occurs at time 480 ns.

Event T4 shows the next Flash access, but this time to a different page (0x1FB8F). Since this is a page miss, the TA signal is asserted after 3 wait states (or 3 rising clock edges).

Of important note is the assertion of the RESET signal prior to the initial access of the Flash device. It is important to assert this signal for at least one clock period prior to the first Flash access, to ensure that all flip-flip storage elements are cleared to predetermined states (so that the Address Comparator latches are cleared to zeroes, and the initial state of the Wait State Generator is Waiting).

Test Bench Source Code

The following text shows the VHDL source for the test bench used to generate the simulation in Figure 4.

```
-- Test Bench for Page Mode Con-
troller
-- Copyright AMD 1998
```

```
library ieee, std;
use std.textio.all;
use ieee.std_logic_1164.all;
```

```
use work.PageModeController;

entity testbench is
end testbench;

architecture test of testbench is
  -- Create a Page Mode Controller

  component PageModeController
    port( pmcCLK, pmcRESET: in
std_logic;
          pmcCS, pmcRW, pmcOE: in
std_logic;
          pmcADDIN: in
std_logic_vector(16 downto 0);
          pmcTA: out std_logic);
  end component;

  -- Local signals
  signal tCLK, tRESET, tCS, tRW,
tOE: std_logic;
  signal tADDIN:
std_logic_vector(16 downto 0);
  signal tTA: std_logic;

begin

  -- Create instance of low level
entity
  PMC1: PageModeController port map
(pmcCLK => tCLK,

pmcRESET => tRESET,

pmcCS => tCS, pmcOE => tOE,

pmcRW => tRW, pmcTA => tTA,

pmcADDIN => tADDIN);

  -- First, set up our running 33
MHz clock
  process
  begin
    tCLK <= '0';
    wait for 0 ns;
    while true loop
      tCLK <= '1';
      wait for 15 ns;
      tCLK <= '0';
      wait for 15 ns;
    end loop;
  end process;

  -- Assert our test stimulus
  process
  begin
    tRESET    <=   '1';   -- Assert
all false
    tADDIN    <=
"0000000000000000";
    tOE     <=  '1';
    tCS     <=  '1';
    tRW     <=  '1';
    wait for 30 ns;

    tRESET    <=   '0';   -- Assert
Reset for 1 clock
    wait for 30 ns;    -- to clear
comparitor

    tRESET    <=   '1';   -- Turn
off Reset
    wait for 60 ns;

    -- We can now apply bus stimu-
lus, using this definition:
    -- A valid read cycle occurs
when:
    -- 1) OE is low
    -- 2) RW is high
    -- 3) CE is low
    -- 4) Valid address on bus

    -- On each read, we will wait
for the TA signal to be
    -- asserted

    tCS <= '0';
    tOE <= '0';
    tRW <= '1';

    tADDIN <= "11111101110111111";
    wait until (tTA = '0');    --
This should be a 3WS read
    wait for 30 ns;

    tCS <= '1';
    tOE <= '1';
    tRW <= '1';

    tADDIN <= "11111101110111111";
    wait for 60 ns; -- read opera-
tion not to flash

    tCS <= '0';
    tOE <= '0';
    tRW <= '1';

    tADDIN <= "11111101110111111";
    wait until (tTA = '0');  --
read same address - should be 1WS
```

```
        wait for 30 ns;

        tCS <= '1';
        tOE <= '1';
        tRW <= '0';
        wait for 60 ns;

        tCS <= '0';
        tOE <= '0';
        tRW <= '1';

        tADDIN <= "1111110111011111";
        wait until (tTA = '0');  --
    read same address - should be 1WS
        wait for 30 ns;

        tCS <= '1';
        tOE <= '1';
        tRW <= '0';
        wait for 60 ns;

        tCS <= '0';
        tOE <= '0';
        tRW <= '1';

        tADDIN <= "1111110111000111";
        wait until (tTA = '0');  --
    read diff address - should be 3WS
        wait for 30 ns;

        wait for 300 ns;

        wait for 1 ns;

        wait;

      end process;

    end test;
    -- end code
```

Other Considerations

The VHDL representation of the Page Mode Controller can be considered an ideal system. System margin, interconnect latency, and internal signal/gate delays are not considered due to the wide variation in synthesis techniques. For a more accurate simulation, VHDL after statements can be added to next state and signal assignments for a more accurate circuit simulation. For example, the statement:

```
iAWS <= (pmcOE or pmcCS or (not pmcRW));
```

could be replaced by:

```
iAWS <= (pmcOE or pmcCS or (not pmcRW))
after 2 ns;
```

This would more accurately simulate a typical gate delay in a FPGA or ASIC implementation. Appropriate values can be substituted depending on the implementation method used by the system designer.