CHAPTER 10

EXTERNAL SRAM

10.1 INTRODUCTION

Random access memory (RAM) is used for massive storage in a digital system since a RAM cell is much simpler than an FF cell. A commonly used type of RAM is the asynchronous static RAM (SRAM). Unlike a register, in which the data is sampled and stored at an edge of a clock signal, accessing data from an asynchronous SRAM is more complicated. A read or write operation requires that the data, address, and control signals be asserted in a specific order, and these signals must be stable for a certain amount of time during the operation.

It is difficult for a synchronous system to access an SRAM directly. We usually use a *memory controller* as the interface, which takes commands from the main system synchronously and then generates properly timed signals to access the SRAM. The controller shields the main system from the detailed timing and makes the memory access appears like a synchronous operation. The performance of a memory controller is measured by the number of memory accesses that can be completed in a given period. While designing a simple memory controller is straightforward, achieving optimal performance involves many timing issues and is quite difficult.

The S3 board has two 256K-by-16 asynchronous SRAM devices, which total 1M bytes. In this chapter, we demonstrate the construction of a memory controller for these devices. Since the timing characteristics of each RAM device are different, the controller is applicable only to this particular device. However, the same design principle can be used for similar

FPGA Prototyping by VHDL Examples. By Pong P. Chu

SRAM devices. The Xilinx Spartan-3 device also contains smaller embedded memory blocks. The use of this memory is discussed in Chapter 11.

10.2 SPECIFICATION OF THE IS61LV25616AL SRAM

10.2.1 Block diagram and I/O signals

The S3 board has two IS61LV25616AL devices, which are 256K-by-16 SRAM manufactured by Integrated Silicon Solution, Inc. (ISSI). A simplified block diagram is shown in Figure 10.1(a). This device has an 18-bit address bus, `ad`, a bidirectional 16-bit data bus, `dio`, and five control signals. The data bus is divided into upper and lower bytes, which can be accessed individually. The five control signals are:

- `ce_n` (chip enable): disables or enables the chip
- `we_n` (write enable): disables or enables the write operation
- `oe_n` (output enable): disables or enables the output
- `lb_n` (lower byte enable): disables or enables the lower byte of the data bus
- `ub_n` (upper byte enable): disables or enables the upper byte of the data bus

All these signals are active low and the `_n` suffix is used to emphasize this property. The functional table is shown in Figure 10.1(b). The `ce_n` signal can be used to accommodate memory expansion, and the `we_n` and `oe_n` signals are used for write and read operations. The `lb_n` and `ub_n` signals are used to facilitate the byte-oriented configuration.

In the remainder of the chapter, we illustrate the design and timing issues of a memory controller. For clarity, we use one SRAM device and access the SRAM in 16-bit word format. This means that the `ce_n`, `lb_n`, and `ub_n` signals should always be activated (i.e., tied to '0'). The simplified functional table is shown in Figure 10.1(c).

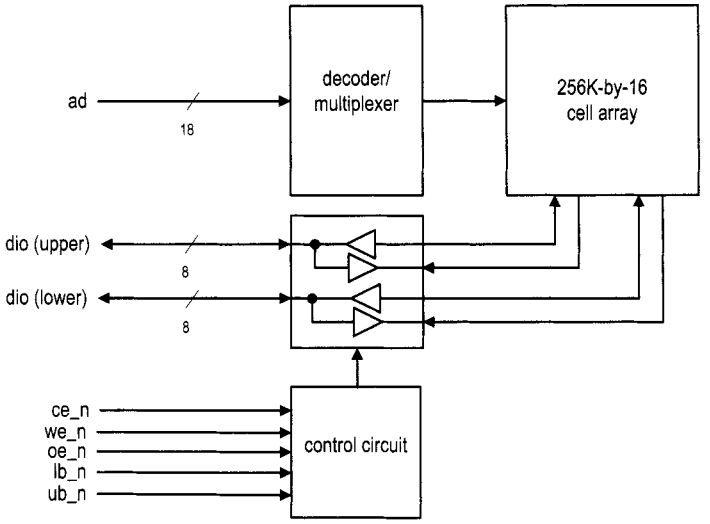
10.2.2 Timing parameters

The timing characteristics of an asynchronous SRAM are quite complex and involve more than two dozen parameters. We concentrate only on a few key parameters that are relevant to our design.

The simplified timing diagrams for two types of read operations are shown in Figure 10.2(a) and (b). The relevant timing parameters are:

- t_{RC}: read cycle time, the minimal elapsed time between two read operations. It is about the same as t_{AA} for SRAM.
- t_{AA}: address access time, the time required to obtain stable output data after an address change.
- t_{OHA}: output hold time, the time that the output data remains valid after the address changes. This should not be confused with the hold time of an edge-triggered FF, which is a constraint for the `d` input.
- t_{DOE}: output enable access time, the time required to obtain valid data after `oe_n` is activated.
- t_{HZOE}: output enable to high-Z time, the time for the tri-state buffer to enter the high-impedance state after `oe_n` is deactivated.
- t_{LZOE}: output enable to low-Z time, the time for the tri-state buffer to leave the high-impedance state after `oe_n` is activated. Note that even when the output is no longer in the high-impedance state, the data is still invalid.

Values of these parameters for the IS61LV25616AL device are shown in Figure 10.2(c).



(a) Block diagram

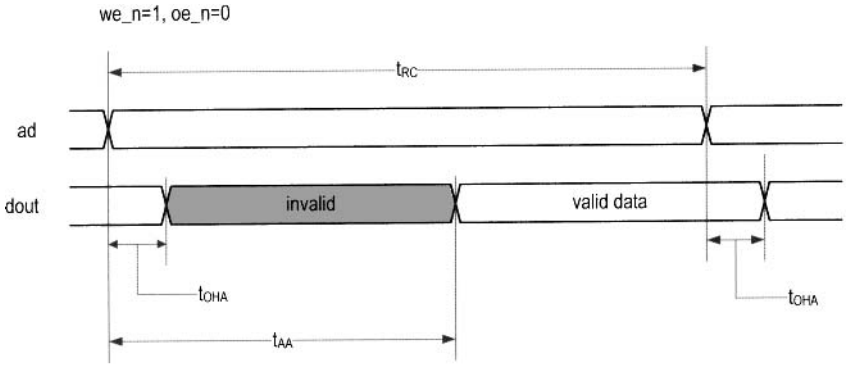
Operation	ce_n	we_n	oe_n	lb_n	ub_n	dio (lower)	dio (upper)
disabled	1	-	-	-	-	Z	Z
	0	1	1	-	-	Z	Z
	0	-	-	1	1	Z	Z
read	0	1	0	0	1	data out	Z
	0	1	0	1	0	Z	data out
	0	1	0	0	0	data out	data out
write	0	0	-	0	1	data in	Z
	0	0	-	1	0	Z	data in
	0	0	-	0	0	data in	data in

(b) Functional table

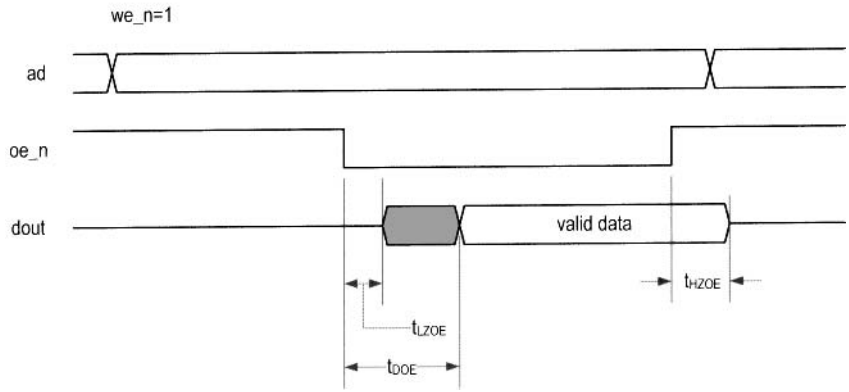
Operation	we_n	oe_n	dio (16 bits)
output disabled	1	1	Z
read 16-bit word	1	0	data out
write 16-bit word	0	-	data in

(c) Simplified functional table

Figure 10.1 Block diagram and functional table of the ISSI 256K-by-16 SRAM.



(a) Timing diagram of an address-controlled read cycle

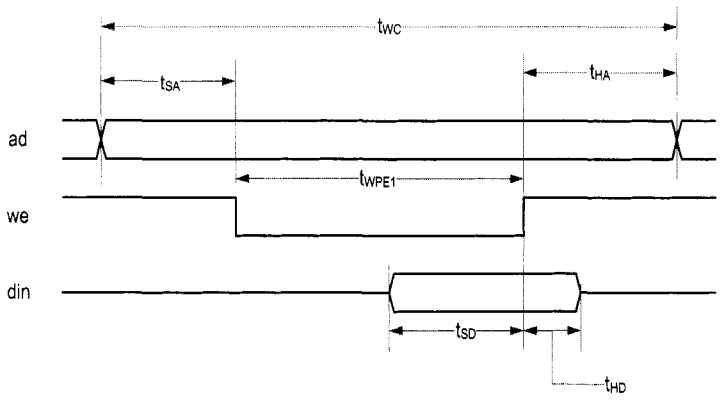


(b) Timing diagram of an oe_n-controlled read cycle

parameter		min	max
t_{RC}	read cycle time	10	–
t_{AA}	address access time	–	10
t_{OHA}	output hold time	2	–
t_{DOE}	output enable access time	–	4
t_{HZOE}	output enable to high-Z time	–	4
t_{LZOE}	output enable to low-Z time	0	–

(c) Timing parameters (in ns)

Figure 10.2 Timing diagrams and parameters of a read operation.



(a) Timing diagram of a write cycle

parameter		min	max
t_{WC}	write cycle time	10	–
t_{SA}	address setup time	0	–
t_{HA}	address hold time	0	–
t_{PWE1}	we_n pulse width	8	–
t_{SD}	data setup time	6	–
t_{HD}	data hold time	0	–

(b) Timing parameter (in ns)

Figure 10.3 Timing diagram and parameters of a write operation.

The simplified timing diagram for a we_n-controlled write operation is shown in Figure 10.3(a). The relevant timing parameters are:

- t_{WC}: write cycle time, the minimal elapsed time between two write operations.
- t_{SA}: address setup time, the minimal time that the address must be stable before we_n is activated.
- t_{HA}: address hold time, the minimal time that the address must be stable after we_n is deactivated.
- t_{PWE1}: we_n pulse width, the minimal time that we_n must be asserted.
- t_{SD}: data setup time, the minimal time that data must be stable before the latching edge (the edge in which we_n moves from '0' to '1').
- t_{HD}: data hold time, the minimal time that data must be stable after the latching edge.

The values of these parameters for the IS61LV25616AL device are shown in Figure 10.3(b). The complete timing information can be found in the data sheet of the IS61LV25616AL device.

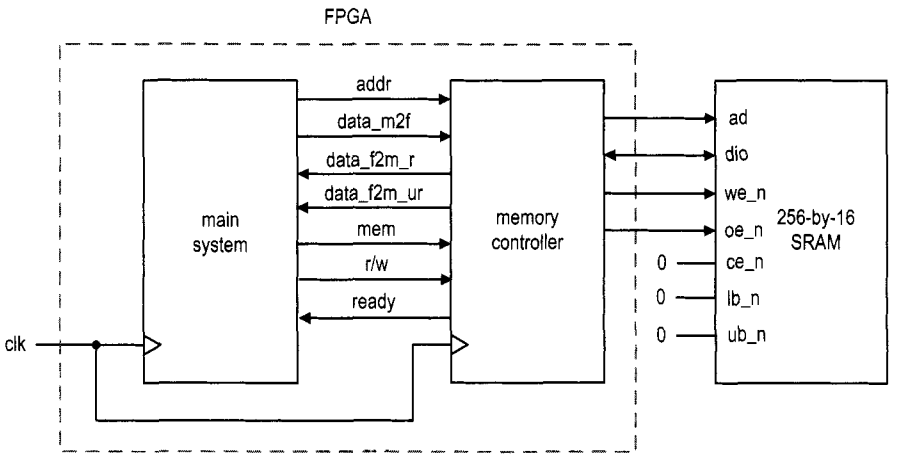


Figure 10.4 Role of an SRAM memory controller.

10.3 BASIC MEMORY CONTROLLER

10.3.1 Block diagram

The role of a memory controller and its I/O signals are shown in Figure 10.4. The signals to the SRAM side are discussed in Section 10.2.1. The signals to the main system side are:

- `mem`: is asserted to '1' to initiate a memory operation.
- `rw`: specifies whether the operation is a read ('1') or write ('0') operation.
- `addr`: is the 18-bit address.
- `data_f2s`: is the 16-bit data to be written to the SRAM (the `_f2s` suffix stands for FPGA to SRAM).
- `data_s2f_r`: is the 16-bit registered data retrieved from the SRAM (the `_s2f` suffix stands for SRAM to FPGA).
- `data_s2f_ur`: is the 16-bit unregistered data retrieved from SRAM.
- `ready`: is a status signal indicating whether the controller is ready to accept a new command. This signal is needed since a memory operation may take more than one clock cycle.

The memory controller basically provides a "synchronous wrap" around the SRAM. When the main system wants to access the memory, it places the address and data (for a write operation) on the bus and activates the command (i.e., the `mem` and `rw` signals). At the rising edge of the clock, all signals are sampled by the memory controller and the desired operation is performed accordingly. For a read operation, the data becomes available after one or two clock cycles.

The block diagram of a memory controller is shown in Figure 10.5. Its data path contains one address register, which stores the address, and two data registers, which store the data from each direction. Since the data bus, `dio`, is a bidirectional signal, a tri-state buffer is needed. The control path is an FSM, which follows the timing diagrams and specifications in Figures 10.2 and 10.3 to generate a proper control sequence.

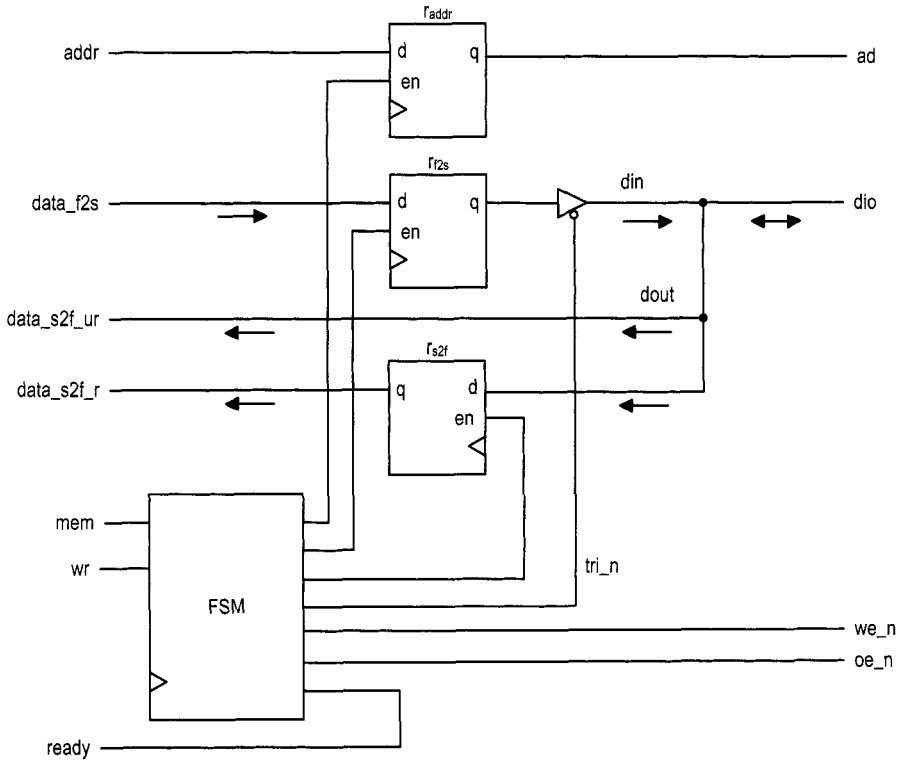


Figure 10.5 Block diagram of a memory controller.

10.3.2 Timing requirement

Although the timing diagrams appear to be complicated at first glance, the control sequences are fairly simple. Let us first consider a read cycle. The `we_n` should be deactivated during the entire operation. Its basic operation sequence is:

1. Place the address on the `ad` bus and activate the `oe_n` signal. These two signals must be stable for the entire operation.
2. Wait for at least t_{AA}. The data from the SRAM becomes available after this interval.
3. Retrieve the data from `dio` and deactivate the `oe_n` signal.

We use the `we_n`-controlled write cycle in our design, as shown in Figure 10.3(a). The basic operation sequence is:

1. Place the address on the `ad` bus and data on the `dio` bus and activate the `we_n` signal. These signals must be stable for the entire operation.
2. Wait for at least t_{PWE1}.
3. Deactivate the `we_n` signal. The data is latched to the SRAM at the '0'-to-'1' transition edge.
4. Remove the data from the `dio` bus.

Note that t_{HD} (data hold time after write ends) is 0 ns for this SRAM, which implies that it is theoretically possible to remove the data and deactivate `we_n` simultaneously. However, because of the variations in propagation delays, this condition cannot be guaranteed in a

real circuit. To achieve proper latching, we need to ensure that the we_n signal is always deactivated first.

10.3.3 Register file versus SRAM

We discuss the design of a register file in Section 4.2.3. Its basic storage elements are D FFs and thus it is completely synchronous. Although a memory controller wraps the SRAM in a synchronous interface, there are several differences:

- A register file usually has one write port and multiple read ports.
- The read and write ports of a register file can be accessed at the same time (i.e., the read and write operations can be done at the same time).
- Writing to a register takes only one clock cycle.
- Data from a register's read ports is always available and the read operation involves no clock or additional control signals.

In summary, a register file is faster and more flexible. However, due to the circuit size of an FF, a register file is feasible only for small storage.

10.4 A SAFE DESIGN

With the block diagram of Figure 10.5, the remaining task is to derive the controller. Our first scheme uses a "safe" design, which means that the design provides large timing margins and does not impose any stringent timing constraints. The control signals are generated directly from the FSM. The controller uses two clock cycles (i.e., 40 ns) to complete memory access and requires three clock cycles (i.e., 60 ns) for back-to-back operations.

10.4.1 ASMD chart

The ASMD chart for this controller is shown in Figure 10.6. The FSM has five states and is initially in the idle state. It starts the memory operation when the mem signal is activated. The rw signal determines whether it is a read or write operation.

For a read operation, the FSM moves to the rd1 state. The memory address, addr, is sampled and stored in the addr_reg register at the transition. The oe_n signal is activated in the rd1 and rd2 states. At the end of the read cycle, the FSM returns to the idle state. The retrieved data is stored in the data_s2f_reg register at the transition, and the oe_n signal is deactivated afterward. Note that the block diagram of Figure 10.5 has two read ports. The data_s2f_r signal is a registered output and becomes available *after* the FSM exits the r2 state. The data remains unchanged until the end of the next read cycle. The data_s2f_ur signal is connected directly to the SRAM's dio bus. Its data should become valid at the end of the rd2 state but will be removed after the FSM enters the idle state. In some applications, the main system samples and stores the memory readout in its own register, and the unregistered output allows this action to be completed one clock cycle earlier.

For a write operation, the FSM moves to the wr1 state. The memory address, addr, and data, data_f2s, are sampled and stored in the addr_reg and data_f2s_reg registers at the transition. The we_n and tri_n signals are both activated in the wr1 state. The latter enables the tri-state buffer to put the data over the SRAM's dio bus. When the FSM moves to the wr2 state, we_n is deactivated but tri_n remains asserted. This ensures that the data is properly latched to the SRAM when we_n changes from'0' to '1'. At the end of the write

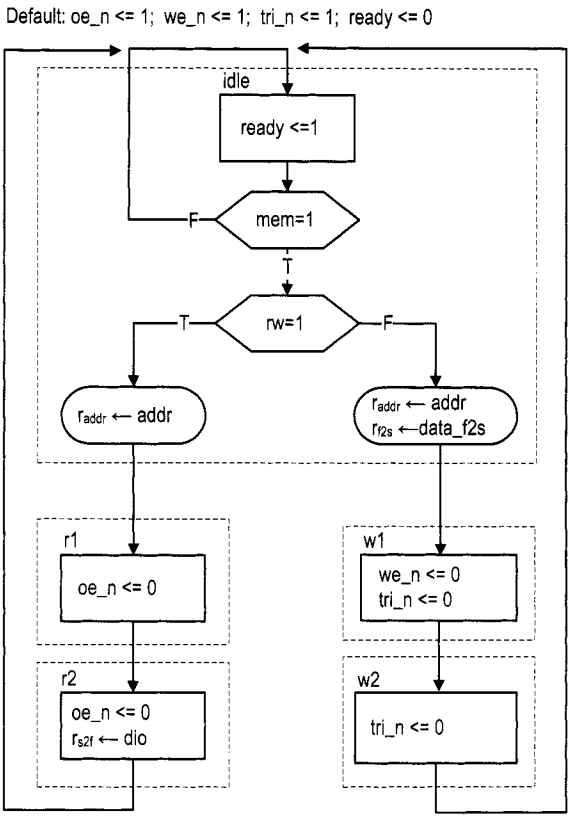


Figure 10.6 ASMD chart of a safe SRAM controller.

cycle, the FSM returns to the `idle` state and `tri_n` is deactivated to remove data from the `dio` bus.

10.4.2 Timing analysis

To ensure correct operation of a memory controller, we must verify that the design meets various timing requirements. Recall that the FSM is controlled by a 50-MHz clock signal and thus stays in each state for 20 ns.

During the read cycle, `oe_n` is asserted for two states, totaling 40 ns, which provides a 30-ns margin over the 10-ns t_{AA}. Although it appears that `oe_n` can be deasserted in the `rd2` state, this imposes a more stringent timing constraint. This issue is explained in Section 10.5.3. The data is stored in the `data_s2f` register when the FSM moves from the `rd2` state to the `idle` state. Although `oe_n` is deasserted at the transition, the data remains valid for a small interval because of the FPGA's pad delay and the t_{HZOE} delay of the SRAM chip. It can be sampled properly by the clock edge.

During the write cycle, `we_n` is asserted in the `wr1` state, and the 20-ns interval exceeds the 8-ns t_{PWE1} requirement. The `tri_n` signal remains asserted in the `wr2` state and thus ensures that the data is still stable during the '0'-to-'1' transition edge of the `we_n` signal.

In terms of performance, both read and write operations take two clock cycles to complete. During the read operation, the unregistered data (i.e., `data_s2f_ur`) is available at the end of the second clock cycle (i.e., just before the rising edge of the second clock cycle) and the registered data (i.e., `data_s2f_r`) is available right after the rising edge of the second clock cycle. Although a memory operation can be done in two clocks, the main system cannot access memory at this rate. Both read and write operations must return to the `idle` state after completion. The main system must wait for another clock cycle to issue a new memory operation, and thus the back-to-back memory access takes three clock cycles.

10.4.3 HDL implementation

The HDL code can be derived by following the block diagram in Figure 10.5 and the ASMD chart in Figure 10.6. The memory controller must generate fast, glitch-free control signals. One method is to modify the output logic to include *look-ahead output buffers* for the Moore output signals. This scheme adds a buffer (i.e., D FF) for each output signal to remove glitches and reduce clock-to-output delay. To compensate the one clock cycle delay introduced by the buffer, we "look ahead" at the state's future value (i.e., the `state_next` signal) and use it to replace the state's current value (i.e., the `state_reg` signal) in the FSM's output logic.

The complete HDL code is shown in Listing 10.1. To facilitate future expansion, we label the S3 board's two SRAM chips as `a` and `b` and add an `_a` suffix to the SRAM's I/O signals in port declaration. Note that tri-state buffers are required for the bidirectional data signal `dio_a`.

Listing 10.1 SRAM controller with three-cycle back-to-back operation

```
library ieee;
use ieee.std_logic_1164.all;
entity sram_ctrl is
   port(
      clk, reset: in std_logic;
      -- to/from main system
      mem: in std_logic;
      rw: in std_logic;
      addr: in std_logic_vector(17 downto 0);
      data_f2s: in std_logic_vector(15 downto 0);
      ready: out std_logic;
      data_s2f_r, data_s2f_ur:
           out std_logic_vector(15 downto 0);
      -- to/from chip
      ad: out std_logic_vector(17 downto 0);
      we_n, oe_n: out std_logic;
      -- SRAM chip a
      dio_a: inout std_logic_vector(15 downto 0);
      ce_a_n, ub_a_n, lb_a_n: out std_logic
   );
end sram_ctrl;

architecture arch of sram_ctrl is
   type state_type is (idle, rd1, rd2, wr1, wr2);
   signal state_reg, state_next: state_type;
   signal data_f2s_reg, data_f2s_next:
```

```
            std_logic_vector(15 downto 0);
    signal data_s2f_reg, data_s2f_next:
            std_logic_vector(15 downto 0);
    signal addr_reg, addr_next: std_logic_vector(17 downto 0);
    signal we_buf, oe_buf, tri_buf: std_logic;
    signal we_reg, oe_reg, tri_reg: std_logic;
begin
   -- state & data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         addr_reg <= (others=>'0');
         data_f2s_reg <= (others=>'0');
         data_s2f_reg <= (others=>'0');
         tri_reg <= '1';
         we_reg <= '1';
         oe_reg <= '1';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         addr_reg <= addr_next;
         data_f2s_reg <= data_f2s_next;
         data_s2f_reg <= data_s2f_next;
         tri_reg <= tri_buf;
         we_reg <= we_buf;
         oe_reg <= oe_buf;
      end if;
   end process;
   -- next-state logic
   process(state_reg,mem,rw,dio_a,addr,data_f2s,
           data_f2s_reg,data_s2f_reg,addr_reg)
   begin
      addr_next <= addr_reg;
      data_f2s_next <= data_f2s_reg;
      data_s2f_next <= data_s2f_reg;
      ready <= '0';
      case state_reg is
         when idle =>
            if mem='0' then
               state_next <= idle;
            else
               addr_next <= addr;
               if rw='0' then --write
                  state_next <= wr1;
                  data_f2s_next <= data_f2s;
               else -- read
                  state_next <= rd1;
               end if;
            end if;
            ready <= '1';
         when wr1 =>
            state_next <= wr2;
         when wr2 =>
```

```
            state_next <= idle;
         when rd1 =>
            state_next <= rd2;
         when rd2=>
            data_s2f_next <= dio_a;
            state_next <= idle;
      end case;
   end process;
   -- "look-ahead" output logic
   process(state_next)
   begin
      tri_buf <= '1';  -- default
      we_buf <= '1';
      oe_buf <= '1';
      case state_next is
         when idle =>
         when wr1 =>
            tri_buf <= '0';
            we_buf <= '0';
         when wr2 =>
            tri_buf <= '0';
         when rd1 =>
            oe_buf <= '0';
         when rd2=>
            oe_buf <= '0';
      end case;
   end process;
   -- to main system
   data_s2f_r <= data_s2f_reg;
   data_s2f_ur <= dio_a;
   -- to SRAM
   we_n <= we_reg;
   oe_n <= oe_reg;
   ad <= addr_reg;
   --i/o for SRAM chip a
   ce_a_n <='0';
   ub_a_n <='0';
   lb_a_n <='0';
   dio_a <= data_f2s_reg when tri_reg='0' else (others=>'Z');
end arch;
```

To minimize the off-chip pad delay (discussed in Section 10.5.1), the corresponding FPGA's I/O pins should be configured properly. This can be done by adding additional information in the constraint file. A typical line is

```
NET "ad<17>"  LOC = "L3" | IOSTANDARD = LVCMOS33 | SLEW=FAST ;
```

10.4.4 Basic testing circuit

We use two circuits to verify operation of the SRAM controller. The first one is a basic testing circuit that allows us manually to perform a single read or write operation. In addition to the SRAM chip I/O signals, the circuit has the following signals:

- `sw`. It is 8 bits wide and used as data or address input.

- `led`. It is 8 bits wide and used to display the retrieved data.
- `btn(0)`. When it is asserted, the current value of `sw` is loaded to a data register. The output of the register is used as the data input for the write operation.
- `btn(1)`. When it is asserted, the controller uses the value of `sw` as a memory address and performs a write operation.
- `btn(2)`. When it is asserted, the controller uses the value of `sw` as a memory address and performs a read operation. The readout is routed to the `led` signal.

During a write operation, we first specify the data value and load it to the internal register and then specify the address and initiate the write operation. During a read operation, we specify the address and initiate the read operation. The retrieved data is displayed in eight discrete LEDs. The complete HDL code is shown in Listing 10.2.

Listing 10.2 Basic SRAM testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_ctrl_test is
  port(
      clk, reset: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(2 downto 0);
      led: out std_logic_vector(7 downto 0);
      ad: out std_logic_vector(17 downto 0);
      we_n, oe_n: out std_logic;
      dio_a: inout std_logic_vector(15 downto 0);
      ce_a_n, ub_a_n, lb_a_n: out std_logic
  );
end ram_ctrl_test;

architecture arch of ram_ctrl_test is
   constant ADDR_W: integer:=18;
   constant DATA_W: integer:=16;
   signal addr: std_logic_vector(ADDR_W-1 downto 0);
   signal data_f2s, data_s2f:
          std_logic_vector(DATA_W-1 downto 0);
   signal mem, rw: std_logic;
   signal data_reg: std_logic_vector(7 downto 0);
   signal db_btn: std_logic_vector(2 downto 0);

begin
   ctrl_unit: entity work.sram_ctrl
      port map(
         clk=>clk, reset=>reset,
         mem=>mem, rw =>rw, addr=>addr, data_f2s=>data_f2s,
         ready=>open, data_s2f_r=>data_s2f,
         data_s2f_ur=>open, ad=>ad,
         we_n=>we_n, oe_n=>oe_n, dio_a=>dio_a,
         ce_a_n=>ce_a_n, ub_a_n=>ub_a_n, lb_a_n=>lb_a_n);

    debounce_unit0: entity work.debounce
       port map(
          clk=>clk, reset=>reset, sw=>btn(0),
```

```
        db_level=>open, db_tick=>db_btn(0));
    debounce_unit1: entity work.debounce
       port map(
          clk=>clk, reset=>reset, sw=>btn(1),
          db_level=>open, db_tick=>db_btn(1));
    debounce_unit2: entity work.debounce
       port map(
          clk=>clk, reset=>reset, sw=>btn(2),
          db_level=>open, db_tick=>db_btn(2));

  --data registers
  process(clk)
  begin
     if (clk'event and clk='1') then
        if (db_btn(0)='1') then
           data_reg <= sw;
        end if;
     end if;
  end process;
  -- address
  addr <= "0000000000" & sw;
  -- command
  process(db_btn,data_reg)
  begin
    data_f2s <= (others=>'0');
    if db_btn(1)='1' then -- write
       mem <= '1';
       rw <= '0';
       data_f2s <= "00000000" & data_reg;
    elsif db_btn(2)='1' then -- read
       mem <= '1';
       rw <= '1';
    else
       mem <= '0';
       rw <= '1';
     end if;
  end process;
  -- output
  led <= data_s2f(7 downto 0);
end arch;
```

10.4.5 Comprehensive SRAM testing circuit

The second circuit performs comprehensive testing. It verifies operation of the SRAM controller and checks the integrity of the SRAM chip as well. This circuit has three functions:

- Write testing data patterns to the entire SRAM at the maximal rate.
- Read the entire SRAM at the maximal rate, check the retrieved data against the original patterns, and record the number of erroneous readouts.
- Inject erroneous data.

These functions can be initiated by three debounced pushbuttons.

The ASMD chart is shown in Figure 10.7. It contains three branches, corresponding to

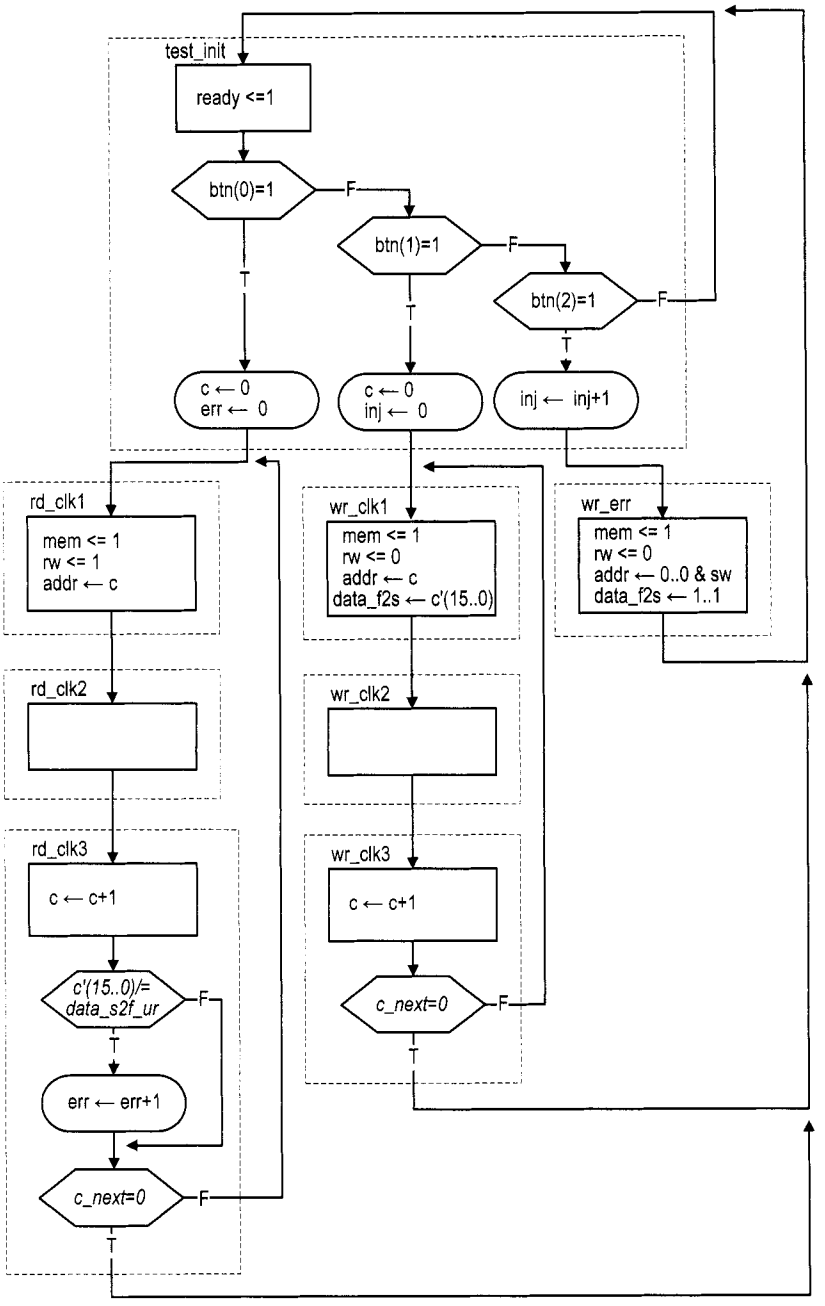


Figure 10.7 ASMD chart of a comprehensive SRAM testing circuit.

three functions. The middle branch writes the test patterns to the SRAM. The wr_clk1, wr_clk2, and wr_clk3 states correspond to the idle, wr1, and wr2 states of the SRAM controller. The FSMD uses the 18-bit c register as a counter to loop through this branch 2^{18} times. The content of the c register is used as an address and the reversed 16 LSBs are used as data during a write operation. The FSMD writes all memory locations while looping through this branch. The left branch reads data from the SRAM. The three states correspond to the idle, rd1, and rd2 states of the SRAM controller. The FSMD again loops through the branch 2^{18} times. The retrieved data is compared with the original test patterns, and the err register is used to keep track of the number of mismatches. The right branch performs a single write operation. It uses the 8-bit switch to form a memory address and writes an erroneous pattern to that address. The inj counter is used to keep track of the number of injected errors. The complete HDL code is shown in Listing 10.3.

Listing 10.3 Comprehensive SRAM testing circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sram_test is
  port(
      clk, reset: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      btn: in std_logic_vector(2 downto 0);
      led: out std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0);
      ad: out std_logic_vector(17 downto 0);
      we_n, oe_n: out std_logic;
      dio_a: inout std_logic_vector(15 downto 0);
      ce_a_n, ub_a_n, lb_a_n: out std_logic
  );
end sram_test;

architecture arch of sram_test is
   constant ADDR_W: integer:=18;
   constant DATA_W: integer:=16;
   signal addr: std_logic_vector(ADDR_W-1 downto 0);
   signal data_f2s, data_s2f:
          std_logic_vector(DATA_W-1 downto 0);
   signal mem, rw: std_logic;
   type state_type is (test_init, rd_clk1, rd_clk2, rd_clk3,
                       wr_err, wr_clk1, wr_clk2, wr_clk3);
   signal state_reg, state_next: state_type;
   signal c_next, c_reg: unsigned(ADDR_W-1 downto 0);
   signal c_std: std_logic_vector(ADDR_W-1 downto 0);
   signal inj_next, inj_reg: unsigned(7 downto 0);
   signal err_next, err_reg: unsigned(15 downto 0);
   signal db_btn: std_logic_vector(2 downto 0);

begin
   --================================================
   -- component instantiation
   --================================================
```

```
    ctrl_unit: entity work.sram_ctrl
40  port map(
       clk=>clk, reset=>reset,
       mem=>mem, rw =>rw, addr=>addr,
       data_f2s=>data_f2s, ready=>open,
       data_s2f_r=>open, data_s2f_ur=>data_s2f,
45     ad=>ad,  dio_a=>dio_a,
       we_n=>we_n, oe_n=>oe_n,
       ce_a_n=>ce_a_n, ub_a_n=>ub_a_n, lb_a_n=>lb_a_n);

     debounce_unit0: entity work.debounce
50      port map(
           clk=>clk, reset=>reset, sw=>btn(0),
           db_level=>open, db_tick=>db_btn(0));
     debounce_unit1: entity work.debounce
        port map(
55         clk=>clk, reset=>reset, sw=>btn(1),
           db_level=>open, db_tick=>db_btn(1));
     debounce_unit2: entity work.debounce
        port map(
           clk=>clk, reset=>reset, sw=>btn(2),
60         db_level=>open, db_tick=>db_btn(2));
     disp_unit: entity work.disp_hex_mux
       port map(
          clk=>clk, reset=>'0',dp_in=>"1111",
          hex3=>std_logic_vector(err_reg(15 downto 12)),
65        hex2=>std_logic_vector(err_reg(11 downto 8)),
          hex1=>std_logic_vector(err_reg(7 downto 4)),
          hex0=>std_logic_vector(err_reg(3 downto 0)),
          an=>an, sseg=>sseg);

70  --=================================================
    -- FSMD
    --=================================================
    -- state & data registers
    process(clk,reset)
75  begin
       if (reset='1') then
          state_reg <= test_init;
          c_reg <= (others=>'0');
          inj_reg <= (others=>'0');
80        err_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
          c_reg <= c_next;
          inj_reg <= inj_next;
85        err_reg <= err_next;
      end if;
    end process;
    c_std <= std_logic_vector(c_reg);
    -- fsmd next-state logic / data path operations
90  process(state_reg,sw,db_btn,c_reg,c_std,
            c_next,inj_reg,err_reg,data_s2f)
```

```
begin
   c_next <= c_reg;
   inj_next <= inj_reg;
   err_next <= err_reg;
   addr <= (others=>'0');
   rw <= '1';
   mem <= '0';
   data_f2s <= (others=>'0');
   case state_reg is
      when test_init =>
         if db_btn(0)='1' then
            state_next <= rd_clk1;
            c_next <=(others=>'0');
            err_next <=(others=>'0');
         elsif db_btn(1)='1' then
            state_next <= wr_clk1;
            c_next <=(others=>'0');
            inj_next <=(others=>'0'); -- clear injected err
         elsif db_btn(2)='1' then
            state_next <= wr_err;
            inj_next <= inj_reg + 1;
         else
            state_next <= test_init;
         end if;
      when wr_err => -- write 1 err; done in next 2 clocks
         state_next <= test_init;
         mem <= '1';
         rw <= '0';
         addr <= "0000000000" & sw;
         data_f2s <= (others=>'1');
      when wr_clk1 => -- in idle state of sram_ctrl
         state_next <= wr_clk2;
         mem <= '1';
         rw <= '0';
         addr <= c_std;
         data_f2s <= not c_std(DATA_W-1 downto 0);
      when wr_clk2 => -- in wr1 state of sram_ctrl
         state_next <= wr_clk3;
      when wr_clk3 => -- in wr2 state of sram_ctrl
         c_next <= c_reg + 1;
         if c_next=0 then
            state_next <= test_init;
         else
            state_next <= wr_clk1;
         end if;
      when rd_clk1 => -- in idle state of sram_ctrl
         state_next <= rd_clk2;
         mem <= '1';
         rw <= '1';
         addr <= c_std;
      when rd_clk2 => -- in rd1 state of sram_ctrl
         state_next <= rd_clk3;
      when rd_clk3 => -- in rd2 state of sram_ctrl
```

```
145             -- compare readout; must use unregistered output
                if (not c_std(DATA_W-1 downto 0))/=data_s2f then
                   err_next <= err_reg + 1;
                end if;
                c_next <= c_reg + 1;
150             if c_next=0 then
                   state_next <= test_init;
                else
                   state_next <= rd_clk1;
                end if;
155        end case;
        end process;
        led <= std_logic_vector(inj_reg);
    end arch;
```

Note that the number of write–read mismatches is connected to the seven-segment LED display and shown as a four-digit hexadecimal number, and the number of injected errors is connected to the eight discrete LEDs.

We can use this circuit as follows:

- Perform the read function. Since the SRAM is not written yet, it is in the initial "power-on" state. The seven-segment LED display should show a large number of mismatches.
- Perform the write function.
- Perform the read function. The number of mismatches should be zero if both the SRAM controller and the SRAM device work properly.
- Inject error data a few times (to different memory locations).
- Perform the read function again. The number of mismatches should be the same as the number of injected errors.

10.5 MORE AGGRESSIVE DESIGN

Although the previous memory controller functions properly, it does not have optimal performance. While both the read and write cycles are 10 ns of the SRAM device, the back-to-back memory access of this controller takes 60 ns (i.e., three clock cycles). In this section, we study the timing issue in more detail, examine several more aggressive designs and their potential problems, and discuss some FPGA features that help to remedy the problems.

10.5.1 Timing issues

Timing issues on asynchronous SRAM There are two subtle timing issues in designing a high-performance asynchronous SRAM controller. The first issue is deactivation of the `we_n` signal. The '0'-to-'1' transition of `we_n` functions somewhat like a clock edge of an FF, in which the data is latched and stored to the internal memory element. Note that the data hold time (t_{HD}) is zero for this SRAM. Although it appears that it is fine to deactivate `we_n` and remove data at the same time, this approach is not reliable because of the variations in propagation delays. We must ensure that `we_n` is deactivated *before* data is removed from the bus.

The second issue is the potential conflict on the data bus, `dio`. Recall that the data bus is a bidirectional bus. The controller places data on the bus during a write operation, and the

SRAM places data on the bus during a read operation. A condition known as `fighting` occurs if the controller and SRAM place data on the bus at the same time. This condition should be avoided to ensure reliable operation.

Estimation of propagation delay Designing a good memory controller requires having a good understanding about the propagation delays of various signals. However, it is a difficult task. First, during synthesis, an RT-level description is optimized and mapped to logic cells and wire interconnects. The final implementation may not resemble the block diagram depicted by the initial description, and thus it is difficult to estimate the propagation delay from the initial description.

Second, a memory operation involves *off-chip* data access. Additional propagation delay is introduced when a signal propagates through the FPGA's I/O pads. The delay, sometimes known as *pad delay*, is usually much larger than the internal wiring delay and its exact value depends on a variety of factors, including the type of FPGA device, the location of the output register (in LE or IOB), the I/O standards, the slew rate, the driver strength, and external loading.

It requires intimate knowledge of the FPGA device and the synthesis software to perform a good timing analysis and to estimate the propagation delays of various signals.

10.5.2 Alternative design I

The first alternative design is targeted to reduce the back-to-back operation overhead. Instead of always returning to the `idle` state, the memory controller can check the `mem` signal at the end of current memory operation (i.e., in the `rd2` or `wr2` state) and determine what to do next. It initiates a new memory operation immediately if there is a pending request.

The revised ASMD chart for this controller is shown in Figure 10.8. In the `rd2` and `wr2` states, the `mem` and `rw` signals are examined and the FSMD may move directly to the `rd1` or `wr1` state if another memory operation is required.

Timing analysis Most of the original timing analysis in Section 10.4.2 can still be applied to this design. However, skipping the `idle` state introduces subtle new complications when different types of back-to-back memory operations are performed. The issue is the potential fighting on the data bus.

Let us consider a write operation performed immediately after a read operation. During the read operation, the signal flows from the SRAM to the FPGA. To facilitate this operation, the tri-state buffer of the SRAM should be "turned on" (i.e., passing signal) and the tri-state buffer of the FPGA should be "turned off" (i.e., high impedance). During the write operation, the signal flows from the FPGA to the SRAM, and the roles of the two tri-state buffers are reversed. Note that a small delay is required to turn on or off a tri-state buffer. In the SRAM chip, these delays are specified by t_{HZOE} (`oe_n` to high-impedance time) and t_{LZOE} (`oe_n` to low-impedance time) in Figure 10.2.

In the original SRAM controller, both tri-state buffers are turned off in the `idle` state. The state provides enough time for the data bus to settle to the high-impedance condition. The new design requires the two tristate buffers to reverse directions simultaneously during back-to-back operations. For example, when moving from the `rd2` state to the `wr1` state, the FSMD generates signals to turn off the SRAM's tri-state buffer and to turn on the FPGA's tri-state buffer. A problem may occur in this transition if the SRAM's tri-state buffer is turned off too slowly or the FPGA's tri-state buffer is turned on too quickly. In a small interval, both buffers may allow data to be placed on the bus and fighting occurs. Similarly, fighting may occur when a read operation is performed immediately after a write operation.

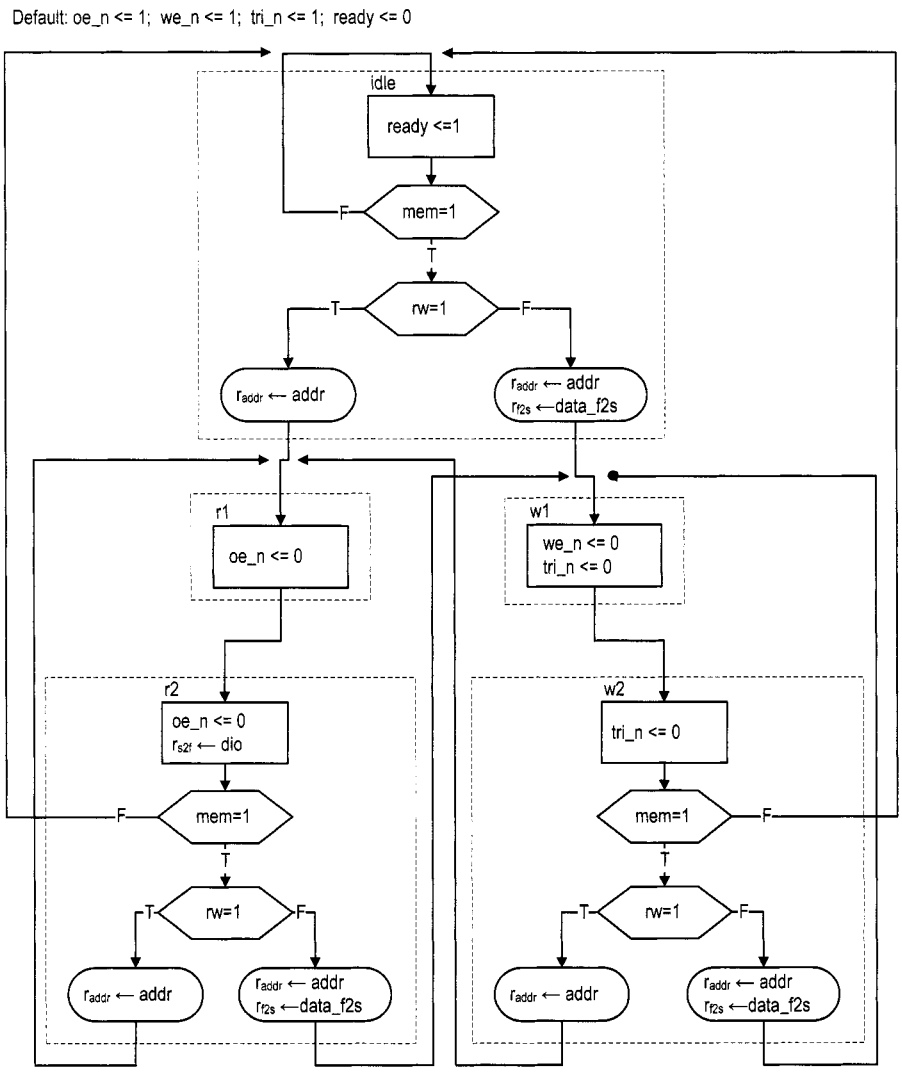


Figure 10.8 ASMD chart of SRAM controller design I.

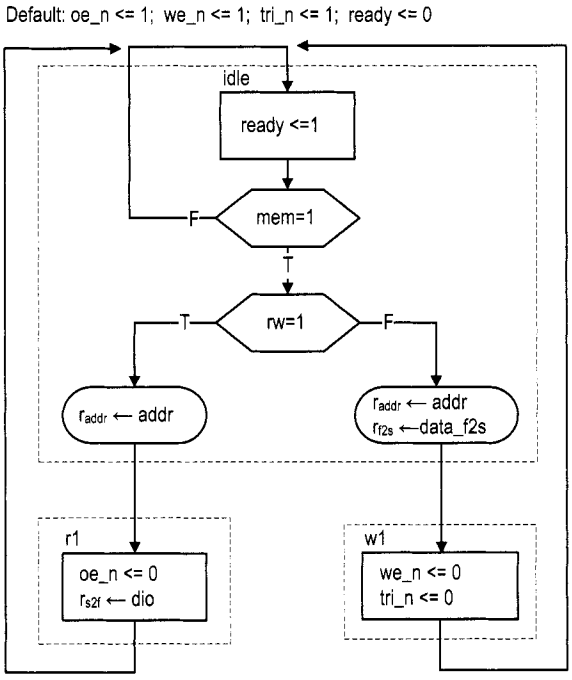


Figure 10.9 ASMD chart of SRAM controller design II.

Since the interval tends to be very small, the fighting should not cause severe damage to the devices but may introduce a large transient current which makes the design less reliable. We must do a detailed timing analysis to examine whether fighting occurs, and may even need to fine-tune the timing to fix the problem. As discussed in Section 10.5.1, it is a difficult task.

10.5.3 Alternative design II

Timing analysis in Section 10.4.2 shows that the initial design provides a large safety margin. In this controller, a memory operation takes two clock cycles, which amount to 40 ns. Since the read and write cycles of the SRAM are each 10 ns, we naturally wonder whether it is possible to reduce the operation time to a single 20-ns clock cycle. This can be done by eliminating the `rd2` and `wr2` states in the ASMD chart. The second alternative design uses this approach. The revised ASMD chart is shown in Figure 10.9. It takes one clock cycle to complete the memory access and requires two clock cycles to complete the back-to-back operations.

Timing analysis Reducing a state from the original controller imposes much tighter timing constraints for both read and write operations. Let us first consider the read operation. During operation, the address signal first propagates through the FPGA's I/O pads to the SRAM's address bus, and the retrieved data then propagates back through the I/O pads to FPGA's internal logic. All of this must be completed within a 20-ns clock cycle. In addition to the 10-ns SRAM address access time (i.e., t_{AA}), the cycle must accommodate

two pad delays. The pad delay of a Spartan-3 device can range from 4 ns to more than 10 ns. Therefore, we need to "fine-tune" the synthesis to achieve this margin.

Unlike the read operation, a write operation is "one-way" and only needs to propagate the address, data, and control signals to the SRAM chip. If we assume that the signals experience similar pad delays, the absolute value of the delay is a lesser issue. Instead, the key is the *order* of signals being activated and deactivated. As discussed in Section 10.5.1, `we_n` must be deactivated before data to latch the data properly to the SRAM. In the original design, this is achieved by including the second state in the write operation, `wr2`, in which `we_n` is deactivated but the data is still available (i.e., `tri_n` is still active). In the revised controller, the `we_n` and `tri_n` signals are deactivated simultaneously at the end of the `wr1` state. Due to the variations in the internal logic and pad delays, normal synthesis cannot guarantee that `we_n` is deactivated before the data is removed from the external data bus. Again, for a reliable design, we need to fine-tune the synthesis to satisfy this goal.

10.5.4 Alternative design III

We can combine the features from the two preceding revisions to derive the third alternative design. This new controller eliminates the second clock cycle in the read and write operations and allows back-to-back operation without first returning to the `idle` state. This is the most aggressive design. The revised ASMD chart is shown in Figure 10.10. It combines the modifications from the previous two ASMD charts. The revised design takes one clock cycle to complete the memory access and one clock cycle to complete back-to-back operations.

Note that the `we_n` signal must be asserted for a fraction of the clock period and cannot be shown in the ASMD chart. We use the `we_tmp` in the `wr1` state and later derive `we_n` from this signal.

Timing analysis Since the new design combines the features of the two previous designs, all the timing issues discussed in the two preceding subsections must be considered for this design as well. One additional issue is generation of the `we_n` signal. During back-to-back write operations, the ASMD stays on the `wr1` state. In the original design, the `we_n` signal is a Moore output. It will be asserted to '0' continuously in this case. The controller does not function properly since the data is latched to the SRAM at the '0'-to-'1' transition of the `we_n` signal. To solve the problem, the `we_n` signal must be asserted in only a fraction of the clock period.

One possible way to solve the problem is to assert the signal only at the first half of the clock, which is 10 ns and can satisfy the t_{WPE1} requirement in theory. Intuitively, we are tempted to do this by gating the `we_tmp` signal with the clock signal, `clk`:

```
we_n <= we_tmp or (not clk);
```

However, this is not a reliable solution because of the potential glitches and delay variation. A better alternative is discussed in the next subsection.

10.5.5 Advanced FPGA features$^{Xilinx\ specific}$

The memory controller examples in this section illustrate the limitations of the FSM-based controller and synchronous design methodology. Basically, an FSM cannot generate a control sequence that is "finer" than the period of its clock signal. The operation of these alternative designs relies on factors that cannot be specified by an RT-level HDL description.

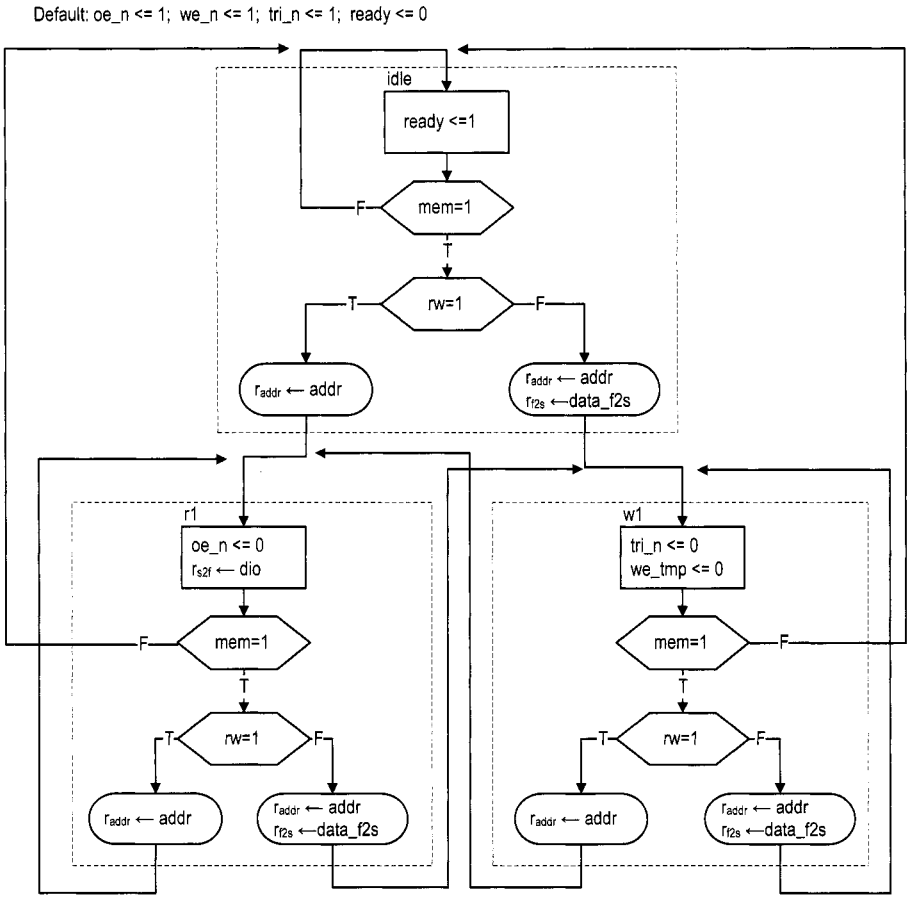


Figure 10.10 ASMD chart of SRAM controller design III.

Due to the variations in propagation delays, the synthesized circuits are not reliable and may or may not work.

There are some ad hoc features to obtain better control. These features are usually device and software dependent. For example, the digital clock manager (DCM) circuit and input/output block (IOB) of the Spartan-3 device can help to remedy some of the previously discussed problems. Detailed discussion of DCM and IOB is beyond the scope of this book. In this subsection, we sketch a few ideas and illustrate how to apply these features to obtain a more reliable controller.

DCM A Spartan-3 FPGA device contains up to eight *digital clock managers* (DCMs). As its name indicates, a DCM is a circuit that manipulates the system clock signal. It can multiply or divide the frequency or shift the phase of the incoming clock signal to generate new clock signals.

One way to obtain a "finer" control sequence is to use a faster clock. Since implementation of a memory controller is fairly simple, the circuit itself can operate at a faster clock rate. For example, we can isolate the memory controller and drive it with a DCM-generated 200-MHz clock signal, whose period is only 5 ns. Consider the write operation of the ASMD chart in Figure 10.6. In the new controller, each state lasts only 5 ns. To satisfy the 10-ns `we_n` requirement, we need to expand the `wr1` state to two states and assert the `we_n` signal in these states. The complete write operation now requires four states. However, because of the faster clock rate, the four clock cycles amount to only 20 ns, which is much better than the original 60-ns design.

A simple application of clock phase shift is discussed in the next subsection.

IOB An *input/output block* (IOB) of a Spartan-3 FPGA device provides a programmable interface between an I/O pin and the device's internal logic. It contains several storage registers and tri-state buffers as well as analog driver circuits that can be configured to provide different slew rates and driver strength and to support a variety of I/O standards.

To minimize the off-chip pad delay discussed in Section 10.5.3, we can put the output registers of the memory controller to the FFs inside the IOBs and configure the driver with the proper slew rate and strength. This can be done by specifying the desired condition and configuration in the constraint file.

An IOB also contains a *double data rate* (DDR) register, which has two clocks and two inputs. Conceptually, we can think that the two inputs are sampled independently by the two clocks and the sampled values are stored in the same register. The DDR register and DCM can be combined to generate a control signal whose width is a fraction of a clock signal, as the `we_n` signal discussed in Section 10.5.4. The block diagram is shown in Figure 10.11(a). The regular output register is replaced with a DDR register. The top portion of the DDR consists of the *we_tmp* signal and the original clock signals, `clk`. The bottom input of the DDR is tied to '1' and the clock is connected to the out-of-phase clock signal, `clk180`, which is generated by a DCM. The '1' is always loaded at the rising edge of the `clk180` signal, which corresponds to the falling edge of the `clk` signal. It essentially deactivates the second half of the *we_n* signal. The timing diagram is shown in Figure 10.11(b). This approach generates a clean half-cycle signal and is far more reliable than the clock gating scheme discussed in Section 10.5.4.

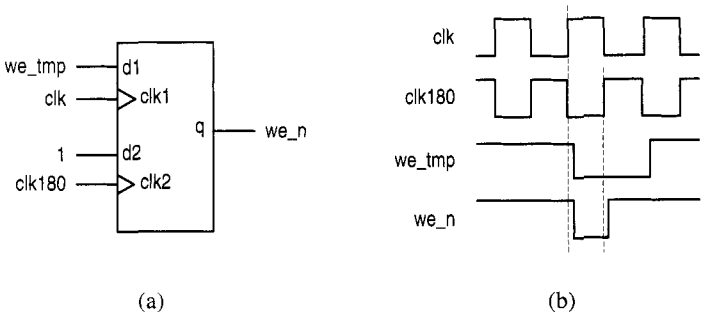


Figure 10.11 Generating a half-cycle signal with DDR.

10.6 BIBLIOGRAPHIC NOTES

The data sheet published by ISSI provides detailed information for the IS61LV25616AL SRAM device. The Xilinx application note, *XAPP462 Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*, discusses the use of DCM, and the data sheet, *DS099 Spartan-3 FPGA Family: Complete Data Sheet*, explains the architecture and configuration of the IOB and the DDR register.

10.7 SUGGESTED EXPERIMENTS

10.7.1 Memory with a 512K-by-16 configuration

There are two 256K-by-16 SRAM chips, and their I/O connections are shown in the manual of the S3 board. We can expand them to form a 512K-by-16 SRAM.

1. Derive a scheme to combine the two chips.
2. Follow the procedure in Section 10.4 to design a memory controller for the 512K-by-16 SRAM. Derive the HDL description.
3. Modify the testing circuit in Section 10.4.5 for the new controller and derive the HDL description.
4. Synthesize the testing circuit and verify operation of the controller and SRAM chips.

10.7.2 Memory with a 1M-by-8 configuration

Repeat Experiment 10.7.1 but configure the two chips as a 1M-by-8 SRAM. The `lb_n` and `ub_n` signals can be used for this purpose.

10.7.3 Memory with an 8M-by-1 configuration

A single bit of the 256K-by-16 SRAM can be written as follows:

- Read a 16-bit word.
- Modify the designated bit in the word.
- Write the 16-bit word back.

Repeat Experiment 10.7.1 but configure the two chips as an 8M-by-1 SRAM.

10.7.4 Expanded memory testing circuit

The memory testing circuit in Section 10.4.5 conducts exhaustive back-to-back read and back-to-back write tests. We can expand the circuit to include an exhaustive "read-after-write" test, in which the testing circuit issues write and read operations alternately for the entire memory space. To make the test more effective, the writing and reading addresses should be different. For example, we can make the read operation retrieve the data written 16 positions earlier (i.e., if the current writing address is c, the reading address will be c-16). Create a modified ASMD chart, derive an HDL description, synthesize the circuit, and verify its operation.

10.7.5 Memory controller and testing circuit for alternative design I

Derive the HDL code for alternative design I in Section 10.5.2 and create an expanded testing circuit similar to the one in Experiment 10.7.4. Synthesize the testing circuit and examine whether any error occurs during operation.

10.7.6 Memory controller and testing circuit for alternative design II

Repeat the process in Experiment 10.7.5 for alternative design II discussed in Section 10.5.3.

10.7.7 Memory controller and testing circuit for alternative design III

Repeat the process in Experiment 10.7.5 for alternative design III discussed in Section 10.5.4.

10.7.8 Memory controller with DCM

Study the application note on DCM and follow the discussion in Section 10.5.5 to drive the safe memory controller discussed in Section 10.4 with a higher clock rate (150 MH or even 200 MHz). Derive an ASMD chart and HDL code, and create a new testing circuit. Synthesize the circuit and verify operation of the memory controller and the SRAM.

10.7.9 High-performance memory controller

Study the documentation of the DCM and the IOB, and apply these features to reconstruct alternative design III discussed in Section 10.5.4. Create a new testing circuit. Synthesize the circuit and verify operation of the memory controller and the SRAM.