CHAPTER 12

REGISTER TRANSFER METHODOLOGY: PRACTICE

RT methodology is a powerful and versatile design technique. It can be applied to a wide variety of applications. In this chapter, we use several examples to illustrate how this methodology can be used in different types of problems and to highlight the design procedure and relevant issues.

12.1 INTRODUCTION

As discussed in Chapter 11, RT methodology can be thought of as a design technique that realizes an algorithm in hardware. The algorithm can be a complex process or just a simple sequential execution, and thus RT methodology is very flexible and versatile. We study five examples in this chapter, including a one-shot pulse generator, SRAM controller, universal asynchronous receiver and transmitter (UART), greatest common divisor (GCD) circuit, and square-root approximation circuit. The one-shot pulse generator is used to compare and contrast the differences among the regular sequential circuit, FSM and RT methodology. The SRAM controller illustrates the process of generating level-sensitive control signals to meet the timing requirement of a clockless device. The GCD circuit is another example of realizing a sequential algorithm in hardware. It shows how the hardware can be used to accelerate the performance. The UART receiver is a typical control-oriented application, which involves complex control structure and decision conditions. The square-root circuit, on the other hand, is a typical data-oriented application, which involves mainly arithmetic operations over data.

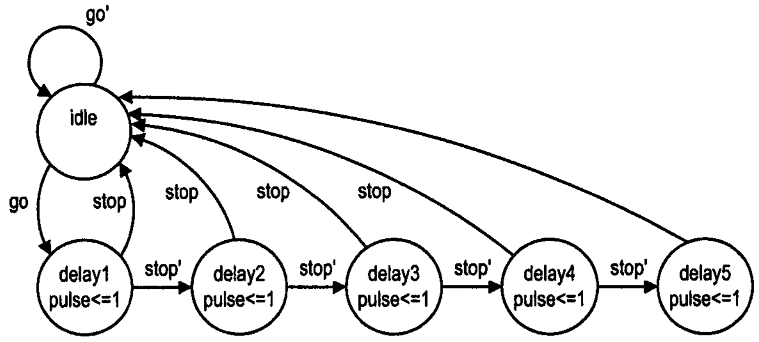


Figure 12.1 State diagram of a one-shot pulse generator.

12.2 ONE-SHOT PULSE GENERATOR

In Section 8.2.3, we divided sequential circuits into three categories based on the characteristics of the next-state logic:

- Regular sequential circuit. The next-state logic is regular.
- FSM. The next-state logic is random.
- RT methodology. The next-state logic consists of a regular part and a random part.

The RT methodology is the most flexible and capable scheme since it can accommodate both types of next-state logic.

The division is created to assist the circuit design and code development. There are no formal definitions of *regular* and *random*, and some applications can be designed as either type. In this section, we use a one-shot pulse generator as an example to illustrate the differences among the three types of circuits and to demonstrate the advantages and flexibility of the RT methodology.

A one-shot pulse generator is a circuit that generates a single fixed-width pulse upon activation of a trigger signal. We assume that the width of the pulse is five clock cycles. The detailed specifications are listed below.

- There are two input signals, go and `stop`, and one output signal, `pulse`.
- The go signal is the trigger signal that is usually asserted for only one clock cycle. During normal operation, assertion of the go signal activates the `pulse` signal for five clock cycles.
- If the go signal is asserted again during this interval, it will be ignored.
- If the `stop` signal is asserted during this interval, the `pulse` signal will be cut short and return to '0'.

Although the circuit is simple, it includes a regular part, which counts five clock cycles, and a random part, which keeps track of whether the circuit is idle or currently generating the pulse. Because of the simplicity, this circuit can be implemented as a pure regular sequential circuit, a pure FSM or a design using RT methodology.

12.2.1 FSM implementation

We first examine the FSM implementation. The state diagram is shown in Figure 12.1. The diagram consists of an idle state and five delay states, which activate the `pulse` signal for five clock cycles. The five delay states essentially function as a regular sequential circuit that counts for five clock cycles. The identical transition patterns of these five states hints at the "regularity" of this part of the operation. The corresponding VHDL code is shown in Listing 12.1.

Listing 12.1 FSM implementation of a one-shot pulse generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pulse_5clk is
   port(
      clk, reset: in std_logic;
      go, stop: in std_logic;
      pulse: out std_logic
   );
end pulse_5clk;

architecture fsm_arch of pulse_5clk is
   type fsm_state_type is
      (idle, delay1, delay2, delay3, delay4, delay5);
   signal state_reg, state_next: fsm_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic & output logic
   process(state_reg,go,stop)
   begin
      pulse <= '0';
      case state_reg is
         when idle =>
            if go='1' then
               state_next <= delay1;
            else
               state_next <= idle;
            end if;
         when delay1 =>
            if stop='1' then
               state_next <=idle;
            else
               state_next <=delay2;
            end if;
            pulse <= '1';
         when delay2 =>
            if stop='1' then
               state_next <=idle;
            else
```

```
                  state_next <=delay3;
               end if;
               pulse <= '1';
            when delay3 =>
               if stop='1' then
                  state_next <=idle;
               else
                  state_next <=delay4;
               end if;
               pulse <= '1';
            when delay4 =>
               if stop='1' then
                  state_next <=idle;
               else
                  state_next <=delay5;
               end if;
               pulse <= '1';
            when delay5 =>
               state_next <=idle;
               pulse <= '1';
         end case;
      end process;
end fsm_arch;
```

12.2.2 Regular sequential circuit implementation

We can also implement the pulse generator as a regular sequential circuit. It can be considered a mod-5 counter with a special control circuit to enable or disable the counting. To accommodate the generation of a single pulse, an additional FF is needed to flag whether the counter is active or idle. The VHDL code is shown in Listing 12.2.

Listing 12.2 Regular sequential circuit implementation of a one-shot pulse generator

```
architecture regular_seq_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   signal c_reg, c_next: unsigned(3 downto 0);
   signal flag_reg, flag_next: std_logic;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         c_reg <= (others=>'0');
         flag_reg <= '0';
      elsif (clk'event and clk='1') then
         c_reg <= c_next;
         flag_reg <= flag_next;
      end if;
   end process;
   -- next-state logic
   process(c_reg,flag_reg,go,stop)
   begin
      c_next <= c_reg;
      flag_next <= flag_reg;
      if (flag_reg='0') and (go='1') then
         flag_next <= '1';
         c_next <= (others=>'0');
      elsif (flag_reg='1') and
            ((c_reg=P_WIDTH-1) or (stop='1')) then
         flag_next <= '0';
      elsif (flag_reg='1') then
         c_next <= c_reg + 1;
      end if ;
   end process;
   -- output logic
   pulse <= '1' when flag_reg='1' else '0';
end regular_seq_arch;
```

There are two registers. The c_reg register is used for the counter, and the flag_reg register indicates whether the counter is active. The critical part of the description is the if statement of the next-state logic. The first condition, (flag_reg='0') and (go='1'), indicates that the counter is currently idle and the go signal is asserted. Under this condition, the flag is asserted and the counter enters the active counting state at the next rising edge of the clock. The second condition indicates that the counter reaches 5 or the stop signal is asserted and the counting should stop. The last condition indicates that the counter is in the active state and should keep on counting.

In this code, the flag_reg register functions as some sort of state register to keep track of the current condition of the circuit. The state transitions are implicitly embedded in the if statement of the next-state logic.

12.2.3 Implementation using RT methodology

The RT methodology can separate the regular and random logic, and the ASMD chart is shown in Figure 12.2. Two states in the chart indicate whether the counter is active, and the arcs show the transitions under various conditions. The RT operation in the delay state specifies the desired increment of the counter. Following the ASMD chart, we can easily derive the VHDL code, as shown in Listing 12.3.

Listing 12.3 FSMD implementation of a one-shot pulse generator

```
architecture fsmd_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   type fsmd_state_type is (idle, delay);
   signal state_reg, state_next: fsmd_state_type;
   signal c_reg, c_next: unsigned(3 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         c_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         c_reg <= c_next;
      end if;
```

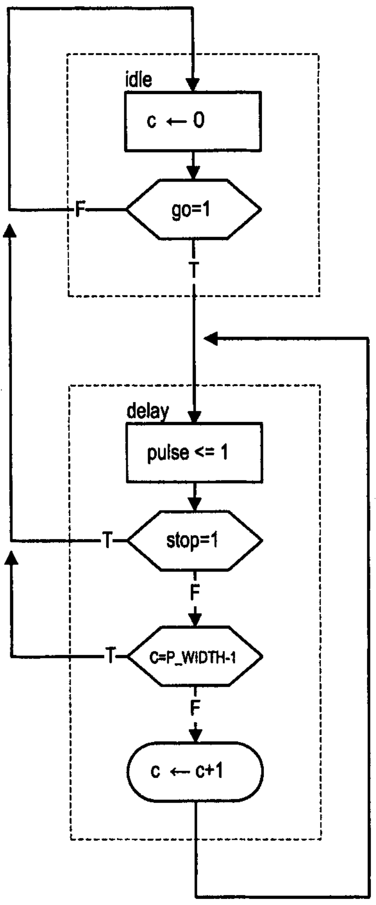


Figure 12.2 ASMD chart of a one-shot pulse generator.

```
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,go,stop,c_reg)
   begin
      pulse <= '0';
      c_next <= c_reg;
      case state_reg is
         when idle =>
            if go='1' then
               state_next <= delay;
            else
               state_next <= idle;
            end if;
            c_next <= (others=>'0');
         when delay =>
            if stop='1' then
               state_next <=idle;
            else
               if (c_reg=P_WIDTH-1) then
                  state_next <=idle;
               else
                  state_next <=delay;
                  c_next <= c_reg + 1;
               end if;
            end if;
            pulse <= '1';
      end case;
   end process;
end fsmd_arch;
```

12.2.4 Comparison

The pulse generator example shows that we can use an FSM to emulate a regular sequential circuit, and vice versa. However, the emulation is cumbersome and convolved, and is only possible for a small design. On the other hand, the RT methodology can capture the essence of both regular and random logic, and the description is simple, flexible, clear and informative. That is why it is such a powerful methodology.

To further illustrate the capability of the RT methodology, let us consider an expanded programmable one-shot pulse generator. In this circuit, the width of the pulse can be programmed between 1 and 7. The "programming" is done as follows:

- The go and stop signals are asserted at the same time to indicate the beginning of the program mode.
- The desired value is shifted in via the go signal in the next three clock cycles.

With the RT methodology, we can easily incorporate the extension into the ASMD chart, as shown in Figure 12.3. The corresponding VHDL code is shown in Listing 12.4.

Listing 12.4 Programmable one-shot pulse generator

```
architecture prog_arch of pulse_5clk is
   type fsmd_state_type is (idle, delay, sh1, sh2, sh3);
   signal state_reg, state_next: fsmd_state_type;
   signal c_reg, c_next: unsigned(2 downto 0);
```

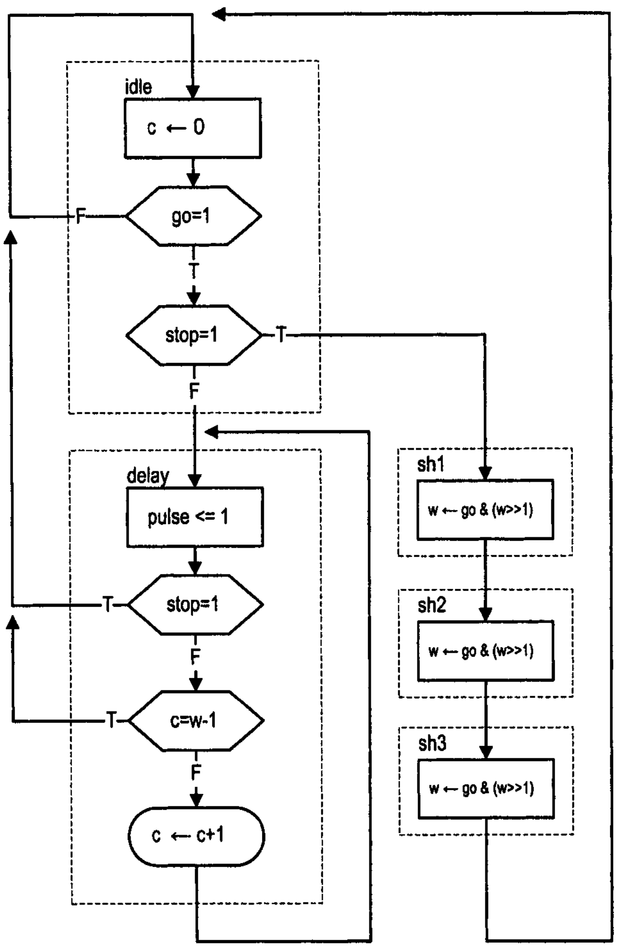


Figure 12.3 ASMD chart of a programmable one-shot pulse generator.

```
   signal w_reg, w_next: unsigned(2 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         c_reg <= (others=>'0');
         w_reg <= "101";  -- default 5-cycle delay
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         c_reg <= c_next;
         w_reg <= w_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,go,stop,c_reg,w_reg)
   begin
      pulse <= '0';
      c_next <= c_reg;
      w_next <= w_reg;
      case state_reg is
         when idle =>
            if go='1' then
               if stop='1' then
                  state_next <= sh1;
               else
                  state_next <= delay;
               end if;
            else
               state_next <= idle;
            end if;
            c_next <= (others=>'0');
         when delay =>
            if stop='1' then
               state_next <=idle;
            else
               if (c_reg=w_reg-1) then
                  state_next <=idle;
               else
                  c_next <= c_reg + 1;
                  state_next <=delay;
               end if;
            end if;
            pulse <= '1';
         when sh1 =>
            w_next <= go & w_reg(2 downto 1);
            state_next <= sh2;
         when sh2 =>
            w_next <= go & w_reg(2 downto 1);
            state_next <= sh3;
         when sh3 =>
            w_next <= go & w_reg(2 downto 1);
```

```
               state_next <= idle;
         end case;
      end process;
end prog_arch;
```

While we can implement the extended pulse generator as a pure FSM circuit or a pure regular sequential circuit in theory, the emulation becomes very involved and error-prone. It will require lots of effort to derive the code.

12.3 SRAM CONTROLLER

Random access memory (RAM) provides massive storage for digital systems. It is constructed as a two-dimensional array of memory cells. A cell is designed and optimized at the transistor level to achieve maximal efficiency. Since the silicon real estate is the primary concern, a memory cell is kept as simple as possible. Its control is level sensitive and uses no clock signal. To incorporate a RAM device into a synchronous digital system, we need a special circuit, known as a *memory controller*, to act as an interface to the synchronous system. Design of the memory controller illustrates control of a clockless subsystem.

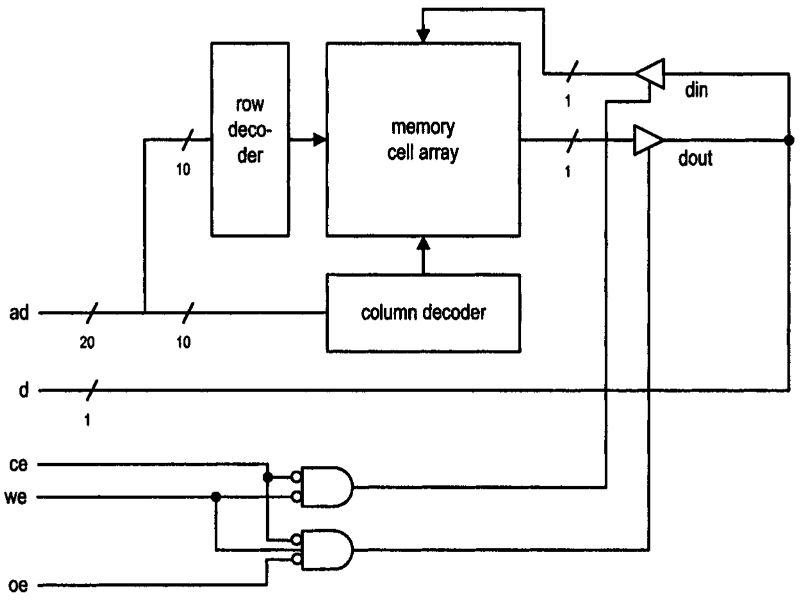
12.3.1 Overview of SRAM

RAM is organized as a two-dimensional array of memory cells with special decoding and multiplexing circuits. The block diagram of a typical 2^{20}-by-1 static RAM (SRAM) is shown in Figure 12.4(a). It contains a 2^{10}-by-2^{10} cell array, two 10-to-2^{10} decoders and an I/O control circuit. The I/O of the SRAM includes a 20-bit address signal, ad, a 1-bit bidirectional data signal, d, and three control signals, ce, we and oe. The ad signal is split and connected to two decoders, which, in turn, enable the cell of the specified location. The three control signals are used to control SRAM operation. The chip select signal, cs, specifies whether to enable the SRAM. The output enable signal, oe, and the write enable signal, we, choose between write and read modes and control the direction of data flow. The function table is shown in Figure 12.4(b). Note that these signals are active low.

Because of the lack of a clock signal, SRAM timing is quite involved. A set of minimum and maximum timing constraints has to be satisfied to ensure proper operation. We first examine the timing of a read operation. There are two methods to read data. In the first method, both the oe and cs signals are already activated (i.e., '0') and the address signal is used to access the desired data. It is known as an *address-controlled* read cycle and the timing diagram is shown in Figure 12.5(a). In the second method, the address signal is already stable and the cs signal already activated, and the oe signal is used to initiate the read operation. It is known as an *oe-controlled* read cycle, and the timing diagram is shown in Figure 12.5(b). Note the activities of the tri-state data bus when the oe signal is activated and deactivated.

The relevant timing parameters associated with a read cycle are:

- T_{aa}: address access time, the required time to obtain stable output data after an address change. It is somewhat like the propagation delay of the read operation and is used to characterize the speed of an SRAM, as in "50-ns SRAM."
- T_{oh}: output hold from address change 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.

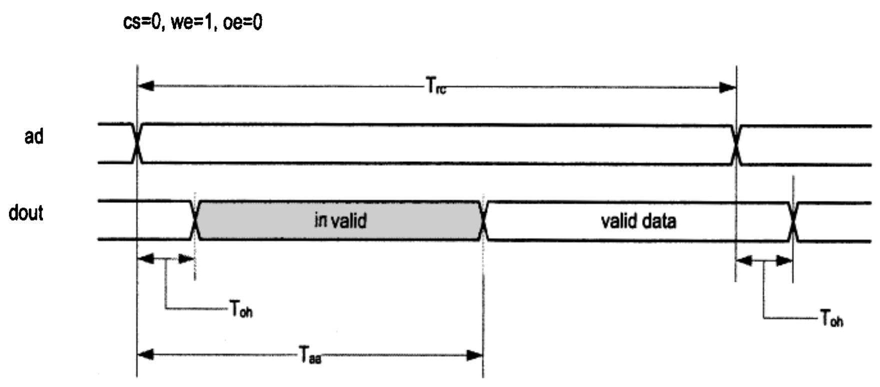


(a) Block diagram

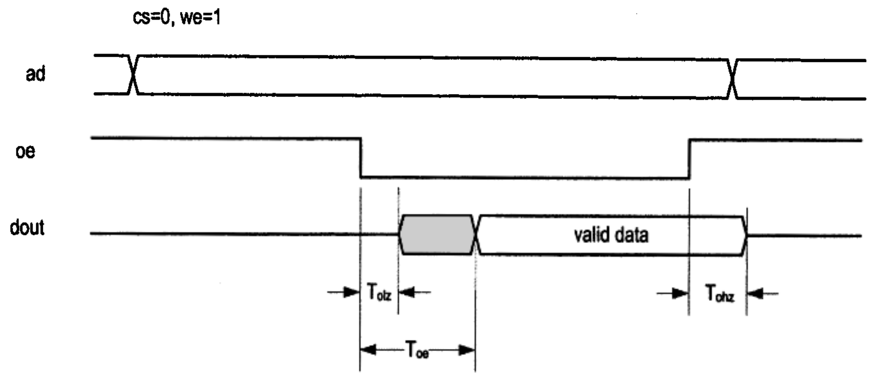
ce	we	oe	**Operation**	**Data pin** d
1	-	-	no operation	Z
0	0	-	write	data in
0	1	0	read	data out
0	1	1	no operation	Z

(b) Function table

Figure 12.4 Block diagram and functional table of a 2^{20}-by-1 SRAM.



(a) Address-controlled read cycle



(b) oe-controlled read cycle

Figure 12.5 Timing diagrams of an SRAM read cycle.

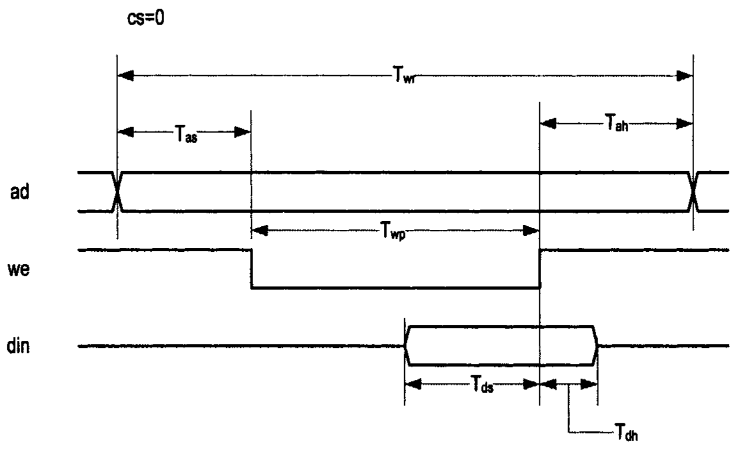


Figure 12.6 Timing diagram of an SRAM write cycle.

- T_{olz}: output enable to output in low-impedance time, the time for the tri-state buffer to leave from the high-impedance state after oe is activated. Note that even when the output is no longer in the high-impedance state, the data is still invalid.
- T_{oe}: output enable to output valid time, the time required to obtain valid data after oe is activated.
- T_{ohz}: output to Z time, the time for the tri-state buffer to enter the high-impedance state.
- T_{rc}: read cycle time, the minimal elapsed time between two read operations. It is about the same as T_{aa} for SRAM.

The write cycle is more complex. The timing diagram of a write cycle is shown in Figure 12.6. The key to understanding the write cycle timing is the assertion of the we signal, which latches the input data into the designated memory cell and plays a key role in the write operation. There are three major constraints:

- To latch data into the designated memory cell, the we signal must be activated (i.e., being '0') for a certain amount of time. This is specified by T_{wp}.
- The address needs to be stable for the entire write operation. Actually, it must be stable before we is activated and remain stable for a small amount of time after we is deactivated. The two time intervals are specified by T_{as} and T_{ah}.
- The input data must be stable in a small window when it is latched. The latch operation occurs at the edge when we transits from '0' to '1'. The input data has to be stable before and after the edge for a small amount of time. The two time intervals are specified by T_{ds} and T_{dh}. This constraint is somewhat like the constraint imposed on the d signal of a D FF at the rising edge of the clock.

These timing parameters are formally defined as follows:

- T_{wp}: write pulse width, the minimal time that the we signal must be activated.
- T_{as}: address setup time, the minimal time that the address must be stable before we is activated.
- T_{ah}: address hold time, the minimal time that the address must be stable after we is deactivated.

Table 12.1 Timing parameters of two SRAMs

Parameter	120-ns SRAM	20-ns SRAM
T_{aa} (max)	120 ns	20 ns
T_{oh} (min)	10 ns	3 ns
T_{olz} (min)	10 ns	0 ns
T_{oe} (max)	80 ns	9 ns
T_{ohz} (max)	40 ns	9 ns
T_{rc} (min)	120 ns	20 ns
T_{wp} (min)	70 ns	12 ns
T_{as} (min)	20 ns	0 ns
T_{ah} (min)	5 ns	0 ns
T_{ds} (min)	35 ns	1 ns
T_{dh} (min)	5 ns	0 ns
T_{wr} (min)	120 ns	20 ns

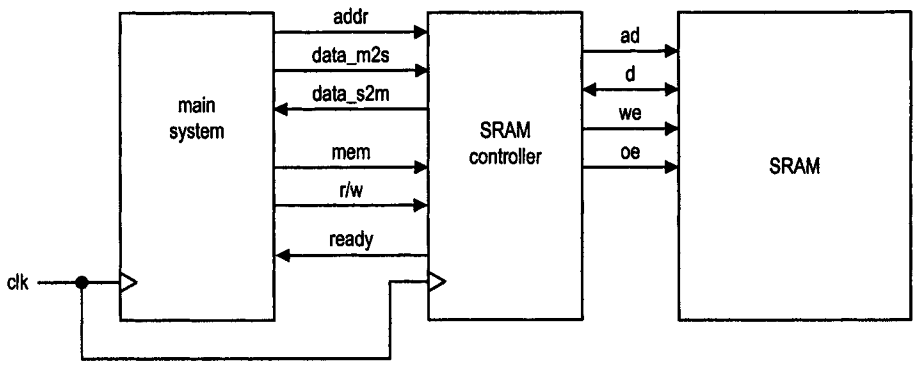


Figure 12.7 Role of an SRAM controller.

- T_{ds}: data setup time, the minimal time that data must be stable before the latching edge (the edge in which we moves from '0' to '1').
- T_{dh}: data hold time, the minimal time that data must be stable after the latching edge.
- T_{wr}: write cycle time, the minimal elapsed time between two write operations.

While there has been little change in the basic SRAM architecture over the years, its capacity and speed have improved significantly. The address access time (T_{aa}) can range from a few nanoseconds to several hundred nanoseconds. The typical timing parameters of an older, slow 120-ns SRAM and a more recent 20-ns SRAM are shown in Table 12.1.

12.3.2 Block diagram of an SRAM controller

The purpose of a memory controller is to interface the clockless memory and a synchronous system. The role of an SRAM controller is shown in Figure 12.7. It takes command from the main system and generates proper signals to store data into or retrieve data from the SRAM. The main system is a synchronous system. There are two command signals, mem and rw, and one status signal, ready. The main system activates the mem signal when a

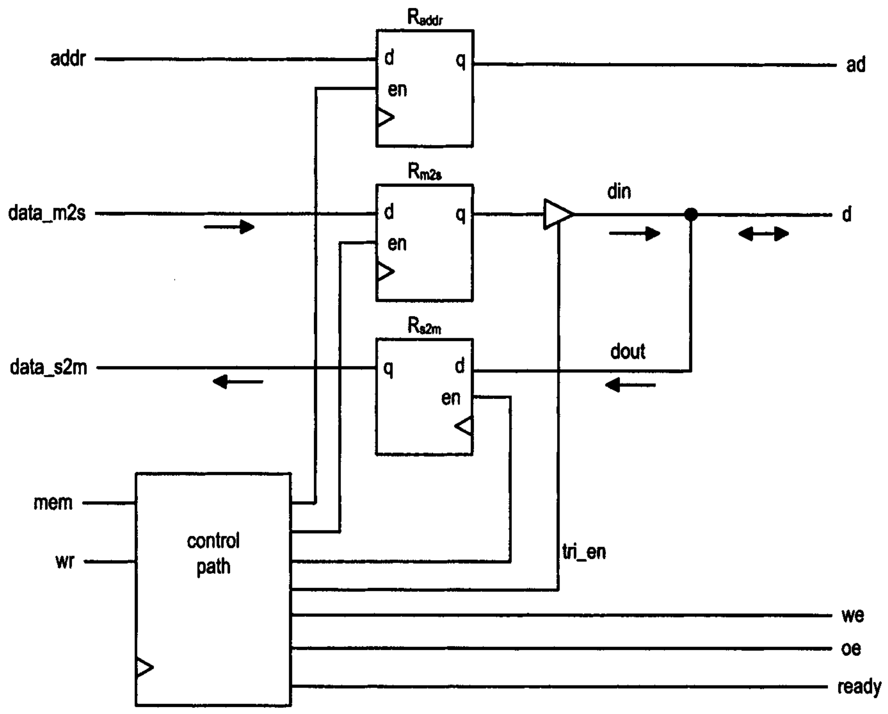


Figure 12.8 Block diagram of an SRAM controller.

memory operation is required and uses rw to specify the type of operation ('0' for write and '1' for read). The SRAM controller uses the ready signal to indicate whether it is ready for the operation. The addr signal is the address used to indicate the location of the memory. The data_m2s and data_s2m signals are the data transferred from the main system to the SRAM and from the SRAM to the main system respectively.

The main system treats the memory operation as a synchronous operation. For a write operation, it activates mem, makes rw '0', and places the address on addr and data on data_m2s for one clock cycle. At the rising edge of the clock, this information will be sampled by the SRAM controller, which, in turn, initiates an SRAM write cycle and generates proper control signals. It may take several clock cycles to complete an operation. For a read operation, the main system activates mem, makes rw '1', and places the address on addr for one clock cycle. Again, this information will be sampled by the SRAM controller at the rising edge of the clock, and an SRAM read cycle is initiated. After a predetermined number of clock cycles, the SRAM controller will put the data on data_s2m and make the data available to the main system.

Note that the main system and memory controller are controlled by the same clock. From the main system's point of view, the memory operation is completely synchronous. The combined memory controller and SRAM function somewhat like the register file of Section 9.3.1. However, whereas accessing a location in a register file can be done in one clock cycle, it takes many clock cycles to complete an SRAM read or write operation.

The block diagram of the SRAM controller is shown in Figure 12.8. The data path contains three registers, R_{addr}, R_{m2s} and R_{s2m}, which are used to store the address, the data from the main system to the SRAM, and the data from the SRAM to the main system respectively. Since the data input of the SRAM is bidirectional, a tri-state buffer is used to avoid conflict. The output from the register R_{m2s} will be placed in the data line, `d`, when the tri-state buffer is enabled.

The control path coordinates the overall SRAM access and generates the control signals, which include the `we` and `oe` signals of the SRAM and the enable signals of tri-state buffer and registers in the data path. There are several requirements for these control signals. First, the signals must be activated in the order specified in the read and write cycles. Second, the signals must meet various timing constraints of the SRAM. Finally, the signals need to ensure that there is no conflict (i.e., fighting) on the bidirectional data line.

12.3.3 Control path of an SRAM controller

We design the control path in two steps:

- Derive a sketch of an FSM according to the activities in read and write cycles.
- Refine the FSM with the actual SRAM timing parameters and clock period.

In the first step, we derive a sketch of an FSM that can activate and deactivate various signals in the desired order. This can be done by dividing the read or write cycles into multiple parts according to the activities of the signals and assigning a state for each part. For example, the write cycle can be divided into five parts, as shown in Figure 12.9(a).

A segment of an FSM can be constructed accordingly, as shown in Figure 12.10(a). We assume that the address and data are stored into the registers before the FSM moves to the `s1` state. The data can be placed on the bidirectional line by activating the `tri_en` signal. The task of the FSM is essentially to generate two output signals, `we` and `tri_en`, in the following order: "10", "00", "01", "11" and "10".

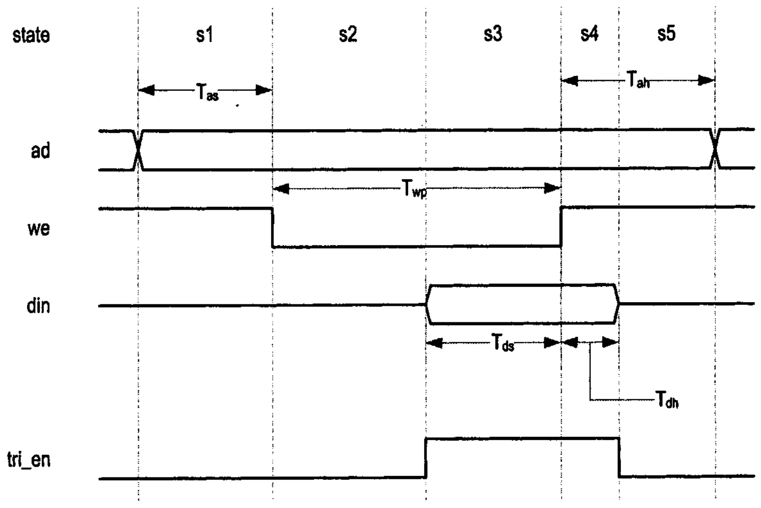
Closer examination of the SRAM's timing specifications can help us to simplify the FSM. For example, T_{wp} is normally much larger than T_{ds} in most SRAMs, and there is no harm in placing data on the `din` line earlier. Thus, we can merge the `s2` and `s3` states into a single state. Also, since there in no constraint specified between the order of deactivation of address and data, we can merge the `s4` and `s5` states. The revised division and FSM segment are shown in Figures 12.9(b) and 12.10(b) respectively.

There are two issues with the initial sketch. First, the length of a state in the FSM corresponds to the period of the clock signal. The period must be large enough to accommodate the most strenuous timing parameter. Since T_{wp} is much larger than other parameters, the time allocated to T_{as} and T_{dh} in the `ss1` and `ss3` states are unnecessarily inflated. Second, in a practical design, the memory controller is usually a part of a larger system, and the clock rate is determined by the main system. The memory controller cannot have a separate clock and must work with the system clock.

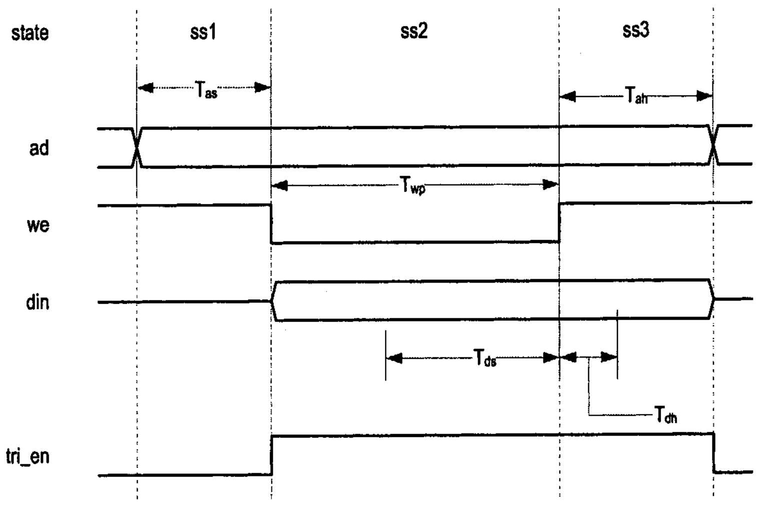
In the second step, we refine the FSM according to the system clock period and SRAM timing parameters. The SRAM's access time and the main system's clock rate are the two key factors in the final design of the control path. The following examples illustrate the design and relevant issues for a slow SRAM and a fast SRAM.

Control path for a slow SRAM The term *slow* here means that the SRAM's address access time (T_{aa}) is relatively large to the main system's clock period. For example, if we assume that the main system's clock period is 25 ns (i.e., the clock rate is 40 MHz), the 120-ns SRAM shown in Table 12.1 will be considered as a slow SRAM to this system since it takes about five clock cycles to complete a memory operation.

Because of the slow SRAM speed, it takes five (i.e., $\lceil \frac{120}{25} \rceil$) clock cycles to cover T_{aa} and three (i.e., $\lceil \frac{70}{25} \rceil$) cycles to cover T_{wp} We need to use multiple states in the FSM to



(a) Five-state division



(b) Three-state division

Figure 12.9 Divisions of a write cycle.

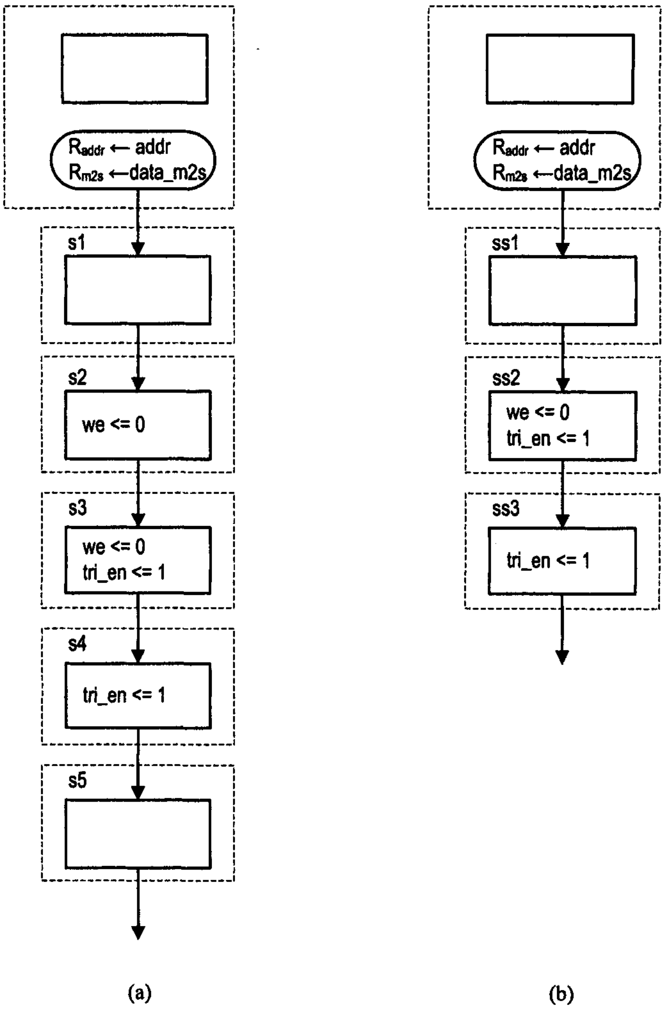

$R_{addr} \leftarrow$ addr
$R_{m2s} \leftarrow$data_m2s
s1
s2
we <= 0
s3
we <= 0
tri_en <= 1
s4
tri_en <= 1
s5
(a)
$R_{addr} \leftarrow$ addr
$R_{m2s} \leftarrow$data_m2s
ss1
ss2
we <= 0
tri_en <= 1
ss3
tri_en <= 1
(b)

Figure 12.10 FSM segments for a write cycle.

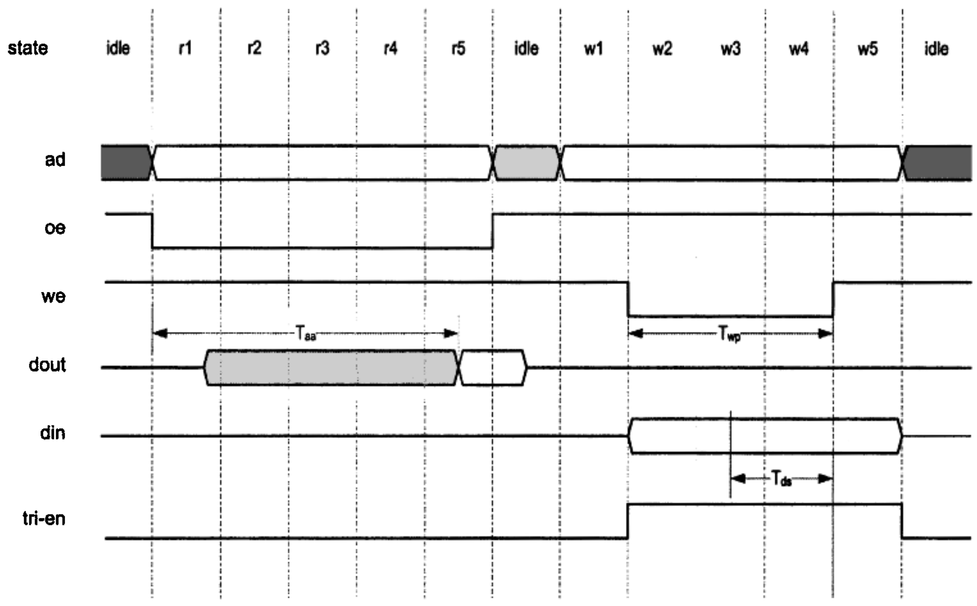


Figure 12.11 Division of read and write cycles of a slow SRAM.

accommodate the timing requirement. Figure 12.11 shows the division in the write and read cycles. An extra clock cycle, which represents the `idle` state, is inserted between the two operations. A read or write operation takes six clock cycles (i.e., 150 ns) to complete. The periods include one for the `idle` state and five for a read or write cycle. We can do a quick check on the SRAM timing parameters:

- T_{aa}: 120 ns < 125 ns (5*25 ns)
- T_{wp}: 70 ns < 75 ns (3*25 ns)
- T_{as}: 20 ns < 25 ns
- T_{ah}: 5 ns < 25 ns
- T_{ds}: 35 ns < 75 ns (3*25 ns)
- T_{dh}: 5 ns < 25 ns

It is clear that all timing parameters are satisfied and there is a margin of at least 5 ns.

The quick check is based on an ideal FSMD. To obtain more detailed timing information, we also need to consider the various propagation delays introduced by the data path and control path of the memory controller. For example, the `oe` signal is disabled in the end of the `r5` state and the data on the `d` line is sampled and stored when the FSM moves from the `r5` state to the `idle` state. We must perform a detailed timing analysis to ensure that there is no setup or hold time violation for the R_{s2m} register. The detailed timing diagram is shown in Figure 12.12. The read operation progresses as follows. At t_1, the FSM moves to the `r1` state. After the T_{ctrl} delay (at t_2), the `oe` signal is activated and the SRAM starts the read operation. After T_{aa} (at t_3), the data is available. At t_4, the FSM moves from the `r5` state to the `idle` state, and the memory controller samples and stores the data into the R_{s2m} register. After the T_{ctrl} delay (at t_5), the `oe` signal is deactivated. The data line (`dout`) of the SRAM returns to the high-impedance state after the T_{oz} delay (at t_6).

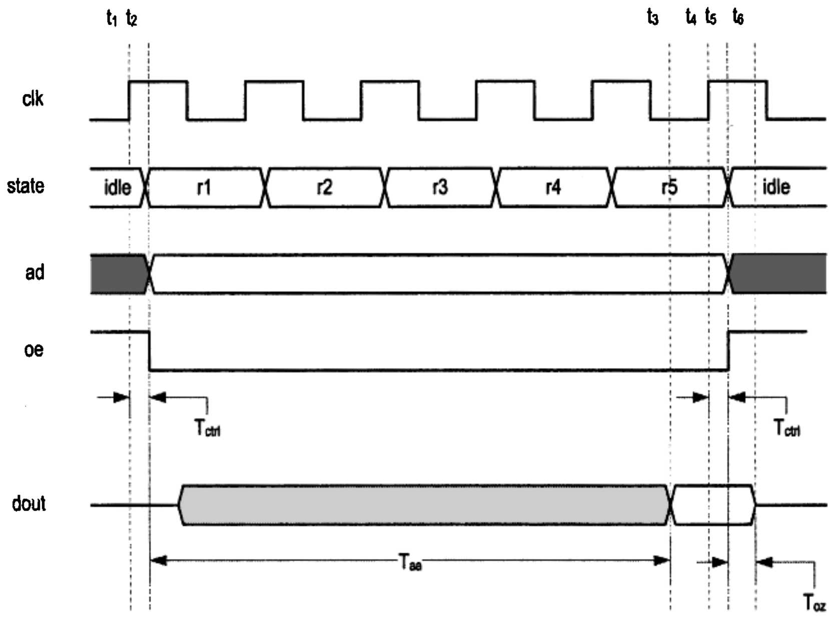


Figure 12.12 Detailed timing diagram of the read cycle.

To avoid the setup time violation, the data has to be stable before T_{setup} of the rising edge of the clock; that is,

$$T_{setup} < 5T_c - T_{ctrl} - T_{aa}$$

We can use the look-ahead output buffer to minimize T_{ctrl} and reduce it to the clock-to-q delay (T_{cq}) of the FF. With a 25-ns clock and 120-ns SRAM, the inequality becomes

$$T_{setup} + T_{cq} < 5 \text{ ns}$$

This condition can be met by most of today's device technology.

To avoid the hold time violation, the data has to be stable after T_{hold} of the rising edge of the clock:

$$T_{hold} < T_{ctrl} + T_{oz}$$

Since T_{ctrl} is T_{cq}, this condition can be easily satisfied.

Other timing requirements, such as the data bus conflict, the exact timing on the SRAM's T_{ds} and T_{dh} requirement, can be analyzed in a similar way. Because of the relatively large safety margin of this design, the initial checking should still be valid.

Following the division and signal activation, we can derive the ASMD chart accordingly, as shown in Figure 12.13. The VHDL code of the complete memory controller is shown in Listing 12.5. It includes both the data path and control path. Note that we use the look-ahead output buffer scheme for the we, oe and tri_en signals to ensure that the signals are glitch-free and to minimize the clock-to-output delay.

Listing 12.5 Memory controller of a slow SRAM

```
library ieee;
use ieee.std_logic_1164.all;
entity sram_ctrl is
```

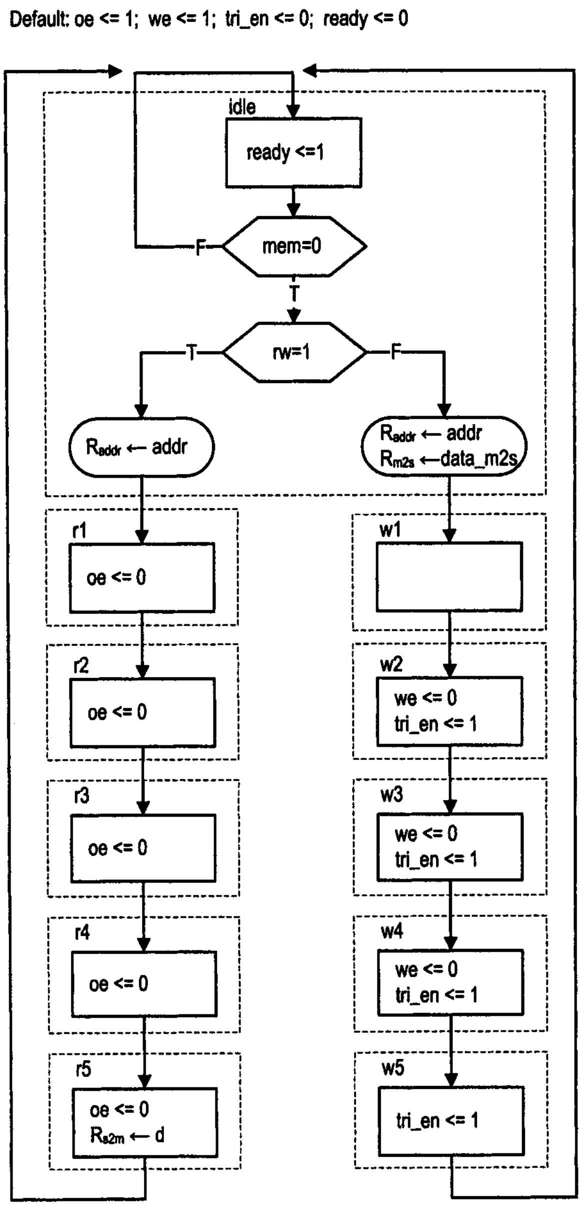


Figure 12.13 ASMD chart for a slow SRAM controller.

```
   port(
      clk, reset: in std_logic;
      mem: in std_logic;
      rw: in std_logic;
      addr: in std_logic_vector(19 downto 0);
      data_m2s: in std_logic;
      we, oe: out std_logic;
      ready: out std_logic;
      data_s2m: out std_logic;
      d: inout std_logic;
      ad: out std_logic_vector(19 downto 0)
   );
end sram_ctrl;

architecture arch of sram_ctrl is
   type state_type is
      (idle, r1, r2, r3, r4, r5, w1, w2, w3, w4, w5);
   signal state_reg, state_next: state_type;
   signal data_m2s_reg, data_m2s_next: std_logic;
   signal data_s2m_reg, data_s2m_next: std_logic;
   signal addr_reg, addr_next: std_logic_vector(19 downto 0);
   signal tri_en_buf, we_buf, oe_buf: std_logic;
   signal tri_en_reg, we_reg, oe_reg: std_logic;
begin
   -- state & data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         addr_reg <= (others=>'0');
         data_m2s_reg <= '0';
         data_s2m_reg <= '0';
         tri_en_reg <= '0';
         we_reg <= '1';
         oe_reg <='1';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         addr_reg <= addr_next;
         data_m2s_reg <= data_m2s_next;
         data_s2m_reg <= data_s2m_next;
         tri_en_reg <= tri_en_buf;
         we_reg <= we_buf;
         oe_reg <= oe_buf;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,mem,rw,d,addr,data_m2s,
           data_m2s_reg,data_s2m_reg,addr_reg)
   begin
      addr_next <= addr_reg;
      data_m2s_next <= data_m2s_reg;
      data_s2m_next <= data_s2m_reg;
      ready <= '0';
      case state_reg is
         when idle =>
            if mem='0' then
               state_next <= idle;
            else
               if rw='0' then --write
                  state_next <= w1;
                  addr_next <= addr;
                  data_m2s_next <= data_m2s;
               else -- read
                  state_next <= r1;
                  addr_next <= addr;
               end if;
            end if;
            ready <= '1';
         when w1 =>
            state_next <= w2;
         when w2 =>
            state_next <= w3;
         when w3 =>
            state_next <= w4;
         when w4 =>
            state_next <= w5;
         when w5 =>
            state_next <= idle;
         when r1 =>
            state_next <= r2;
         when r2 =>
            state_next <= r3;
         when r3 =>
            state_next <= r4;
         when r4 =>
            state_next <= r5;
         when r5 =>
            state_next <= idle;
            data_s2m_next <= d;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      tri_en_buf <='0';
      oe_buf <= '1';
      we_buf<= '1';
      case state_next is
         when idle =>
         when w1 =>
         when w2 =>
            we_buf <= '0';
            tri_en_buf <= '1';
         when w3 =>
            we_buf <= '0';
            tri_en_buf <= '1';
            when w4 =>
               we_buf <= '0';
               tri_en_buf <= '1';
            when w5 =>
               tri_en_buf <= '1';
            when r1 =>
               oe_buf <= '0';
            when r2 =>
               oe_buf <= '0';
            when r3 =>
               oe_buf <= '0';
            when r4 =>
               oe_buf <= '0';
            when r5 =>
               oe_buf <= '0';
         end case;
      end process;
      -- output
      we <= we_reg;
      oe <= oe_reg;
      ad <= addr_reg;
      d <= data_m2s_reg when tri_en_reg ='1' else 'Z';
      data_s2m <= data_s2m_reg;
   end arch;
```

Control path for a fast SRAM The major problem with the previous memory system is its speed. Since it takes six clock cycles to read or write a data item from the SRAM, it can be used only if the main system accesses the memory sporadically. One way to improve the memory performance is to use a faster SRAM. For example, we can use the 20-ns SRAM of Table 12.1, whose address access time (T_{aa}) is smaller than the 25-ns clock period of the main system. Figure 12.14 shows the timing of one possible design, in which a read cycle and a write cycle are done in one clock cycle. We can again check the division against the SRAM timing parameters:

- T_{aa}: 20 ns < 25 ns
- T_{wp}: 12 ns < 25 ns
- T_{as}: 0 ns $\leq$ 0 ns
- T_{ah}: 0 ns $\leq$ 0 ns
- T_{ds}: 12 ns < 25 ns
- T_{dh}: 0 ns $\leq$ 0 ns

Although all constraints are satisfied, the timing is very tight. The timing of T_{as}, T_{ah} and T_{dh} just meet the specification and there is no safety margin. The propagation delays of the control path and data path may cause timing violations. We may need to manually fine-tune the propagation delays of various signals to ensure correct operation.

In this design, a read or write operation requires two clock cycles because the FSM must return to the `idle` state after each operation. Since performance is the goal of a fast SRAM controller, it is desirable to perform back-to-back memory operations without returning to the `idle` state. This requires an ad hoc circuit to generate a `we` pulse whose activation time is only a fraction of a clock period and more manual fine-tuning on propagation delays to avoid data bus fighting.

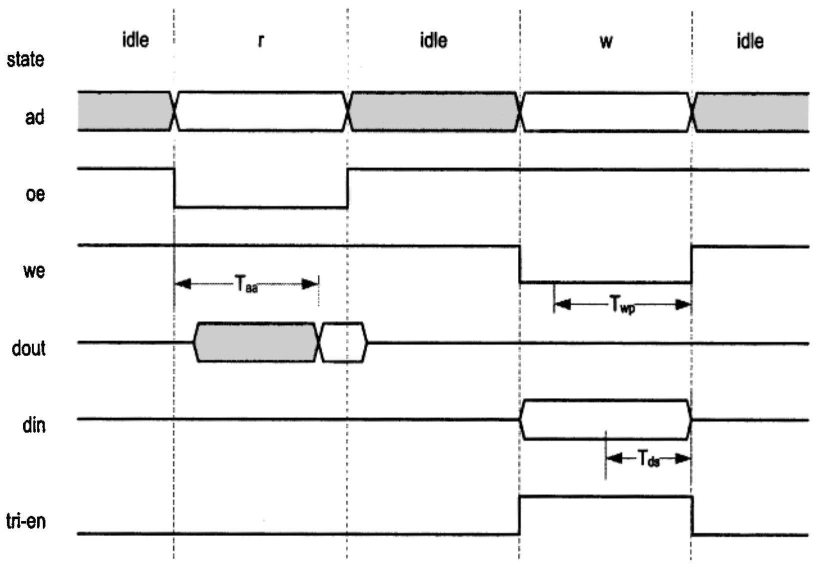


Figure 12.14 Division of read and write cycles of a fast SRAM.

In summary, while it is possible to perform single-clock back-to-back memory operations, the design imposes very strict and tricky timing requirements on the control signals. These requirements are delay-sensitive and cannot be expressed or implemented by a regular FSM. This kind of circuit is not suitable for RT-level synthesis. To implement the controller, we need to manually derive the schematic using cells from the device library and even to manually do the placement and routing. Many device manufacturers have recognized the design difficulty and incorporated the memory controller into a memory chip. This kind of device is known as *synchronous memory*. Since the main system only needs to issue commands, place address and data, or retrieve data at rising edges of the clock, this type of device greatly simplifies the memory interface to a synchronous system.

12.4 GCD CIRCUIT

The $\gcd(a, b)$ function returns the greatest common divisor (GCD) of two positive integers, a and b. For example, $\gcd(1, 10)$ is 1 and $\gcd(12, 9)$ is 3. The gcd function can be obtained by using subtraction, which is based on the equation

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b, b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } a < b \end{cases}$$

Assume that a_in and b_in are positive nonzero integers and their GCD is r. The equation can easily be converted into the following pseudocode:

```
a = a_in;
b = b_in;
while (a /= b) {
   if (a > b) then
      a = a - b;
```

```
            else
               b = b - a;
            end if
         }
         r = a;
```

To make the pseudocode more compatible with the ASMD chart, we convert the while loop into a goto statement and use a swap operation to reduce the number of required subtractions. The revised pseudocode becomes

```
         a = a_in;
         b = b_in;
  swap:  if (a = b) then
            goto stop;
         else
            if (a < b) then -- swap a and b
               a = b;   -- assume the two operations
               b = a;   -- can be done in parallel
            end if;
            a = a - b;
            goto swap;
         end if;
  stop:  r = a;
```

The code first moves the larger value into a and then performs a single subtraction of a - b. This code can easily be converted into an ASMD chart, as shown in Figure 12.15. As the sequential multiplier circuit of Chapter 11, the `start` and `ready` signals are added to interface external systems. The corresponding VHDL code is shown in Listing 12.6.

Listing 12.6 Initial implementation of a GCD circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gcd is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(7 downto 0)
   );
end gcd ;

architecture slow_arch of gcd is
   type state_type is (idle, swap, sub);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
```

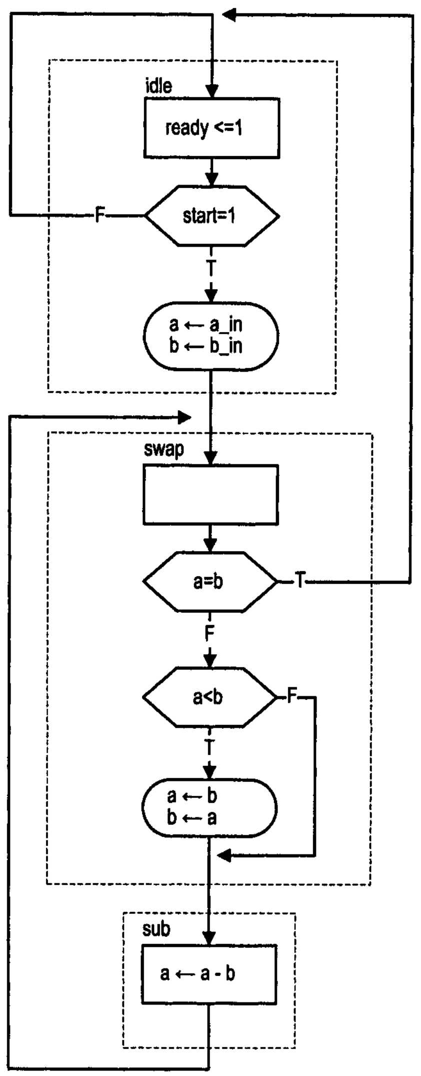


Figure 12.15 ASMD chart of the initial GCD circuit.

```
         b_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         b_reg <= b_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,a_reg,b_reg,start,a_in,b_in)
   begin
      a_next <= a_reg;
      b_next <= b_reg;
      case state_reg is
         when idle =>
            if start='1' then
               a_next <= unsigned(a_in);
               b_next <= unsigned(b_in);
               state_next <= swap;
            else
               state_next <= idle;
            end if;
         when swap =>
            if (a_reg=b_reg) then
               state_next <= idle;
            else
               if (a_reg < b_reg) then
                  a_next <= b_reg;
                  b_next <= a_reg;
               end if;
               state_next <= sub;
            end if;
         when sub =>
            a_next <= a_reg - b_reg;
            state_next <= swap;
         end case;
   end process;
   -- output
   ready <= '1' when state_reg=idle else '0';
   r <= std_logic_vector(a_reg);
end slow_arch;
```

As discussed in Section 11.5, one factor in the performance of an FSMD is the number of clock cycles required to complete the computation. In this design, the input values are subtracted successively until the a_reg=b_reg condition is reached. The number of clock cycles required to complete computation of this GCD circuit depends on the input values. It requires more time if only a small value is subtracted each time. The calculation of $\gcd(1, 2^8 - 1)$ represents the worst-case scenario. The loop has to be repeated $2^8 - 1$ times until the two values are equal. For a circuit with an N-bit input, the computation time is on the order of $O(2^N)$, and thus this is not an effective design.

One way to improve the design is to take advantage of the binary number system. For a binary number, we can tell whether it is odd or even by checking the LSB. Based on the

LSBs of two inputs, several simplification rules can be applied in the derivation of the GCD function:

- If both a and b are even, $\gcd(a,b) = 2\gcd(\frac{a}{2}, \frac{b}{2})$.
- If a is odd and b is even, $\gcd(a,b) = \gcd(a, \frac{b}{2})$.
- If a is even and b is odd, $\gcd(a,b) = \gcd(\frac{a}{2}, b)$.

Since the divided-by-2 operation corresponds to shifting right one position, it can be implemented easily in hardware. The previous equation can be extended:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ 2\gcd(\frac{a}{2}, \frac{b}{2}) & \text{if } a \neq b \text{ and } a, b \text{ even} \\ \gcd(a, \frac{b}{2}) & \text{if } a \neq b \text{ and } a \text{ odd, } b \text{ even} \\ \gcd(\frac{a}{2}, b) & \text{if } a \neq b \text{ and } a \text{ even, } b \text{ odd} \\ \gcd(a-b, b) & \text{if } a > b \text{ and } a, b \text{ odd} \\ \gcd(a, b-a) & \text{if } a < b \text{ and } a, b \text{ odd} \end{cases}$$

To incorporate the new rules into the algorithm, the main issue is how to handle computation of $2\gcd(\frac{a}{2}, \frac{b}{2})$. One way is ignoring the factor 2 in initial iterations and using an additional register, n, to keep track of the number of occurrences in which both operands are even. The final GCD value can be restored by multiplying the initial result by 2^n, which corresponds to shifting the initial result left n positions.

The expanded ASMD chart is shown in Figure 12.16. It has several modifications. In the swap state, the LSBs of the a and b registers are checked. The register is shifted right one position (i.e., divided by 2) if it is even. Furthermore, the n register is incremented if both are even. If the a and b registers are odd, they are compared and, if necessary, swapped, and the FSM moves to the sub state. An extra state, labeled res (for "restore"), is added to restore the final GCD value. The initial result in a is shifted left repeatedly until the n counter reaches 0. The corresponding VHDL code is shown in Listing 12.7.

Listing 12.7 More efficient implementation of a GCD circuit

```
architecture fast_arch of gcd is
   type state_type is (idle, swap, sub, res);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
   signal n_reg, n_next: unsigned(2 downto 0);
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         b_reg <= (others=>'0');
         n_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         b_reg <= b_next;
         n_reg   <= n_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
```

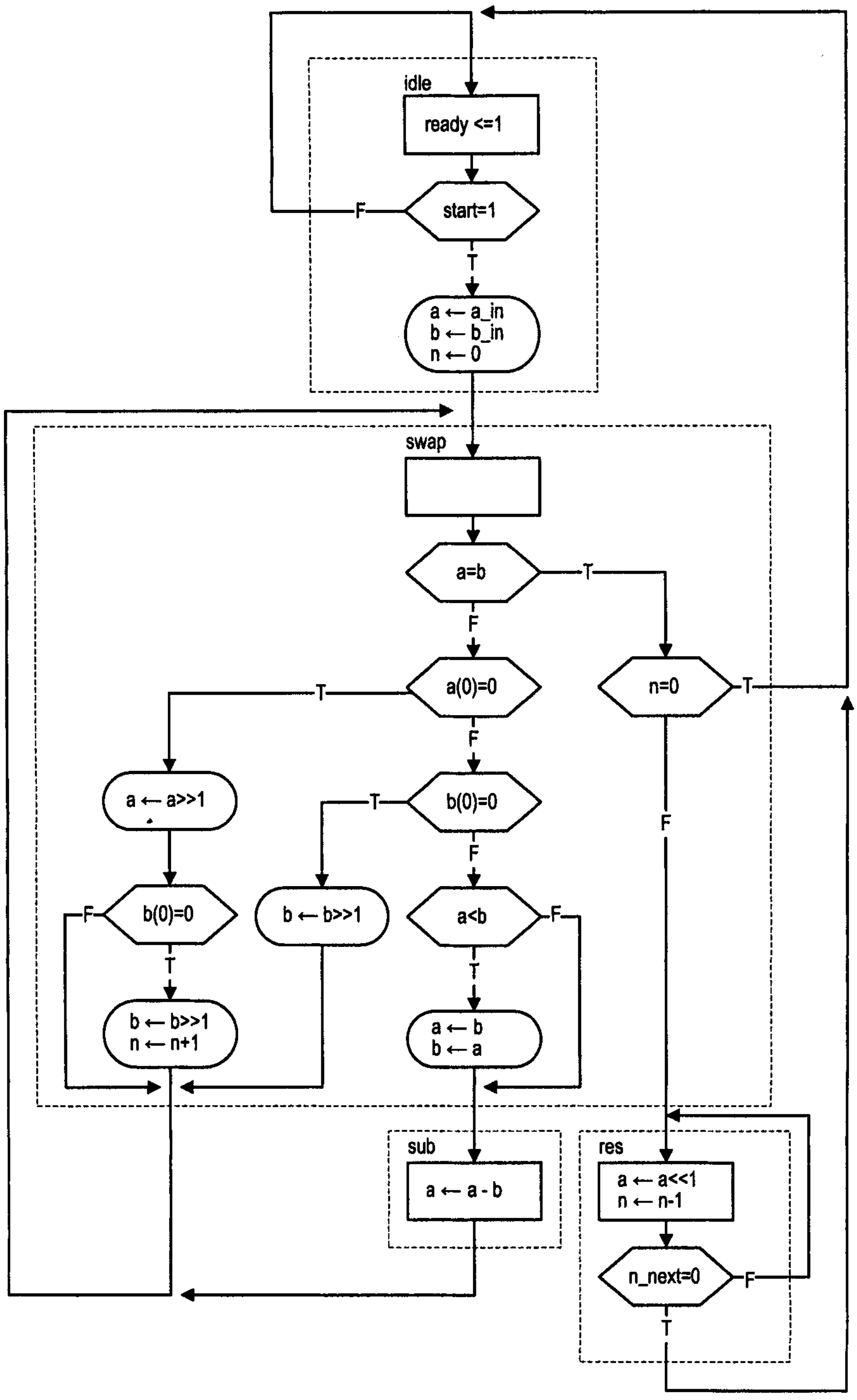


Figure 12.16 ASMD chart of the revised GCD circuit.

```
process(state_reg,a_reg,b_reg,n_reg,start,a_in,b_in,n_next)
begin
   a_next <= a_reg;
   b_next <= b_reg;
   n_next <= n_reg;
   case state_reg is
      when idle =>
         if start='1' then
            a_next <= unsigned(a_in);
            b_next <= unsigned(b_in);
            n_next <= (others=>'0');
            state_next <= swap;
         else
            state_next <= idle;
         end if;
      when swap =>
         if (a_reg=b_reg) then
            if (n_reg=0) then
               state_next <= idle;
            else
               state_next <= res;
            end if;
         else
            if (a_reg(0)='0') then -- a_reg even
               a_next <= '0' & a_reg(7 downto 1);
               if (b_reg(0)='0') then  -- both even
                  b_next <= '0' & b_reg(7 downto 1);
                  n_next <= n_reg + 1;
               end if;
               state_next <= swap;
            else -- a_reg odd
               if (b_reg(0)='0') then  -- b_reg even
                  b_next <= '0' & b_reg(7 downto 1);
                  state_next <= swap;
               else -- both a_reg and b_reg odd
                  if (a_reg < b_reg) then
                     a_next <= b_reg;
                     b_next <= a_reg;
                  end if;
                  state_next <= sub;
               end if;
            end if;
         end if;
      when sub =>
         a_next <= a_reg - b_reg;
         state_next <= swap;
      when res =>
         a_next <= a_reg(6 downto 0) & '0';
         n_next <= n_reg - 1;
         if (n_next=0) then
            state_next <= idle;
         else
            state_next <= res;
            end if;
         end case;
      end process;
      --output
      ready <= '1' when state_reg=idle else '0';
      r <= std_logic_vector(a_reg);
   end fast_arch;
```

Now let us consider the number of clock cycles needed to complete one computation. Assume that the width of the input operand is N bits. The algorithm gradually reduces the values in the a_reg and b_reg until they are equal. In the worst case, there are $2N$ bits to be processed initially. If a value is even, the LSB is shifted out and thus the number of bits is reduced by 1. If both values are odd, a subtraction is performed and the difference is even, and the number of bits can be reduced by 1 in the next iteration. In the most pessimistic scenario, the $2N$ bits can be processed in $2 * 2N$ iterations, and the required computation time is on the order of $O(N)$, which is much better than the $O(2^N)$ of the original algorithm.

Because of the flexibility of hardware implementation, it is possible to invest extra hardware resources to improve the performance. For example, instead of handling the data bit by bit in the swap and res states, we can use more sophisticated combinational circuits to process the data in parallel. In the swap state, the circuit checks and shifts out the trailing 0's of a and b. In the res state, a shift-left barrel shifter restores the final result in a single step. The revised VHDL code is shown in Listing 12.8.

Listing 12.8 Performance-oriented implementation of a GCD circuit

```
architecture fastest_arch of gcd is
   type state_type is (idle, swap, sub, res);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
   signal n_reg, n_next, a_zero, b_zero: unsigned(2 downto 0);
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         b_reg <= (others=>'0');
         n_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         b_reg <= b_next;
         n_reg   <= n_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,a_reg,b_reg,n_reg,start,
           a_in,b_in,a_zero,b_zero)
   begin
      a_next <= a_reg;
      b_next <= b_reg;
      n_next <= n_reg;
      a_zero <= (others=>'0');
      b_zero <= (others=>'0');
      case state_reg is
         when idle =>
            if start='1' then
               a_next <= unsigned(a_in);
               b_next <= unsigned(b_in);
               n_next <= (others=>'0');
               state_next <= swap;
            else
               state_next <= idle;
            end if;
         when swap =>
            if (a_reg=b_reg) then
               if (n_reg=0) then
                  state_next <= idle;
               else
                  state_next <= res;
               end if;
            else
               if (a_reg(0)='1' and b_reg(0)='1') then -- swap
                  if (a_reg < b_reg) then
                     a_next <= b_reg;
                     b_next <= a_reg;
                  end if;
                  state_next <= sub;
               else
                  -- shift out 0s of a_reg
                  if (a_reg(0)='1') then
                       a_zero <="000";
                  elsif (a_reg(1)='1') then
                     a_next <= "0" & a_reg(7 downto 1);
                       a_zero <="001";
                  elsif (a_reg(2)='1') then
                     a_next <= "00" & a_reg(7 downto 2);
                       a_zero <="010";
                  elsif (a_reg(3)='1') then
                     a_next <= "000" & a_reg(7 downto 3);
                       a_zero <="011";
                  elsif (a_reg(4)='1') then
                     a_next <= "0000" & a_reg(7 downto 4);
                       a_zero <="100";
                  elsif (a_reg(5)='1') then
                     a_next <= "00000" & a_reg(7 downto 5);
                       a_zero <="101";
                  elsif (a_reg(6)='1') then
                     a_next <= "000000" & a_reg(7 downto 6);
                       a_zero <="110";
                  else  -- a_reg(7)='1'
                     a_next <= "0000000" & a_reg(7);
                       a_zero <="111";
                  end if;
                  -- shift out 0s of b_reg
```

```
                    if (b_reg(0)='1') then
                       b_zero <="000";
                    elsif (b_reg(1)='1') then
                       b_next <= "0" & b_reg(7 downto 1);
                       a_zero <="001";
                    elsif (b_reg(2)='1') then
                       b_next <= "00" & b_reg(7 downto 2);
                       b_zero <="010";
                    elsif (b_reg(3)='1') then
                       b_next <= "000" & b_reg(7 downto 3);
                       b_zero <="011";
                    elsif (b_reg(4)='1') then
                       b_next <= "0000" & b_reg(7 downto 4);
                       b_zero <="100";
                    elsif (b_reg(5)='1') then
                       b_next <= "00000" & b_reg(7 downto 5);
                       b_zero <="101";
                    elsif (b_reg(6)='1') then
                       b_next <= "000000" & b_reg(7 downto 6);
                       b_zero <="110";
                    else  -- b_reg(7)='1'
                       b_next <= "0000000" & b_reg(7);
                       b_zero <="111";
                    end if;
                    -- find common number of 0s
                    if (a_zero > b_zero) then
                       n_next <= n_reg + b_zero;
                    else
                       n_next <= n_reg + a_zero;
                    end if;
                    state_next <= swap;
                 end if;
              end if;
           when sub =>
              a_next <= a_reg - b_reg;
              state_next <= swap;
           when res =>
              case n_reg is
                 when "000" =>
                    a_next <= a_reg;
                 when "001" => a_next <=
                    a_reg(6 downto 0) & '0';
                 when "010" =>
                    a_next <= a_reg(5 downto 0) & "00";
                 when "011" =>
                    a_next <= a_reg(4 downto 0) & "000";
                 when "100" => a_next <=
                    a_reg(3 downto 0) & "0000";
                 when "101" =>
                    a_next <= a_reg(2 downto 0) & "00000";
                 when "110" =>
                    a_next <= a_reg(1 downto 0) & "000000";
                 when others =>
```

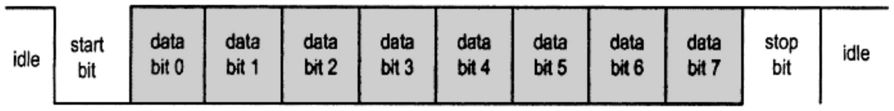


Figure 12.17 Transmission of a byte.

```
                    a_next <= a_reg(0) & "0000000";
              end case;
              state_next <= idle;
        end case;
     end process;
     -- output
     ready <= '1' when state_reg=idle else '0';
     r <= std_logic_vector(a_reg);
  end fastest_arch;
```

12.5 UART RECEIVER

Universal asynchronous receiver and transmitter (UART) is a scheme that sends bytes of data through a serial line. The transmission of a single byte is shown in Figure 12.17. The serial line is in the '1' state when it is idle. The transmission is started with a *start bit*, which is '0', followed by eight data bits and ended with a *stop bit*, which is '1'. It is also possible to insert an optional parity bit in the end of the data bits to perform error detection. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits, and use of the parity bit.

The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. Construction of a UART receiver is more involved since no clock information is conveyed through the serial line. The receiver can retrieve the data bits only by using the predetermined parameters. It uses an oversampling scheme to ensure that the data bits are retrieved at the correct point. This scheme utilizes a high-frequency sampling signal to estimate the middle point of a data bit and then retrieve data bits at these points. For example, assume that the sampling rate is 16 times the baud rate (i.e., there are 16 sampling pulses for each bit). The incoming stream can be recovered as follows:

1. When the incoming line becomes '0' (i.e., the beginning of the start bit), initiate the sampling pulse counter.
2. When the counter reaches 7, clear it to 0 and restart. At this point, the incoming signal reaches about a half of the start bit (i.e., the middle point of the start bit).
3. When the counter reaches 15, clear it to 0 and restart. At this point, the incoming signal progresses for one bit and reaches the middle of the first data bit. The data in the serial line should be retrieved and shifted into a register.
4. Repeat Step 3 seven times to retrieve the remaining seven data bits.
5. Repeat Step 3 one more time but without shifting. The incoming signal should reach the middle of the stop bit at this point, and its value should be '1'.

The idea behind this scheme is to use oversampling to overcome the uncertainty of the initiation of the start bit. Even when we don't know the exact onset point of the start bit, it can be off by at most $\frac{1}{16}$. The subsequent data bit retrievals are off by at most $\frac{1}{16}$ from the middle point as well.

With understanding of the oversampling procedure, we can derive the ASMD chart accordingly. One issue is the creation of sampling pulses. The easiest way is to treat the UART as a separate subsystem that utilizes a clock signal whose frequency is just 16 times that of the baud rate. This approach violates the synchronous design principle and should be avoided. A better alternative is to use a single-clock enable pulse that is synchronized with the system clock, as discussed in Section 9.1.3. Assume that the system clock is 1 MHz and the baud rate is 1200 baud. The frequency of the sampling enable pulse should be $16 * 1200$, which can be obtained by a mod-52 counter (note that $\frac{1{,}000{,}000}{16*2000} = 52$). It can easily be coded in VHDL:

```
process(clk,reset)
begin
   if reset='1' then
      clk16_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      clk16_reg <= clk16_next;
   end if;
end process;
-- next-state/output logic
clk16_next <= (others=>'0') when clk16_reg=51 else
                 clk16_reg + 1 ;
s_pulse <= '1' when clk16_reg=0 else '0';
```

The ASMD chart of a simplified UART receiver is shown in Figure 12.18. The chart follows the previous steps and includes three major states, start, data and stop, which represent the processing of the start bit, data bits and stop bit respectively. The s_pulse signal is the enable pulse whose frequency is 16 times that of the baud rate. Note that the FSMD stays in the same state unless the s_pulse signal is activated. There are two counters, represented by the s and n registers. The s register keeps track of the number of sampling pulses and counts to 7 in the start state and to 15 in the data and stop states. The n register keeps track of the number of data bits received in the data state. The retrieved bits are shifted into and reassembled in the b register. The corresponding VHDL code is shown in Listing 12.9. We assume that the system clock is 1 MHz and the baud rate is 1200 baud.

Listing 12.9 Simplified UART receiver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_receiver is
   port(
      clk, reset: in std_logic;
      rx: in std_logic;
      ready: out std_logic;
      pout: out std_logic_vector(7 downto 0)
   );
end uart_receiver ;

architecture arch of uart_receiver is
   type state_type is (idle, start, data, stop);
```

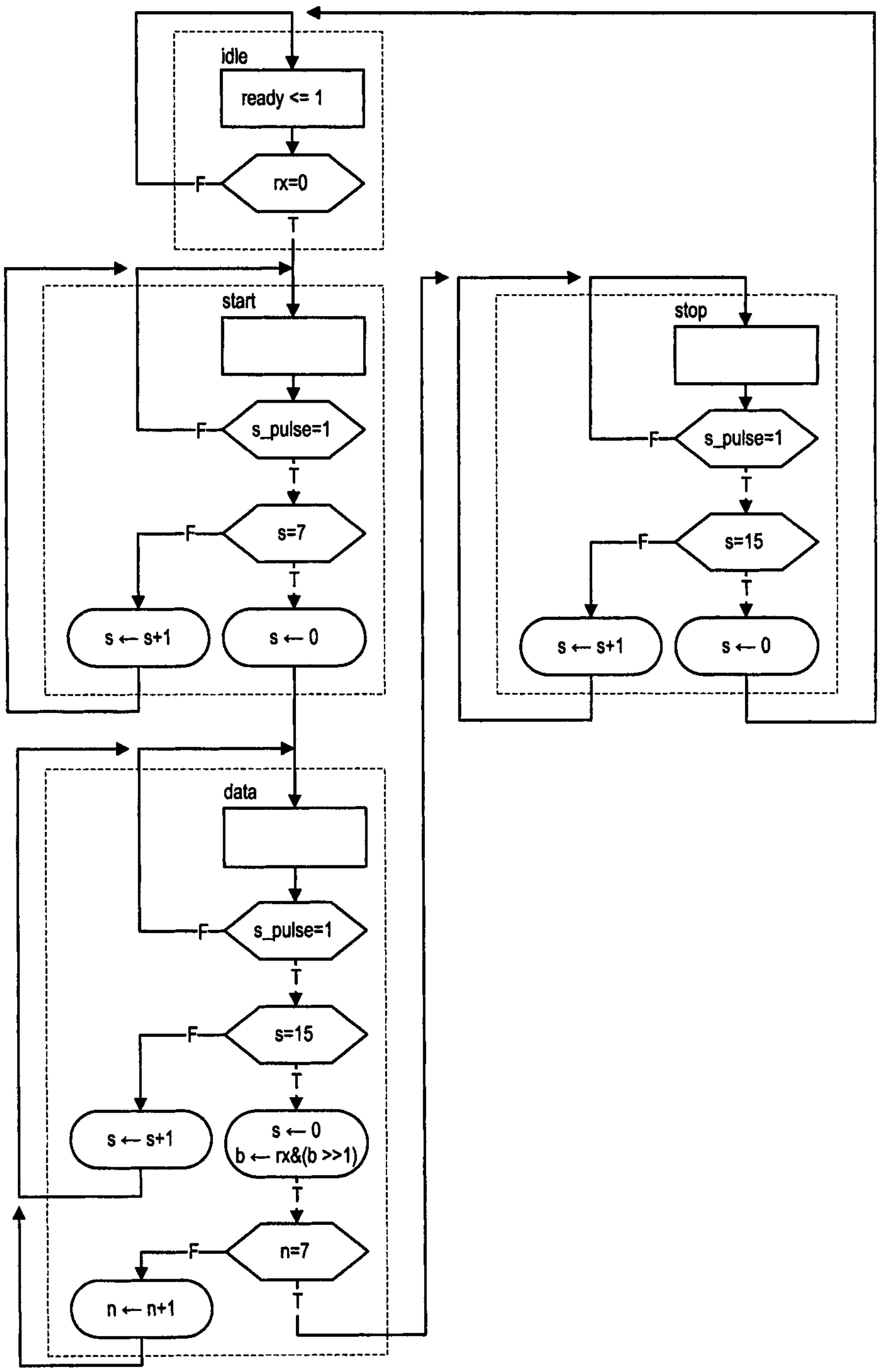


Figure 12.18 ASMD chart of a UART receiver.

```
   signal state_reg, state_next: state_type;
   signal clk16_next, clk16_reg: unsigned(5 downto 0);
   signal s_reg, s_next: unsigned(3 downto 0);
   signal n_reg, n_next: unsigned(2 downto 0);
   signal b_reg, b_next: std_logic_vector(7 downto 0);
   signal s_pulse: std_logic;
   constant DVSR: integer := 52;
begin
   -- free-running mod-52 counter, independent of FSMD
   process(clk,reset)
   begin
      if reset='1' then
         clk16_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         clk16_reg <= clk16_next;
      end if;
   end process;
   -- next-state/output logic
   clk16_next <= (others=>'0') when clk16_reg=(DVSR-1) else
                 clk16_reg + 1 ;
   s_pulse <= '1' when clk16_reg=0 else '0';

   -- FSMD state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         s_reg <= (others=>'0');
         n_reg <= (others=>'0');
         b_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         s_reg <= s_next;
         n_reg <= n_next;
         b_reg <= b_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,s_reg,n_reg,b_reg,s_pulse,rx)
   begin
      s_next <= s_reg;
      n_next <= n_reg;
      b_next <= b_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if rx='0' then
               state_next <= start;
            else
               state_next <= idle;
            end if;
            ready <='1';
         when start =>
```

```
            if (s_pulse = '0') then
               state_next <= start;
            else
               if s_reg=7 then
                  state_next <= data;
                  s_next <= (others=>'0');
               else
                  state_next <= start;
                  s_next <= s_reg + 1;
               end if;
            end if;
         when data =>
            if (s_pulse = '0') then
               state_next <= data;
            else
               if s_reg=15 then
                  s_next <= (others=>'0');
                  b_next <= rx & b_reg(7 downto 1);
                  if n_reg=7 then
                     state_next <= stop ;
                     n_next <= (others=>'0');
                  else
                     state_next <= data;
                     n_next <= n_reg + 1;
                  end if;
               else
                  state_next <= data;
                  s_next <= s_reg + 1;
               end if;
            end if;
         when stop =>
            if (s_pulse = '0') then
               state_next <= stop;
            else
               if s_reg=15 then
                  state_next <= idle;
                  s_next <= (others=>'0');
               else
                  state_next <= stop;
                  s_next <= s_reg + 1;
               end if;
            end if;
      end case;
   end process;
   pout <= b_reg;
end arch;
```

Several extensions are possible for this UART receiver, including adding a parity bit to detect the transmission error, checking the stop bit for the framing error, and making the baud rate adjustable. The main problem with the UART scheme is its performance. Because of the oversampling, the baud rate can be only a small fraction of the system clock rate, and thus this scheme can be used only for a low data rate.

12.6 SQUARE-ROOT APPROXIMATION CIRCUIT

The previous UART example is a typical *control-oriented* application, which is characterized by the dominance of the sophisticated decision conditions and branching structures in the algorithm. The opposite type is a *data-oriented* application, which involves mainly data manipulation and arithmetic operations. It is also known as a *computation-intensive* application.

Although a data-oriented application can be implemented by a combinational circuit in theory, the approach uses a large number of functional units and thus requires a significant amount of hardware resources. RT methodology allows us to share functional units in a time-multiplexed fashion, and we can schedule the operations sequentially to achieve the desired trade-off between performance and circuit complexity. A square-root approximation circuit in this section illustrates the design procedure and relevant issues of data-oriented applications.

The square-root approximation circuit uses simple adder-type components to obtain the approximate value of $\sqrt{a^2+b^2}$, where a and b are signed integers. The approximation is obtained by the following formula:

$$\begin{aligned}\sqrt{a^2+b^2} &\approx \max(((x-0.125x)+0.5y),x)\\ &\quad \text{where } x=\max(|a|,|b|) \text{ and } y=\min(|a|,|b|)\end{aligned}$$

Note that the $0.125x$ and $0.5y$ operations correspond to shift x right three positions and shift y right one position, and that no actual multiplication circuit is needed. The equation can be coded in a traditional programming language. Let the two input operands be a_in and b_in and the output be r. One possible pseudocode is

```
a = a_in;
b = b_in;
t1 = abs(a);
t2 = abs(b);
x = max(t1, t2);
y = min(t1, t2);
t3 = x*0.125;
t4 = y*0.5;
t5 = x - t3;
t6 = t4 + t5;
t7 = max(t6, x)
r = t7;
```

To help VHDL conversion, we intentionally avoid reuse of the same variable name on the left-hand side of the statements. Because of the lack of control structure, the pseudocode can be translated to synthesizable VHDL code directly. The corresponding code is shown in Listing 12.10.

Listing 12.10 Square-root approximation circuit using direct dataflow description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sqrt is
   port(
      a_in, b_in: in std_logic_vector(7 downto 0);
      r: out std_logic_vector(8 downto 0)
   );
end sqrt;

architecture comb_arch of sqrt is
   constant WIDTH: natural:=8;
   signal a, b, x, y: signed(WIDTH downto 0);
   signal t1, t2, t3, t4, t5, t6, t7: signed(WIDTH downto 0);
begin
   a <= signed(a_in(WIDTH-1) & a_in); -- signed extension
   b <= signed(b_in(WIDTH-1) & b_in);
   t1 <= a when a > 0 else
         0 - a;
   t2 <= b when b > 0 else
         0 - b;
   x <= t1 when t1 - t2 > 0 else
        t2;
   y <= t2 when t1 - t2 > 0 else
        t1;
   t3 <= "000" & x(WIDTH downto 3);
   t4 <= "0" & y(WIDTH downto 1);
   t5 <= x - t3;
   t6 <= t4 + t5;
   t7 <= t6 when t6 - x > 0 else
         x;
   r <= std_logic_vector(t7);
end comb_arch;
```

Note that the code consists only of concurrent statements, and thus their order does not matter. The original sequential execution is embedded in the interconnection of components and the flow of data. The VHDL code consists of seven arithmetic components, including one adder and six subtractors. Since the addition and subtractions are not mutually exclusive, sharing is not possible.

For a data-oriented application, it will be helpful to examine the dependency and movement of the data. This information can be visualized by a *dataflow graph*, in which an operation is represented by a node (a circle), and its input and output variables are represented by the incoming and outgoing arcs. The dataflow graph of the square-root approximation algorithm is shown in Figure 12.19.

The graph shows that the algorithm has only a limited degree of parallelism since at most only two operations can be executed concurrently. The seven arithmetic components of the previous VHDL code cannot significantly increase the performance, and most hardware resources are wasted. Thus, while the code is simple, it is not very efficient. RT methodology is a better alternative.

To transform a dataflow chart to an ASMD chart, we need to specify when and how operations in the dataflow graph are executed. The transformation include two major tasks: scheduling and binding. *Scheduling* specifies *when* a function (i.e., a circle) can start execution, and *binding* specifies *which* functional unit is assigned to perform the execution. One important design constraint is the number of functional units allowed to be used in a design. We can allocate a minimal number of functional units to reduce the circuit size, allocate a maximal number of units to exploit full potential parallelism, or find a specific number to achieve the desired trade-off between performance and circuit size. Obtaining

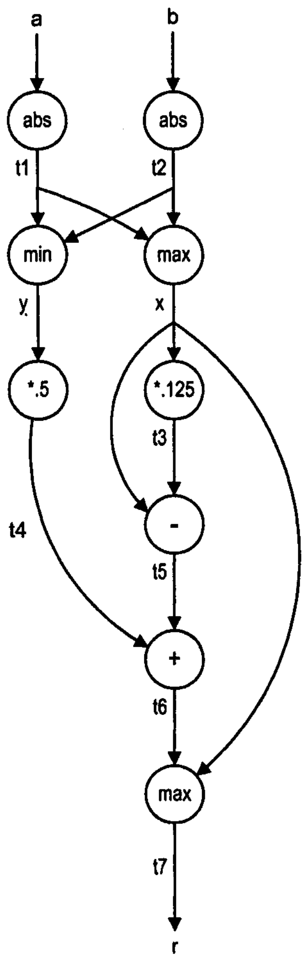
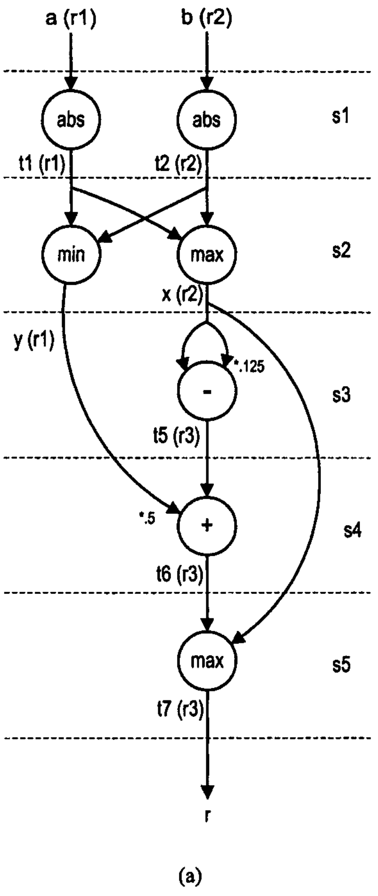


Figure 12.19 Dataflow graph.



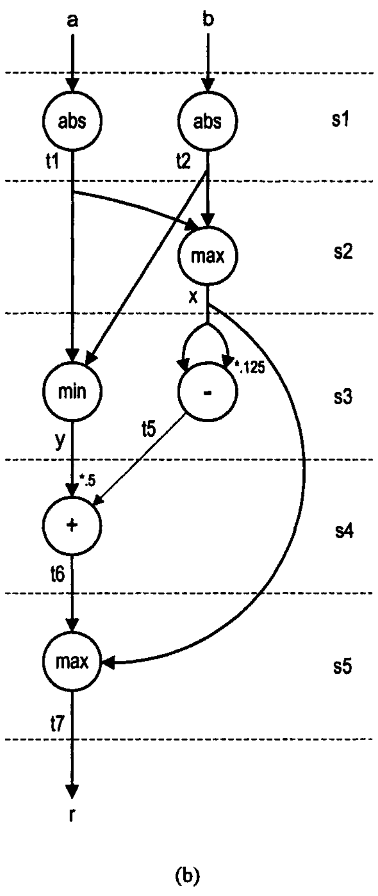


Figure 12.20 Schedules with two functional units.

an optimal schedule involves sophisticated algorithms and is a difficult task. Specialized EDA software tools are needed for a complex dataflow graph.

The dataflow graph of the square-root approximation algorithm involves a variety of operations. The $*.125$ and $*.5$ operations can be implemented by fixed-amount shifting circuits, which require no physical logic and thus should not be considered in the scheduling process. The other operations can be constructed by adders with some "glue" and routing logic. Thus, we can assume that the adder/subtractor is the only functional unit type required for the algorithm. Because at most two operations can be executed in parallel, the ASMD design can only utilize up to two functional units.

One possible schedule is shown in Figure 12.20(a). Note that the $*.125$ and $*.5$ operations are removed from the graph. The parentheses associated with the variables will be explained later. The dataflow graph is divided into five time intervals, which are later mapped into five states of an ASMD chart. It utilizes two units. One possible binding is to assign the two operations in the left column to one unit and the five operations in the right column to another unit. An alternative schedule and binding is shown in Figure 12.20(b), which requires the same amount of time to complete the computation. A schedule that uses only

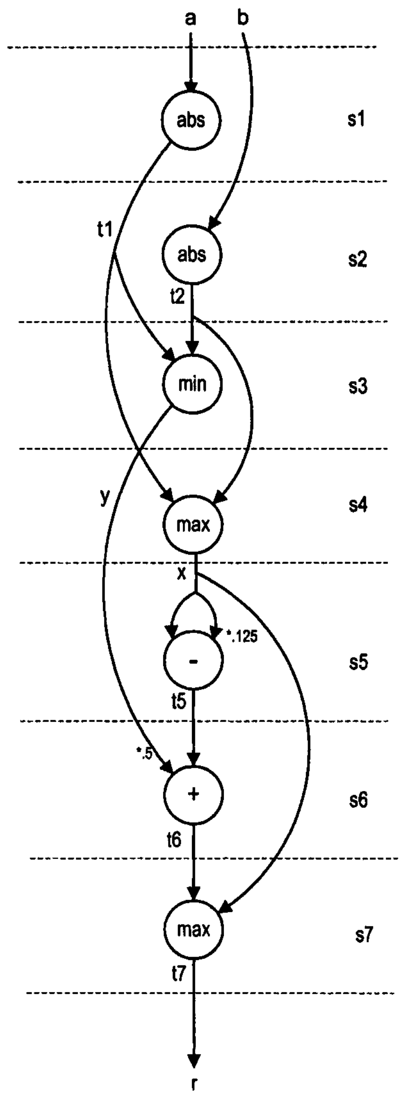


Figure 12.21 Schedule with one functional unit.

one functional unit is shown in Figure 12.21. It needs two extra time intervals to complete the operation.

Once the scheduling and binding are done, the dataflow graph can be transformed into an ASMD chart. Since each time interval represents a state in the chart, a register is needed when a signal is passed through the state boundary. The corresponding ASMD chart of Figure 12.20(a) is shown in Figure 12.22(a). The variables in the graph are mapped into the registers of the ASMD chart. There are two operations in the `s1` and `s2` states and one operation in the `s3`, `s4` and `s5` states. The `start` and `ready` signals and an additional `idle` state are included to interface the circuit with an external system.

Additional optimization schemes can be applied to reduce the number of registers and to simplify the routing structure. For example, instead of creating a new register for each

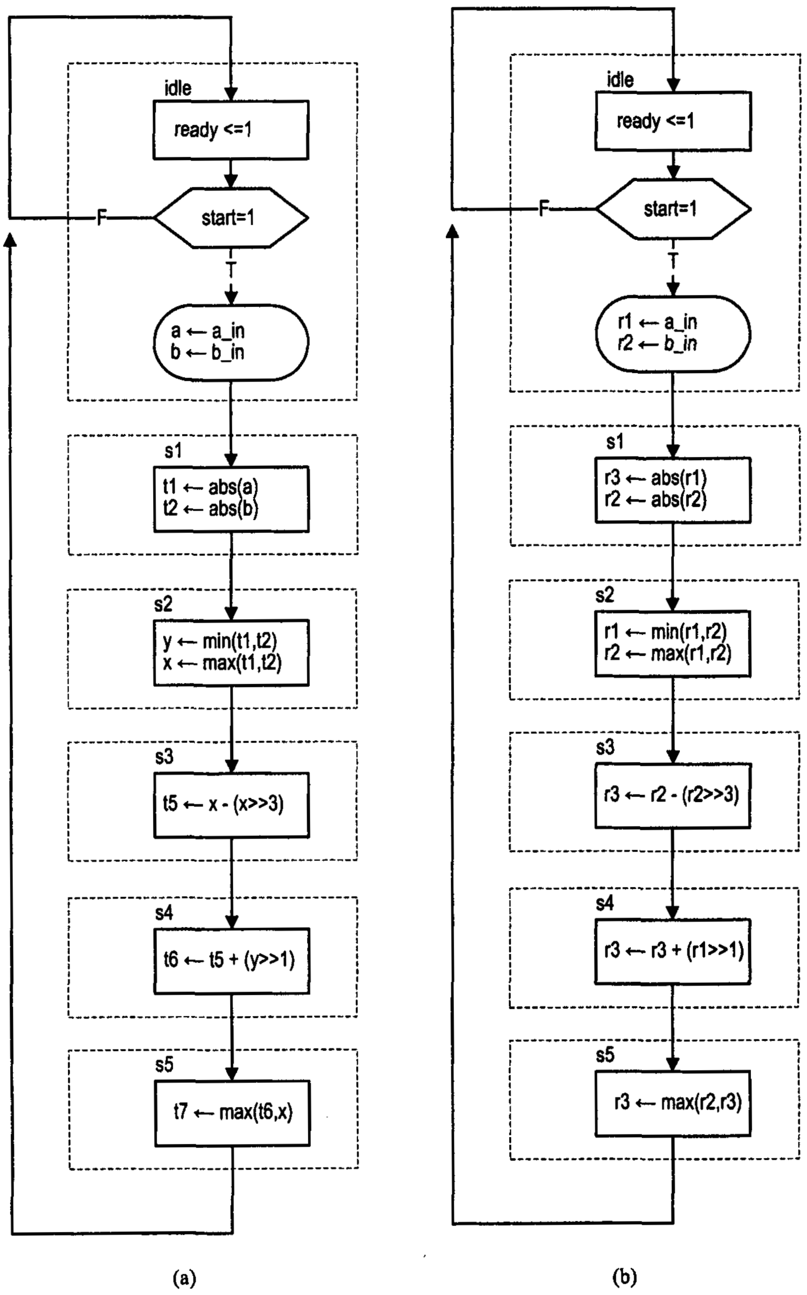


Figure 12.22 ASMD charts of a square-root approximation circuit.

variable, we can reuse an existing register if its value is no longer needed. This corresponds to properly renaming the variables in the dataflow graph. Close examination of Figure 12.20(a) shows that we can use three variables to cover the entire operation. The relationship between the new registers and the original registers is:

- Use r1 to replace a, t1 and y.
- Use r2 to replace b, t2 and x.
- Use r3 to replace t5, t6 and t7.

The replacement variables are shown in parentheses in Figure 12.20(a). The revised ASMD chart is shown in Figure 12.22(b). The number of the registers is reduced from seven to three.

The VHDL code can be derived according to the ASMD chart and is shown in Listing 12.11. To ensure proper sharing, the two functional units are isolated from the other description and coded as two separated segments. The first unit uses a single subtractor to perform the max and abs functions. The second unit uses a single adder to perform the abs and max functions as well as addition and subtraction. For clarity, we use the + operator for the carry-in signal. The synthesis software should be able to map it to the carry-in port of the adder rather than inferring another adder.

Listing 12.11 Square-root approximation circuit using RT methodology

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sqrt is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(8 downto 0)
   );
end sqrt;

architecture seq_arch of sqrt is
   constant WIDTH: integer:=8;
   type state_type is (idle, s1, s2, s3, s4, s5);
   signal state_reg, state_next: state_type;
   signal r1_reg, r2_reg, r3_reg: signed(WIDTH downto 0);
   signal r1_next, r2_next, r3_next: signed(WIDTH downto 0);
   signal sub_op0, sub_op1, diff, au1_out:
      signed(WIDTH downto 0);
   signal add_op0, add_op1, sum, au2_out:
      signed(WIDTH downto 0);
   signal add_carry: integer ;
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         r1_reg <= (others=>'0');
         r2_reg <= (others=>'0');
         r3_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         r1_reg <= r1_next;
         r2_reg <= r2_next;
         r3_reg <= r3_next;
      end if;
   end process;
   -- next-state logic and data path routing
   process(start,state_reg,r1_reg,r2_reg,r3_reg,
           a_in,b_in,au1_out,au2_out)
   begin
      r1_next <= r1_reg;
      r2_next <= r2_reg;
      r3_next <= r3_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if start='1' then
               r1_next <= signed(a_in(WIDTH-1) & a_in);
               r2_next <= signed(b_in(WIDTH-1) & b_in);
               state_next <= s1;
            else
               state_next <= idle;
            end if;
            ready <='1';
         when s1 =>
            r1_next <= au1_out; -- t1=|a|
            r2_next <= au2_out; -- t2=|b|
            state_next <= s2;
         when s2 =>
            r1_next <= au1_out; -- y=min(t1,t2)
            r2_next <= au2_out; -- x=max(t1,t2)
            state_next <= s3;
         when s3 =>
            r3_next <= au2_out; -- t5=x-0.125x
            state_next <= s4;
         when s4 =>
            r3_next <= au2_out; -- t6=0.5y+t5
            state_next <= s5;
         when s5 =>
            r3_next <= au2_out; -- t7=max(t6,x)
            state_next <= idle;
      end case;
   end process;
   -- arithmetic unit 1
   -- subtractor
   diff <= sub_op0 - sub_op1;
   -- input routing
   process(state_reg,r1_reg,r2_reg)
   begin
      case state_reg is
         when s1 => -- 0-a
```

```
            sub_op0 <= (others=>'0');
            sub_op1 <= r1_reg; -- a
         when others =>  -- s2: t2-t1
            sub_op0 <= r2_reg; -- t2
            sub_op1 <= r1_reg; -- t1
      end case;
   end process;
   -- output routing
   process(state_reg,r1_reg,r2_reg,diff)
   begin
      case state_reg is
         when s1 => --|a|
            if diff(WIDTH)='0' then  -- (0-a)>0
               au1_out <= diff; -- - a
            else
               au1_out <= r1_reg; -- a
            end if;
         when others =>  -- s2: min(a,b)
            if diff(WIDTH)='0' then --(t2-t1)>0
               au1_out <= r1_reg; -- t1
            else
               au1_out <= r2_reg; -- t2
            end if;
      end case;
   end process;
   -- arithmetic unit 2
   -- adder
   sum <= add_op0 + add_op1 + add_carry;
   -- input routing
   process(state_reg,r1_reg,r2_reg,r3_reg)
   begin
      case state_reg is
         when s1 => -- 0-b
            add_op0 <= (others=>'0'); --0
            add_op1 <= not r2_reg;  -- not b
            add_carry <= 1;
         when s2 => -- t1-t2
            add_op0 <= r1_reg; --t1
            add_op1 <= not r2_reg; --not t2
            add_carry <= 1;
         when s3 => -- -- x-0.125x
            add_op0 <= r2_reg; --x
            add_op1 <= not ("000" & r2_reg(WIDTH downto 3));
            add_carry <= 1;
         when s4 => -- 0.5*y + t5
            add_op0 <= "0" & r1_reg(WIDTH downto 1);
            add_op1 <= r3_reg;
            add_carry <= 0;
         when others => -- t6 - x
            add_op0 <= r3_reg; --t1
            add_op1 <= not r2_reg; --not x
            add_carry <= 1;
      end case;
   end process;
   -- output routing
   process(state_reg,r1_reg,r2_reg,r3_reg,sum)
   begin
      case state_reg is
         when s1 => -- |b|
            if sum(WIDTH)='0' then  -- (0-b)>0
               au2_out <= sum; -- -b
            else
               au2_out <= r2_reg; -- b
            end if;
         when s2 =>
            if sum(WIDTH)='0' then
               au2_out <= r1_reg;
            else
               au2_out <= r2_reg;
            end if;
         when s3|s4 => -- +,-
            au2_out <= sum;
         when others => -- s5
            if sum(WIDTH)='0' then
               au2_out <= r3_reg;
            else
               au2_out <= r2_reg;
            end if;
      end case;
   end process;
   -- output
   r <= std_logic_vector(r3_reg);
end seq_arch;
```

12.7 HIGH-LEVEL SYNTHESIS

The square-root approximation circuit of Section 12.6 shows that deriving the optimal RT design for data-oriented applications is by no means a simple task. The procedure is complex and involves many sophisticated algorithms. Derivation of this type of circuit belongs to a specific class of design, known as *high-level synthesis* or as somewhat misleading *behavioral synthesis*.

The synthesis starts with a set of constraints and an abstract VHDL description similar to the algorithm's pseudocode. The high-level synthesis software converts the initial description into an FSMD and automatically derives code for the control path and data path. In other words, the high-level synthesis software basically transforms from code in the form of Listing 12.10 to code in the form of Listing 12.11. The main task of the synthesis is to find an optimal schedule and binding to minimize the required hardware resources, to maximize performance or to obtain the best trade-off within a given constraint.

High-level synthesis is best for data-oriented, computation-intensive applications, such as those encountered in signal processing. It requires a separate software package, and its output is fed to regular synthesis software.

12.8 BIBLIOGRAPHIC NOTES

High-level synthesis covers primarily algorithms to perform the *binding* and *scheduling* of hardware resources, with emphasis on functional units. The treatment is normally very theoretical. The texts, *Synthesis and Optimization of Digital Circuits* by G. De Micheli, and *High-Level Synthesis: Introduction to Chip and System Design* by D. D. Gajski et al., provide good coverage on this topic. The square-root approximation circuit is adopted from the text, *Principles of Digital Design* by D. D. Gajski, which uses the circuit to demonstrate the procedures and various optimization algorithms of high-level synthesis.

Problems

12.1 In the ASMD chart of the programmable one-shot pulse generator of Section 12.2, shifting the desired values requires three states. This operation can be done by using a single state and a counter.

(a) Revise the ASMD chart to accommodate the change.

(b) Derive the VHDL code of the revised chart.

12.2 Redesign the programmable one-shot pulse generator of Section 12.2 as a pure regular sequential circuit. Derive the VHDL code.

12.3 Redesign the programmable one-shot pulse generator of Section 12.2 as a pure FSM.

(a) Derive the state diagram.

(b) Derive the VHDL code.

12.4 For the memory controller in Section 12.3, assume that the period of the system clock is 50 ns. Redesign the circuit for the 120-ns SRAM. The design should use a minimal number of states in the FSMD.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) Determine the required time to perform a read operation.

12.5 Repeat the Problem 12.4 with a system clock of 15 ns.

12.6 The memory controller of Listing 12.5 must return to the `idle` state after each operation. We can improve performance by skipping this state when back-to-back memory operations are issued.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) When a read operation follows immediately after a write operation, the direction of data flow in the bidirectional `d` line changes. Do a detailed timing analysis to examine whether a conflict can occur. We can assume that the timing parameters of the tri-state buffer in the data path are similar to those of the SRAM.

(d) Repeat part (c) for a write operation immediately following a read operation.

12.7 The FIFO buffer of Section 9.3.2 uses a register file as temporary storage. Revise the design to use an SRAM device for storage. Assume that the 120-ns SRAM is used and the system clock is 25 ns. We wish to design a FIFO controller for this system. Since it takes several clock cycles to complete a memory operation, the controller should have an additional status signal, `ready`, to indicate whether the SRAM is currently in operation.

(a) Derive the ASMD chart for the FIFO controller.