CHAPTER 11

REGISTER TRANSFER METHODOLOGY: PRINCIPLE

To accomplish a complex task, we frequently describe the process by an *algorithm*, which is a sequence of steps or actions. Algorithms are generally implemented by programs written in a traditional programming language (i.e., by software) and executed in a general-purpose computer. However, to obtain better performance and efficiency, it is sometimes beneficial or even necessary to realize an algorithm in custom hardware. The *register transfer methodology* (*RT methodology*) is a design methodology that describes system operation by a sequence of data transfers and manipulations among the registers. This methodology can support the variables and sequential execution of an algorithm and provide a systematic way to convert an algorithm into hardware.

11.1 INTRODUCTION

11.1.1 Algorithm

An algorithm is a detailed sequence of actions or steps to accomplish a task or to solve a problem. Since the semantics of traditional programming languages is also based on sequential execution, an algorithm can easily be converted into a program using the constructs of these languages. The program is then compiled into the machine instructions and executed in a general-purpose computer. Let us consider a simple task that sums the four elements of an array, divides the sum by 8 and rounds the result to the closest integer. The pseudocode of one possible algorithm is

```
size = 4
sum = 0;
for i in (0 to size-1) do {
   sum = sum + a(i);}
q = sum / 8;
r = sum rem 8;
if (r > 3) {
   q = q + 1;}
outp = q;
```

The algorithm first adds individual elements and stores the result in a variable called sum. It then uses the division (/) and remainder (**rem**) operations to find the quotient and remainder. If the remainder is greater than 3, an extra 1 is added to quotient for rounding. The example demonstrates two basic characteristics of an algorithm:

- *Use of variables.* A variable in an algorithm or pseudocode can be interpreted as a "memory location with a symbolic address" (i.e., the name of the variable). It is used to store an intermediate computation result. For example, in the second statement, 0 is stored into the memory location with a symbolic address of sum. Inside the for loop, a(i) is added with the current content of sum, and then summation is stored back into the same memory location. In the fourth statement, the content of sum is divided by 8, and the result is stored into a memory location with a symbolic address of q.
- *Sequential execution.* The execution of an algorithm is performed sequentially and the order of the steps is important. For example, the summation of the elements must be obtained before the division operation can be performed. Note that the order of execution may rely on certain conditions, as in the for loop and if statements.

In VHDL, the variables and sequential execution are treated as a special case and encapsulated inside a process. Although a description with variables can be synthesized in some cases, the variables are mapped to signals and are not interpreted or realized as "memory locations with symbolic addresses."

11.1.2 Structural data flow implementation

To achieve better performance and efficiency, we frequently want to implement an algorithm in custom hardware. The variable and sequential semantics of algorithm are very different from the concurrent model of hardware. What we have learned so far is to transform "sequential execution" into "structural data flow" by mapping an algorithm into a system of cascading hardware blocks, in which each block represents a statement in the algorithm. For example, we can unroll the loop of the previous algorithm and convert the variables into internal connection signals. Assume that sum is an 8-bit signal. The corresponding VHDL code becomes

```
sum <= 0;
sum0 <= a(0);
sum1 <= sum0 + a(1);
sum2 <= sum1 + a(2);
sum3 <= sum2 + a(3);
q <= "000" & sum3(8 downto 3);
r <= "00000" & sum3(2 downto 0);
outp <= q + 1 when (r > 3) else
        q;
```

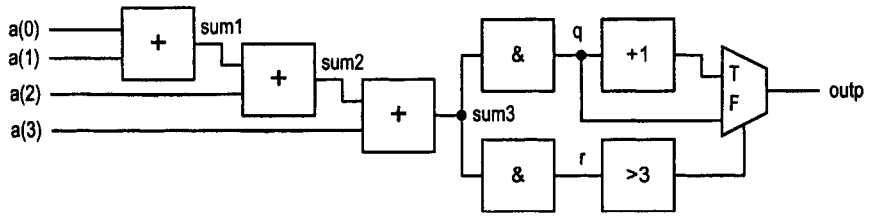


Figure 11.1 Structural data flow implementation.

Note that the `sum / 8` and `sum` **rem** `8` operations are implemented by concatenation (i.e., &) operations. The corresponding block diagram is shown in Figure 11.1.

Although the circuit can carry out the task, the operation of the hardware is very different from the sequential semantics of the original algorithm. In this construction, the circuit is a pure combinational logic, and the adders and dividers (concatenation operators) execute in parallel. The implementation does not use any concept of variable, and the sequential execution is implicitly embedded in the interconnection of components and the flow of data. To some degree, the synthesis essentially utilizes extra hardware to accelerate the operation. Instead of using a single arithmetic unit of a computer to perform these operations sequentially, the custom hardware utilizes multiple adders and division circuits to calculate the result in parallel.

The structural data flow implementation is not general and can be applied only for simple, trivial algorithms. The following two variations of the previous algorithm illustrate the limitation of this approach. First, let us consider an array with 10 elements. In the pseudocode, this can be done by replacing 4 with 10 in the first statement. This increases the number of loop iterations. We can unroll the loop and derive the structural data flow implementation, which needs nine adders. If the number of elements of the array continues to grow, the number of adders increases accordingly. Clearly, this approach needs excessive hardware resource and is not practical for a larger array. Second, let us assume that the size of the array is not fixed but is specified by an additional input, `n`. To accomplish this in the algorithm, we only need to substitute `n` into the first statement and make it `size = n`. This will be very difficult for structural data flow implementation. Since the hardware cannot expand or shrink dynamically, we have to construct a circuit that can calculate the results for *all* possible values of `n` and then use a multiplexer to route the desired value to output. The resulting hardware will be extremely complicated, and this approach is not practical in reality.

11.1.3 Register transfer methodology

The previous example shows the limitation and inflexibility of structural data flow implementation. To realize an algorithm in hardware, we need hardware constructs that resemble the variable and sequential execution model. The *register transfer methodology* (*RT methodology*) is aimed for this purpose. The key characteristics of this methodology are:

- Use registers to store the intermediate data and to imitate the variables used in an algorithm.
- Use a custom *data path* to realize all the required register operations.
- Use a custom *control path* to specify the order of the register operations.

We have utilized registers for regular sequential circuits and FSMs in previous chapters. They are usually dedicated to a specific circuit, as in a counter or an FSM. In the RT methodology, the registers are used as general storage that keeps the intermediate computed values, just as the variables of an algorithm. For example, consider a typical statement in pseudocode:

```
a = a + b
```

We can use two registers, `a_reg` and `b_reg`, to imitate the `a` and `b` variables. When this statement is executed, the content of the `a_reg` and `b_reg` registers will be added, and the result will be stored back into the `a_reg` register at the next rising edge of the clock.

When an algorithm is realized in RT methodology, the necessary data manipulation and data routing are performed by dedicated hardware. For example, an adder is required for the previous statement. The data manipulation circuit, routing network and the registers together are known as the *data path*.

Since an algorithm is described as a sequence of actions, we need a circuit to control *when* and *what* RT operations should take place. The circuit is known as the *control path*. A control path can be realized by an FSM, which can use states to enforce the order of the desired steps and use the decision boxes to imitate the branches and iterations (loops) in an algorithm.

We call this implementation methodology *register transfer methodology* since an algorithm is transformed into a sequence of actions that specifies how the data is manipulated and transferred among registers. A typical RT implementation includes a data path and a control path. We can use an extended FSM to describe the overall system operation. It is known as *FSM with a data path (FSMD)*.

As we mentioned in Section 1.4.3, use of the term *register transfer* is somewhat abused. It sometimes is used rather vaguely to represent a level of abstraction (i.e., the RT level) between the gate and processor levels. In this book, we use the term *RT methodology* for this specific design methodology and the term *RT level* for module-level abstraction.

11.2 OVERVIEW OF FSMD

An FSMD is the key to realizing the RT methodology. This section provides an overview of FSMD, including RT operation, data path, control path and extended ASM chart. The subsequent sections use examples to illustrate the detailed derivation and construction of an FSMD.

11.2.1 Basic RT operation

A basic action in RT methodology is a *register transfer operation*. We use the following notation for an RT operation:

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

In this notation, the register on the left-hand side (i.e., r_{dest}) is the destination register. The registers on the right-hand side (i.e., r_{src1}, r_{src2} and r_{srcn}) are the source registers and they represent the outputs (i.e., the contents) of these registers. The $f(\cdot)$ function is the operation to be performed. It is an expression composed of source registers and sometimes external inputs. The overall notation means that the new value of r_{dest} is calculated according to $f(r_{src1}, r_{src2}, \ldots, r_{srcn})$, and the result will be stored into r_{dest} at the next rising edge of

the clock. Note that the $\leftarrow$ notation is not defined in VHDL. It is only used in this book to denote the register transfer operation.

There is no specific restriction on the $f(\cdot)$ function. It can be any expression as long as it can be realized by a combinational circuit. A few representative RT operations are shown below.

- `r` $\leftarrow$ 1: A constant 1 is stored into the `r` register.
- `r` $\leftarrow$ `r`: The content of the `r` register is stored back into itself. The content, of course, remains unchanged.
- `r` $\leftarrow$ `r << 3`: The content of the `r` register is shifted left three positions and then stored back into itself.
- `r0` $\leftarrow$ `r1`: The content of the `r1` register is stored (or transferred) into the `r0` register.
- `n` $\leftarrow$ `n` $-$ 1: The content of the `n` register is decremented by 1 and the result is stored back into itself.
- `y` $\leftarrow$ `a` $\dot{\oplus}$ `b` $\oplus$ `c` $\oplus$ `d`: The contents of the `a`, `b`, `c` and `d` registers are xored and the result is stored into the `y` register.
- `s` $\leftarrow$ $\mathtt{a}^2 + \mathtt{b}^2$: The summation of `a` squared and `b` squared is stored into the `s` register. We can write this expression only if the predesigned combinational multiplier module is available.

The major difference between a variable of an algorithm and a register is that a *system clock* is embedded implicitly in an RT operation. Consider the register operation

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

Its detailed actions are las follows:

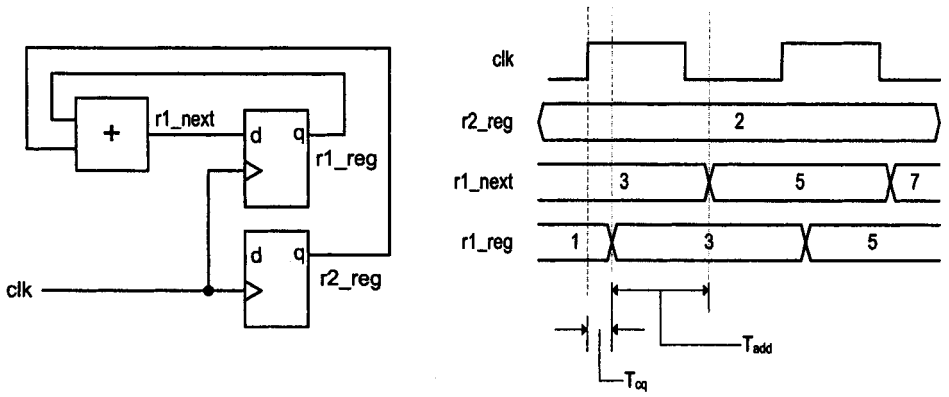
1. At the rising edge of the clock, new data from the source registers is available after the clock-to-q delay of the source registers.
2. The data is computed by a combinational circuit that realizes the $f(\cdot)$ function. We assume that the clock period is long enough to accommodate the propagation delay of the combinational circuit and the setup time of the r_{dest} register. The result is routed to the input of the r_{dest} register.
3. At the next rising edge of the clock, the result will be sampled and stored into the r_{dest} register.

In our discussion of sequential circuits, we use the suffixes `_reg` and `_next` for the current output and next input of a register. A more accurate description of an RT operation can be expressed using these suffixes. For example, consider the `r1` $\leftarrow$ `r1 + r2` operation. It actually means

- `r1_next <= r1_reg + r2_reg;`
- `r1_reg <= r1_next` at the rising edge of the clock;

Note that the `<=` notation is used for regular signal assignment.

Realizing an RT operation is straightforward. We basically construct the $f(\cdot)$ function using combinational components and then connect its output to the input of the destination register. Again, consider the `r1` $\leftarrow$ `r1 + r2` operation. It involves an addition in $f(\cdot)$. Its block diagram is shown in Figure 11.2(a) and the corresponding timing diagrams is shown in Figure 11.2(b). Note that the `r1` register will not be updated until the next rising edge of the clock.



(a) Block diagram (b) Timing diagram

Figure 11.2 Single RT operation.

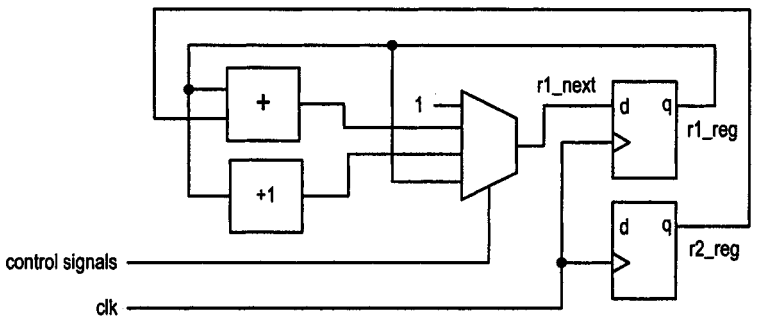


Figure 11.3 Block diagram of a set of RT operations with the same destination register.

11.2.2 Multiple RT operations and data path

An algorithm consists of many steps, and a destination register is not loaded with the same data in these steps. For example, the `r1` register may be set to 1 in the initialization step, added with the content of `r2` in a summation step, incremented in the two counting steps, and kept unchanged in the final step. Thus, four RT operations use `r1` as the destination register:

- `r1` ← 1;
- `r1` ← `r1` + `r2`;
- `r1` ← `r1` + 1;
- `r1` ← `r1`;

Because of the multiple possibilities, a multiplexing circuit is needed to route the desired value to the input of the `r1` register. The block diagram is shown in Figure 11.3. We can choose the desired RT operation by setting the proper selection signal in the multiplexing circuit.

A design with RT methodology normally involves many registers. We can repeat this procedure for every register. The resulting circuit constitutes the basic, unoptimized data path, which can perform every needed RT operation of an algorithm.

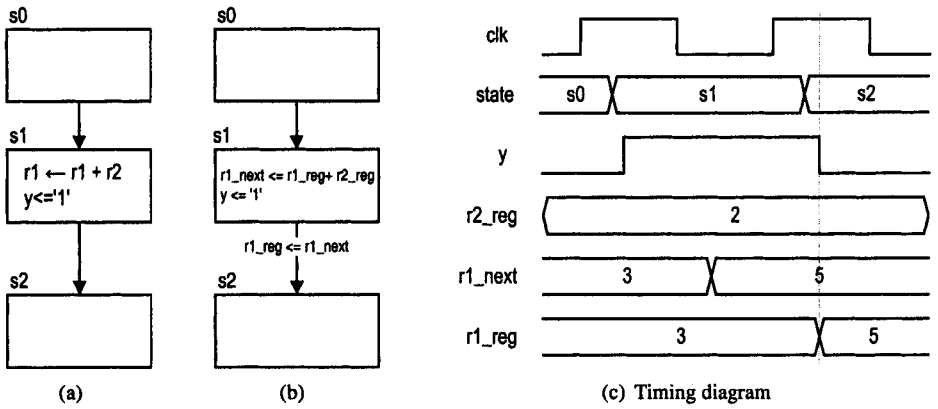


Figure 11.4 RT operation in a segment of an ASMD chart.

11.2.3 FSM as the control path

While a data path realizes all required RT operations in an algorithm, we need a mechanism to specify when and which RT operations should be performed. A control path is used to enforce the order of RT operations and to selectively perform certain RT operations based on the external commands or internal status. A control path can be realized by a custom FSM. An FSM is a natural match for this task for several reasons:

- The state transition of an FSM is performed on a clock-by-clock basis. Since an RT operation is also updated on a clock-by-clock basis, an RT operation can be specified in a state of the FSM.
- An FSM can enforce a specific sequence of actions.
- Upon an examination of input conditions, an FSM can branch to different paths and thus can alter the sequence of actions. This can be used to implement various branch constructs, such as the if and loop statements, in an algorithm.

11.2.4 ASMD chart

Since an RT operation is performed in a state of the FSM, we can extend the FSM to FSMD to indicate the desired RT operation in each state. The state representation and state transition of an FSMD are similar to those of an FSM. However, RT operations, in addition to output signals, are specified in states or transition arcs. We use an extended ASM chart, in which an RT operation can be specified either inside a state box or in a conditional output box, to describe the operation of an FSMD. It is known as an *ASM with a data path chart* (*ASMD chart*).

The construction and operation of an ASMD chart can best be explained by an example. A segment of an ASMD is shown in Figure 11.4(a). An RT operation, `r1` ← `r1 + r2`, is specified in the `s1` state. For comparison purposes, we also include a regular activated output, `y`, in the `s1` state. When the FSMD enters the `s1` state, the `r1 + r2` expression is calculated and its result becomes the next value of `r1`. At the next rising edge of the clock, the FSMD transits from `s1` to `s2` and `r1` is updated with the new value. Note that the `r1` register is not updated inside the state box but during the transition between the `s1` and `s2` states. The new value of `r1` is available only when the FSMD reaches the `s2` state.

A clumsy, but more accurate, notation is shown in Figure 11.4(b), in which the `r1_next` signal is calculated in the `s1` state, independent of the clock edge, and the `r1_reg` signal is updated at the transition. Note that the regular signal assignment notation, <=, is used in the diagram.

The timing diagram is shown in Figure 11.4(c). When the FSMD enters the `s1` state, the computation of `r1 + r2` starts but the output of `r1` remains unchanged. Note that the regular output, y, is activated after the clock-to-q delay in the `s1` state. At the next rising edge of the clock, the FSMD moves to the `s2` state and the new value is sampled and stored into `r1`. After the clock-to-q delay, the new value is propagated to the output of `r1`. Note that the y signal is deactivated in the `s2` state.

The `r1` register samples and stores the input data at every rising edge of the clock. Thus, `r1` is updated in the `s0` and `s2` states as well, even when no operation is needed. Since the system is synchronous, a register cannot be disabled or suspended. Instead, it just keeps its old value by sampling its own output; i.e., performing the `r1` ← `r1` operation. To reduce the clutter, we don't include this operation in an ASMD chart. If a destination register `r` is not associated with an RT operation in a state, we assume that it performs the default `r` ← `r` operation.

Although the appearances of an ASMD chart and a regular flowchart are somewhat similar, their operations are different. The operation of an FSMD is operated on a clock-by-clock basis. The register in an ASMD chart is not updated until the exit of the current state, and thus an RT operation exhibits some sort of *delayed-store* behavior. The most error-prone part of deriving an ASMD chart is this delayed-store operation. To obtain a correct and efficient ASMD chart, we need to have a clear understanding of the timing of an RT operation and know when a register is updated. Section 11.3.4 provides a comprehensive discussion of this issue.

11.2.5 Basic FSMD block diagram

The conceptual block diagram of an FSMD is shown in Figure 11.5. It is divided into a data path and a control path. The data path can perform all the required RT operations and is composed of three major parts:

- *Data registers.* The registers store the intermediate computation results.
- *Functional units.* The functional units perform the functions specified by RT operations. Typical functional units include an adder, subtractor, incrementor, decrementor and shifter.
- *Routing circuit.* The circuit routes the source registers' outputs to the proper functional units and routes the calculated results from the functional units to proper destination registers. It is normally constructed by customized multiplexers.

A data path normally includes the following input and output signals:

- `data input`: the external input data, which is to be processed by the FSMD.
- `data output`: the processed results of the FSMD.
- `control signal`: input signal used to specify which RT operations should be performed. It is generated by the control path.
- `internal status`: output signal indicating certain conditions of the data path, such as whether a specific register is 0. This signal is used by the control path to determine the future course of action.

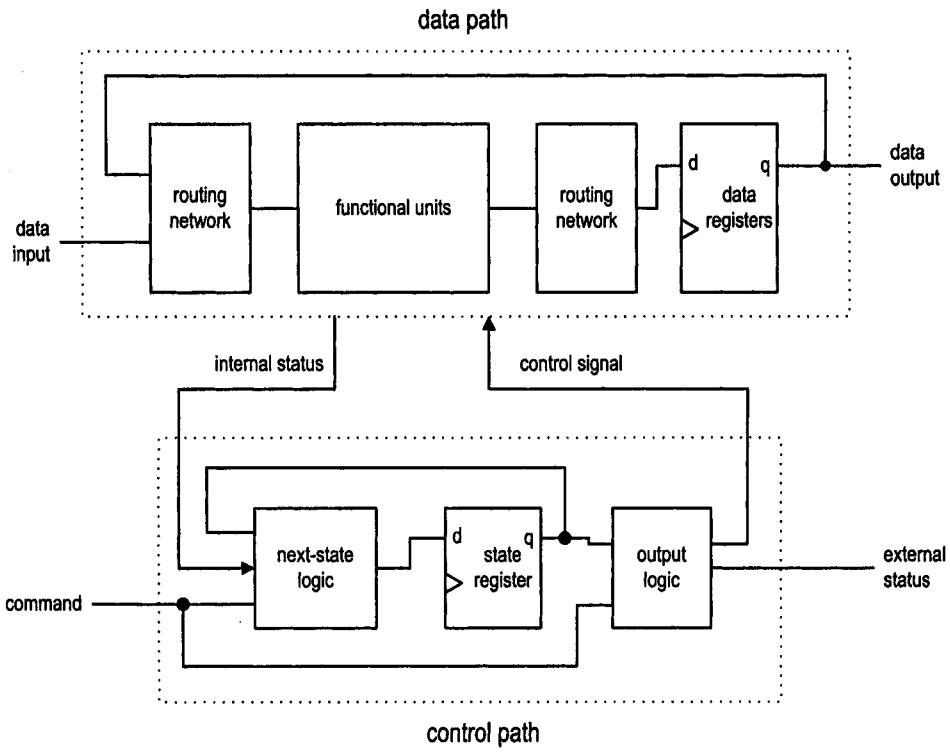


Figure 11.5 Basic block diagram of an FSMD.

The control path is an FSM. As a regular FSM, it contains a state register, next-state logic and output logic. A control path normally includes the following input and output signals:

- `command`: the external command signal to the FSMD, such as the start of the operation. It is an input to the FSM.
- `internal status`: signal from the data path, which is also an input to the FSM. The FSM uses it and the external command to determine the next state.
- `control signal`: output of the FSM used to control data path operation.
- `external status`: output of the FSM used to indicate the status of the FSMD operation, such as whether the system is busy.

In addition to these signals, the registers of the data path and control path are connected to the same clock signal and to an optional asynchronous reset signal.

Note that the data path resembles a regular sequential circuit, and the control path is an FSM and thus is a random sequential circuit. Therefore, an FSMD can be considered a combined sequential circuit, as discussed in Section 8.2.3. Although the FSMD consists of two types of sequential circuits, both circuits are synchronized by the same clock, and thus the FSMD still follows the same synchronous design methodology.

11.3 FSMD DESIGN OF A REPETITIVE-ADDITION MULTIPLIER

The derivation of an ASMD chart and the construction of an FSMD can best be explained by closely examining several examples. This section illustrates how to convert a simple repetitive-addition multiplication algorithm into an ASMD chart and realize it in hardware. Various alternatives are discussed in subsequent sections.

11.3.1 Converting an algorithm to an ASMD chart

We learned to implement a combinational multiplier in Section 7.5.4. The design utilized multiple adders and is somewhat like the data-flow implementation of Section 11.1.2. An alternative is to use one adder to perform the additions sequentially. Assume that the two operands of the multiplication are `a_in` and `b_in`. One simple sequential algorithm is to add `a_in` repetitively for `b_in` times. For example, 7*5 can be computed as $7+7+7+7+7$. While this method is not efficient, it is simple and we can concentrate on the derivation of the ASMD chart and hardware.

Consider a multiplier with input `a_in` and `b_in`, and with output `r_out`. All three signals are in unsigned integer format. The repetitive-addition algorithm can be formalized in the following pseudocode:

```
if (a_in=0 or b_in=0) then {
   r = 0;}
else{
   a = a_in;
   n = b_in;
   r = 0;
   while (n != 0 ){
      r = r + a;
      n = n - 1;}
}
r_out = r;
```

Note that the ASMD chart does not have a loop construct. Its decision box uses a Boolean condition to choose one of two possible exit paths and thus is somewhat like a combined if and goto statement. To make it closer to an ASMD chart, we convert the while loop using an if statement and two goto statements. The revised pseudocode becomes

```
        if (a_in=0 or b_in=0) then {
           r = 0;}
        else{
           a = a_in;
           n = b_in;
           r = 0;
op:        r = r + a;
           n = n - 1;
           if (n = 0) then{
              goto stop;}
           else{
              goto op;}
        }
stop:   r_out = r;
```

To realize this algorithm in hardware, we must first define its input and output signals. The input signals are:

- a_in and b_in: input operands. They are 8-bit signals with the std_logic_vector data type and interpreted as unsigned integers.
- start: command. The multiplier starts operation when the start signal is activated.
- clk: system clock.
- reset: asynchronous reset signal for system initialization.

The output signals are:

- r_out: the product. It is a 16-bit signal with the std_logic_vector data type and interpreted as an unsigned integer.
- ready: external status signal. It is asserted when the multiplication circuit is idle and ready to accept new inputs. It can also be interpreted that the previous operation has been completed and the result is ready.

Note that the start and ready signals are added to accommodate sequential operation. We can imagine that the sequential multiplier is part of a large system. When the main system wants to do a multiplication operation, it first checks the ready signal and then places the two operands on the two data inputs and asserts the start signal. When the start signal is activated, the sequential multiplier takes the two data inputs and begins computation. It activates the ready signal to inform the main system once the computation has been completed.

The ASMD chart is shown in Figure 11.6. It closely follows the pseudo algorithm. It uses n, a and r data registers to imitate the three variables, uses decision boxes to implement the two if statements, and uses RT operations to realize regular sequential statements.

Unlike the pseudocode, in which one statement is executed at a time, the ASMD chart allows some degree of parallelism. When the RT operations are scheduled in the same state, it means that they are performed in the same clock cycle and thus are done in parallel. For example, both r ← r + a and n ← n - 1 operations are scheduled in the op state. This implies that there are an adder and a decrementor in the physical circuit and that two calculations can be performed simultaneously. In general, we can schedule RT operations in the same state (i.e., the same clock cycle) as long as there is no data dependency, and enough hardware resources are available.

There are four states in the ASMD chart. The idle state indicates that the circuit is currently idle. The ready signal is asserted accordingly. If the start signal is asserted, the FSMD checks whether one of the inputs is zero and branches to the ab0 or load state. In the ab0 state, r is assigned to 0 and the FSMD returns to the idle state. Although not required, we assume that a and n are loaded with a_in and b_in. In the load state, r is initialized to 0, and a and n are loaded with the external input values. The FSMD then enters the loop and iterates through the op (for "operation") state b_in times. In each iteration, it adds the content of a to r and decrements n by 1. The n register is used to keep track of the number of operations. The loop stops when it reaches 0 and the FSMD returns to the idle state. We intentionally use a vague Boolean expression, count_0=1, inside the decision box. This is elaborated in Section 11.3.4. As in an ASM chart, we use a dashed box to represent an ASMD block and to emphasize that all operations inside the block are done in parallel at the same clock cycle.

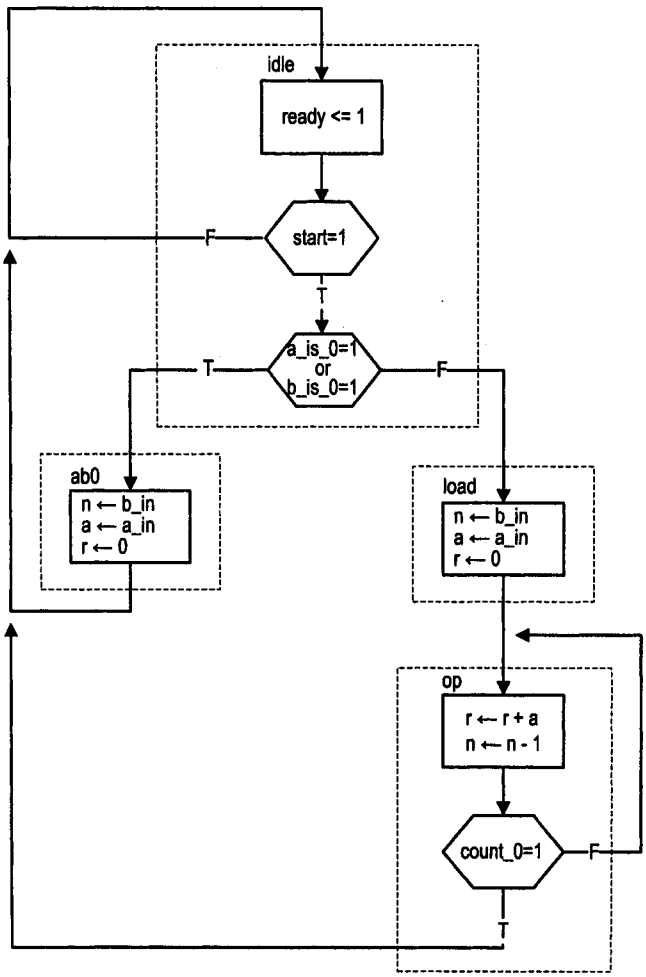


Figure 11.6 ASMD chart of a repetitive-addition multiplier.

11.3.2 Construction of the FSMD

Once the ASMD chart is constructed, more detailed information is available. We can refine the basic sketch of Figure 11.5 and derive a more detailed conceptual block diagram. We first divide the system into a control path and a data path.

The construction of the control path is the same as with the FSM. Recall that the signals inside the decision box constitute the input of the FSM. In the ASMD chart, the Boolean expressions use four signals: `start`, `a_is_0`, `b_is_0` and `count_0`. The `start` signal is the external command, and the other three are internal status signals from the data path. They are asserted when the corresponding conditions are met. The output of the control path includes the external `ready` status signal and the control signals that specify the RT operations of the data path. In this example, we use the output of the state register as the control signal. The block diagram is shown at the bottom of Figure 11.8.

At first glance, construction of the data path seems to be more involved. However, it can be derived systematically by following simple guidelines. The basic data path can be constructed as follows:

1. List all possible RT operations in the ASMD chart.
2. Group RT operations according to their destination registers.
3. For each group, derive the circuit following the process of Section 11.2.2:
   (a) Construct the destination register.
   (b) Construct the combinational circuits involved in each RT operation.
   (c) Add multiplexing and routing circuits if the destination register is associated with multiple RT operations.
4. Add the necessary circuits to generate the status signals.

The RT operations of the repetitive-addition multiplication ASMD are grouped as follows:

- RT operations with the `r` register:
  - `r` ← `r` (in the `idle` state)
  - `r` ← 0 (in the `load` and `ab0` states)
  - `r` ← `r + a` (in the `op` state)
- RT operations with the `n` register:
  - `n` ← `n` (in the `idle` state)
  - `n` ← `b_in` (in the `load` and `ab0` states)
  - `n` ← `n - 1` (in the `op` state)
- RT operations with the `a` register:
  - `a` ← `a` (in the `idle` and `op` states)
  - `a` ← `a_in` (in the `load` and `ab0` states)

Note that we must include the default RT operations for the three registers.

Let us consider the circuit associated with the `r` register. The conceptual diagram is shown in Figure 11.7. It has three possible sources for the input: 0, `r` and `r + a`. The routing of the next value is done by an abstract multiplexer, as the one discussed in Section 4.3.2. It uses the output of the state register as the select signal, and its input ports are labeled with the four possible symbolic values. The connection indicates that `r_reg` is routed to `r_next` if the `state_reg` signal is `idle`, 0 is routed to `r_next` if the `state_reg` signal is `ab0` or `load`, and `r_reg + a_reg` is routed to `r_next` if the `state_reg` signal is `op`.

We can repeat the process for two other registers and use three comparators to implement the three status signals. The complete data path, combined with the control path, is shown

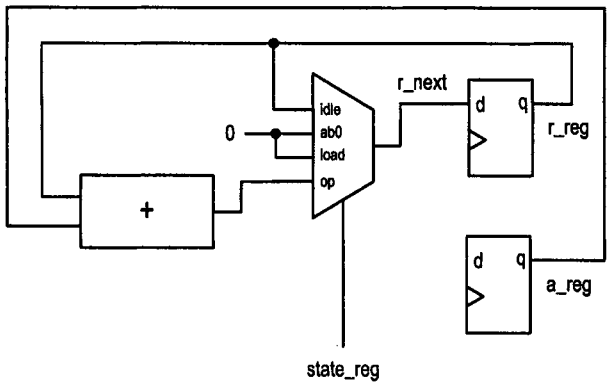


Figure 11.7 Data path associated with the r register.

in Figure 11.8. The clock and reset signals are connected to all registers. To reduce clutter, they are not shown on the diagram. The four major parts of the data path, functional units, routing circuit, data registers and status circuit, are grouped as shaded blocks.

Since this is a simple design, Figure 11.8 is somewhat unnecessarily complicated. For example, the multiplexing circuit for the a_next signal can be replaced by a register with an enable signal. The purpose of the diagram is to illustrate the derivation process. This process is very general and thus can be applied to any properly designed ASMD chart. Since the block diagram will eventually be described by VHDL code and synthesized, the multiplexing circuit will be optimized during logic synthesis.

11.3.3 Multi-segment VHDL description of an FSMD

After understanding the construction of an FSMD, we can derive the VHDL program accordingly. Our first VHDL description follows the detailed block diagram of Figure 11.8. The diagram is divided into seven blocks, which include the state register, next-state logic and output logic of the control path, and the data registers, functional units, routing network and status circuit of the data path. We use a VHDL segment for each block, and the code is shown in Listing 11.1.

Listing 11.1 Multi-segment description of a repetitive-addition multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity seq_mult 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(15 downto 0)
   );
end seq_mult;

architecture mult_seg_arch of seq_mult is
```

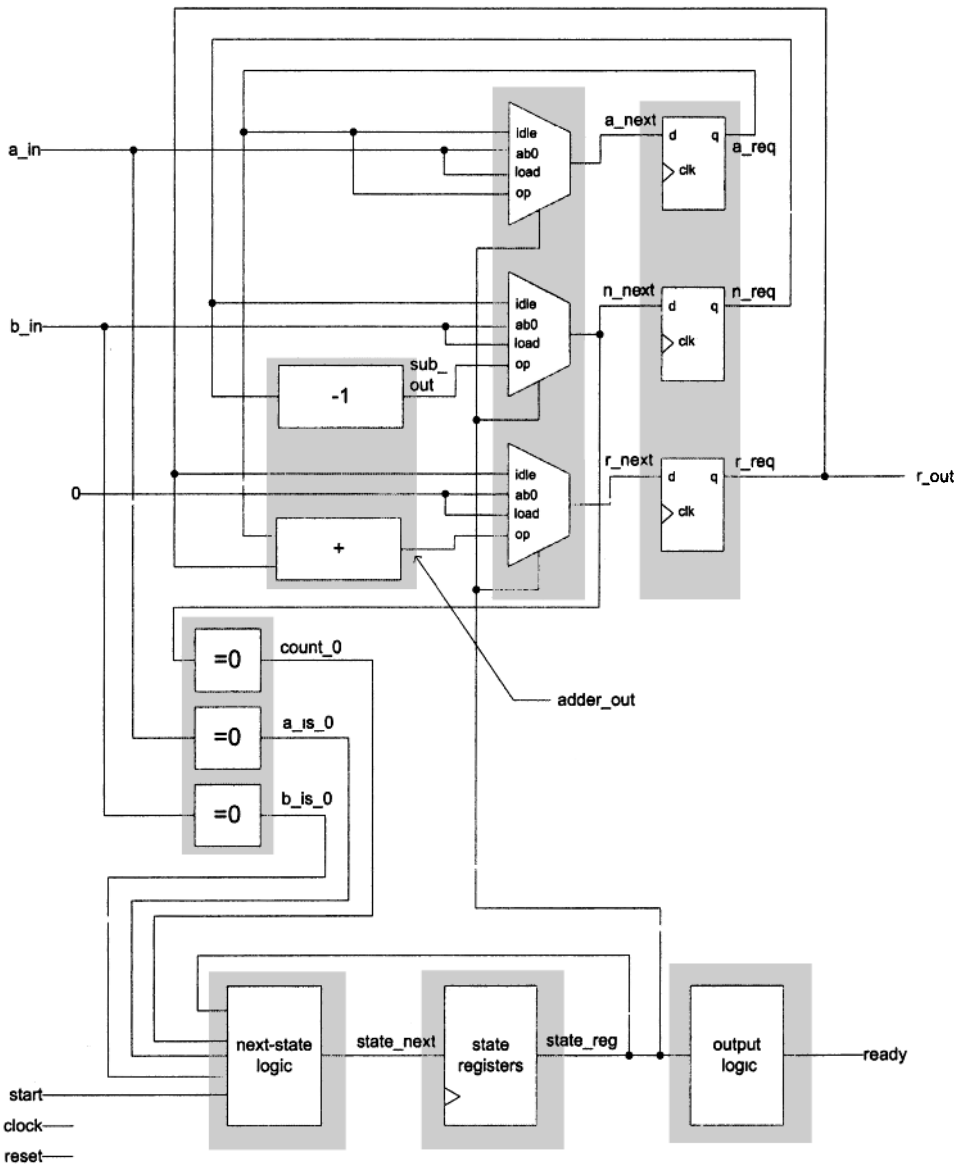


Figure 11.8 Complete block diagram of a repetitive-addition multiplier.

```
    constant WIDTH: integer:=8;
    type state_type is (idle, ab0, load, op);
    signal state_reg, state_next: state_type;
    signal a_is_0, b_is_0, count_0: std_logic;
    signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
    signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
    signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
    signal adder_out: unsigned(2*WIDTH-1 downto 0);
    signal sub_out: unsigned(WIDTH-1 downto 0);
begin
   -- control path: 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;
   -- control path: next-state/output logic
   process(state_reg,start,a_is_0,b_is_0,count_0)
   begin
      case state_reg is
         when idle =>
            if start='1' then
               if (a_is_0='1' or b_is_0='1') then
                  state_next <= ab0;
               else
                  state_next <= load;
               end if;
            else
               state_next <= idle;
            end if;
         when ab0 =>
            state_next <= idle;
         when load =>
            state_next <= op;
         when op =>
            if count_0='1' then
               state_next <= idle;
            else
               state_next <= op;
            end if;
      end case;
   end process;
   -- control path: output logic
   ready <= '1' when state_reg=idle else '0';
   -- data path: data register
   process(clk,reset)
   begin
      if reset='1' then
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
```

```
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         a_reg <= a_next;
         n_reg <= n_next;
         r_reg <= r_next;
      end if;
   end process;
   -- data path: routing multiplexer
   process(state_reg,a_reg,n_reg,r_reg,
           a_in,b_in,adder_out,sub_out)
   begin
      case state_reg is
         when idle =>
            a_next <= a_reg;
            n_next <= n_reg;
            r_next <= r_reg;
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when op =>
            a_next <= a_reg;
            n_next <= sub_out;
            r_next <= adder_out;
         end case;
   end process;
   -- data path: functional units
   adder_out <= ("00000000" & a_reg) + r_reg;
   sub_out <= n_reg - 1;
   -- data path: status
   a_is_0 <= '1' when a_in="00000000" else '0';
   b_is_0 <= '1' when b_in="00000000" else '0';
   count_0 <= '1' when n_next="00000000" else '0';
   -- data path: output
   r <= std_logic_vector(r_reg);
end mult_seg_arch;
```

11.3.4 Use of a register value in a decision box

The key to realizing RT methodology is to derive an efficient and correct ASMD description of an algorithm. Once this is accomplished, the VHDL derivation is more or less a mechanical procedure. The most subtle part of the ASMD derivation is using a register in Boolean expressions of the decision boxes. We intentionally avoided this issue in the ASMD of Figure 11.6 and used the somewhat vague a_is_0, b_is_0 and count_0 status signals inside the decision boxes. A more descriptive way is to express the Boolean conditions with registers or input signals.

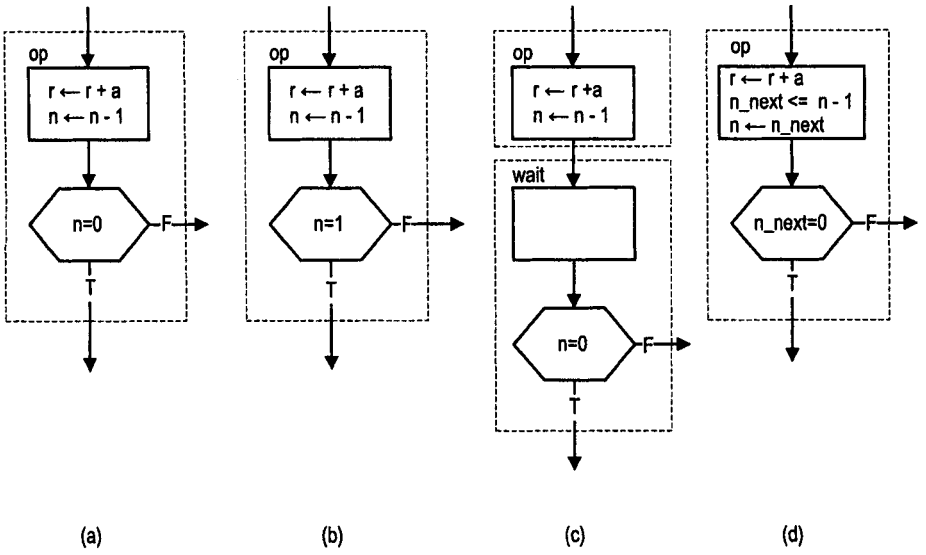


Figure 11.9 Register used in a decision box.

In the second decision box, the `a_is_0=1 or b_is_0=1` expression can easily be translated into `a_in=0 or b_in=0`. In the third decision box, the condition for the `count_0=1` expression is more subtle. The `n` register is used as a counter to keep track of the number of iterations. The iteration stops when `n` reaches 0. In pseudocode, it is expressed as

```
n = n - 1;
if (n = 0) then{
   goto stop;}
else{
   goto op;}
```

Since the execution is sequential, the `n` variable is updated in the `n = n - 1` statement, and then the new value is used in the `n = 0` expression of the if statement.

In the corresponding ASMD chart, the `n ← n - 1` operation and the decision box are in the same ASMD block. Since `n` is updated when the FSMD exits the block, the old value of `n` is used in decision box. If we write the condition as `n = 0` inside the decision box, as in Figure 11.9(a), one extra iteration is introduced and thus the result is not correct.

One way to fix the problem is to use the condition of the previous iteration, `n = 1`, to terminate the loop, as in Figure 11.9(b). This approach may not work for other algorithms when the condition of the previous iteration cannot be determined in advance. The discrepancy between the pseudocode and the ASMD chart also makes the ASMD chart less intuitive.

One clumsy way to solve the problem is to insert an artificial `wait` state so that the content of `n` can be updated before it is used in a decision box. This approach is shown in Figure 11.9(c). While this makes the ASMD looks like the original algorithm, the `wait` state introduces one extra clock cycle in the iteration and thus severely degrades the performance.

A better way is to use the *next value* of the `n` register in the Boolean expression of the decision box. Since the next value is calculated during the op state, it is available at the end

of the clock cycle and can be used in the decision box. Note that the previous VHDL code actually uses this value to generate the count_0 status signal:

```
count_0 <= '1' when n_next=0 else '0';
```

To express this idea in the ASMD chart, we have to split the RT operation r ← $f(\cdot)$ into two parts:

- r_next <= $f(\cdot)$
- r ← r_next;

The first part means that the next value of the r register is calculated and updated within the current clock cycle. We use the signal assignment notation, <=, to emphasize that the assignment is independent of the clock. The second part indicates that the r_next signal is then assigned to r at the exit of the current state, as a regular RT operation. We can use this notation to replace the count_0=0 expression of the ASMD chart, as shown in Figure 11.9(d). This approach is the preferred method since it does not use the condition of the previous iteration, maintains consistency with the original sequential algorithm and introduces no performance penalty.

11.3.5 Four- and two-segment VHDL descriptions of FSMD

The previous multi-segment description follows the detailed FSMD block diagram. For a simple design, some blocks are very straightforward, and partitioning the VHDL code into so many code segments is overkill. We can merge some blocks to make the code more compact.

For the FSMD block diagram in Figure 11.5, we can merge the combinational circuits of the data path and control path respectively, and divide the code into four segments: the data path registers, data path combinational circuit, control path register and control path combinational circuit. The detailed VHDL code is shown in Listing 11.2. Some duplicated segments are omitted. Note that we eliminate the a_is_0, b_is_0 and count_0 status signals, and use the a_in, b_in and n_next signals directly in the Boolean conditions of the control path.

Listing 11.2 Four-segment description of a repetitive-addition multiplier

```
architecture four_seg_arch of seq_mult is
    -- declarations same as mult_seg_arch , omitted
    . . .
begin
  -- control path: state register
    -- same as mult_seg_arch , omitted
    . . .
  -- control path: combinational logic
  process(start,state_reg,a_in,b_in,n_next)
  begin
    ready <='0';
    case state_reg is
      when idle =>
        if start='1' then
          if (a_in="00000000" or b_in="00000000") then
            state_next <= ab0;
          else
            state_next <= load;
```

```
                end if;
            else
                state_next <= idle;
            end if;
            ready <='1';
         when ab0 =>
            state_next <= idle;
         when load =>
            state_next <= op;
         when op =>
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op;
            end if;
      end case;
   end process;
   -- data path: data register
     -- same as mult_seg_arch, omitted
     . . .
   -- data path: combinational circuit
   process(state_reg,a_reg,n_reg,r_reg,a_in,b_in)
   begin
      -- default value
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
      case state_reg is
         when idle =>
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when op =>
            n_next <= n_reg - 1;
            r_next <= ("00000000" & a_reg) + r_reg;
      end case;
   end process;
   . . .
end four_seg_arch;
```

Since the data registers and the state register are synchronized by the same clock signal, we can merge them into a single code segment. Similarly, since the descriptions of both combinational circuits are based on the state of the FSM, we can merge them into one segment. The resulting code consists of only two segments, one for the registers and one for the combinational circuits. The VHDL code is shown in Listing 11.3.

Listing 11.3 Two-segment description of a repetitive-addition multiplier

```
architecture two_seg_arch of seq_mult is
    -- declarations same as mult_seg_arch , omitted
    . . .
begin
  -- state and data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      a_reg <= (others=>'0');
      n_reg <= (others=>'0');
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      a_reg <= a_next;
      n_reg <= n_next;
      r_reg <= r_next;
    end if;
  end process;
  -- combinational circuit
  process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
          n_next)
  begin
    -- default value
    a_next <= a_reg;
    n_next <= n_reg;
    r_next <= r_reg;
    ready <='0';
    case state_reg is
      when idle =>
        if start='1' then
          if (a_in="00000000" or b_in="00000000") then
            state_next <= ab0;
          else
            state_next <= load;
          end if;
        else
            state_next <= idle;
        end if;
        ready <='1';
      when ab0 =>
        a_next <= unsigned(a_in);
        n_next <= unsigned(b_in);
        r_next <= (others=>'0');
        state_next <= idle;
      when load =>
        a_next <= unsigned(a_in);
        n_next <= unsigned(b_in);
        r_next <= (others=>'0');
        state_next <= op;
      when op =>
        n_next <= n_reg - 1;
```

```
            r_next <= ("00000000" & a_reg) + r_reg;
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op;
            end if;
         end case;
      end process;
      r <= std_logic_vector(r_reg);
   end two_seg_arch;
```

The combinational segment basically follows the ASMD chart. It uses a case statement to list the states of the ASMD chart and specifies the actions needed in each state, which include the RT operations to be performed in the data path, the next state of the control path and the external status signal of the control path.

In the beginning of the process, we use the default signal assignment statements:

```
a_next <= a_reg;
n_next <= n_reg;
r_next <= r_reg;
ready <='0';
```

These imply that registers will keep their previous values and the output signal will be unasserted if they are not assigned in a branch of the case statement. Use of the default signal assignment statements is consistent with our notation of the ASMD chart, in which only the non-default RT operations and asserted output signals are listed inside a state box. Following the two-segment coding style, we can derive the VHDL code directly from an ASMD chart and quickly realize it in hardware.

The four- and two-segment coding styles are just some possible ways to merge the blocks of an FSMD. Since an FSMD is a sequential circuit, it is a good practice to separate the registers from the combinational circuit. Other than that, we can combine or isolate combinational blocks as needed and exercise different degrees of control over the underlying hardware configuration. In an FSMD design, the functional units of the data path are normally the most complex components and are the dominant factor in circuit size and system performance. We should pay more attention to these parts and may need to separate them from the remaining code to achieve the desired area or performance constraints. The other portion of the combinational circuit can be treated as "random logic" and will be optimized during logic synthesis.

11.3.6 One-segment coding style and its deficiency

In VHDL, it is possible to combine the registers and combinational circuit into a single segment. This style may introduce some subtle mistakes, as discussed in Section 8.7. We can use this style to code FSMD as well and it allows us to translate an ASMD chart directly into one-segment VHDL code. Although this approach seems to be quick and compact at first glance, it may again introduce many subtle problems and is not recommended. The following example illustrates some of the problems. The one-segment VHDL code of the repetitive-addition multiplier is shown in Listing 11.4.

Listing 11.4 One-segment description of a repetitive-addition multiplier

```
architecture one_seg_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, ab0, load, op);
   signal state_reg: state_type;
   signal a_reg, n_reg: unsigned(WIDTH-1 downto 0);
   signal r_reg: unsigned(2*WIDTH-1 downto 0);
begin
   process(clk,reset)
      variable n_next: unsigned(WIDTH-1 downto 0);
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         case state_reg is
            when idle =>
               if start='1' then
                  if (a_in="00000000" or b_in="00000000") then
                     state_reg <= ab0;
                  else
                     state_reg <= load;
                  end if;
               end if;
            when ab0 =>
               a_reg <= unsigned(a_in);
               n_reg <= unsigned(b_in);
               r_reg <= (others=>'0');
               state_reg <= idle;
            when load =>
               a_reg <= unsigned(a_in);
               n_reg <= unsigned(b_in);
               r_reg <= (others=>'0');
               state_reg <= op;
            when op =>
               n_next := n_reg - 1;
               n_reg <= n_next;
               r_reg <= ("00000000" & a_reg) + r_reg;
               if (n_next="00000000") then
                  state_reg <= idle;
               end if;
         end case;
      end if;
   end process;
   ready <='1' when (state_reg=idle) else '0';
   r <= std_logic_vector(r_reg);
end one_seg_arch;
```

There are several subtle problems in the code. First, since a register is inferred for any signal within the `clk'event` **and** `clk='1'` branch, the next value of a data register cannot be referred by a signal. To overcome this, we must define `n_next` as a variable for

immediate assignment. Note that the variable here is used to achieve the effect of immediate assignment and has nothing to do with the variables used in the pseudocode. Second, to avoid the unnecessary output buffer, the `ready` output signal has to be moved outside the process and be coded as a separate segment. The problems encountered in the one-segment coding style usually require more attention and offset the original hope for quick and clear coding. We avoid this coding style in this book.

11.4 ALTERNATIVE DESIGN OF A REPETITIVE-ADDITION MULTIPLIER

After studying the basic FSMD construction and VHDL coding of the repetitive-addition algorithm, we examine two variations in this section. The variations introduce the concept of sharing and Mealy-controlled RT operation.

11.4.1 Resource sharing via FSMD

We discussed combinational resource sharing in Section 7.2. It can be applied only to few restricted scenarios. Since an FSMD provides a mechanism to schedule RT operations, sharing can be achieved in a *time-multiplexed* fashion; i.e., we can assign the same functional unit in different states (i.e., different clock cycles) and use it repeatedly. For example, if an algorithm needs to perform three additions, instead of using three adders to perform the three additions at the same time, we can use one adder and schedule the additions in three states. The FSMD allows us to have another dimension of flexibility to obtain a good trade-off between the circuit size and performance.

When we convert an algorithm into an FSMD, the functional units of the data path are usually the most complex components. Since many RT operations perform the same or similar functions, some functional units can be shared as long as these operations are scheduled in different states. In the previous repetitive-addition multiplier implementation, the function units include a 16-bit adder and an 8-bit decrementor. In the original ASMD of Figure 11.6, both addition and decrementing RT operations are scheduled in the op state, and thus no sharing is possible. If we wish to reduce the circuit size, one possibility is to split the operations and schedule them into two states. This idea is shown in the revised ASMD chart in Figure 11.10, in which the original op state is split into the `op1` and `op2` states. Note that an iteration now travels through two states and thus requires two clock cycles. Since the main calculation of the algorithm is done through the iterations, it takes almost twice the number of clock cycles to complete the same task.

The block diagram of the revised data path is shown in Figure 11.11. Note that there is only one adder. Two 2-to-1 multiplexers route the desired inputs to the adder. The inputs can be either `a_reg` and `r_reg`, or `n_reg` and "11$\cdots$11", which is -1 in 2's-complement format. The former will be routed to the adder only if the current state of the control path is `op1`. Since there are already multiplexing circuits for the registers' inputs, no special routing circuit is needed for the output of the adder. Note that the output of the adder, `add_out`, is routed to the `op1` port of the `r_reg` register's input multiplexer and to the `op2` port of the `n_reg` register's input multiplexer.

As we discussed in Chapter 6, synthesis software is weak in performing RT-level optimization. If we use the two-segment coding style, the software may not be able to detect the intended sharing in the data path. To ensure the proper hardware construction, we can explicitly specify the desired functional unit sharing in VHDL code. The revised VHDL code is shown in Listing 11.5. It basically uses the two-segment coding style but isolates

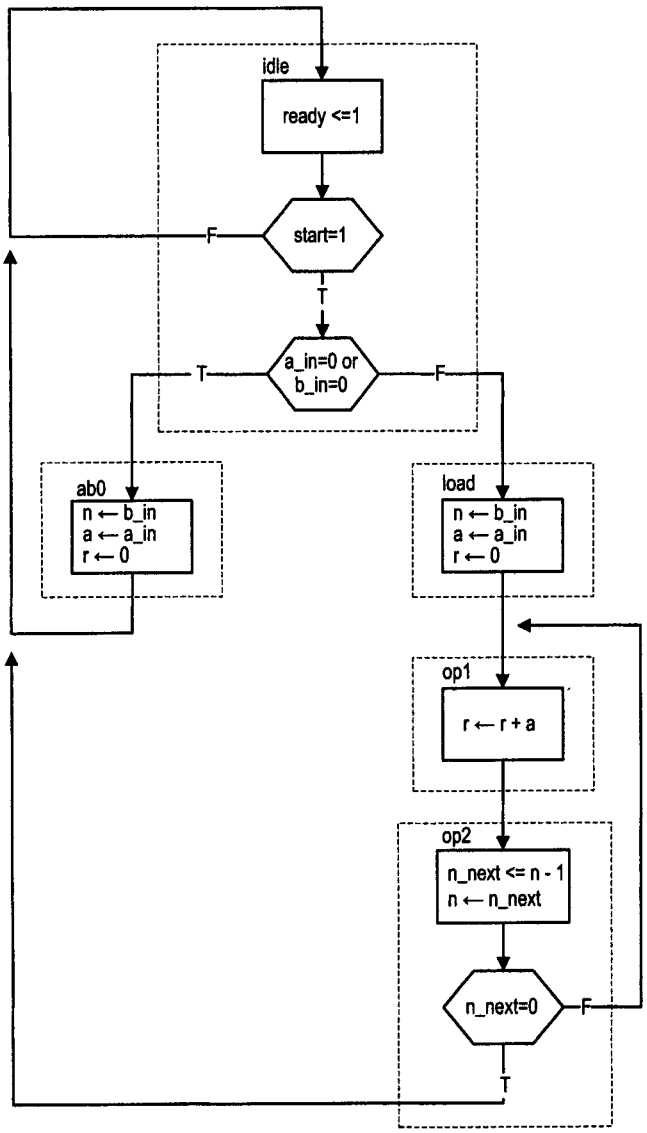


Figure 11.10 ASMD chart with sharing.

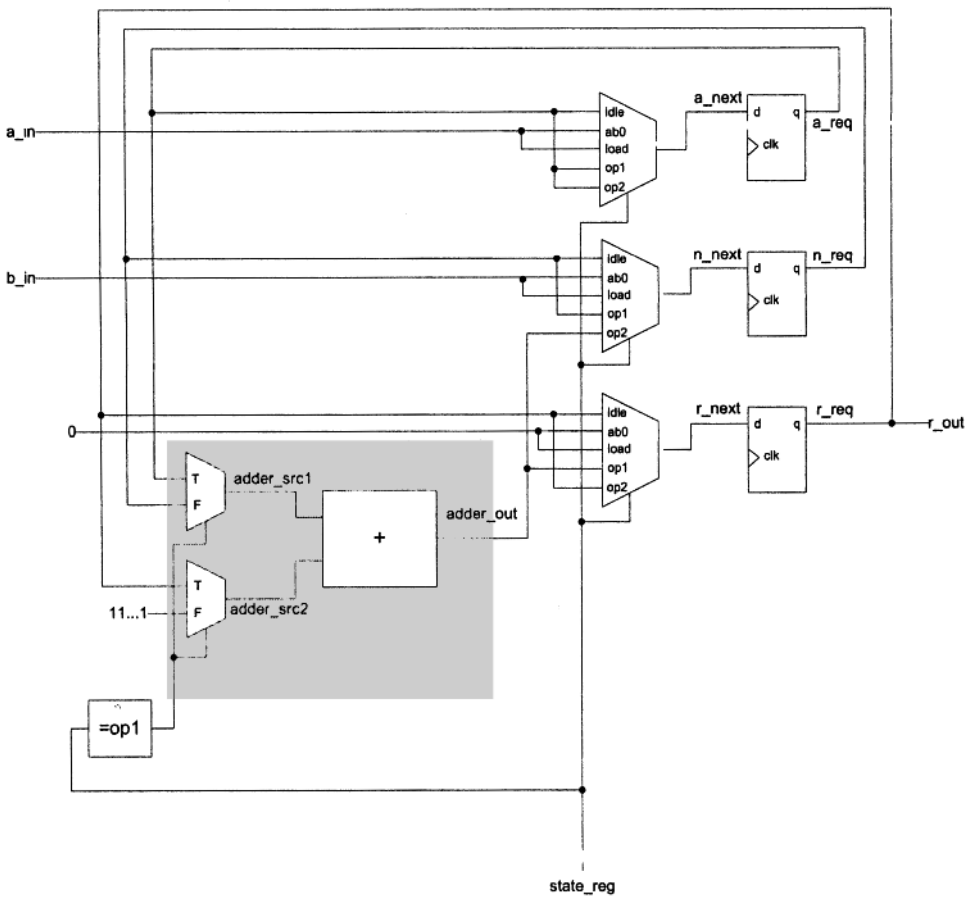


Figure 11.11 Conceptual block diagram of a sharing data path.

the functional unit from the remaining code. Note that the n register is only 8 bits wide and some adjustments are made in code to accommodate the 16-bit adder.

Listing 11.5 Sharing on a repetitive-addition multiplier

```
architecture sharing_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, ab0, load, op1, op2);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
   signal adder_src1,adder_src2: unsigned(2*WIDTH-1 downto 0);
   signal adder_out: unsigned(2*WIDTH-1 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         n_reg <= n_next;
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic/ouput logic and data path routing
   process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
           adder_out,n_next)
   begin
      -- defaut value
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if start='1' then
               if (a_in="00000000" or b_in="00000000") then
                  state_next <= ab0;
               else
                  state_next <= load;
               end if;
            else
                 state_next <= idle;
            end if;
            ready <='1';
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
```

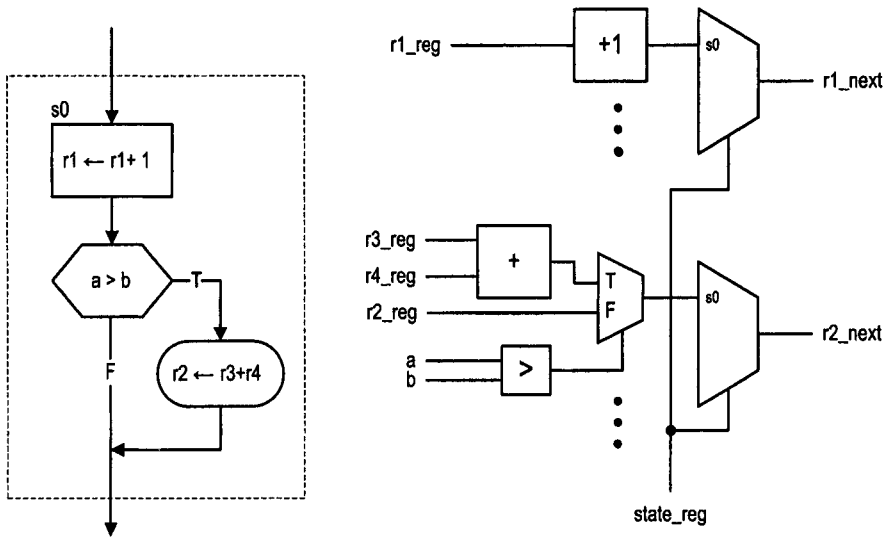
```
            r_next <= (others=>'0');
            state_next <= idle;
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
            state_next <= op1;
         when op1 =>
            r_next <= adder_out;
            state_next <= op2;
         when op2 =>
            n_next <= adder_out(WIDTH-1 downto 0);
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op1;
            end if;
      end case;
   end process;
   -- data path input routing and functional units
   process(state_reg,r_reg, a_reg, n_reg)
   begin
      if (state_reg=op1) then
         adder_src1 <= r_reg;
         adder_src2 <= "00000000" & a_reg;
      else  -- for op2 state
         adder_src1 <= "00000000" & n_reg;
         adder_src2 <= (others=>'1');
      end if;
   end process;
   adder_out <= adder_src1 + adder_src2;
   -- output
   r <= std_logic_vector(r_reg);
end sharing_arch;
```

Because the 8-bit decrementor is a relatively simple functional unit, the new design will not reduce the circuit size significantly, and the sharing is probably overkill for this particular example. Clearly, the sharing will become more predominant if complex functional units, such as combinational multipliers, are involved.

11.4.2 Mealy-controlled RT operations

In Section 10.4, we discussed the difference between a Mealy output and a Moore output. A Mealy output is preferred for an edge-sensitive control signal because it responds faster and requires fewer states in an FSM. Since the control path and data path are synchronized by the same clock signal, the control signals connected to the data path are edge-sensitive, and thus the Mealy output can be used. In terms of the FSMD, this means that we can specify RT operations in a conditional output box of an ASMD chart.

A representative ASMD block with a conditional output box is shown in Figure 11.12(a). The conditional output box indicates that the `r2` ← `r3 + r4` operation will be performed if the `a > b` condition is `true`. If the condition is `false`, `r2` remains unchanged, which



(a) ASMD block

(b) Conceptual block diagram

Figure 11.12 ASMD block with a conditional output box.

means that the `r2` ← `r2` operation will be performed. For comparison purposes, a Moore output-controlled operation, `r1` ← `r1 + 1`, is included in the state box.

If this is a regular flowchart, the condition `a > b` is first evaluated and, if the condition is met, the `r2` ← `r3 + r4` operation will be performed accordingly. However, in an ASMD chart, all operations inside an ASMD block are evaluated in parallel. When the FSMD is in the `s0` state, evaluations of `a > b`, `r3 + r4` and `r1 + 1` are performed at the same time. At the end of the clock cycle, the FSMD checks the result of `a > b` and stores the value of `r2` or the result of `r3 + r4` to `r2` accordingly.

When an RT operation is specified inside a state box, as in `r1` ← `r1 + 1`, there is only one possible next value (i.e., `r1 + 1`) in the `s0` state. On the other hand, when a conditional output box exists, there are several possible next values (i.e., `r2` or `r3 + r4`). This implies that an additional multiplexing circuit is needed. The corresponding conceptual block diagram is shown in Figure 11.12(b). An additional 2-to-1 multiplexer is added to handle the conditional output box. The result of the `a > b` operation is used as a selection signal and routes the desired next value to the `s0` port of the abstract multiplexer.

We can apply this idea to the repetitive-addition multiplier. The original ASMD chart in Figure 11.6 is actually somewhat awkward. In the `idle` state, the `start`, `a_in` and `b_in` signals are used in the decision box, and thus they have to be available at the exit of the `idle` state. If the `start` signal is asserted, `a_in` and `b_in` will be loaded into the `a` and `n` registers in the `load` or `ab0` state. Because of the delayed store, the actual sampling of the `a_in` and `b_in` signals occurs when the FSMD exits the `load` or `ab0` state, and thus the `a_in` and `b_in` signals must be available at this clock edge again. For an external system that uses the multiplication circuit, this means that it has to place the two operands on `a_in` and `b_in` ports for two consecutive clocks. To release the external system from this artificial timing constraint, a better design should be able to sample the `start`, `a_in` and `b_in` signals at the same time and at only one clock edge.

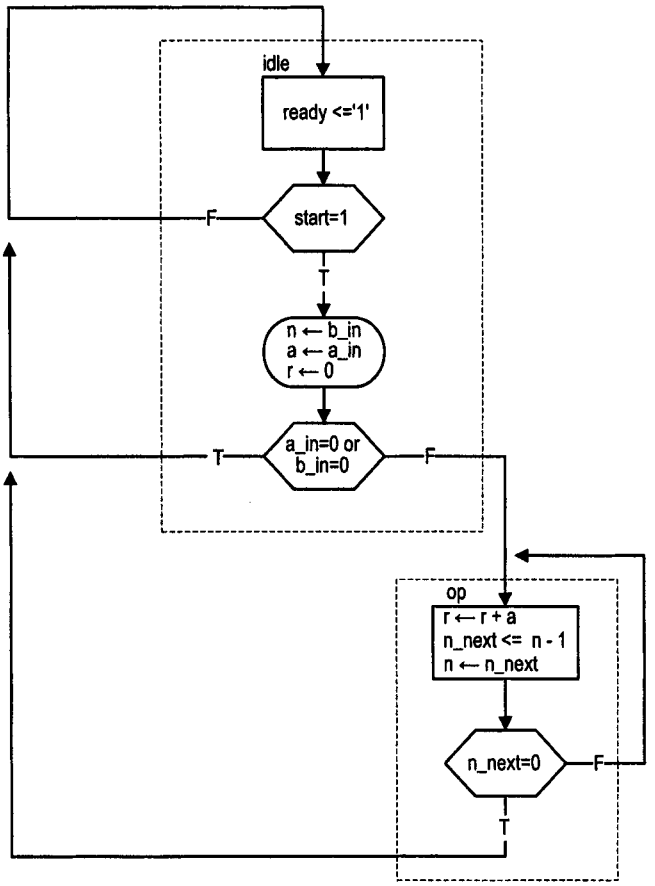


Figure 11.13 ASMD chart with Mealy-controlled RT operations.

This design can be achieved by using Mealy-controlled RT operations. The revised ASMD chart is shown in Figure 11.13. It merges the `ab0` and `load` states into the `idle` state and moves the corresponding RT operations into a conditional output box. In addition to relaxing the timing constraint on the external system, the revised design reduces the number of states from four to two and improves the overall performance. The VHDL code is shown in Listing 11.6. It uses the two-segment coding style. Note that some next-value statements, such as `a_next <= unsigned(a_in)`, are within the then branch of the if statement, which corresponds to the conditional output box of the ASMD chart.

Listing 11.6 Mealy-controlled RT operations for a repetitive-addition multiplier

```
architecture mealy_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, op);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
begin
```

```
    -- state and data registers
    process(clk,reset)
    begin
       if reset='1' then
          state_reg <= idle;
          a_reg <= (others=>'0');
          n_reg <= (others=>'0');
          r_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
          a_reg <= a_next;
          n_reg <= n_next;
          r_reg <= r_next;
       end if;
    end process;
    -- combinational circuit
    process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
            n_next)
    begin
       a_next <= a_reg;
       n_next <= n_reg;
       r_next <= r_reg;
       ready <='0';
       case state_reg is
          when idle =>
             if start='1' then
                a_next <= unsigned(a_in);
                n_next <= unsigned(b_in);
                r_next <= (others=>'0');
                if a_in="00000000" or b_in="00000000" then
                   state_next <= idle;
                else
                   state_next <= op;
                end if;
             else
                state_next <= idle;
             end if;
             ready <='1';
          when op =>
             n_next <= n_reg - 1;
             r_next <= ("00000000" & a_reg) + r_reg;
             if (n_next="00000000") then
                state_next <= idle;
             else
                state_next <= op;
             end if;
       end case;
    end process;
    r <= std_logic_vector(r_reg);
end mealy_arch;
```

11.5 TIMING AND PERFORMANCE ANALYSIS OF FSMD

An FSMD is a synchronous circuit and thus is subject to similar setup and hold time constraints. The setup time constraint, in turn, imposes the maximal clock rate. Unlike a regular sequential circuit, an algorithm described by an FSMD requires a sequence of RT operations to complete. Thus, in addition to the clock rate, the total computation time of an FSMD depends on the number of clock cycles needed to complete the computation as well. The following subsections discuss these issues.

11.5.1 Maximal clock rate

We analyzed timing of a regular sequential circuit and an FSM in Chapters 8 and 10. Both analyses are based on the basic block diagram shown in Figure 8.5. The basic diagram of an FSMD, shown in Figure 11.5, is somewhat different. It has two separate but interactive feedback loops, one for the control path and one for the data path. In theory, we can merge the two feedback loops, convert the FSMD block diagram into the standard diagram, and then analyze it as an ordinary sequential circuit. Because of the interaction between the two loops, it will be difficult to manually analyze the merged combinational circuit. We must rely on a software tool to do the timing analysis and determine the maximal clock rate.

Although the manual analysis cannot determine the exact maximal clock rate, it is possible to determine the boundaries of the rate. This analysis provides more insights into the FSMD operation and helps us to derive a more efficient design. The basic FSDM block diagram of Figure 11.5 has two feedback loops. The data path loop is based on the data register, and the control path loop is based on the state register. The two loops are not independent but interact via the control signals and status signals. For example, a function unit in the data path cannot operate until the control signals set the selection signal of the input multiplexer, and the next-state logic in the control path cannot proceed until the status signals are available. The exact maximal clock rate depends on where the control signals are needed and where the status signals are generated. This depends on the individual implementation and cannot be generalized. Our analysis considers the best- and worst-case scenarios and thus determines the boundaries of the maximal clock rate.

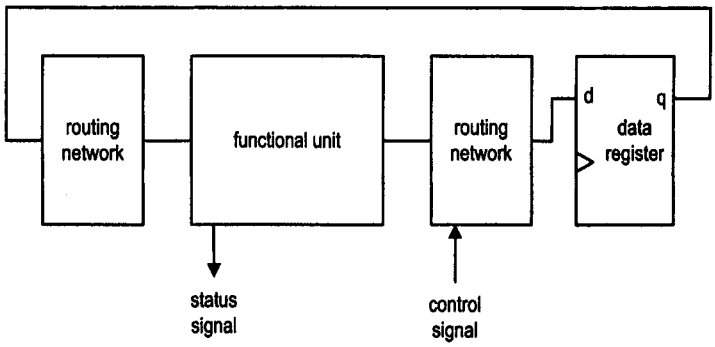
The control path is the bottom part of Figure 11.5. Its timing parameters are the same as those of an FSM. They are defined as follows:

- $T_{cq(state)}$: clock-to-q delay of the state register.
- $T_{setup(state)}$: setup time of the state register.
- T_{next}: maximal propagation delay of the next-state logic of the control path FSM.
- T_{output}: maximal propagation delay of the output-state logic of the control path FSM.

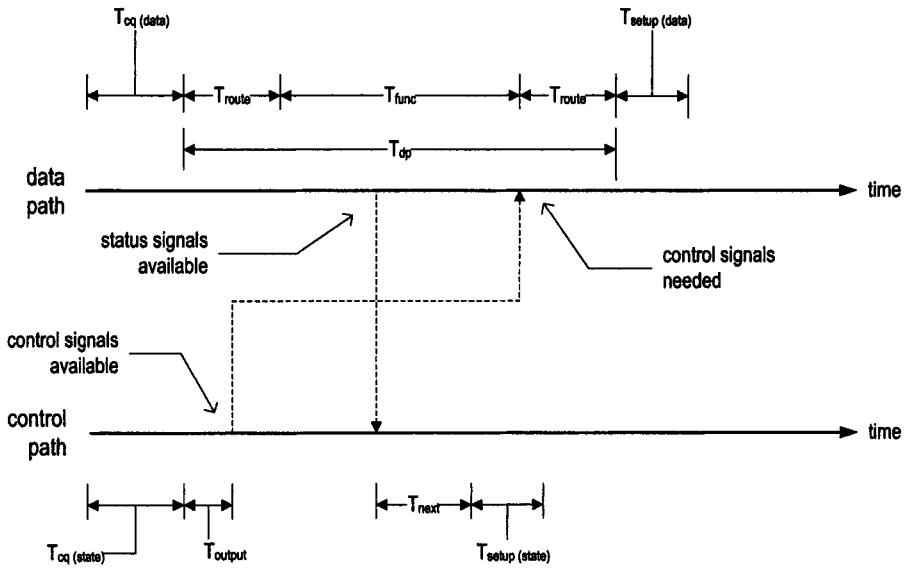
The conceptual diagram of the data path is shown at the top of Figure 11.5. The relevant timing parameters are:

- $T_{cq(data)}$: clock-to-q delay of the data register.
- $T_{setup(data)}$: setup time of the data register.
- T_{func}: maximal propagation delay of the functional units.
- T_{route}: maximal propagation delay of the routing multiplexing circuit.
- T_{dp}: maximal propagation delay of the combinational circuit of the data path, which is the sum of T_{func} and $2T_{route}$.

As before, we use T_c for the the clock period. In a normal design, T_{func} is likely to be the largest and the most dominant of all timing parameters. We use this assumption in the analysis.



(a) Conceptual data path diagram

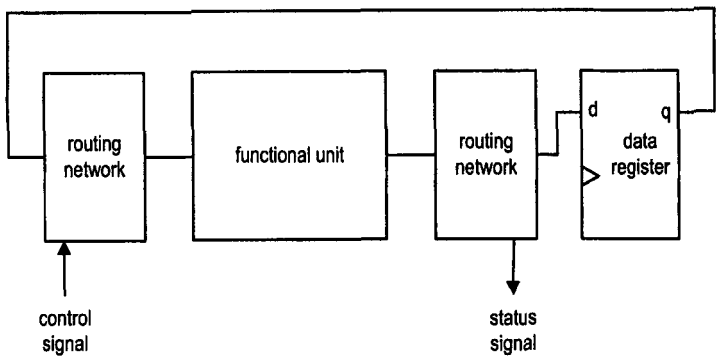


(b) Time lines

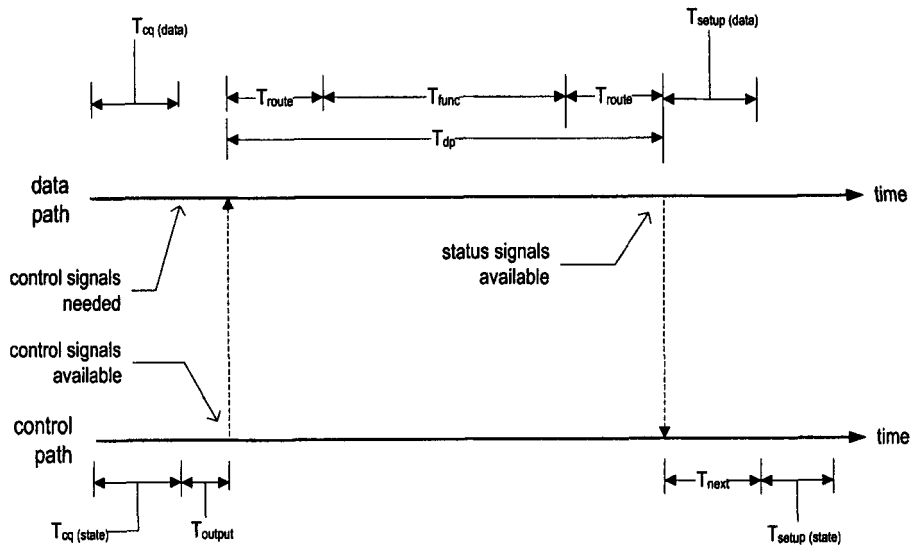
Figure 11.14 Time lines for the best-case scenario.

We first consider the best-case scenario. In this scenario, the control signals are required at a late stage of data path operation, and the status signals are generated in an early stage of data path operation, as shown in the conceptual data path diagram in Figure 11.14(a). The time lines of the data- and control-path operations are shown in Figure 11.14(b). They start with the rising edge of the clock signal. Since the data path uses control signals in the late stage, the operation of the output logic overlaps with the operation of data path and thus contributes no extra delay for the data path loop. Similarly, since the status signal is available at an early stage, the operation of the next-state logic of the control path and the computation of data path are done in parallel. When the data path computation is complete, the next-state value is also ready in the control path. The time lines show that the minimal clock period of the FSMD is the same as the clock period of the data path, which is

$$T_c = T_{cq(data)} + T_{dp} + T_{setup(data)}$$



(a) Conceptual data path diagram



(b) Time lines

Figure 11.15 Time lines for the worst-case scenario.

The worst-case scenario reverses the conditions of the best-case scenario. In this scenario, control signals are required at the beginning of data path operation, and the status signals are generated at the end of data path operation. The conceptual diagram of the data path and the time lines are shown in Figure 11.15. The data path must wait for the FSM to generate the output signals, and the control path must wait for status signals to generate the next-state value. Except for the register, there is no overlapped operation between the control path and the data path. The minimal clock period can be found by following the time lines, and it includes the propagation delays of all combinational components:

$$T_c = T_{cq(state)} + T_{output} + T_{dp} + T_{next} + T_{setup(state)}$$

Assume that the state register and data register have similar timing characteristics, and clock-to-q delay and setup time are T_{cq} and T_{setup} respectively. From the two extreme

scenarios, we can establish the boundaries of the minimal clock period:

$$T_{cq} + T_{dp} + T_{setup} \leq T_c \leq T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}$$

Consequently, the maximal clock rate is bound by

$$\frac{1}{T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}} \leq f \leq \frac{1}{T_{cq} + T_{dp} + T_{setup}}$$

For a design with a wide, complex data path, T_{dp} will be much larger than T_{next} and T_{output}, and thus variation in the minimal clock period is relatively small. For a circuit with a complex control path, we may need to minimize T_{next} and T_{output} to obtain better performance. For this kind of design, we can isolate the control path FSM in VHDL code, as in the multi- or four-segment coding styles, and apply special FSM optimization software to obtain a more efficient FSM implementation.

11.5.2 Performance analysis

In an FSMD, computation is performed in a sequence of steps, and it usually takes many clock cycles to complete a task. Thus, the total required time becomes

$$T_{total} = K * T_c$$

where K is the number of clock cycles and T_c is the clock period. K is determined by the algorithm, the width of the input and the value of the input. The determination of K is an ad hoc process and can sometimes be very difficult. For certain algorithms, K and T_c may work against each other. For example, we can merge more computation steps into a single state. This will reduce the number of states (and thus the clock cycles) but increase the clock period due to the larger data path propagation delay (T_{dp}). On the other hand, we can sometimes divide an operation into several smaller steps and schedule them in multiple clock cycles. This will decrease T_{dp} and the clock period, but requires more clock cycles to complete the same computation.

Consider the original ASMD design in Figure 11.6. The width of input operands is 8 bits. The K of this algorithm is not a constant but depends on the value of the `b_in` input. In the best case, `b_in` is 0, and the FSMD goes through the `idle` and `ab0` states. The computation takes two clock cycles (i.e., $K=2$). In the worst case, `b_in` is 255 (i.e., 2^8-1) and `a_in` is not 0, the FSMD goes through the `idle` and `load` states once and loops the `op` state 255 times. K becomes 257. We can generalize this for n-bit input operands. In the worst case, the FSMD goes through the `idle` and `load` states once and loops the `op` state 2^n-1 times. Thus, it takes 2^n+1 clock cycles to complete the computation. We can apply the same analysis for the sharing ASMD design in Figure 11.10, in which the `op` state is split into two states. It requires two clock cycles for each loop iteration. For n-bit input operands, the worst-case K becomes $2+2(2^n-1)$, which is 2^{n+1}. Whereas the data path size is smaller in this design, the required computation time is nearly doubled.

11.6 SEQUENTIAL ADD-AND-SHIFT MULTIPLIER

Although simple, the previous repetitive-addition algorithm is not practical since the required computation time is on the order of $O(2^n)$. This section introduces a more efficient sequential multiplication algorithm. The algorithm is based on the add-and-shift method

					a_3	a_2	a_1	a_0	multiplicand
$\times$					b_3	b_2	b_1	b_0	multiplier
					a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			a_3b_2	a_2b_2	a_1b_2	a_0b_2			
$+$		a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	product

Figure 11.16 Multiplication as a summation of a_ib_j terms.

discussed in Section 7.5.4. The multiplication of two 4-bit numbers is illustrated in Figure 11.16. It includes three tasks:

1. Multiply the digits of the multiplier (b_3, b_2, b_1 and b_0) by the multiplicand (A) one at a time to obtain $b_3{*}A$, $b_2{*}A$, $b_1{*}A$ and $b_0{*}A$. The $b_i{*}A$ operation is bitwise and operation of b_i and the digits of A; that is,

$$b_i{*}A = (a_3{\cdot}b_i,\ a_2{\cdot}b_i,\ a_1{\cdot}b_i,\ a_0{\cdot}b_i)$$

2. Shift $b_i{*}A$ to left i positions.
3. Add the shifted $b_i{*}A$ terms to obtain the final product.

11.6.1 Initial design

The add-and-shift method can easily be converted into a sequential algorithm. We can process one digit of the multiplier (i.e., b_i) at a time and iterate through all digits of the multiplier (B). In each iteration, we calculate $b_i{*}A$, shift it to the left i positions, and then add it to the partial product. Since b_i is a binary digit, it can be either 0 or 1. Instead of computing $b_i{*}A$, we use an if statement to check the value of b_i and add the shifted A to the partial product when b_i is 1. Assume that the inputs are `a_in` and `b_in`. The pseudocode is

```
n = 0;
p = 0;
while (n!=8) {
   if (b_in(n)=1) then{
      p = p + (a_in << n);}
   n = n + 1;
}
r_out = p;
```

In hardware, it is expensive to do indexing (i.e., `b_in(n)`) and general shifting (i.e., `a_in << n`). To overcome the problem, we can "intelligently" shift `a_in` and `b_in` one position in each iteration. The pseudocode of this algorithm is

```
a = a_in;
b = b_in;
n = 8;
p = 0;
while (n!=0) {
```

```
        if (b(0)=1) then{
           p = p + a;}
        a = a << 1;
        b = b >> 1;
        n = n - 1;
     }
     r_out = p;
```

Four variables are used in the algorithm. The p variable is used to store the partial product, and the n variable is used to keep track of the number of iteration. Note that the counting direction of the n is reversed from the previous pseudocode to accommodate future hardware implementation. The a variable is used to store the shifted multiplicand (A), which is shifted left one position in each iteration. The b variable is used for the multiplier (B). It is shifted right one position in each iteration, and thus b_i of B becomes LSB (i.e., b(0)) of b in the ith iteration.

To facilitate development of an ASMD chart, we can convert the while loop into if and goto statements:

```
     a = a_in;
     b = b_in;
     n = 8;
     p = 0;
op:  if b(0)=1 then {
        p = p + a;}
     a = a << 1;
     b = b >> 1;
     n = n - 1
     if (n!=0) then{
        goto op;}
     r_out = p;
```

This pseudocode can easily be converted to an ASMD chart, as shown in Figure 11.17. The FSMD has three states. In the `idle` state, the FSMD checks the `start` signal. If it is asserted, the FSMD loads the initial values to registers and moves to either the add or `shift` state. If the corresponding bit of the multiplier is '1', the FSMD moves to the add state, in which the shifted multiplicand (A) is added to the partial product. Otherwise, the FSMD moves to the `shift` state, in which the multiplicand (A) is shifted left one position, the multiplier (B) is shifted right one position, and the counter is decremented by 1. The add-and-shift process continues to iterate until the counter reaches 0.

While the chart basically follows the pseudocode, there are two differences. First, since the two shift operations and the counter decrementing operation are independent, they are scheduled in the same state and performed in parallel. Second, due to the delayed store of RT operations, we use the next values of the registers in decision boxes. Note that `b_next(0)` and `n_next` are used in the decision boxes of the `shift` state, and `b_in(0)` is used in the decision box of the `idle` state.

After developing a correct, comprehensive ASMD chart, we can derive the VHDL description accordingly. The VHDL code is shown in Listing 11.7. Note that the two shifting operations are done by the concatenation operations (&).

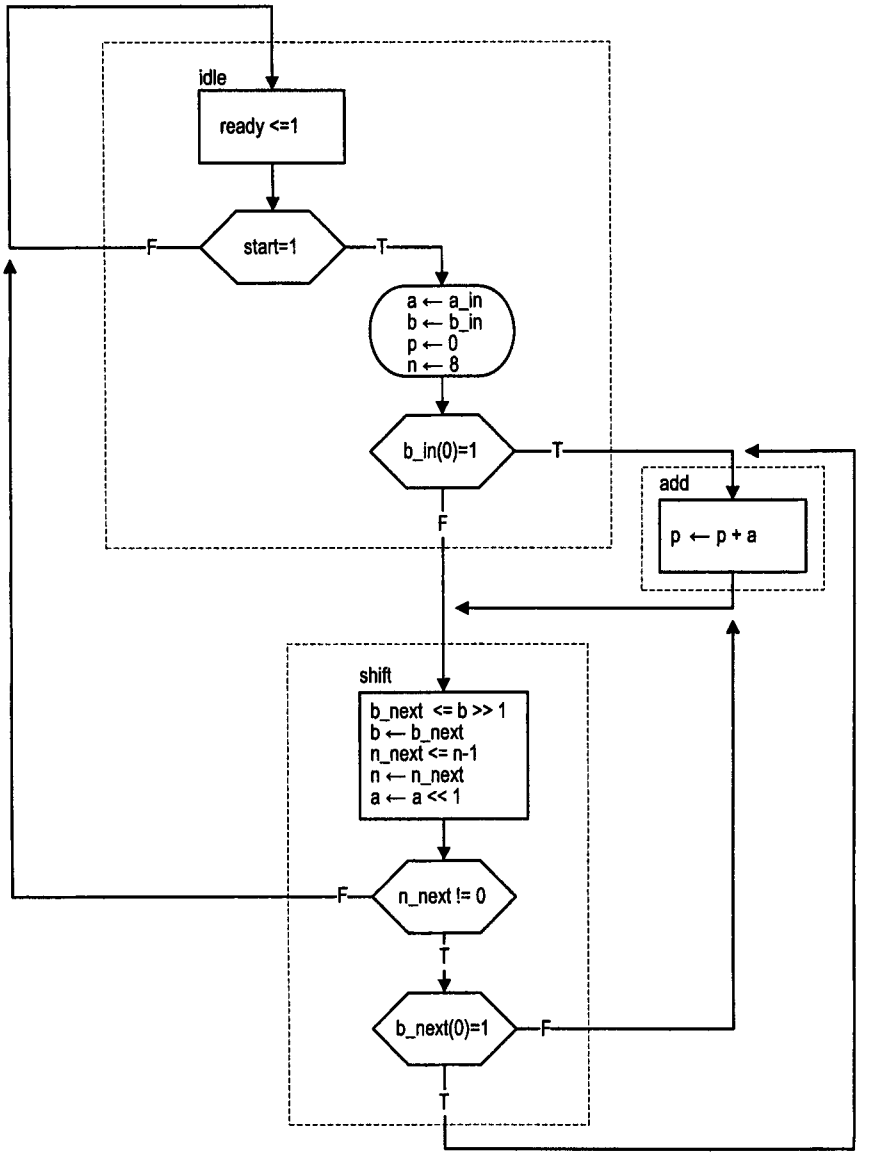


Figure 11.17 ASMD chart of the initial add-and-shift multiplier.

Listing 11.7 Initial description of an add-and-shift sequential multiplier

```
architecture shift_add_raw_arch of seq_mult is
   constant WIDTH: integer:=8;
   constant C_WIDTH: integer:=4; -- width of the counter
   constant C_INIT: unsigned(C_WIDTH-1 downto 0):="1000";
   type state_type is (idle, add, shift);
   signal state_reg, state_next: state_type;
   signal b_reg, b_next: unsigned(WIDTH-1 downto 0);
   signal a_reg, a_next: unsigned(2*WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
   signal p_reg, p_next: unsigned(2*WIDTH-1 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         b_reg <= (others=>'0');
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
         p_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         b_reg <= b_next;
         a_reg <= a_next;
         n_reg <= n_next;
         p_reg <= p_next;
      end if;
   end process;
   -- combinational circuit
   process(start,state_reg,b_reg,a_reg,n_reg,p_reg,
           b_in,a_in,n_next,a_next)
   begin
      b_next <= b_reg;
      a_next <= a_reg;
      n_next <= n_reg;
      p_next <= p_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if start='1' then
               b_next <= unsigned(b_in);
               a_next <= "00000000" & unsigned(a_in);
               n_next <= C_INIT;
               p_next <= (others=>'0');
               if b_in(0)='1' then
                  state_next <= add;
               else
                  state_next <= shift;
               end if;
            else
               state_next <= idle;
            end if;
```

```
            ready <='1';
         when add =>
            p_next <= p_reg + a_reg;
            state_next <= shift;
         when shift =>
            n_next <= n_reg - 1;
            b_next <= '0' & b_reg (WIDTH-1 downto 1);
            a_next <= a_reg(2*WIDTH-2 downto 0) & '0';
            if (n_next /= "0000") then
               if a_next(0)='1' then
                  state_next <= add;
               else
                  state_next <= shift;
               end if;
            else
               state_next <= idle;
            end if;
      end case;
   end process;
   r <= std_logic_vector(p_reg);
end shift_add_raw_arch;
```

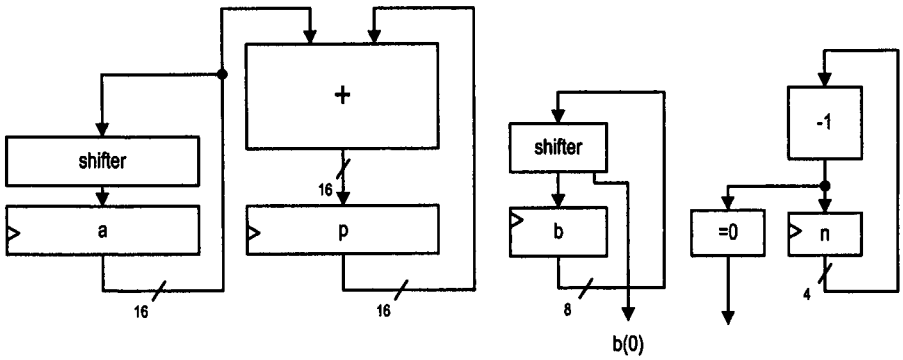
Recall that the functional units used in the data path are normally the most critical components in an FSMD, and understating their basic organization can help us to develop a more efficient design. The sketch of the data path is shown in Figure 11.18(a). To reduce the clutter, only functional units and major data flow are shown. Note that since the amount is fixed in two shift operations, the shifters require no real logic.

In the new algorithm, the number of iterations in the loop is equal to the width of the input operand. An iteration goes through the add and `shift` states if the corresponding multiplier bit is '1' and goes through only the `shift` state otherwise. For n-bit input operands, the computation requires $2n+1$ clock cycles in the worst case (i.e., `b_in` is "$1\cdots1$") and $n+1$ clock cycles in the best case (i.e., `b_in` is "$0\cdots0$"). It is far superior to the 2^n+1 clock cycles of the repetitive-addition algorithm.

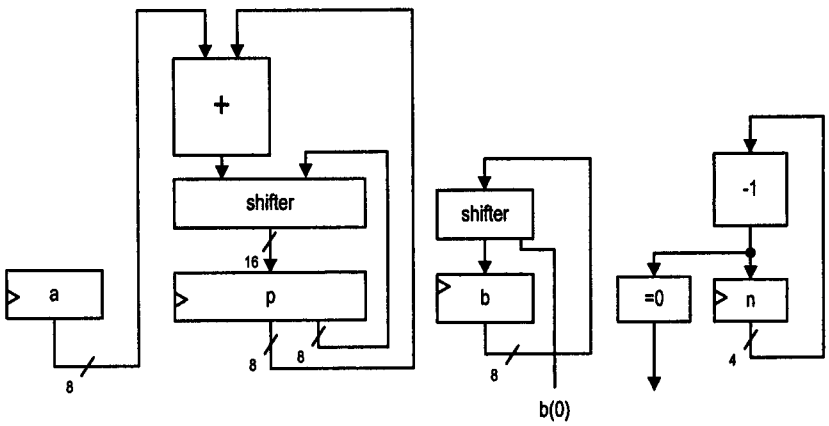
11.6.2 Refined design

Our initial implementation of the add-and-shift multiplier closely follows the sequential pseudocode, which is presumed to be executed in a general-purpose processor. However, hardware implementation provides more flexibility and gives us an opportunity to further streamline the design. There are several possible improvements. We can first improve the efficiency of the ASMD chart. The main computation is done by iterating the add-and-shift loop, and each iteration may go through up to two states. If we examine the add and `shift` states closely, the RT operations in these states are independent. It is possible to merge the two states and utilize a conditional output box for the p $\leftarrow$ p + a operation in the new state. The revised ASMD chart will require only one clock cycle for each iteration.

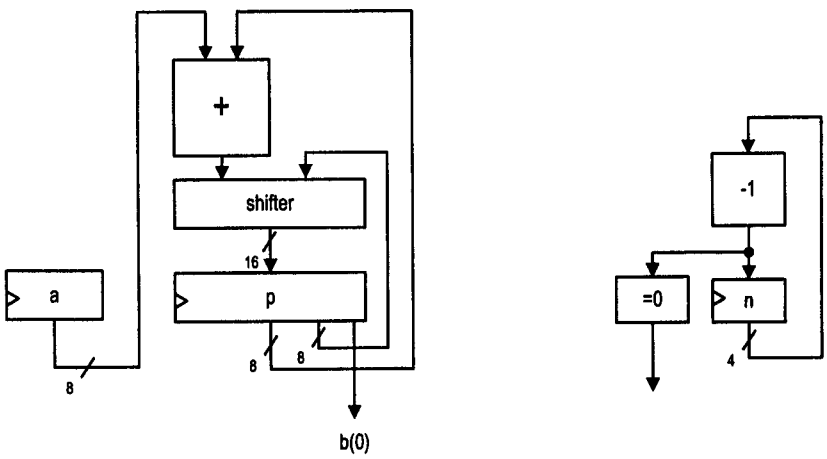
We can also improve the efficiency of the data path. Note that when a is added to the partial products, only the eight leftmost bits of the partial product are involved in the operation and the remaining trailing bits are kept unchanged. Instead of using a 16-bit adder, we can reduce the width of the adder to 9 bits (an 8-bit operand plus a 1-bit carry). This requires to selectively route a portion of the partial product to the adder. The "selective routing" involves complex multiplexing circuits and is not desirable. A better alternative



(a) Initial design



(b) "Shifting p register" design



(c) Final design

Figure 11.18 Sketches of the data path of add-and-shift multipliers.

is to shift the partial product to the right one position in each iteration, and thus the eight current leftmost bits are always connected to the input of the adder. This approach also eliminates the need of shifting multiplier (A) and reduces the width of the a register by half. The sketch of the revised data path is shown in Figure 11.18(b).

The circuit adds the upper half (the left half) of the p register and the a register and then combines the output of the adder with the original lower half (the right half) of the p register to form the new partial product. The p register is then shifted right one bit. Since we wish to merge the addition and shifting operations in the same state, there is no register between the adder and shifter and thus the two operations are performed in the same clock cycle. Because shifting right one bit involves only wiring, merging the two operations will not affect the critical path or increase the clock period.

Another minor improvement is to utilize the right unused portion of the p register. Note that initially only the left half of the p register contains the valid data. The valid portion expands to the right one position in each iteration when the shift-right operation is performed. On the other hand, the b register has eight valid bits initially. In each iteration, it shifts to the right one position and discards the LSB. Since the expansion of the left part of the p register matches the shrinkage of the b register, we can utilize the unused right part of the p register to function as the b register and eliminate the original b register. The sketch of the final data path is shown in Figure 11.18(c).

The refined and improved ASMD chart is shown in Figure 11.19. We use the notations pu and pl as the aliases for the upper half (left half) and lower half (right half) of the p register. Since the addition and shifting operations are performed in the same state, we must use the addition results for the following shift operation. To achieve the desired effect, we use the regular assignment notation (<=) instead of the RT notation (←) in the two conditional output boxes (i.e., pu_next <= pu and pu_next <= pu + a). The result (i.e., pu_next) is then used in a regular RT shift operation (i.e., p ← p_next >> 1). The VHDL code can be derived following the ASMD chart and is shown in Listing 11.8.

Listing 11.8 Refined description of an add-and-shift sequential multiplier

```
architecture shift_add_better_arch of seq_mult is
   constant WIDTH: integer:=8;
   constant C_WIDTH: integer:=4; -- width of the counter
   constant C_INIT: unsigned(C_WIDTH-1 downto 0):="1000";
   type state_type is (idle, add_shft);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
   signal p_reg, p_next: unsigned(2*WIDTH downto 0);
   -- alias for the upper and lower parts of p_reg
   alias pu_next: unsigned(WIDTH downto 0) is
                  p_next(2*WIDTH downto WIDTH);
   alias pu_reg: unsigned(WIDTH downto 0) is
                  p_reg(2*WIDTH downto WIDTH);
   alias pl_reg: unsigned(WIDTH-1 downto 0) is
                  p_reg(WIDTH-1 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
```

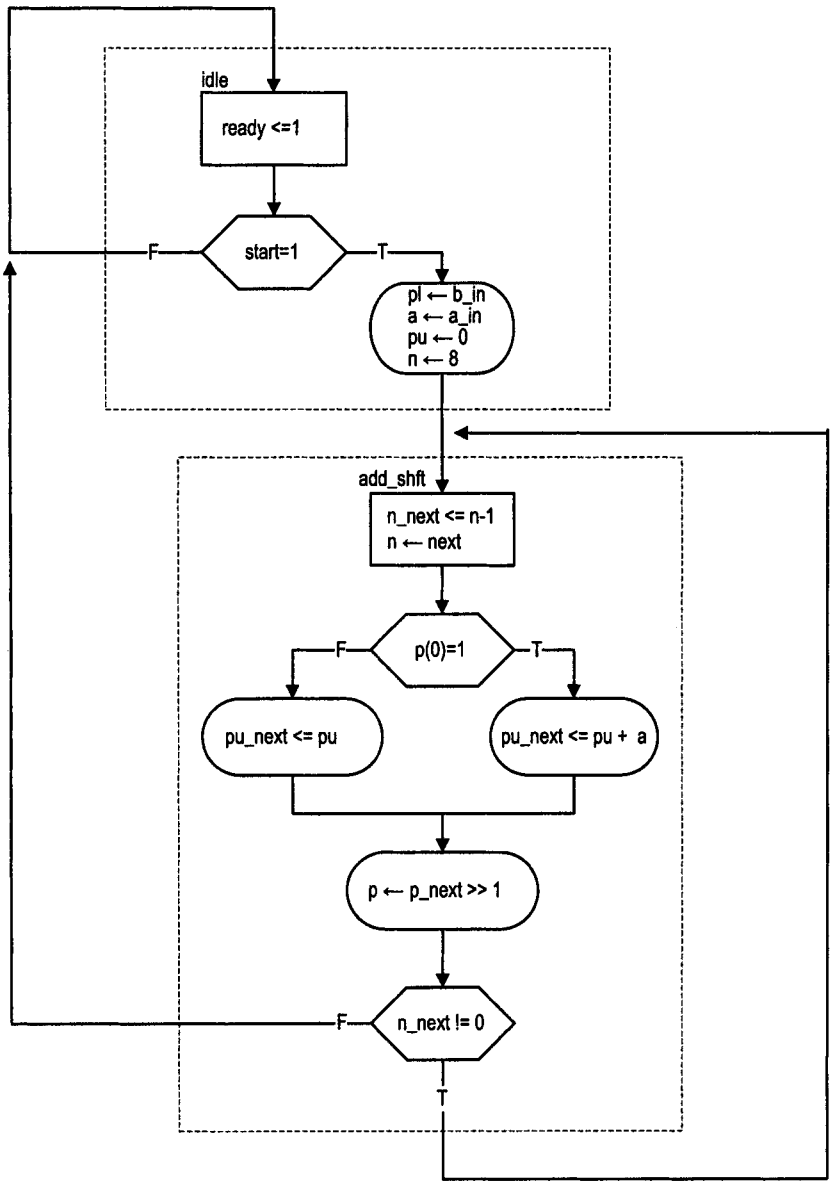


Figure 11.19 ASMD chart of the refined add-and-shift multiplier.

```
        a_reg <= (others=>'0');
        n_reg <= (others=>'0');
        p_reg <= (others=>'0');
     elsif (clk'event and clk='1') then
        state_reg <= state_next;
        a_reg <= a_next;
        n_reg <= n_next;
        p_reg <= p_next;
     end if;
  end process;
  -- combinational circuit
  process(start,state_reg,a_reg,n_reg,p_reg,
          a_in,b_in,n_next,p_next)
  begin
     a_next <= a_reg;
     n_next <= n_reg;
     p_next <= p_reg;
     ready <='0';
     case state_reg is
        when idle =>
           if start='1' then
              p_next <= "000000000" & unsigned(b_in);
              a_next <= unsigned(a_in);
              n_next <= C_INIT;
              state_next <= add_shft;
           else
              state_next <= idle;
           end if;
           ready <='1';
        when add_shft =>
           n_next <= n_reg - 1;
           -- add if multiplier bit is '1'
           if (p_reg(0)='1') then
              pu_next <= pu_reg + ('0' & a_reg);
           else
              pu_next <= pu_reg;
           end if;
           --shift
           p_next <= '0' & pu_next &
                     pl_reg(WIDTH-1 downto 1);
           if (n_next /= "0000") then
              state_next <= add_shft;
           else
              state_next <= idle;
           end if;
     end case;
  end process;
  r <= std_logic_vector(p_reg(2*WIDTH-1 downto 0));
end shift_add_better_arch;
```

Table 11.1 Comparison of performance and circuit complexity of three multipliers

Design method	# Clock cycles	Size of functional units	# Register bits
Repetitive-addition	2 to $2^n + 1$	$2n$-bit adder, n-bit decrementor	$4n$
Add-and-shift (initial)	$n + 1$ to $2n + 1$	$2n$-bit adder, $\lceil \log_2(n + 1) \rceil$-bit dec	$5n + \lceil \log_2(n + 1) \rceil$
Add-and-shift (refined)	$n + 1$	n-bit adder, $\lceil \log_2(n + 1) \rceil$-bit dec	$3n + \lceil \log_2(n + 1) \rceil + 1$

11.6.3 Comparison of three ASMD designs

We have examined several designs of a sequential multiplier. Table 11.1 summarizes the key characteristics of these designs, including the Mealy-based repetitive-addition design of Section 11.4.2 and two add-and-shift designs in this section. We assume that the width of the input operands is n bits. The table lists the range of the number of clock cycles to complete the multiplication, the size of the functional units in the data path, and the total number of bits in the data registers. Note that in add-and-shift designs, the counter counts from n to 0 for n-bit operands. There are $n + 1$ patterns in the counting sequence, and thus the width of the counter is $\lceil \log_2(n + 1) \rceil$ bits.

The table shows that the hardware complexity of the repetitive-addition multiplier and the initial add-and-shift multiplier are comparable but that the latter significantly improves performance by reducing the worst-case clock cycles from about 2^n to $2n$. The refined add-and-shift design reduces the hardware complexity roughly by half and decreases the worst-case clock cycles from about $2n$ to n. The adder is the dominant part in the design and contributes most to the propagation delay of the data path. Because of the smaller adder, we can expect the refined add-and-shift design to have a smaller clock period as well.

After we become familiar with the process of converting an ASMD chart to VHDL code, it becomes more or less a mechanical procedure. The key is to find an efficient algorithm and researching an effective data path to support the RT operations in the algorithm. We then can derive the ASMD chart and VHDL code accordingly. As the sequential multiplier examples show, the effectiveness of a design ultimately relies on our understanding of the problem and hardware. No synthesis software can convert the repetitive-addition algorithm into an add-and-shift algorithm or convert the initial add-and-shift design to the refined design.

11.7 SYNTHESIS OF FSMD

The design methodology and VHDL coding style discussed in this chapter impose no new synthesis requirement. From the software's point of view, an FSMD code is just a code with both regular sequential circuits and an FSM and thus can be synthesized accordingly. We can separate the control path and data path in VHDL code if we want to use special FSM optimization software for the control path synthesis.

The synthesis of an algorithm can also be performed in a more abstract level, known as *high-level synthesis* or *behavioral synthesis*. The synthesis starts with abstract VHDL descriptions that are coded in pure sequential statements, similar to those used in the al-

gorithm's pseudocode. The behavioral synthesis software converts the initial description into RT operations and automatically derives a control path and a data path. This kind of synthesis is limited to certain specialized applications. We discuss this in an example in Chapter 12.

11.8 SYNTHESIS GUIDELINES

An FSMD is a synchronous circuit with a regular sequential circuit (the data path) and an FSM (the control path). We should follow the guidelines from Chapters 8, 9 and 10. Few additional guidelines are related primarily to RT operations and construction of the data path:

- As any sequential circuit, the registers of the FSMD should be separated from the combinational circuits.
- Be aware that an RT operation exhibits a delayed-store behavior. Use of a register in a decision box should be carefully examined.
- The variables used in Boolean expressions of a pseudo algorithm normally correspond to the next values of the registers used in an ASMD.
- The function units are normally the most dominant components in a FSMD design. To exercise more control, we may need to isolate them from the rest of the code.
- Separate the control path from the code if the FSM optimization is needed later.

11.9 BIBLIOGRAPHIC NOTES

FSMD and ASMD chart provide a powerful methodology to realize sequential algorithms in hardware. The text, *Principles of Digital Design* by D. D. Gajski, has a comprehensive chapter on the representation and synthesis of FSMD. The text, *Verilog Digital Computer Design* by M. G. Arnold, applies RT methodology for computer design. As its name shows, Verilog is used for the text.

Problems

11.1 The ASMD chart in Figure 11.6 uses the `n` register to keep track of the number of iterations. It is initialized with `b_in` and counts down to 0. Alternatively, it can be initialized with 0 and counts up to `b_in`. From the implementation point of view, which method is better? Explain.

11.2 The ASMD chart in Figure 11.6 must return to the `idle` state after completion even when the main system is ready with a new set of inputs. An alternative is to allow the circuit to perform back-to-back operation in which the FSMD jumps to the `ab0` or `load` state if the `start` signal is asserted while the current operation is completed.

(a) Modify the ASMD chart to reflect the change.

(b) Derive the VHDL code.

11.3 In the ASMD chart of Figure 11.13, what happens if we replace the `a_in=0 or b_in=0` expression of the `idle` state with the `n=0 or b=0` expression? Explain.

11.4 In Listing 11.4, what happens to the algorithm if `n_next` is declared as a signal?

11.5 Repetitive-subtraction division is an algorithm to implement division operation. Let `y` and `d` be the dividend and divisor respectively. This algorithm obtains the quotient (`q`) and the remainder (`r`) by subtracting `d` from `y` repeatedly until the remaining of `y` is smaller than `d`. Assume that all signals are 8 bits wide and interpreted as unsigned integers.

(a) Derive a pseudo algorithm.
(b) Convert the pseudo algorithm into an ASMD chart.
(c) Derive a detailed conceptual diagram.
(d) Derive the VHDL code according to the blocks of the conceptual diagram (i.e., in multi-segment style).
(e) Derive the VHDL code in two-segment style.
(f) Is this an efficient algorithm? Explain.

11.6 A leading-zero counting circuit counts the number of consecutive 0's from an input signal. We want to design a sequential version of this circuit. The design should check one bit of the input at a time and increment accordingly. The counting stops when the first '1' is encountered.

(a) Derive a pseudo algorithm.
(b) Convert the pseudo algorithm into an ASMD chart.
(c) Derive the VHDL code in two-segment style.
(d) Assume that the input is 16 bits wide. Synthesize both combinational and sequential versions in an ASIC technology. Compare the size and performance.

11.7 The Fibonacci function is defined as

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

We want to implement this function in hardware. Assume that `n` is a 6-bit input and interpreted as an unsigned integer. Note that $fib(63)$ is 6557470319842.

(a) Derive an ASMD chart.
(b) Derive the VHDL code.

11.8 In the ASMD chart of Figure 11.13, we can express the condition in the bottom decision box as `n_next=0` or `n=1`. From the timing's point of view, which one can help to get a higher clock rate? Explain.

11.9 Synthesize the combinational multiplier in Section 7.5.4 and the sequential multiplier described by the ASMD chart of Figure 11.19 using an ASIC technology.

(a) Assume that the input operands are 8 bits wide. Compare the size and performance of the two circuits.
(b) Repeat part (a), but assume that the input operands are 16 bits wide.

11.10 For the sequential multiplier described by the ASMD chart of Figure 11.19, eight iterations of add-and-shift operation are needed. We can improve the design further by reducing the number of iterations to seven.

(a) Derive an ASMD chart.
(b) Derive the VHDL code.
(c) Assume that the width of the input operands is n. Calculate the relevant parameters of Table 11.1 for this improved design.

11.11 In the sequential add-and-shift multiplier, we can use a combinational circuit to process 2 bits at a time. Instead of adding 0 or shifted A to the partial product, we can add 0, shifted A, shifted $2A$, or shifted $3A$ to the partial product.

(a) Derive the revised ASMD chart for this circuit.

(b) Derive the VHDL code.

(c) Synthesize the design with an ASIC technology. Compare the size and performance of this design and the original design.

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.

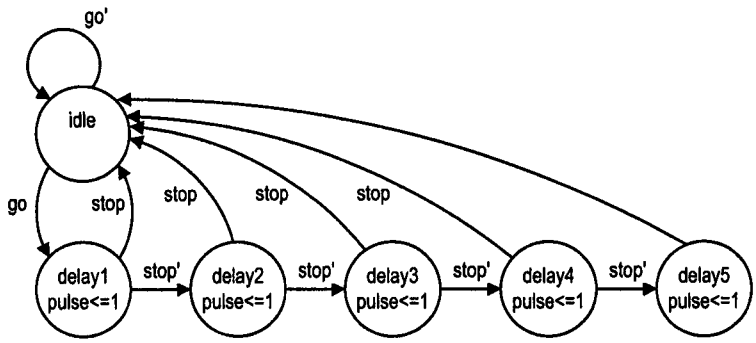


Figure 12.1 State diagram of a one-shot pulse generator.

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;
```

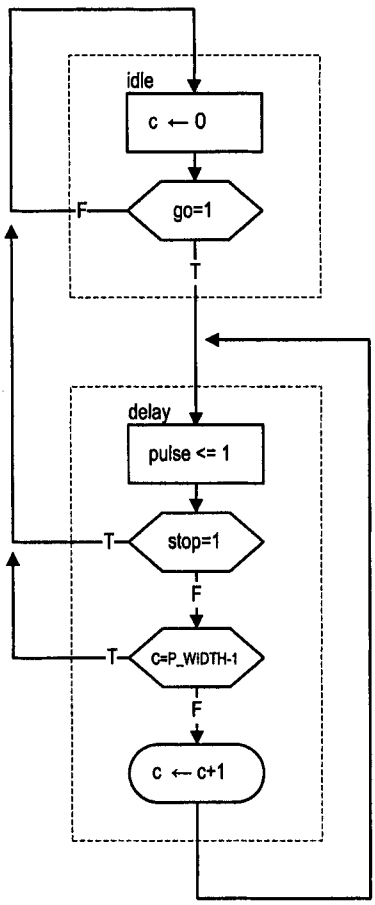


Figure 12.2 ASMD chart of a one-shot pulse generator.

```
    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);
```

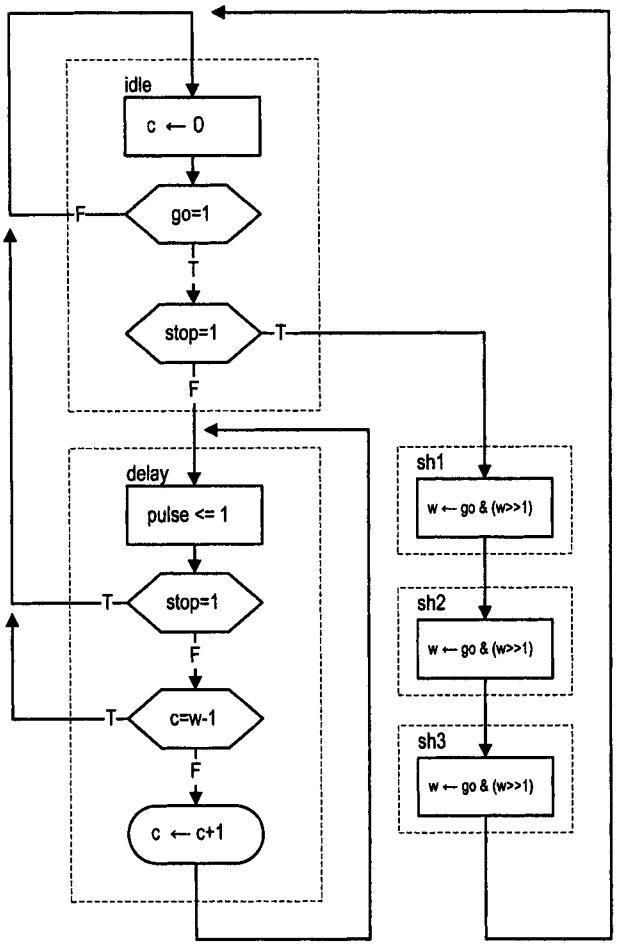


Figure 12.3 ASMD chart of a programmable one-shot pulse generator.

```
5    signal w_reg, w_next: unsigned(2 downto 0);
  begin
     -- state and data registers
     process(clk,reset)
     begin
10      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
15         state_reg <= state_next;
           c_reg <= c_next;
           w_reg <= w_next;
        end if;
     end process;
20   -- next-state logic & data path functional units/routing
     process(state_reg,go,stop,c_reg,w_reg)
     begin
        pulse <= '0';
        c_next <= c_reg;
25      w_next <= w_reg;
        case state_reg is
           when idle =>
              if go='1' then
                 if stop='1' then
30                  state_next <= sh1;
                 else
                    state_next <= delay;
                 end if;
              else
35               state_next <= idle;
              end if;
              c_next <= (others=>'0');
           when delay =>
              if stop='1' then
40               state_next <=idle;
              else
                 if (c_reg=w_reg-1) then
                    state_next <=idle;
                 else
45                  c_next <= c_reg + 1;
                    state_next <=delay;
                 end if;
              end if;
              pulse <= '1';
50         when sh1 =>
              w_next <= go & w_reg(2 downto 1);
              state_next <= sh2;
           when sh2 =>
              w_next <= go & w_reg(2 downto 1);
55            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.

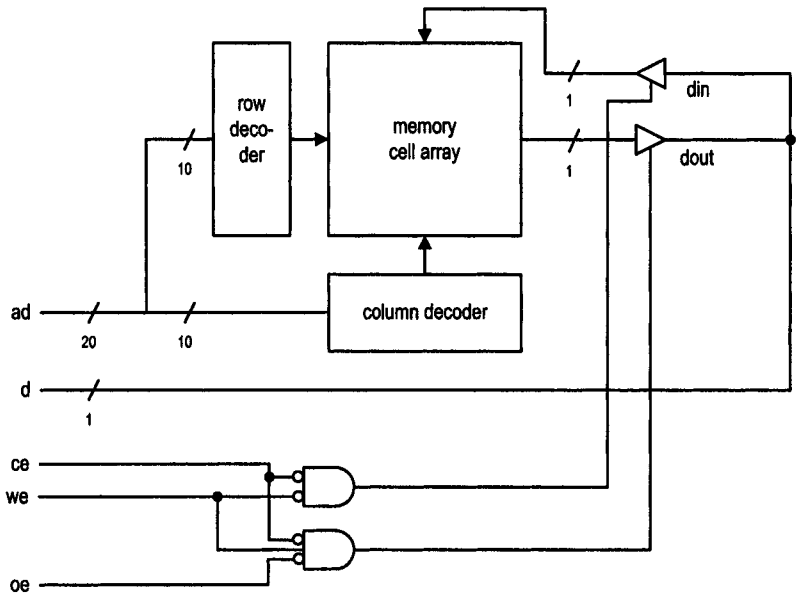
12.3.1 Overview of SRAM

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.

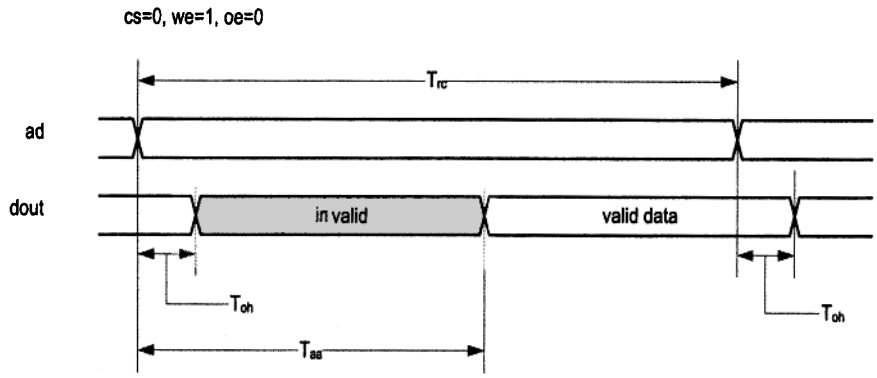


(a) Block diagram

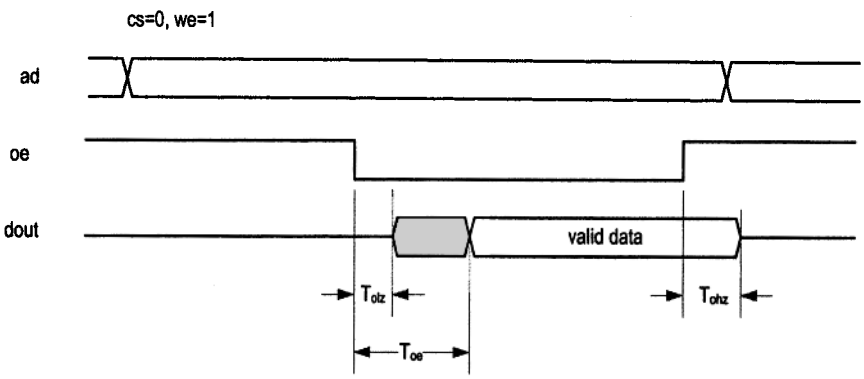
ce	we	oe	**Operation**	**Data pin** d
1	-	-	no operation	Z
0	0	-	write	data in
0	1	0	read	data out
0	1	1	no operation	Z

(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.

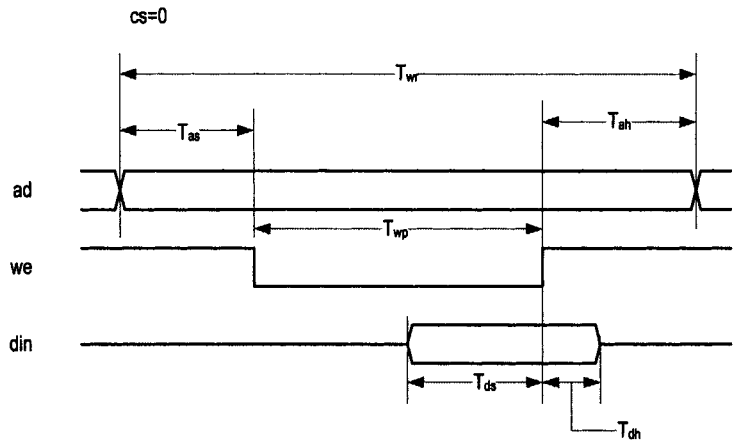


Figure 12.6 Timing diagram of an SRAM write cycle.

- 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

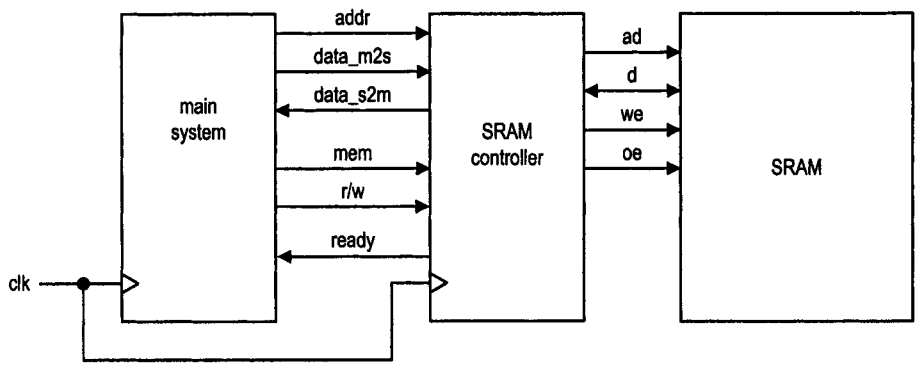


Figure 12.7 Role of an SRAM controller.

- 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

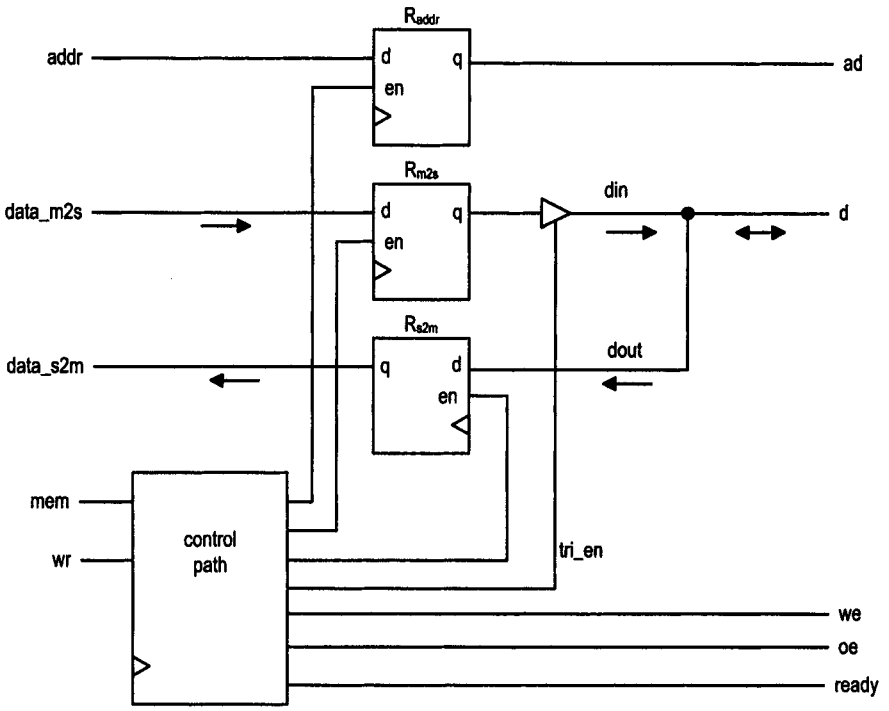


Figure 12.8 Block diagram of an SRAM controller.

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".

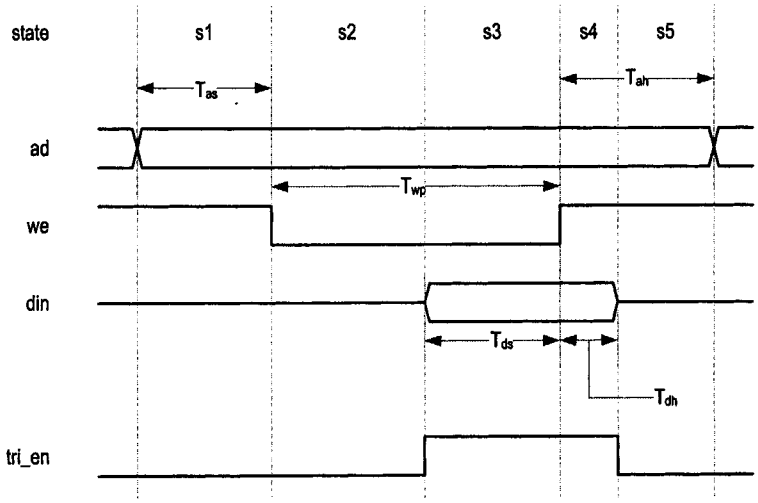
Closer examination of the SRAM's timing specifications can help us to simplify the FSM. For example, T_{wp} is normally much larger than T_{ds} in most SRAMs, and there is no harm in placing data on the din line earlier. Thus, we can merge the s2 and s3 states into a single state. Also, since there in 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.

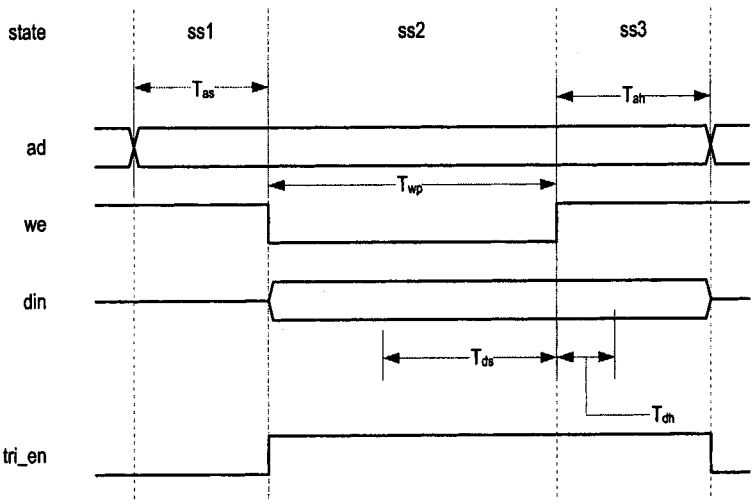
In the second step, we refine the FSM according to the system clock period and SRAM timing parameters. The SRAM's access time and the main system's clock rate are the two key factors in the final design of the control path. The following examples illustrate the design and relevant issues for a slow SRAM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

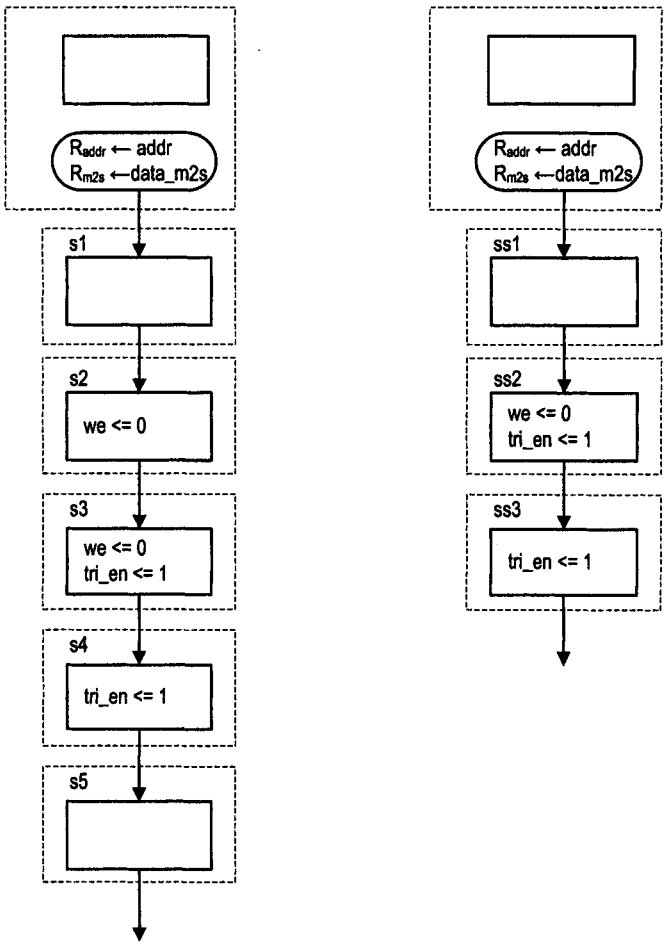


Figure 12.10 FSM segments for a write cycle.

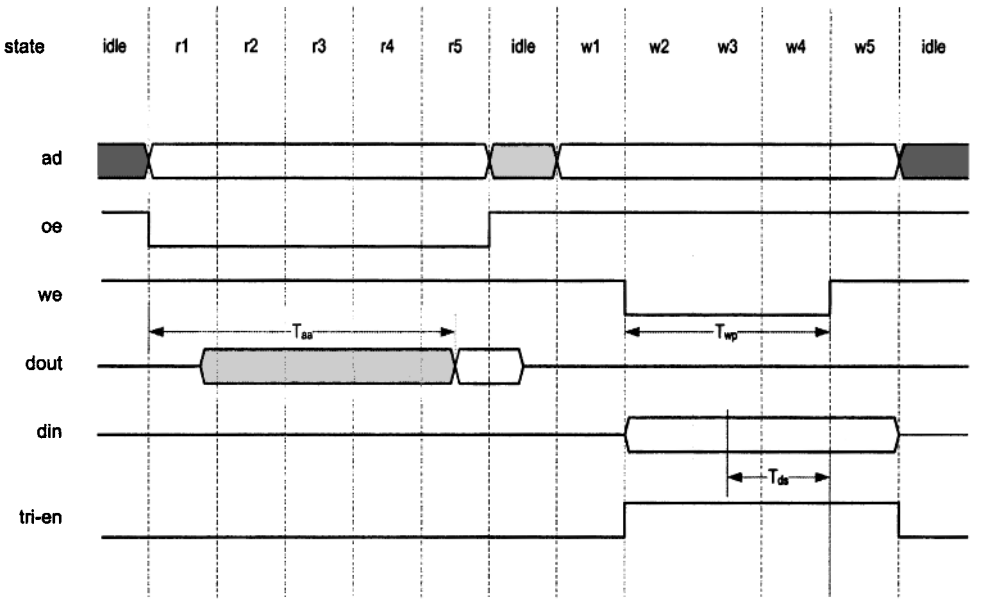


Figure 12.11 Division of read and write cycles of a slow SRAM.

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).

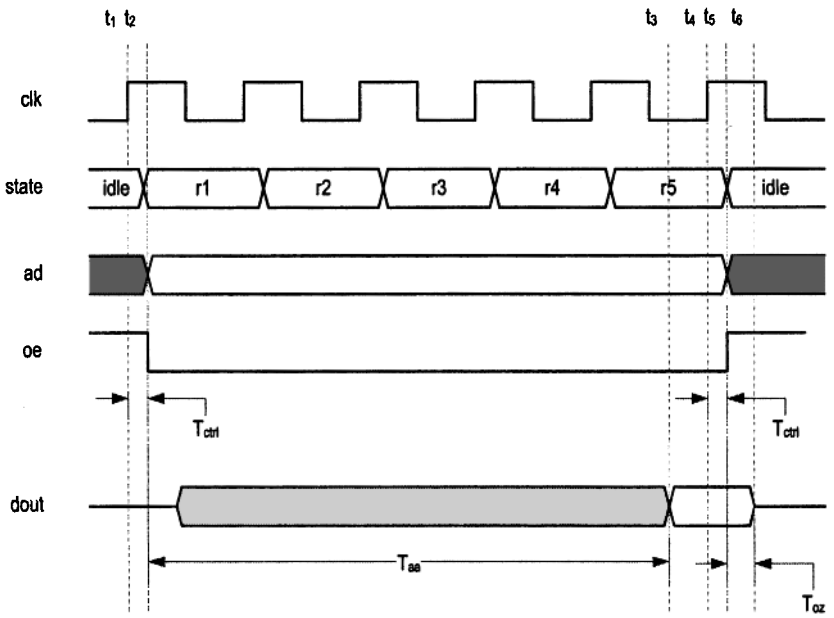


Figure 12.12 Detailed timing diagram of the read cycle.

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
```

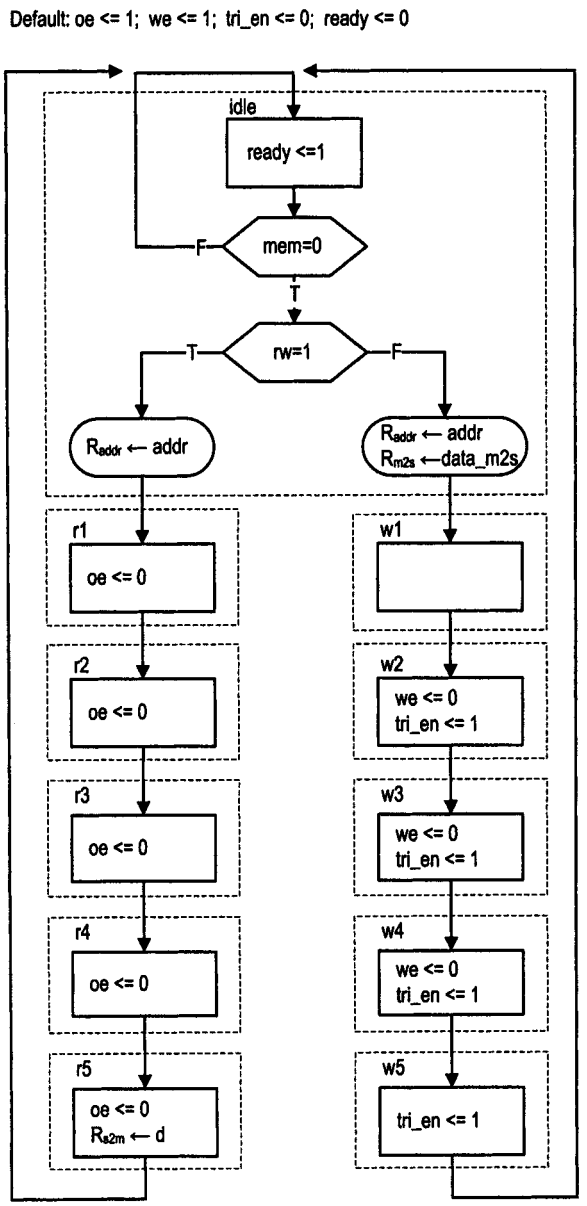


Figure 12.13 ASMD chart for a slow SRAM controller.

```
    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';
```

```
110          when w4 =>
                we_buf <= '0';
                tri_en_buf <= '1';
             when w5 =>
                tri_en_buf <= '1';
115          when r1 =>
                oe_buf <= '0';
             when r2 =>
                oe_buf <= '0';
             when r3 =>
120             oe_buf <= '0';
             when r4 =>
                oe_buf <= '0';
             when r5 =>
                oe_buf <= '0';
125       end case;
      end process;
      -- output
      we <= we_reg;
      oe <= oe_reg;
130   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.

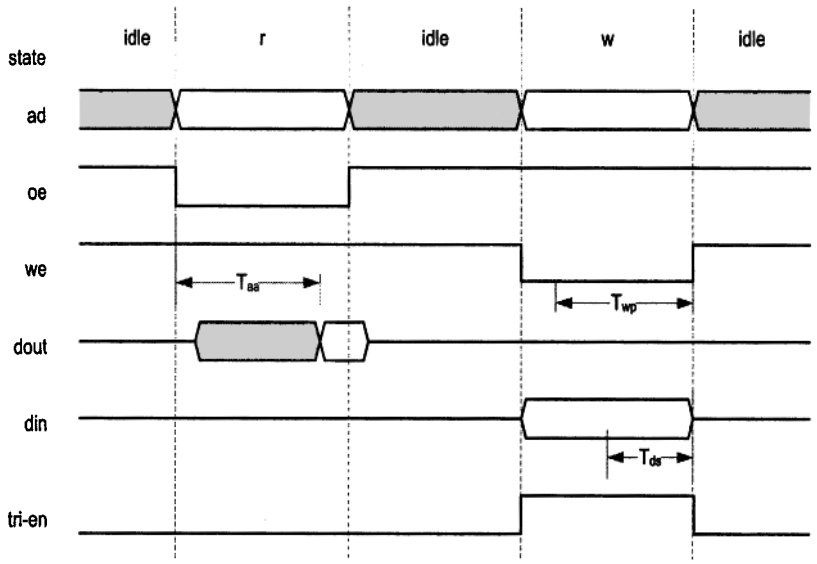


Figure 12.14 Division of read and write cycles of a fast SRAM.

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');
```

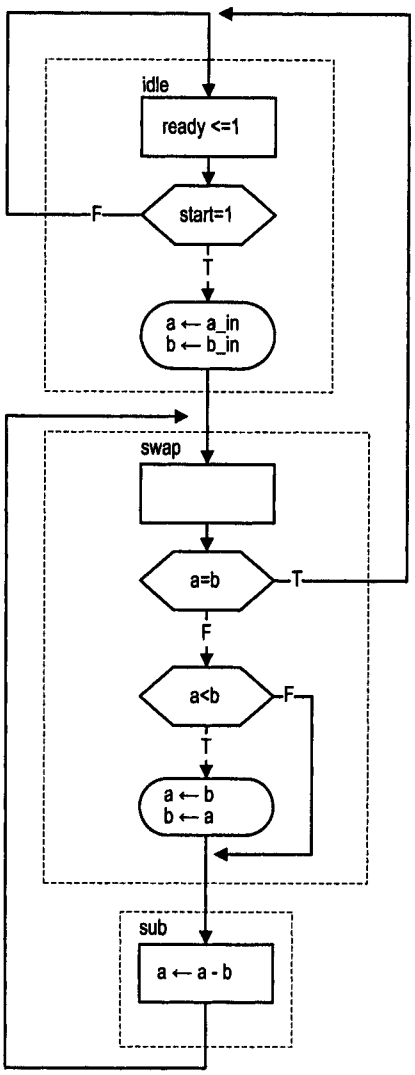


Figure 12.15 ASMD chart of the initial GCD circuit.

```
25          b_reg <= (others=>'0');
         elsif (clk'event and clk='1') then
            state_reg <= state_next;
            a_reg <= a_next;
            b_reg <= b_next;
30       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
35       a_next <= a_reg;
         b_next <= b_reg;
         case state_reg is
            when idle =>
               if start='1' then
40                a_next <= unsigned(a_in);
                  b_next <= unsigned(b_in);
                  state_next <= swap;
               else
                  state_next <= idle;
45             end if;
            when swap =>
               if (a_reg=b_reg) then
                  state_next <= idle;
               else
50                if (a_reg < b_reg) then
                     a_next <= b_reg;
                     b_next <= a_reg;
                  end if;
                  state_next <= sub;
55             end if;
            when sub =>
               a_next <= a_reg - b_reg;
               state_next <= swap;
            end case;
60    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
```

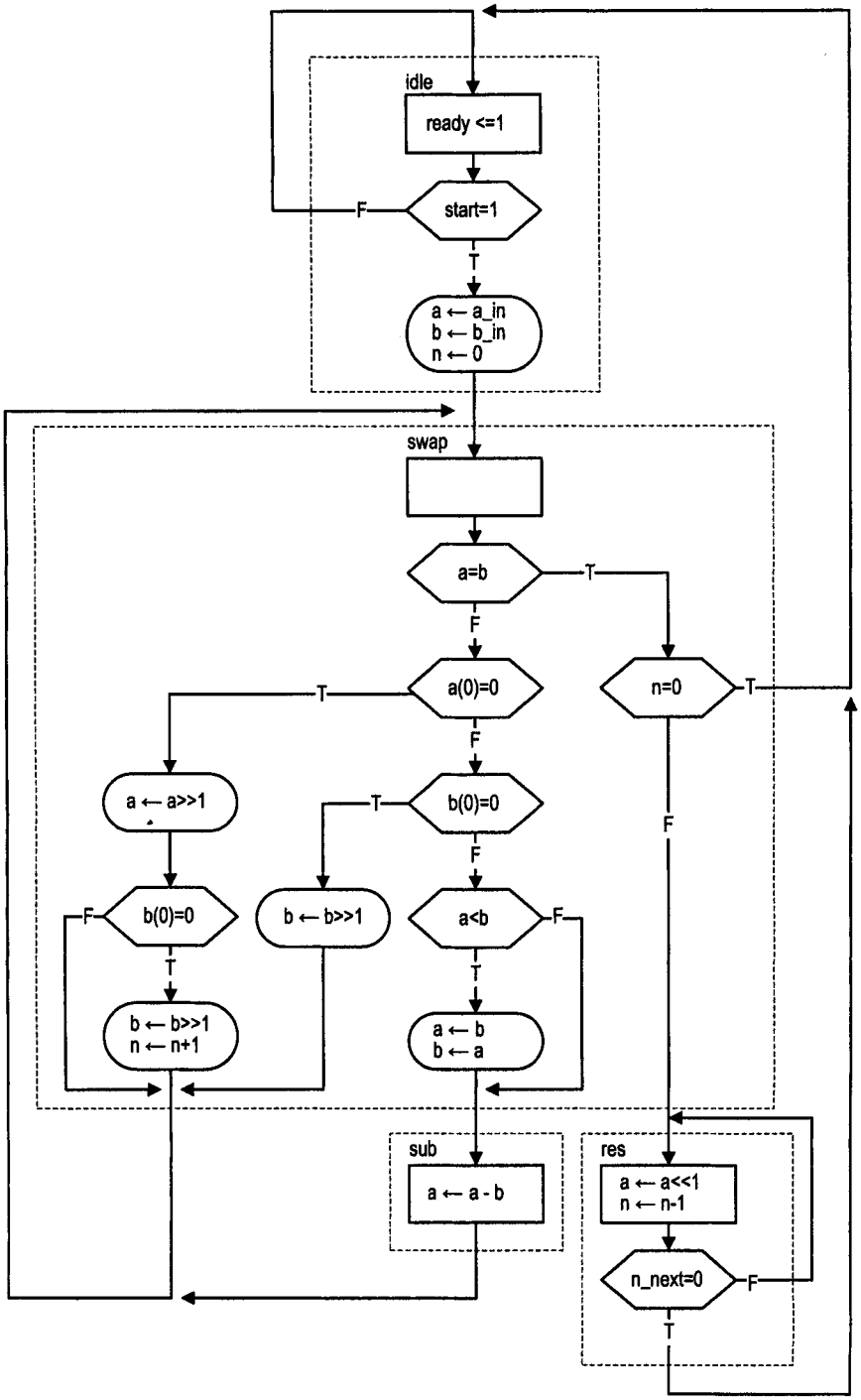


Figure 12.16 ASMD chart of the revised GCD circuit.

```
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');
30      b_zero <= (others=>'0');
        case state_reg is
           when idle =>
              if start='1' then
                 a_next <= unsigned(a_in);
35               b_next <= unsigned(b_in);
                 n_next <= (others=>'0');
                 state_next <= swap;
              else
                 state_next <= idle;
40            end if;
           when swap =>
              if (a_reg=b_reg) then
                 if (n_reg=0) then
                    state_next <= idle;
45               else
                    state_next <= res;
                 end if;
              else
                 if (a_reg(0)='1' and b_reg(0)='1') then -- swap
50                  if (a_reg < b_reg) then
                       a_next <= b_reg;
                       b_next <= a_reg;
                    end if;
                    state_next <= sub;
55               else
                    -- shift out 0s of a_reg
                    if (a_reg(0)='1') then
                         a_zero <="000";
                    elsif (a_reg(1)='1') then
60                     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";
65                  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);
70                       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
75                     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";
80                  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 =>
```

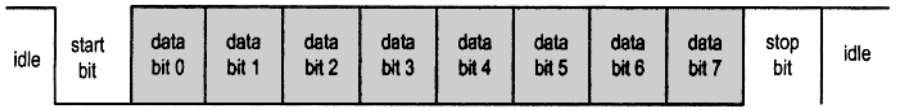


Figure 12.17 Transmission of a byte.

```
                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);
```

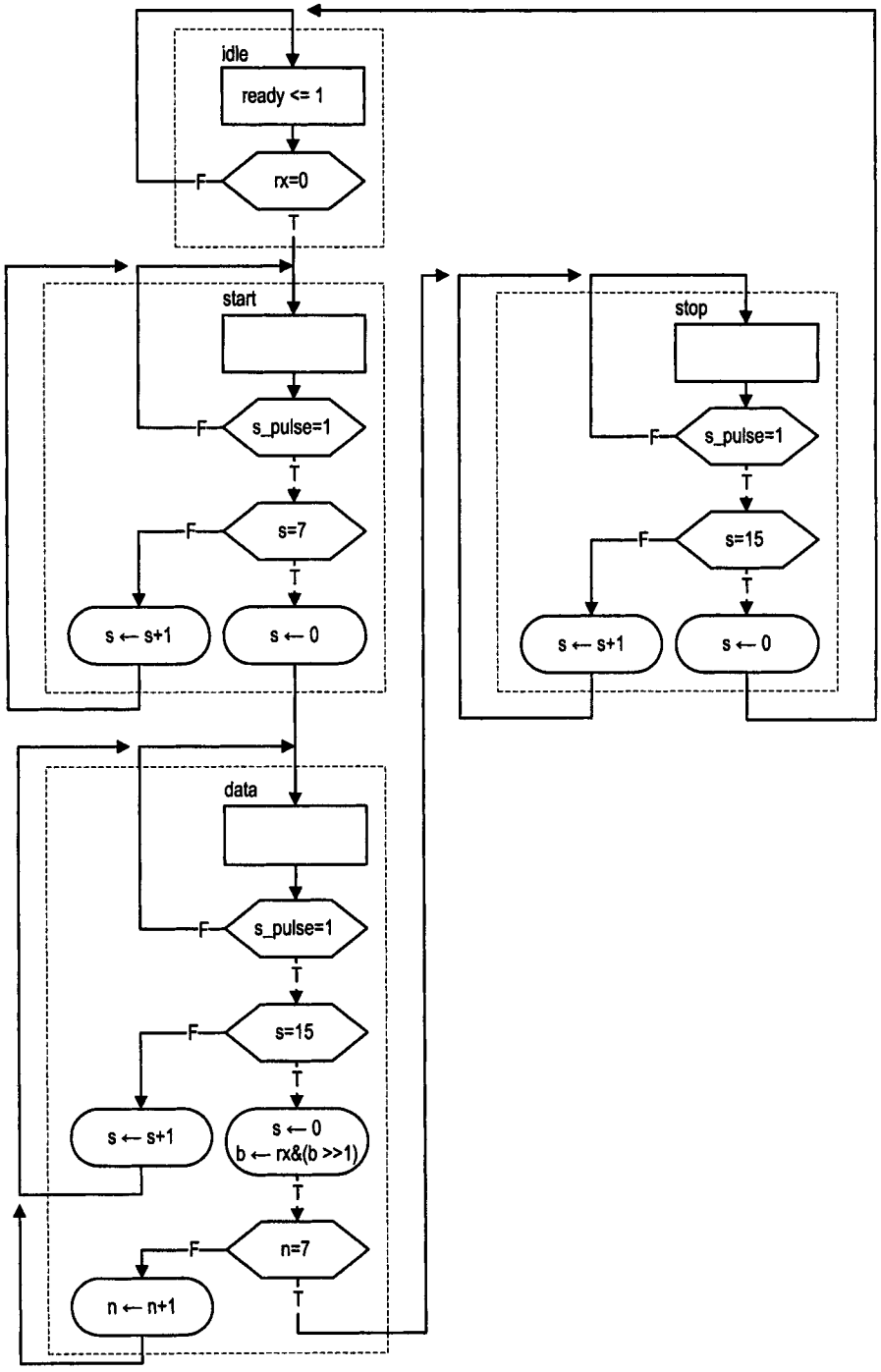


Figure 12.18 ASMD chart of a UART receiver.

```
    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:

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

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

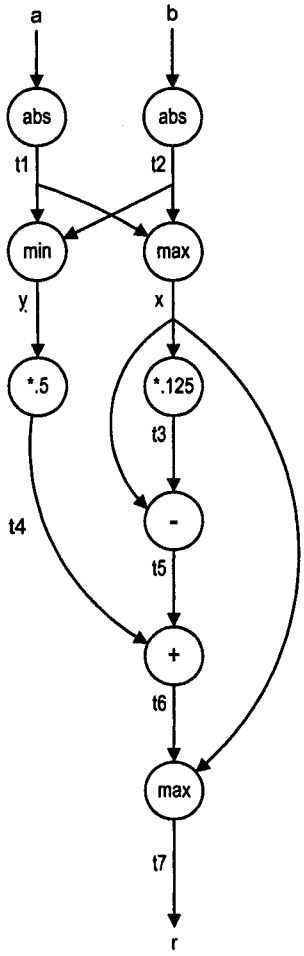


Figure 12.19 Dataflow graph.

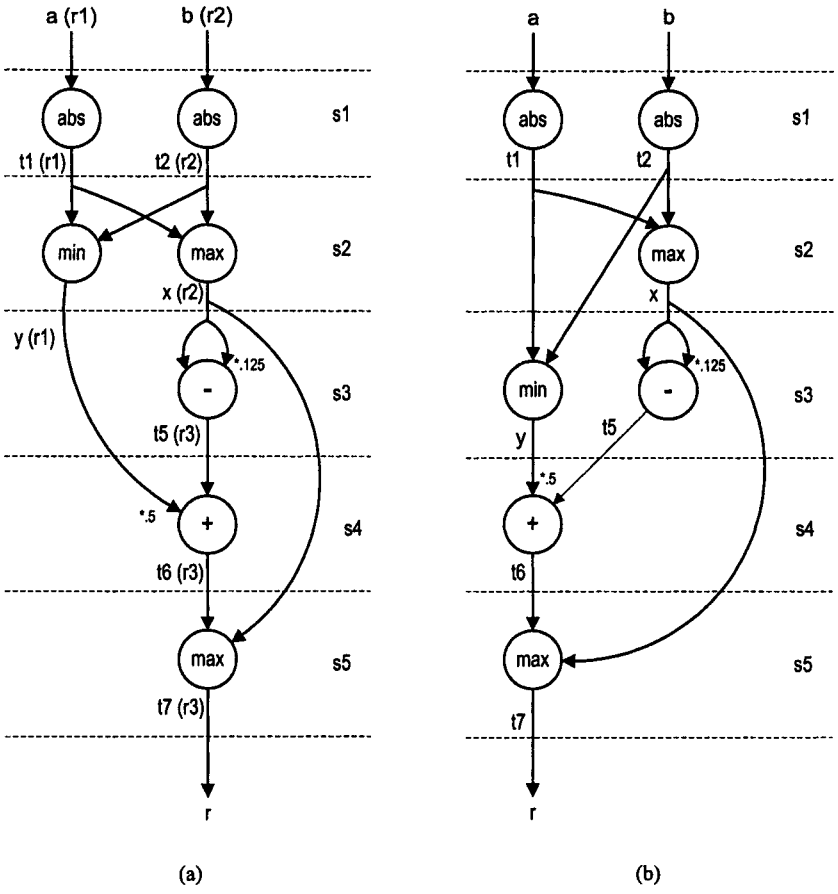


Figure 12.20 Schedules with two functional units.

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

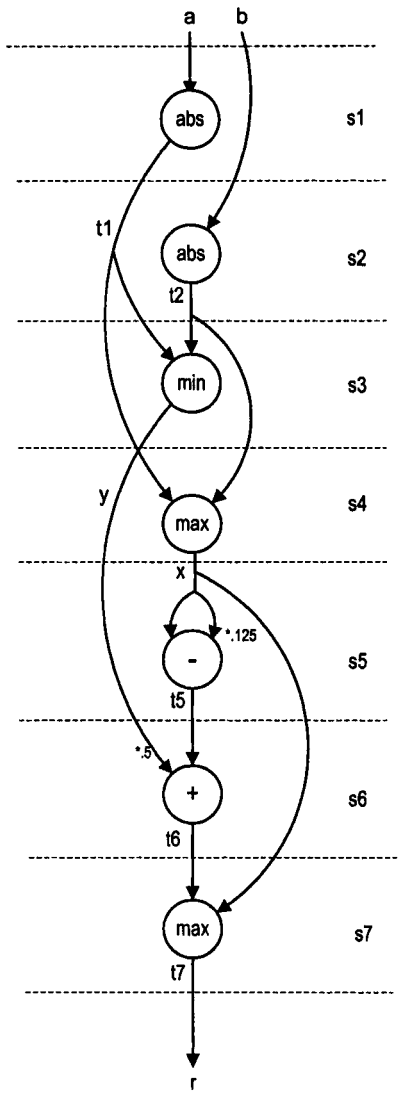


Figure 12.21 Schedule with one functional unit.

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

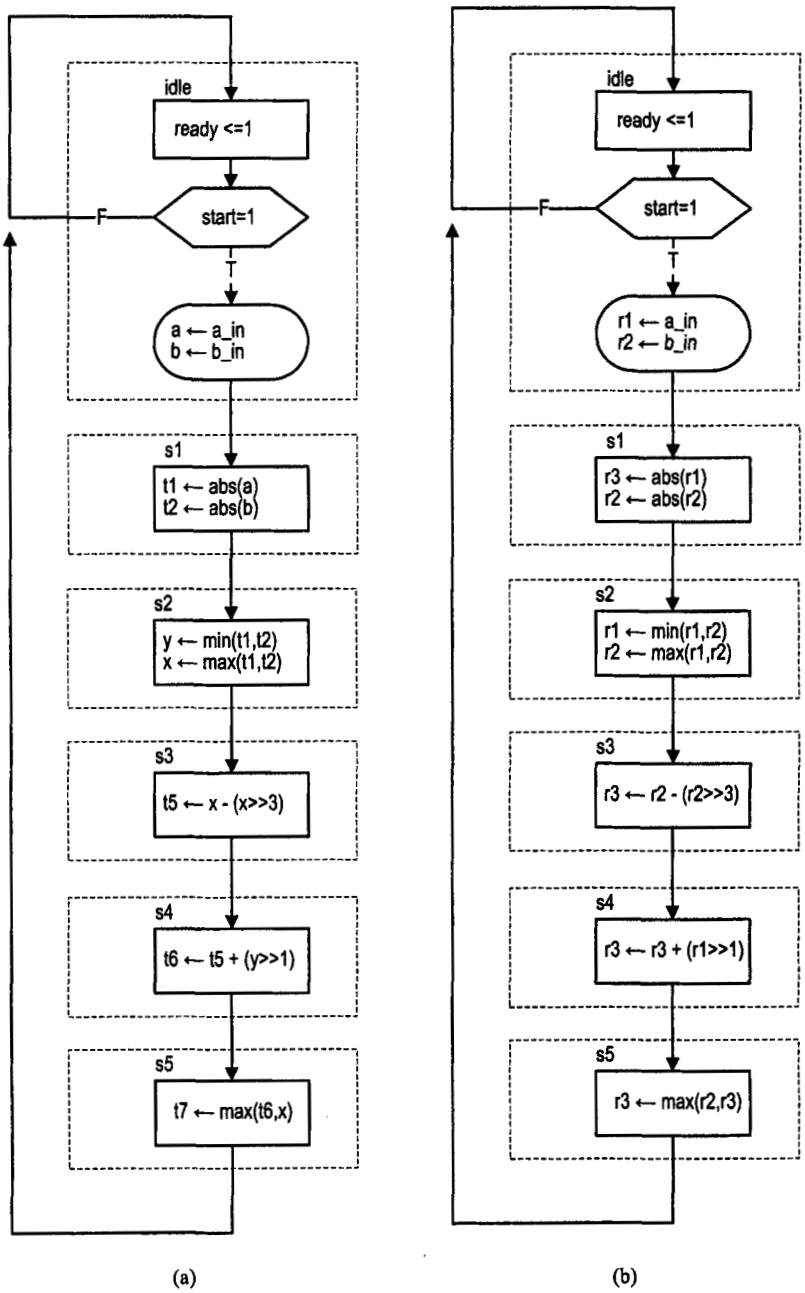


Figure 12.22 ASMD charts of a square-root approximation circuit.

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;
140 -- output routing
    process(state_reg,r1_reg,r2_reg,r3_reg,sum)
    begin
       case state_reg is
          when s1 => -- |b|
145          if sum(WIDTH)='0' then  -- (0-b)>0
                au2_out <= sum; -- -b
             else
                au2_out <= r2_reg; -- b
             end if;
150       when s2 =>
             if sum(WIDTH)='0' then
                au2_out <= r1_reg;
             else
                au2_out <= r2_reg;
155          end if;
          when s3|s4 => -- +,-
             au2_out <= sum;
          when others => -- s5
             if sum(WIDTH)='0' then
160             au2_out <= r3_reg;
             else
                au2_out <= r2_reg;
             end if;
       end case;
165 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.

(b) Derive the VHDL code.

12.8 Repeat Problem 12.7 for a stack controller.

12.9 Consider the GCD circuit in Section 12.4. Assume that inputs are N-bit unsigned integers. For each architecture:

(a) Determine the input pattern that leads to the maximal number of clock cycles.

(b) Calculate the exact number of clock cycles for the pattern.

12.10 For the `fast_arch` architecture of the GCD circuit, we can combine the subtraction with the comparison and merge the `sub` state into the `swap` state.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) Assume that the clock period is doubled because of the merge. Discuss whether the merge actually increases the performance (i.e., completes the computation in less time).

(d) Repeat part (c), but assume that the clock period is increased by only 50%.

12.11 In the `fastest_arch` architecture of the GCD circuit, up to two shifting operations are performed in the `swap` state and one is performed in the `res` state. Since a barrel shifter is a complex circuit, we want the three operations to share one unit.

(a) Derive the VHDL code of a barrel shifter that can perform both shift-right and shift-left operations.

(b) Derive the revised ASMD chart.

(c) Derive the VHDL code.

12.12 Design a transmitter for the UART discussed in Section 12.5.

(a) Derive the ASMD chart.

(b) Derive the VHDL code.

12.13 Expand the UART receiver of Section 12.5 to make the baud rate adjustable. Assume that there is an additional 2-bit control signal, `baud_sel`, which specifies the desired baud rate, which can be 1200, 2400, 4800 or 9600 baud.

12.14 Revise the UART of Section 12.5 to include an even-parity bit. The length of the data is now 7 bits, and the eighth bit is the parity bit. The parity bit is asserted when there is an odd number of 1's in data bits (and thus makes the 8 received bits always have an even number of 1's). Design a transmitter and a receiver for the modified UART.

(a) Derive the ASMD chart for the transmitter.

(b) Derive the VHDL code for the transmitter.

(c) Derive the ASMD chart for the receiver. The receiver should include an extra output signal to indicate the occurrence of a parity error.

(d) Derive the VHDL code for the receiver.

12.15 Expand the UART receiver of Section 12.5 to accommodate different parity schemes. Assume that there is an extra 2-bit control signal, `pairty_sel`, which selects the desired parity scheme, which can be odd parity, even parity or no parity.

12.16 Consider a UART that can communicate at four baud rates: 1200, 2400, 4800 and 9600 baud. Assume that the actual baud rate is unknown but the transmitter always sends a "11111111" data byte at the beginning of the session. Design a circuit that can automatically determine the baud rate and derive the VHDL code.

12.17 Consider the schedule in Figure 12.20(b).

(a) Map the variables into a minimum number of registers.

(b) Derive the ASMD chart for the schedule. Recall that two arithmetic units are used in this schedule.

(c) Derive the VHDL code.

12.18 Repeat Problem 12.17 for the schedule in Figure 12.21. Note that only one arithmetic unit is used in this schedule.

12.19 Multiplication can be implemented by performing additions of shifted bit-products, as discussed in Section 7.5.4. Let $p_7, p_6, \ldots, p_1, p_0$ be the shifted bit-products of an 8-bit multiplier. The final product can be expressed as

$$y = p_7 + p_6 + \cdots + p_1 + p_0$$

(a) Derive the dataflow graph for the expression. Arrange the additions as a tree to exploit parallelism.

(b) Assume that only one adder is provided. Derive a schedule.

(c) Derive the ASMD chart for the schedule. Use a minimal number of registers in the chart.

(d) Derive the VHDL code.

(e) Discuss the difference between this design and the sequential multiplier discussed in Section 11.6.

12.20 Repeat parts (b), (c) and (d) of Problem 12.19, but use two adders to accelerate the operation.

12.21 Repeat parts (b), (c) and (d) of Problem 12.19, but use three adders to accelerate the operation.