CHAPTER 5

Register-Transfer Level (RTL) Design

5.1 HIGH-LEVEL STATE MACHINE (HLSM) BEHAVIOR

Register-transfer level (RTL) design involves describing the behavior of a design as the transfers that occur between registers every clock cycle. One method for such description uses a high-level state machine (HLSM) computation model. The finite-state machine (FSM) model of Chapter 3 allows only Boolean operations and conditions in states and transitions. In contrast, the HLSM model allows arithmetic operations and conditions, such as the addition or comparison of two 8-bit binary numbers. Furthermore, the HLSM model allows explicit declaration of registers, which may be written to and read from in the HLSM's states and transitions.

Figure 5.1 shows an FSM and an HLSM model for the three-cycles-high laser timer system of Section 3.2, whose behavior is such that when a button press is detected, the system holds an output high for exactly three clock cycles. Figure 5.1(a) shows an FSM description of the system, which uses three states to hold the output high for three cycles. However, what if the system was supposed to hold its output high for 512 cycles rather than just three cycles? Creating an FSM with 512 states to hold the output high would result in an unnecessarily large description. For such a system, a high-level state machine model would be more appropriate. Figure 5.1(b) shows a description having identical behavior with the FSM description of Figure 5.1(a) but achieved using a high-level state machine model. The HLSM has only two states, and explicitly declares a 2-bit register *Cnt*. The register *Cnt* is used to count the number of cycles for which the output has been

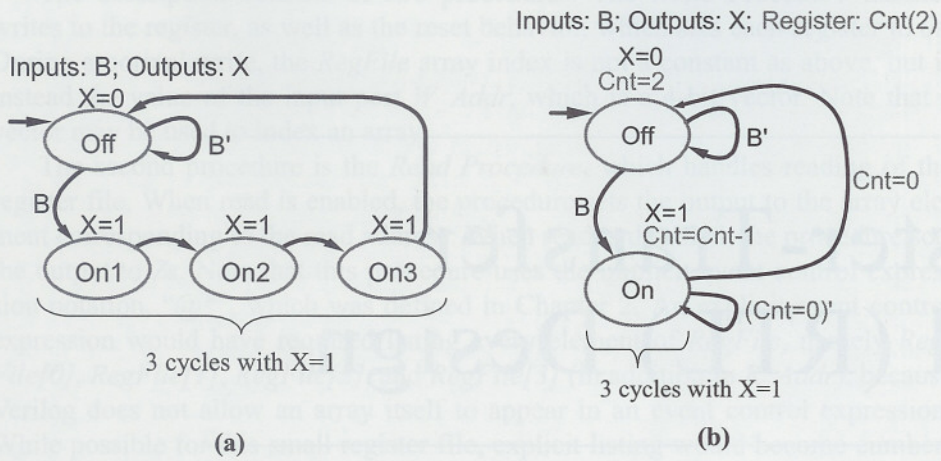


Figure 5.1 Two state machine description types: (a) FSM, (b) high-level state machine (HLSM).

held high. The first state initializes *Cnt* to 2. After a button press has been detected, the second state holds the output high while comparing *Cnt* to *0* and also decrementing *Cnt*. The net result is that the output will be held high for three clock cycles. Initializing *Cnt* to 511 (and also declaring *Cnt* to be a 9-bit register rather than just 2-bits) would result in holding the output high for 512 cycles.

Describing an HLSM in Verilog can be achieved using a straightforward approach similar to the approach for describing an FSM. Similar to the approach for an FSM, the approach for an HLSM considers the target architecture consisting of a combinational logic part and a register part, as shown in Figure 5.2. The earlier-introduced FSM register part consisted only of a state register. The HLSM register part consists of a state register, and of any explicitly declared registers. The figure shows the architecture for the laser timer example system, which has one explicitly declared register, *Cnt*.

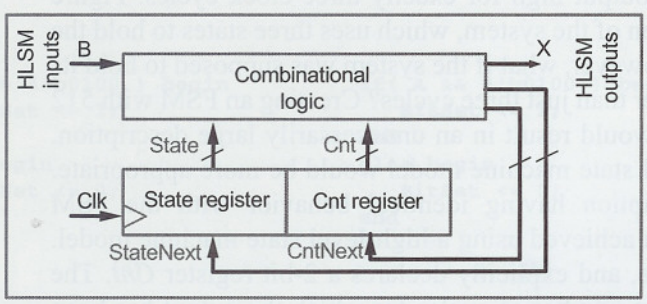


Figure 5.2 Target architecture for an HLSM, consisting of a combinational logic part, and a register part.

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);

   input B;
   output reg X;
   input Clk, Rst;

   parameter S_Off = 0,
             S_On  = 1;

   reg [0:0] State, StateNext;
   reg [1:0] Cnt, CntNext;

   // CombLogic
   always @(State, Cnt, B) begin
      ...
   end

   // Regs
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Figure 5.3 Code template for describing the laser timer's HLSM in Verilog.

Describing an HLSM in Verilog follows straightforwardly from the target architecture, as illustrated in Figure 5.3. The description consists of two procedures, one for the combinational logic and one for the registers. In addition to declaring current and next variables for the state register as for an FSM, the description also declares current and next variables for each explicitly declared register in the HLSM. For the laser timer example's one explicit register *Cnt*, the description declares two variables, *Cnt* and *CntNext*.

Figure 5.4 shows the procedure for the combinational logic part of the HLSM. The always procedure is sensitive to all the inputs of the combinational logic. Figure 5.2 shows those inputs to be the values coming from the registers, namely *State* and *Cnt*, and the external input *B*. The procedure consists of a case statement that executes the actions and transitions associated with the current state. Figure 5.1(b) shows those actions to be not just the setting of the HLSM's external outputs (e.g., "*X=0*"), but also setting of explicitly declared registers (e.g., "*Cnt=2*"). The actions in the procedure of Figure 5.4 therefore include not only setting of the external output values (e.g., "*X <= 0;*"), but also setting of the next values for explicitly defined registers (e.g., "*CntNext <= 2;*").

```
...
   reg [0:0] State, StateNext;
   reg [1:0] Cnt, CntNext;

   // CombLogic
   always @(State, Cnt, B) begin
      case (State)
         S_Off: begin
            X <= 0;
            CntNext <= 2;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
         S_On: begin
            X <= 1;
            CntNext <= Cnt - 1;
            if (Cnt == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
      endcase
   end
...
```

Note: Writes are to "next" variable, reads are from "current" variable. See target architecture to understand why.

Figure 5.4 HLSM combinational part.

Note from the architecture shown in Figure 5.2 that for an explicitly declared register, the combinational logic reads the current variable (*Cnt*) but writes the next variable (*CntNext*) for the *Cnt* register. This dichotomy explains why the action "*Cnt <= Cnt - 1*" of the HLSM in Figure 5.1(b) is described in the procedure of Figure 5.4 as "*CntNext <= Cnt - 1;*". Reading is from *Cnt*, while writing is to *CntNext*. When describing HLSMs in an HDL, care must be taken to ensure that reads are from current variables and writes are to next variables for explicitly declared registers.

The transitions in the procedure in Figure 5.4, when reading from an explicitly declared register, must read from the current variable and not the next variable, again based on the target architecture of Figure 5.2. Thus, the transition that detects *Cnt=0* appears as "*Cnt == 0;*".

Figure 5.5 shows the procedure for the register part of the HLSM. The register part actually describes two registers—the state register (involving variables *State* and *StateNext*), and the *Cnt* register (involving variables *Cnt* and *CntNext*). When the clock is rising and reset is not asserted, the procedure updates each current variable with the corresponding next variable. If instead reset were asserted, the procedure sets each current variable to an initial value. Note that the procedure resets *Cnt* to an initial value (*0*), even though such reset behavior is not strictly

```
...
   // Regs
  always @(posedge Clk) begin
     if (Rst == 1 ) begin
        State <= S_Off;
        Cnt <= 0;
     end
     else begin
        State <= StateNext;
        Cnt <= CntNext;
     end
  end
...
```

Figure 5.5 HLSM register part.

necessary for correct functioning of the HLSM, because the HLSM will initialize *Cnt* to *2* in its first state. Such reset behavior was included to follow the modeling guidelines described in Chapter 3, where it was stated that all registers should have defined reset behavior.

Figure 5.6 provides simulation waveforms generated when using the laser timer example testbench introduced in Chapter 3. Note first that the three-cycle-high behavior is identical to the FSM behavior from Chapter 3. The waveforms show two internal variables, *Cnt* and *State*. Note how the system enters state *S_On* on the first rising clock after *B* becomes *1*, causing *Cnt* to be initialized to *2*. *Cnt* is then decremented on each rising clock while in state *S_On*. After *Cnt* reaches *0*, *State* changes to *S_Off*. Note that *Cnt* also was decremented at that time, causing *Cnt* to wrap around from *0* to *3* (*"00" - 1 = "11"*), but that value of *3* was never used, because state *S_Off* sets *Cnt* to *2* again.

Examining the reset behavior of the system is useful. At the beginning of simulation, *Cnt* is unknown. At the first rising clock, *Cnt* is reset to *0* by the description's explicit reset behavior. At the next rising clock, *Cnt* is set to *2* by state *S_Off*.

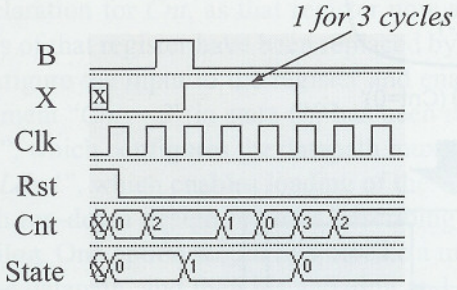


Figure 5.6 HLSM simulation results.

5.2 TOP-DOWN DESIGN— HLSM TO CONTROLLER AND DATAPATH

Recall from Chapters 2 and 3 that top-down design involves first capturing and simulating the behavior of a system, and then transforming that behavior to structure and simulating again. Top-down design divides the design problem into two steps. The first step is to get the behavior right (freed from the complexity of designing structure), and the second step is to derive a structural implementation for that behavior. Dividing into two steps can make the design process proceed more smoothly than trying to directly capture structure. The two-step approach also enables the use of automated tools that automatically convert the behavior to structure.

At the register-transfer level, top-down design involves converting a high-level state machine (HLSM) to a structural design consisting of a controller and a datapath, as shown in Figure 5.7. The datapath carries out the arithmetic operations involved in the HLSM's actions and conditions. The controller sequences those operations in the datapath. The controller itself will be an implementation of an FSM.

The first step in converting an HLSM to a controller and datapath is typically to design a datapath that is capable of implementing all the arithmetic operations of the HLSM. Figure 5.8 shows a datapath capable of implementing the arithmetic operations of the laser timer HLSM.

The datapath includes a register *Cnt*, a decrementer to compute *Cnt-1*, and a comparator to detect when *Cnt* is *0*. Those components are connected to enable the operations needed by the HLSM, with a mux in front of the *Cnt* register to account for the fact that *Cnt* can be loaded from two different sources. The datap-

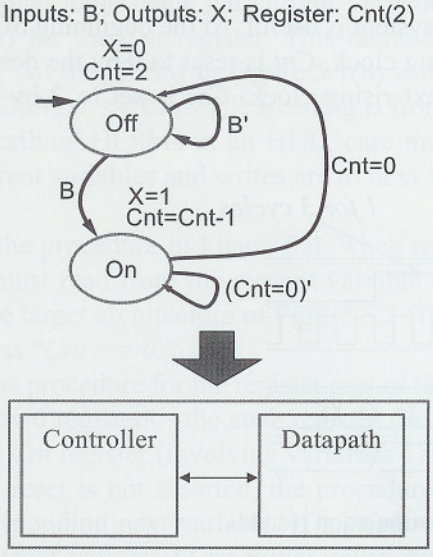


Figure 5.7 RTL top-down design converts an HLSM to a controller and datapath.

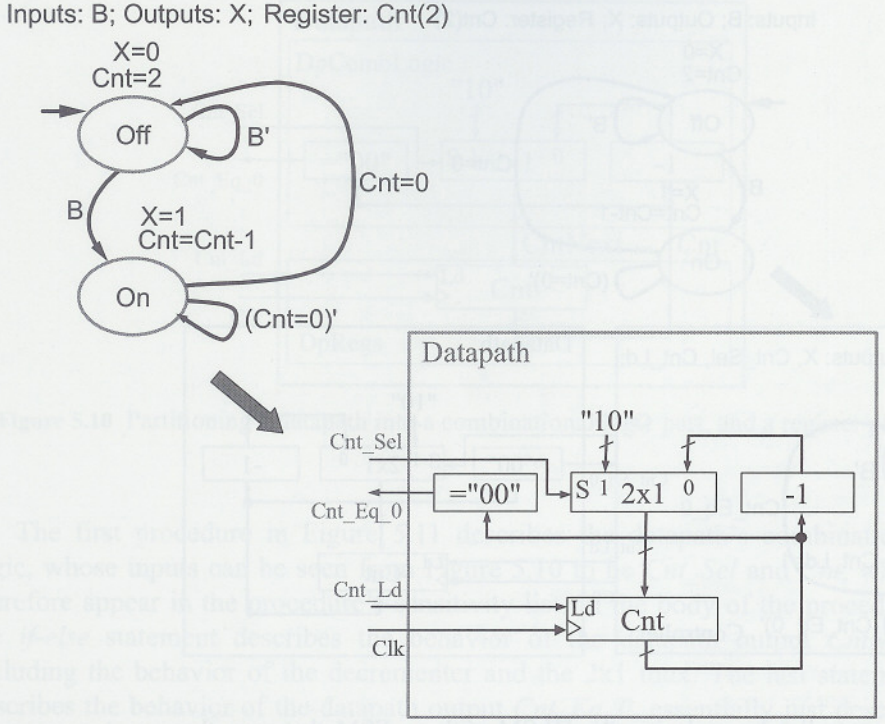


Figure 5.8 Deriving a datapath for the arithmetic operations for the laser timer HLSM.

ath provides clear names for the input and output control signals of the datapath (*Cnt_Sel*, *Cnt_Eq_0*, and *Cnt_Ld*).

After creating a datapath, the next step is to derive a controller by replacing the HLSM with an FSM having the same state and transition structure, but replacing arithmetic operations by Boolean operations that use the datapath input and output control signals to carry out the desired arithmetic operations inside the datapath. Such an FSM is shown in Figure 5.9. The FSM does not contain an explicit register declaration for *Cnt*, as that register now appears in the datapath. Likewise, any writes of that register have been replaced by writes to datapath control signals that configure the input to the register and enable a register load. For example, the assignment "*Cnt = 2*" in state *Off* has been replaced by the Boolean actions "*Cnt_Sel=1*", which configures the datapath mux to pass "*10*" (2) through the mux, and "*Cnt_Ld=1*", which enables loading of the *Cnt* register.

Proceeding with top-down design requires describing Figure 5.9's controller and datapath in Verilog. One option would be to create a module for the controller and a module for the datapath, and then instantiating and connecting a controller module and a datapath module in another higher-level module. However, a simpler approach describes the controller and datapath as procedures within a single module. The simpler approach will now be discussed.

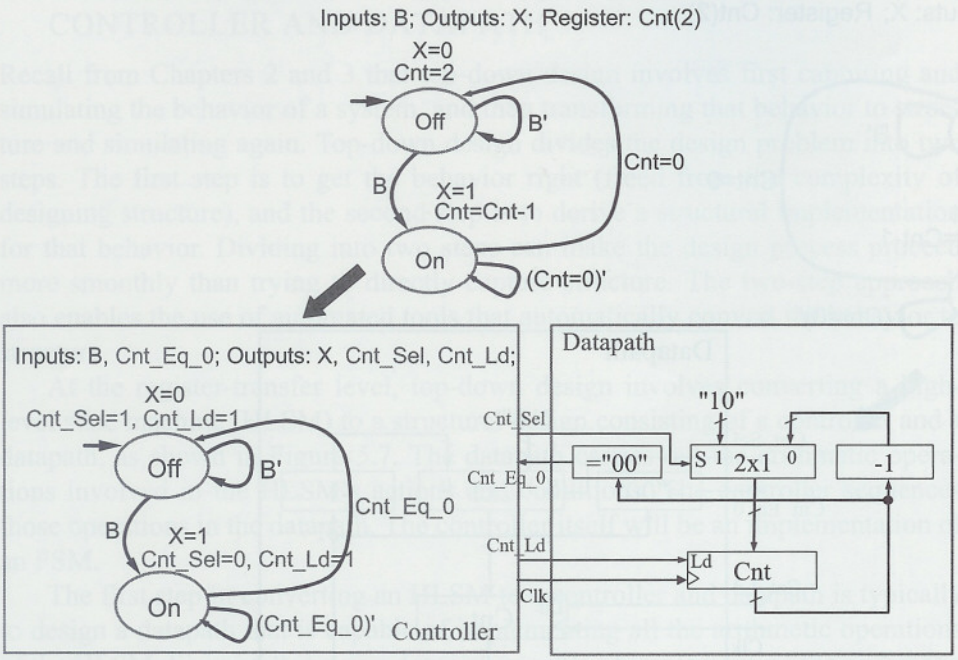


Figure 5.9 Deriving a controller by replacing the HLSM with an FSM that uses the datapath to carry out arithmetic operations.

The datapath itself could be described structurally. Such a description would instantiate four modules: a register, a comparator, a mux, and a decrementer. The description would connect those four components. Each component would require further description as its own module.

The datapath could instead be described behaviorally. Recall from Chapters 2 and 3 that behavioral descriptions are typically easier to create and to understand than structural descriptions. Of course, a behavioral description would need to be further converted to structure to ultimately achieve an implementation, meaning that top-down design must be applied in a hierarchical manner.

One approach to behaviorally describing the datapath involves partitioning the datapath into a combinational logic part and a register part, as shown in Figure 5.10. Notice the similarity of this architecture with the previous architecture in Figure 5.2, which also consists of a combinational logic part and a register part.

Describing the datapath thus proceeds in the same manner as for that previous architecture, namely using two procedures, one for any combinational logic, and one for any registers. Such a two-procedure behavioral description of the datapath is shown in Figure 5.11.

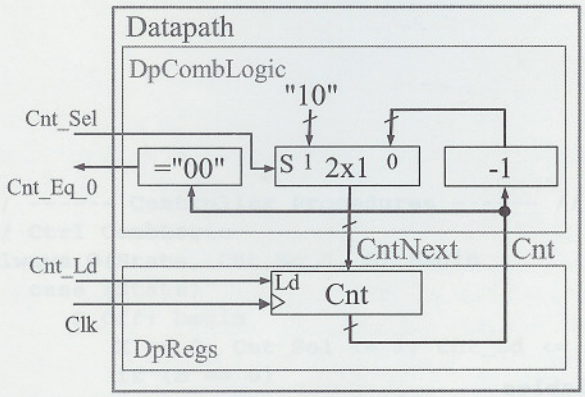


Figure 5.10 Partitioning a datapath into a combinational logic part, and a register part.

The first procedure in Figure 5.11 describes the datapath's combinational logic, whose inputs can be seen from Figure 5.10 to be *Cnt_Sel* and *Cnt*, which therefore appear in the procedure's sensitivity list. In the body of the procedure, the *if-else* statement describes the behavior of the datapath output *CntNext*, including the behavior of the decrementer and the 2x1 mux. The last statement describes the behavior of the datapath output *Cnt_Eq_0*, essentially just describing a comparator with *0*. Notice that those two statements could have appeared in opposite order but would have still described the same behavior.

The second procedure in Figure 5.11 describes the datapath's register. When the clock is rising and the register load input is asserted, the procedure updates the current variable value with the next value. Notice that reset behavior is included for the register, even though such behavior isn't strictly necessary for the correct operation of this particular datapath operation. The reset was included to follow good modeling practice, which involves always including reset behavior for every register. Also notice that only the *Clk* input appears in the procedure's sensitivity list, as the register is synchronous and thus is only updated on (rising) clock edges.

The controller's FSM would also be described using two procedures, one for the combinational part, and one for the register, as was done in Chapter 3, and as shown in Figure 5.12.

Therefore, the controller and datapath description consists of four procedures, two for the datapath (one for combinational logic, one for registers), and two for the controller (one for combinational logic, and one for registers), as shown in Figure 5.13. Those four procedures appear in a single module. The datapath procedures and controller procedures communicate through the variables *Cnt_Eq_0*, *Cnt_Sel*, and *Cnt_Ld*. The other variables are declared either for exclusive use of the datapath (*Cnt*, *CntNext*) or the controller (*State*, *StateNext*).

```
...
   // Shared variables
   reg Cnt_Eq_0, Cnt_Sel, Cnt_Ld;
   // Controller variables
   reg [0:0] State, StateNext;
   // Datapath variables
   reg [1:0] Cnt, CntNext;

   // ------ Datapath Procedures ------ //
   // DP CombLogic
   always @(Cnt_Sel, Cnt) begin
      if (Cnt_Sel==1)
         CntNext <= 2;
      else
         CntNext <= Cnt - 1;

      Cnt_Eq_0 <= (Cnt==0)?1:0;
   end

   // DP Regs
   always @(posedge Clk) begin
      if (Rst == 1 )
         Cnt <= 0;
      else if (Cnt_Ld==1)
         Cnt <= CntNext;
   end
...
```

Figure 5.11 Two-procedure description of a datapath.

```
...
   // ------ Controller Procedures ------ //
   // Ctrl CombLogic
   always @(State, Cnt_Eq_0, B) begin
      case (State)
         S_Off: begin
            X <= 0; Cnt_Sel <= 1; Cnt_Ld <= 1;
            if (B == 0)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
         S_On: begin
            X <= 1; Cnt_Sel <= 0; Cnt_Ld <= 1;
            if (Cnt_Eq_0 == 1)
               StateNext <= S_Off;
            else
               StateNext <= S_On;
         end
      endcase
   end

   // Ctrl Regs
   always @(posedge Clk) begin
      if (Rst == 1 ) begin
         State <= S_Off;
      end
      else begin
         State <= StateNext;
      end
   end
...
```

Figure 5.12 Two-procedure controller description.

```
...
module LaserTimer(B, X, Clk, Rst);
   ...

   parameter S_Off = 0,
             S_On  = 1;

   // Shared variables
   reg Cnt_Eq_0, Cnt_Sel, Cnt_Ld;
   // Controller variables
   reg [0:0] State, StateNext;
   // Datapath variables
   reg [1:0] Cnt, CntNext;

   // ------ Datapath Procedures ------ //
   // DP CombLogic
   always @(Cnt_Sel, Cnt) begin
      ...
   end

   // DP Regs
   always @(posedge Clk) begin
      ...
   end

   // ------ Controller Procedures ------ //
   // Ctrl CombLogic
   always @(State, Cnt_Eq_0, B) begin
      ...
   end

   // Ctrl Regs
   always @(posedge Clk) begin
      ...
   end
endmodule
```

Figure 5.13 Controller and datapath descriptions, consisting of two procedures for the datapath, and two for the controller.

5.3 DESCRIBING A STATE MACHINE USING ONE PROCEDURE

Section 5.1 described how to describe a high-level state machine using two procedures, using one procedure for combinational logic, and a second procedure for registers, as summarized in Figure 5.14(a). The two-procedure description required two variables for each register, a current variable, and a next variable.

A one-procedure description of a high-level state machine is also possible, and is quite commonly used. A one-procedure description is highlighted in Figure 5.14(b). In addition to using only one procedure, the description uses only one variable per register, rather than two variables per register. A one-procedure description thus has advantages of simpler code with clear grouping of related functionality into one procedure. But, as we'll see shortly, the one-procedure description slightly changes the timing behavior.

Figure 5.15 provides a complete one-procedure description of the HLSM for the earlier laser-timer example. The procedure is sensitive only to a rising clock edge. The procedure checks whether the reset input is asserted, in which case the procedure resets all registers (the state register, and explicitly declared registers in the HLSM). If the reset is not asserted, then the procedure executes the HLSM's state actions based on the state variable's value, and sets the next state variable value according to the HLSM transitions.

```
...
module LaserTimer(B, X, Clk, Rst);
...
   reg [0:0] State, StateNext;
   reg [1:0] Cnt, CntNext;

   // CombLogic
   always @(State, Cnt, B) begin
      ...
   end

   // Regs
   always @(posedge Clk) begin
      ...
   end
endmodule
```

(a)

```
...
module LaserTimer(B, X, Clk, Rst);
...
   reg [0:0] State;
   reg [1:0] Cnt;

   always @(posedge Clk) begin
   end
endmodule
```

(b)

Figure 5.14 Alternative approaches for describing a high-level state machine: (a) two-procedure description, (b) one-procedure description.

```
...
module LaserTimer(B, X, Clk, Rst);
   ...
   parameter S_Off = 0,
             S_On  = 1;

   reg [0:0] State;
   reg [1:0] Cnt;

   always @(posedge Clk) begin
      if (Rst == 1 ) begin
         State <= S_Off;
         Cnt <= 0;
      end
      else begin
         case (State)
            S_Off: begin
               X <= 0;
               Cnt <= 2;
               if (B == 0)
                  State <= S_Off;
               else
                  State <= S_On;
            end
            S_On: begin
               X <= 1;
               Cnt <= Cnt - 1;
               if (Cnt == 0)
                  State <= S_Off;
               else
                  State <= S_On;
            end
         endcase
      end
   end
endmodule
```

Figure 5.15 One-procedure description of a laser timer HLSM.

Compared to the two-procedure description shown in Figure 5.4 and Figure 5.5, the one-procedure description in Figure 5.15 is clearly simpler and easier to read. Why then was a two-procedure description introduced? The main reason relates to timing. The one-procedure description of Figure 5.15 does not exactly match the timing of the HLSM in Figure 5.1(b). In particular, the intended HLSM's behavior is such that changes to the HLSM inputs (i.e., to input *B*) would be seen by the HLSM immediately, and thus a change could influence the next state on the rising clock edge immediately following the change. Such behavior is accurately reflected by the two-procedure description, as shown in the waveforms

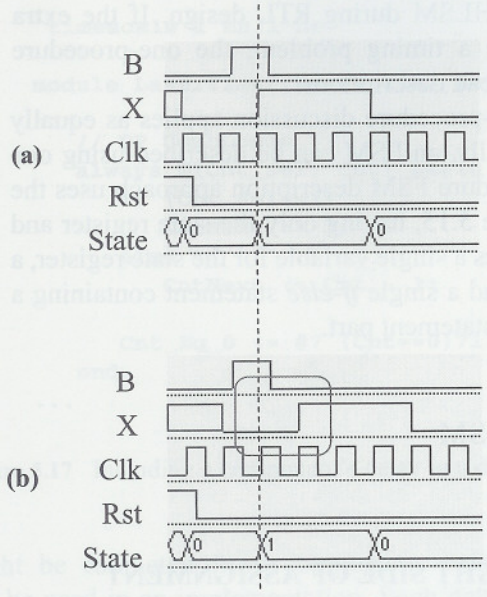


Figure 5.16 Timing differences between different descriptions: (a) In the two-procedure description, a change in *B* appearing sufficiently before a rising clock edge sets up the next state according to the new value of *B*, (b) In the one-procedure synchronous description, *B*'s value is only checked on rising clock edges, and thus a change in *B* is not noticed until the next rising edge, meaning the new value of *B* doesn't impact the next state until *two* rising clock edges after *B* changes.

of Figure 5.16(a), because the combinational logic procedure is independent of the clock input. In contrast, the one-procedure description of Figure 5.15 is synchronous, meaning the procedure only checks the value of *B* on a rising clock edge, and thus a change cannot influence the next state until *two* rising clock edges after the change, as shown in Figure 5.16(b). The one-procedure description thus introduces some delay into the system's behavior.

Introducing such delay is not necessarily a bad thing. In many systems, the extra cycle delay may represent an insignificant change. For example, in Figure 5.16, both waveforms ultimately represent acceptable behavior of the three-cycles-high laser timer system—the output stays high for exactly three clock cycles after a button press. For that system, it is not important if the three cycles are shifted in time by one cycle; what's important is that the output stays high for exactly three cycles. The insignificance of delays in detecting input changes is further illustrated by the fact that external inputs are typically fed through a series of flip-flops in order to isolate circuits from flip-flop metastability issues—that series of flip-flops introduces several cycles of delay itself.

Therefore, a one-procedure synchronous description is actually a reasonable and popular approach for describing a HLSM during RTL design. If the extra cycle for reading inputs does not pose a timing problem, the one-procedure description yields simpler and easier-to-read descriptions.

The same two-procedure versus one-procedure discussion applies as equally to FSMs as it does to HLSMs. Specifically, an FSM can be described using one procedure. The most common one-procedure FSM description approach uses the same structure as the procedure in Figure 5.15, having only the state register and no additional registers. The procedure uses a single variable for the state register, a procedure sensitive to the clock input, and a single *if-else* statement containing a reset part and an actions/transitions case statement part.

5.4 IMPROVING TIMING REALISM

[SIMUL] DELAY CONTROL ON RIGHT SIDE OF ASSIGNMENT STATEMENTS

Real components do not compute their outputs instantly after the components' inputs change. Instead, real components have delay—after inputs change, the correct outputs do not appear until some time later. For example, suppose the comparator in the datapath of Figure 5.9 has a delay of 7 ns. In order to obtain a more accurate RTL simulation of the controller and datapath in that figure, a description could be extended to include such delays. Figure 5.17 shows how the description of Figure 5.11 could be extended to include a 7 ns delay for the comparator component, by using delay control.

Delay control was introduced in Chapter 2 for delaying the execution of a statement, achieved by prepending the delay control to a statement, as in the statement: "*#10 Y <= 0;*". However, delay control can also be inserted on the right side of an assignment statement, such as: "*Y <= #10 0;*". The prepended form delays execution of the statement that follows by 10 time units. In contrast, in the right side form, the statement is not delayed but instead executes immediately. Upon executing, however, the update of *Y* will be scheduled to occur 10 time units in the future.

In Figure 5.17, the assignment to *Cnt_Eq_0*, which models the datapath's comparator, have been extended with delay control indicating a 7 ns delay. To more fully model the datapath component delays, delay controls would also be added to the two *CntNext* assignments, modeling the delay of the mux and the decrementer.

```
`timescale 1 ns/1 ns

module LaserTimer(B, X, Clk, Rst);
...
   // DP CombLogic
   always @(Cnt_Sel, Cnt) begin
      if (Cnt_Sel==1)
         CntNext <= 2;
      else
         CntNext <= Cnt - 1;

      Cnt_Eq_0 <= #7 (Cnt==0)?1:0;
   end
...
```

Figure 5.17 Including a component's delay in a description.

Delays might be estimated from components existing in the component library that will be used in an implementation. Such delays are quite commonly included for low-level components like gates, muxes, registers, and so on. Including such delays in a description results in better timing accuracy during simulation, and can even detect timing problems. For example, if the clock period were shorter than the longest register-to-register delay in the datapath, the designer would notice that the datapath with delays computes incorrect results during simulation.

A tool synthesizing a circuit from a description having delays, like the description in Figure 5.17, would simply ignore the delay controls. Those controls serve to reflect the predicted timing behavior of the eventual implementation, and do *not* serve to guide the synthesis tool in creating that implementation.

Figure 5.18 illustrates the effect of including delays in a description. Figure 5.18(a) shows the simulation waveforms for the system when the comparator does not have any delay. That figure shows several internal variables, *Cnt*, *Cnt_Eq_0* (the comparator output), and *State*. Figure 5.18(b) shows the simulation waveforms for the description having the 7 ns comparator delay. Notice that the comparator's output, *Cnt_Eq_0*, is slightly shifted to the right, due to the delay.

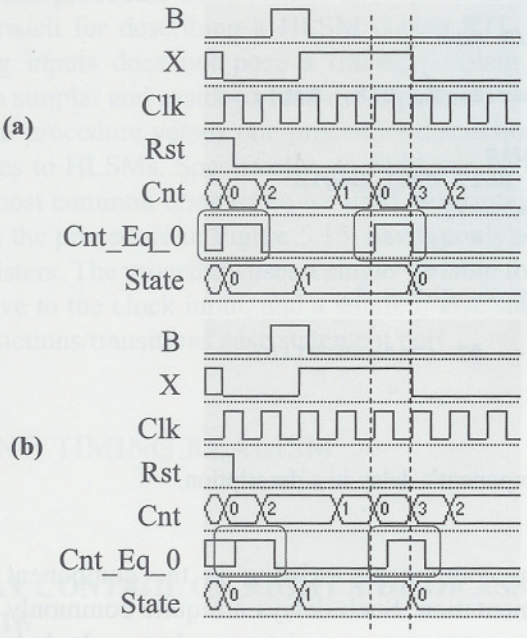
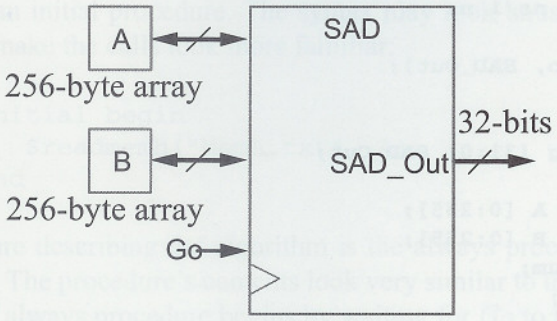


Figure 5.18 Simulation results: (a) without comparator delay, (b) with 7 ns delay.

5.5 ALGORITHMIC-LEVEL BEHAVIOR

In the early stages of design, a designer may want to describe a system's behavior at an even higher level of abstraction than the register-transfer level. Sometimes, the behavior is naturally first described as an algorithm. For example, consider a system that computes the sum of absolute differences (SAD) of the corresponding elements of two 256-element arrays. Such a computation is useful, for example, in video compression to determine the difference between two video frames. A SAD system block diagram appears in Figure 5.19(a). The system's algorithmic-level behavior is shown in Figure 5.19(b). The algorithm, which is written in pseudo-code and not in any particular language, indicates that the system is activated when the input *Go* becomes *1*. The algorithm then generates array indices from 0 to 255, one at a time. For each index, the algorithm computes the absolute value of the difference of the two array elements for that index, and adds that value to a running sum variable. The algorithm then writes the final sum to the system's output. Notice how simple and clear an algorithm-level description can be. A designer may wish to verify the correctness of the algorithm, before proceeding to design the system at the RT level. Some tools (known as behavioral synthesis or high-level synthesis tools) have evolved to attempt to synthesize algorithmic-level

(a)



(b)

```
Wait until Go equals 1
Sum = 0
for I in 0 to 255 loop
    Sum = Sum + |A(I) - B(I)|
end loop
SAD_Out = Sum
```

Figure 5.19 Sum of absolute differences: (a) block diagram, (b) algorithm.

descriptions to circuits automatically, but most HDL users use algorithmic-level descriptions solely for early simulation purposes.

Figure 5.20 shows an algorithmic-level description of the SAD system's behavior. Although the two 256-element arrays *A* and *B* are external to the SAD system itself as shown in Figure 5.19(a), the algorithmic-level description initially declares those two arrays as part of the SAD module. Although not necessary, such declaration makes the initial description easier to create.

The description contains an always procedure having the main algorithm behavior. However, before that procedure, a user-defined function is declared. A ***user-defined function*** is typically intended to compute a value that will be used in an expression. The function defined in Figure 5.20 will be used to compute an absolute value, and is needed because Verilog does not contain a built-in absolute value operator. The first line, "*function integer ABS;*" defined the function's name to be *ABS*, and defines the function to return data of type integer. The second line, "*input integer IntVal;*", defines the function as having one input argument named *IntVal* of type integer. The contents of the function consist of a single statement. More procedural statements would be allowed in the function, as long as they didn't contain any time-controlling statements (delay control or event control) or any calls to tasks. The function's single statement assigns the return value of the function to be a positive version of the input argument. After the function *ABS* has been define, it can then be called by later parts of the description.

```
`timescale 1 ns/1 ns

module SAD(Go, SAD_Out);

   input Go;
   output reg [31:0] SAD_Out;

   reg [7:0] A [0:255];
   reg [7:0] B [0:255];
   integer Sum;
   integer I;

   function integer ABS;
      input integer IntVal;
      begin
         ABS = (IntVal>=0)?IntVal:-IntVal;
      end
   endfunction

   // Initialize Arrays
   initial $readmemh("MemA.txt", A);
   initial $readmemh("MemB.txt", B);

   always begin
      if (!(Go==1)) begin
         @(Go==1);
      end
      Sum = 0;
      for (I=0; I<=255; I=I+1) begin
         Sum = Sum + ABS(A[I] - B[I]);
      end
      #50;
      SAD_Out <= Sum;
   end
endmodule
```

Figure 5.20 Sum of absolute differences algorithmic-level description.

The description also declares memories *A* and *B* as 256-element arrays of bytes. Note that the description initializes those arrays using the built-in system task *$readmemh*. ***$readmemh*** is a system task intended to initialize memories by reading a file of hex numbers (with the file name being the function's first argument) and placing each number into successive elements of an array (with the array name being the second argument). The number of numbers in the file should match the number of elements in the array. The hex numbers in the file should not have a size or base format, and should be separated by white spaces, e.g., 00 FF A1 04 etc. Another system function, ***$readmemb***, instead reads a file of binary

numbers (rather than hex numbers) into an array. Note that functions are called as the sole statement of an initial procedure. The syntax may look unusual; using a begin-end block may make the calls look more familiar:

```
initial begin
   $readmemh("MemA.txt", A);
end
```

The main procedure describing the algorithm is the always procedure at the bottom of Figure 5.20. The procedure's contents look very similar to the algorithm in Figure 5.19(b). The always procedure begins by waiting for *Go* to become *1*. It does so by checking if *Go* is already *1*, and if not, using the event control "*@(Go==1);*" Previous event controls involved a single event, such as "*@(X)*" or "*@(posedge Clk)*", or involved a list of events, such as "*@(X,Y)*". In Figure 5.20, however, the event control uses the expression "*@(Go==1)*". That event control does not merely detect a change in *Go*, but detects a change in *Go* such that *Go* becomes *1*. The word "change" is critical in the previous sentence. If *Go* is already *1* when the event control statement is reached during execution, the procedure still suspends at that event control. The procedure stays suspended at that statement until *Go* changes to *0* and then changes back to *1*. This behavior of an event control is somewhat counterintuitive, as many designers make the mistake of believing that if *Go* was *1* when reaching the statement, execution will simply proceed to the next statement without the procedure suspending. The description therefore uses an *if* statement to achieve the desired behavior of the procedure proceeding to compute the SAD if *Go* is already *1*.

The delay control "#50;" at the end of the procedure creates some delay (albeit a rather short one) between the time that Go becomes 1 and the time that the computed SAD value appears at the output. Including a delay at the end of the procedure also prevents an infinite simulation loop in which the procedure repeatedly executes without ever suspending if Go were always kept at 1.

Figure 5.21 provides a simple test vector procedure for the algorithm-level SAD description. The procedure pulses *Go_s*, waits for some time, and then checks that the computed SAD equals 4 (we happened to define the memory contents such that exactly four elements differed in A and B by one each). Simulation waveforms are shown in Figure 5.22. Note that the SAD output is initially unknown, due to the output not having been explicitly set to some value when the SAD procedure first executed. A better description would set the output to some value, likely during a reset.

```
`timescale 1 ns/1 ns

module Testbench();

   reg Go_s;
   wire [31:0] SAD_Out_s;

   SAD CompToTest(Go_s, SAD_Out_s);

   // Vector Procedure
   initial begin
      Go_s <= 0;
      #10 Go_s <= 1;
      #10 Go_s <= 0;
      #60 if (SAD_Out_s != 4) begin
         $display("SAD failed -- should equal 4");
      end
   end
endmodule
```

Figure 5.21 Simple testbench vector procedure for the SAD algorithmic-level description.

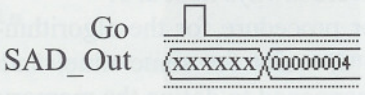


Figure 5.22 Waveforms for the SAD algorithmic-level description.

5.6 TOP-DOWN DESIGN—CONVERTING ALGORITHMIC-LEVEL BEHAVIOR TO RTL

Once satisfied that an algorithm is correct, a designer may wish to proceed to convert the algorithm-level behavior to an RTL description, as a means of moving towards an implementation of the system. The algorithm of Figure 5.20 can be recaptured as the HLSM shown in Figure 5.23. The HLSM can then be described as shown in Figure 5.24.

The HLSM can be tested using a testbench similar to that in Figure 5.21, except that the testbench should wait longer than just 60 ns for the SAD output to appear. By counting the number of HLSM states that must be visited to compute the SAD, one can determine a waiting time of (256*2+3) * (20 ns), where 20 ns is the clock period. Figure 5.25 shows a simple testbench for the HLSM.

Figure 5.26 provides the waveforms resulting from simulating the testbench, showing several internal variables to better demonstrate the HLSM's behavior during simulation.

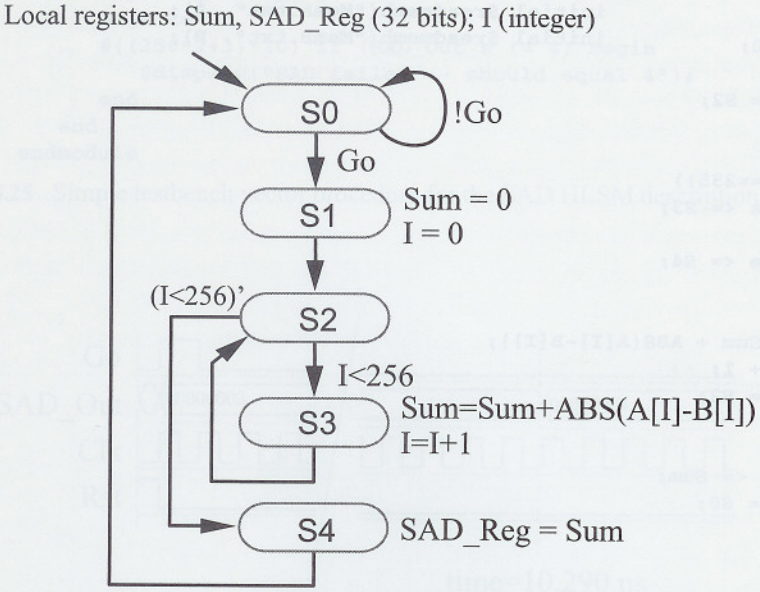


Figure 5.23 HLSM for the SAD system.

```
`timescale 1 ns/1 ns

module SAD(Go, SAD_Out, Clk, Rst);
   input Go;
   output [31:0] SAD_Out;
   input Clk, Rst;

   parameter S0 = 0, S1 = 1,
             S2 = 2, S3 = 3,
             S4 = 4;

   reg [7:0] A [0:255];
   reg [7:0] B [0:255];
   reg [2:0] State;
   integer Sum, SAD_Reg;
   integer I;

   function integer ABS;
      input integer IntVal;
      begin
         ABS = (IntVal>=0)?IntVal:-IntVal;
      end
   endfunction

   // Initialize Arrays
   initial $readmemh("MemA.txt", A);
   initial $readmemh("MemB.txt", B);

// High-level state machine
always @(posedge Clk) begin
   if (Rst==1) begin
      State <= S0;
      Sum <= 0;
      SAD_Reg <= 0;
      I <= 0;
   end
   else begin
      case (State)
         S0: begin
            if (Go==1)
               State <= S1;
            else
               State <= S0;
         end
         S1: begin
            Sum <= 0;
            I <= 0;
            State <= S2;
         end
         S2: begin
            if (!(I==255))
               State <= S3;
            else
               State <= S4;
         end
         S3: begin
            Sum <= Sum + ABS(A[I]-B[I]);
            I <= I + 1;
            State <= S2;
         end
         S4: begin
            SAD_Reg <= Sum;
            State <= S0;
         end
      endcase
   end
end
endmodule
```

Figure 5.24 Description of the HLSM for the SAD system.

```
`timescale 1 ns/1 ns

module Testbench();

   reg Go_s;
   reg Clk_s, Rst_s;
   wire [31:0] SAD_Out_s;

   SAD CompToTest(Go_s, SAD_Out_s, Clk_s, Rst_s);

   // Clock Procedure
   always begin
      Clk_s <= 0;
      #10;
      Clk_s <= 1;
      #10;
   end   // Note: Procedure repeats

   // Vector Procedure
   initial begin
      Rst_s <= 1;
      Go_s <= 0;
      @(posedge Clk_s);
      Rst_s <= 0;
      Go_s <= 1;
      @(posedge Clk_s);
      Go_s <= 0;
      #((256*2+3)*20) if (SAD_Out_s != 4) begin
         $display("SAD failed -- should equal 4");
      end
   end
endmodule
```

Figure 5.25 Simple testbench vector procedure for the SAD HLSM description.

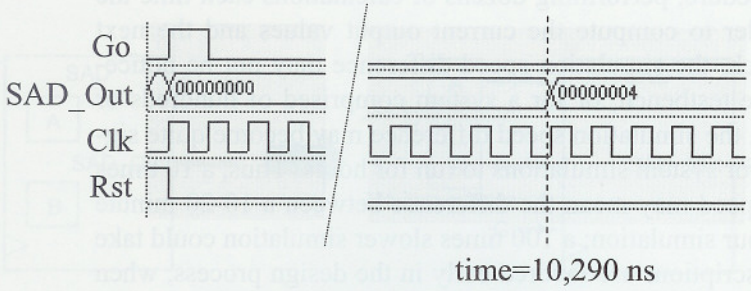


Figure 5.26 Waveforms for HLSM of the SAD system.

[SYNTH] AUTOMATED SYNTHESIS FROM THE ALGORITHMIC-LEVEL

Several commercial tools, known as ***behavioral synthesis*** or ***high-level synthesis*** tools, have been introduced in the 1990s and 2000s intending to automatically synthesize algorithmic-code to RTL code or even directly to circuits. Such tools have not yet been adopted as widely as RTL synthesis tools. Each tool imposes strong requirements and restrictions on the algorithmic-level code that may be synthesized, and those requirements and restrictions vary tremendously across tools. The quality of RTL code or circuits created from such tools also varies tremendously. Nevertheless, behavioral synthesis tools represent an interesting direction and may prove increasingly useful in coming years.

[SIMUL] SIMULATION SPEED

Higher-level descriptions not only have the advantage of requiring less time to create them and thus of being more suitable for initial system behavior definition, but also have the advantage of faster simulation. For example, the algorithmic-level description of the sum of absolute differences system may simulate faster than an HLSM description of that system, which in turn may simulate faster than a lower-level description like a gate-level description. When the system's *Go* input becomes *1*, the algorithmic-level description resumes a procedure that then executes a *for* loop that involves only a few thousand calculations in total to compute the output result. In contrast, the HLSM description would require tens of thousands of calculations by the simulator, which must suspend and resume the HLSM procedure nearly one thousand times in order to simulate the clock-controlled state machine of that procedure, performing dozens of calculations each time the procedure resumes in order to compute the current output values and the next state. For a small testbench, the simulation speed difference may not be noticeable. However, for a large testbench, or for a system comprised of hundreds or thousands of sub-systems, the simulation speed difference may become quite significant. It is not unusual for system simulations to run for hours. Thus, a 10 times difference in simulation speed may mean the difference between a 10-20 minute simulation versus a 2-3 hour simulation; a 100 times slower simulation could take days. Thus, high-level descriptions are favored early in the design process, when system behavior is being defined and refined. Low-level descriptions are necessary to achieve an implementation. High-level descriptions are also useful when integrating components in a large system, to see if those components interact properly from a behavioral perspective—the fast simulation speed allows for testing of a large variety of component interaction scenarios. Lower-level descrip-

tions would be more suitable to verify detailed timing correctness of such large systems, but their slower simulation speed allows for only relatively few scenarios to be examined. For example, a system consisting of several microprocessor components described at a high level might be able to simulate minutes of microprocessor execution in a few hours, but might only be able to simulate a few seconds of microprocessor execution if described at a low level.

We previously showed how state machines could be modeled using two procedures where one procedure was combinational, or using one procedure where that procedure was sensitive only to a clock input. The latter will simulate much faster than the former, due to fewer procedure resumes and suspends. The difference may not be noticeable for small systems, but if a system contains hundreds or thousands of state machines, the difference can become quite significant.

5.7 MEMORY

Desired storage may initially be described merely using variables declared within a system's module, as was done in the SAD example of the previous section and as illustrated in Figure 5.27(a). However, refining the description towards an implementation may mean creating a description with that storage described as a separate memory component, as in Figure 5.27(b).

Describing memory separately begins by creating a new module representing a memory. A description for a simple read-only memory (ROM) appears in Figure 5.28. The memory description has an input port for the address, and an output port for the data. In this case, the bitwidth of those ports are the same, but they obviously can be different depending on the memory size and width. The description declares an array named *Memory* to store the memory's contents. The description uses a continuous assignment statement to always output the *Memory* data element corresponding to the current address input value.

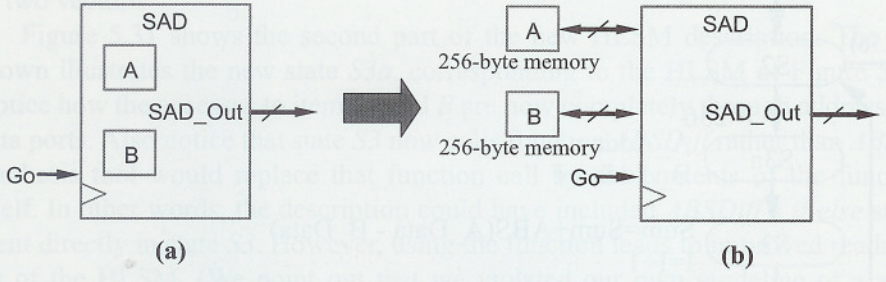


Figure 5.27 A more accurate system description may create memory as a separate component.

```
`timescale 1 ns/1 ns

module SADMem(Addr, Data);

   input [7:0] Addr;
   output [7:0] Data;

   reg [7:0] Memory [0:255];

   assign Data = Memory[Addr];
endmodule
```

Figure 5.28 Description of a simple read-only memory.

Figure 5.29 shows the HLSM for the SAD example, modified to access the external memory components. Rather than simply access *A* and *B* values, the HLSM must now set its address outputs *A_Addr* and *B_Addr*, and then use the returned data inputs *A_Data* and *B_Data*. Furthermore, because the HLSM is fully synchronous due to modeling it using a single procedure sensitive only to the clock input, the HLSM requires an extra state, *S3a*. Although the *A* and *B* memories will be combinational components, the extra state is necessary because the *A_Data* and *B_Data* inputs will only be sampled on clock edges due to the fully synchronous HLSM model being used.

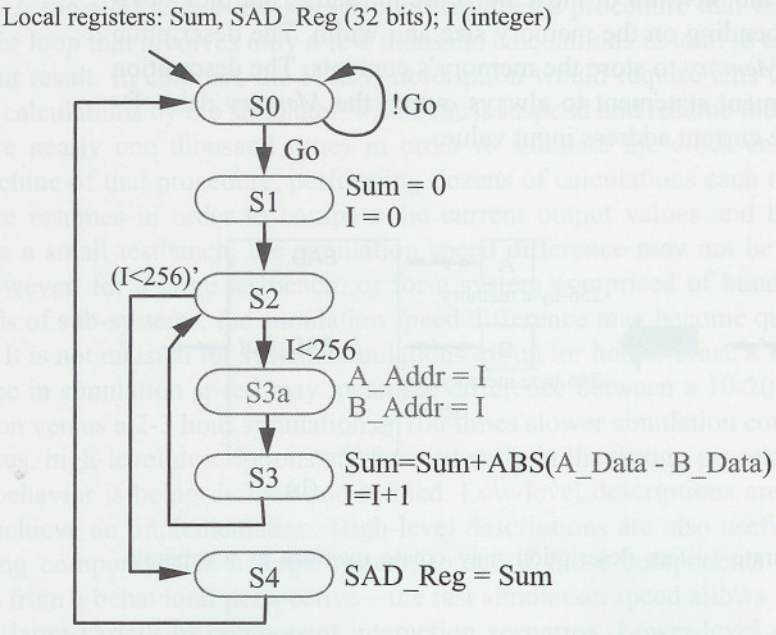


Figure 5.29 SAD HLSM modified to access the external memory components.

```
`timescale 1 ns/1 ns

module SAD(Go, A_Addr, A_Data,
           B_Addr, B_Data,
           SAD_Out, Clk, Rst);

   input Go;
   input [7:0] A_Data, B_Data;
   output reg [7:0] A_Addr, B_Addr;
   output [31:0] SAD_Out;
   input Clk, Rst;

   parameter S0 = 0, S1 = 1,
             S2 = 2, S3 = 3, S3a = 4,
             S4 = 5;

   reg [2:0] State;
   reg [31:0] Sum, SAD_Reg;
   integer I;

   function [7:0] ABSDiff;
      input [7:0] A;
      input [7:0] B;
      begin
         if (A>B) ABSDiff = A - B;
         else ABSDiff = B - A;
      end
   endfunction
```

Figure 5.30 SAD HLSM description with separate memory (part 1).

Figure 5.30 shows the first part of a description of the new SAD HLSM that uses memory components. The module declaration now includes the ports for interfacing with the external memory components. Furthermore, the description now defines a function *ABSDiff* that computes the absolute value of the difference of two vectors.

Figure 5.31 shows the second part of the new HLSM description. The part shown illustrates the new state *S3a*, corresponding to the HLSM of Figure 5.29. Notice how the accesses to items *A* and *B* are now completely through address and data ports. Also notice that state *S3* now calls function *ABSDiff* rather than *ABS*. A synthesis tool would replace that function call by the contents of the function itself. In other words, the description could have included *ABSDiff*'s *if-else* statement directly in state *S3*. However, using the function leads to improved readability of the HLSM. (We point out that we violated our own guideline of always using a begin-end block for an *if* statement. We did so merely so that the code figure would fit in this textbook).

```
always @(posedge Clk) begin
    if (Rst==1) begin
       A_Addr <= 0;
       B_Addr <= 0;
       State <= S0;
       Sum <= 0;
       SAD_Reg <= 0;
       I <= 0;
    end
    else begin
       case (State)
          S0: begin
             if (Go==1)
                State <= S1;
             else
                State <= S0;
          end
          S1: begin
             Sum <= 0;
             I <= 0;
             State <= S2;
          end
          S2: begin
             if (!(I==255))
                State <= S3a;
             else
                State <= S4;
          end
          S3a: begin
             A_Addr <= I;
             B_Addr <= I;
             State <= S3;
          end
          S3: begin
             Sum <= Sum +
              ABSDiff(A_Data, B_Data);
             I <= I + 1;
             State <= S2;
          end
          S4: begin
             SAD_Reg <= Sum;
             State <= S0;
          end
       endcase
    end
 end

 assign SAD_Out = SAD_Reg;

endmodule
```

Figure 5.31 SAD HLSM description with separate memory (part 2).

```
module Testbench();
   reg Go_s;
   wire [7:0] A_Addr_s, B_Addr_s;
   wire [7:0] A_Data_s, B_Data_s;
   reg Clk_s, Rst_s;
   wire [31:0] SAD_Out_s;

   parameter ClkPeriod = 20;

   SAD CompToTest(Go_s, A_Addr_s, A_Data_s,
                  B_Addr_s, B_Data_s,
                  SAD_Out_s, Clk_s, Rst_s);
   SADMem SADMemA(A_Addr_s, A_Data_s);
   SADMem SADMemB(B_Addr_s, B_Data_s);

   // Clock Procedure
   always begin
      Clk_s <= 0; #(ClkPeriod/2);
      Clk_s <= 1; #(ClkPeriod/2);
   end

   // Initialize Arrays
   initial $readmemh("MemA.txt", SADMemA.Memory);
   initial $readmemh("MemB.txt", SADMemB.Memory);

   // Vector Procedure
   initial begin
      ... // Reset behavior not shown
      Go_s <= 1;
      @(posedge Clk_s);
      Go_s <= 0;
      #((256*3+3)*ClkPeriod) if (SAD_Out_s != 4) begin
         $display("SAD failed -- should equal 4");
      end
   end
endmodule
```

Figure 5.32 SAD HLSM testbench with memory components.

Figure 5.32 shows a testbench for the SAD system with separate memory instances. The testbench declares the variables and nets needed for connecting module instances together, and then instantiates and connects the instances. Note that a testbench can instantiate multiple modules for testing, rather than instantiating just one module. To the extent possible, designers should always also test modules separately, before testing them together in a single testbench.

The testbench then defines the clock, memory initialization, and vector procedures. The memory initialization procedures again use the *$readmemh* function to initialize arrays within the *SADMemA* and *SADMemB* memories by specifying the arrays to initialized as *SADMemA.Memory* and *SADMemB.Memory*. The vector procedure is similar to the procedure of the previous SAD HLSM without external

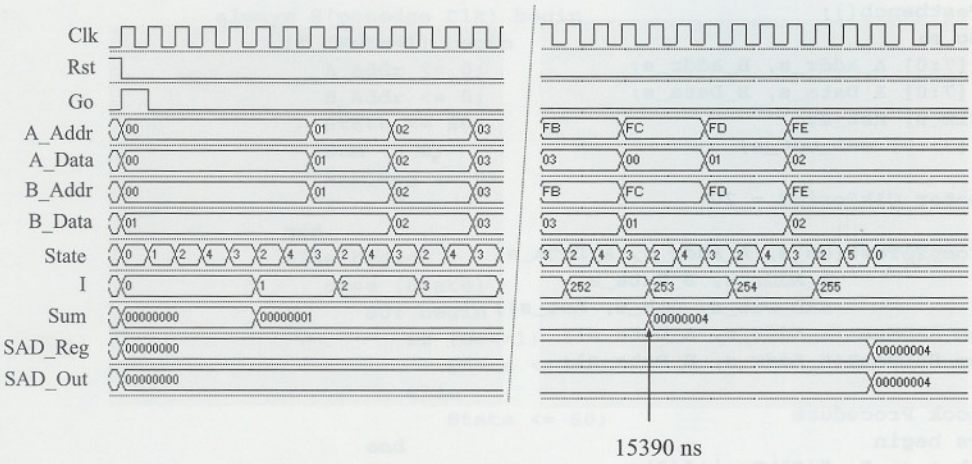


Figure 5.33 Waveforms for SAD HLSM with memory components.

memory components, except that the procedure must wait longer for the SAD output to appear, due to the extra state in the HLSM. Thus, rather than waiting (256*2+3) * (20 ns), the procedure waits (256*3+3) * (20 ns).

The testbench in Figure 5.32 has also been improved by declaring a parameter named *ClkPeriod* to represent the clock's period of 20 ns, and using that parameter in the clock and vector procedures, rather than hardcoding the 20 ns value throughout. Declaring such a parameter enables a designer to easily change the clock period just by changing one number, rather than having to change multiple numbers scattered throughout the code (and possibly forgetting to change the number in one place or changing a number when it should not have been changed).

Finally, Figure 5.33 shows waveforms generated from the testbench. Waveforms for the address and data signals now appear. The final SAD output also appears, although it appears later than in Figure 5.26, due to the extra state in the HLSM. Generally, converting a design from higher-levels to levels closer to an implementation yield increasing timing accuracy.