CHAPTER 7

Design and Synthesis of Datapath Controllers

Digital systems range from those that are control-dominated to those that are data-dominated. Control-dominated systems are reactive systems, responding to external events; data-dominated systems are shaped by the requirements of high-throughput data computation and transport, as in telecommunications and signal processing [1]. Sequential machines are commonly classified and partitioned into datapath units and control units.

Most datapaths include arithmetic units, such as arithmetic and logic units (ALUs), adders, multipliers, shifters, and digital signal processors, but some do not, such as graphics coprocessors. Datapath units consist of computational resources (e.g., ALUs and storage registers), logic for moving data through the system and between the computation units and the internal registers, and datapaths for moving data to and from the external environment. The datapath unit in Figure 7-1 is controlled by a finite-state machine (FSM) that coordinates the execution of instructions that perform operations on the datapath. Datapath units are characterized by repetitive operations on different sets of data, as in signal processing, image processing, and multimedia [2]. Architectures that are dominated by control units will generally have a significant amount of random (irregular) logic, together with some regular structures, like multiplexers for steering signals, and comparators.

7.1 Partitioned Sequential Machines

Partitioning a sequential machine into a datapath and a controller clarifies the architecture and simplifies the design of the system. The process by which the machine is designed is said to be application-driven, because the sequence of operations that must

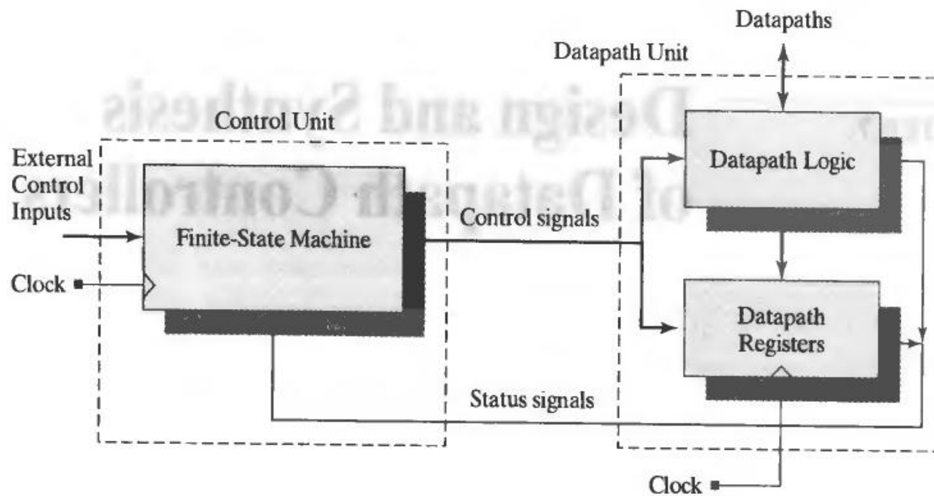


FIGURE 7-1 State-machine controller for a datapath.

be performed by the datapath unit in a particular application determines the resources composing its architecture, the set of instructions that must be executed by the datapath, and, ultimately, the FSM that controls the datapath.

The sequence of steps in an application-driven design process is illustrated in Figure 7-2. Once the architecture of the datapath unit has been selected to support the instruction set of an application, sequences of operations (control states) that support the instruction set can be identified. The control states are used to schedule assertions of the signals that control the movement and manipulation of data as the machine executes instructions. Then an FSM can be designed to generate the control signals. In this section we will illustrate the design of datapath controllers for some simple functional units, to prepare for the design of a stored-program reduced instruction-set computer in the next section.

Control units orchestrate, coordinate, and synchronize the operations of datapath units. The control unit of a machine generates the signals that load, read, and shift the contents of storage registers; fetch instructions and data from memory; store data in memory; steer signals through muxes; control three-state devices; and control the operations of ALUs and other complex datapath units. In synchronous machines, a common clock synchronizes the activities of the controller and datapath functional units. Note that the control unit in Figure 7-1 is implemented as an FSM, and is itself controlled by external input signals and by status signals from the datapath unit. The FSM produces the signals that control the operation of the datapath unit.

Datapath units are commonly described by dataflow graphs; control units are commonly modeled by state transition graphs and/or algorithmic-state machine (ASM) charts for FSMs. Partitioned sequential machines can be modeled by an FSM and datapath (FSMD), a combined control-dataflow graph, which expresses datapath operations in the context of a state-transition graph (STG). We favor using an ASM and datapath (ASMD) chart, which likewise links an ASM chart for a control unit to the operations of the datapath that it controls.

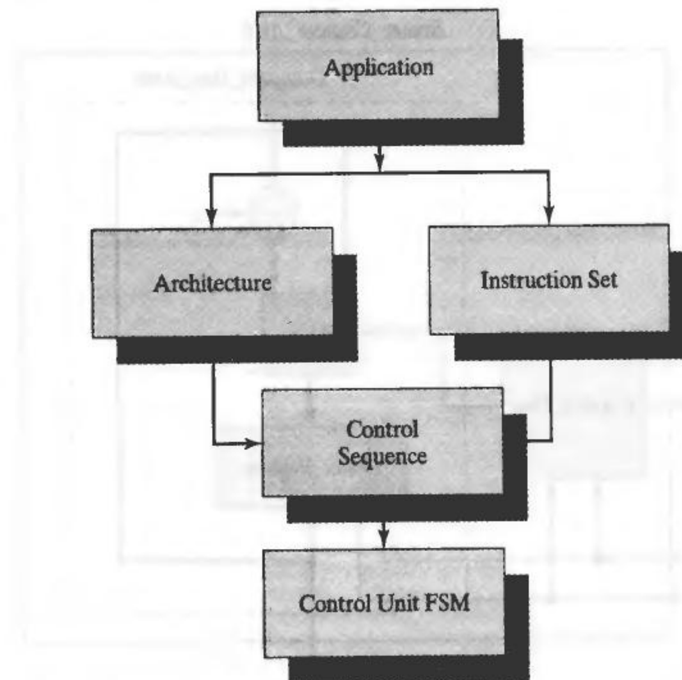


FIGURE 7-2 Application-driven architecture, instruction set, and control sequence for a datapath controller.

7.2 Design Example: Binary Counter

Consider a synchronous 4-bit binary counter that is to be incremented by a count of 1 at each active edge of the clock, and whose count is to wrap around to 0 when the count reaches 1111_2. We could describe the counter by an implicit state machine, *Binary_Counter_Imp*, executing a register transfer operation ($count <= count + 1$) conditionally, in every clock cycle, depending on *enable*, and then synthesize a hardware realization directly.[1] Other approaches are possible. One is to partition the machine into an architecture of separate datapath and control units, as shown in Figure 7-3 for *Binary_Counter_Arch*.

The functional elements of the architecture of the datapath unit consist of (1) a 4-bit register to hold *count*, (2) a mux that steers either *count* or the sum of *count* and 0001_2 to the input of the register, and (3) a 4-bit adder to increment *count*. The signal *enable* must be asserted for counting to occur, and the signal *rst* overrides all activity and drives *count* to a value of 0000_2. The input *rst* must be de-asserted and *enable* must be asserted for the machine to begin counting and to continue counting. The control unit for this simple machine passes *enable* directly to the datapath unit.

Now view the counter itself as an explicit-state machine, *Binary_Counter_STG*, having state *count* (the contents of the counter), and inputs *enable, clk*, and *rst*. The

[1]See Problem 41 in Chapter 6.

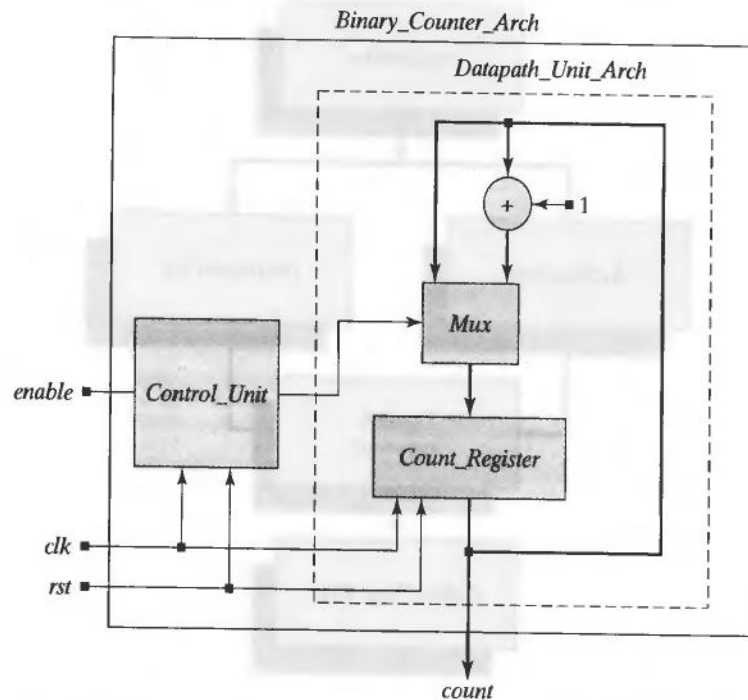


FIGURE 7-3 Architecture for a synchronous 4-bit binary counter.

simplified STG of the machine is shown in Figure 7-4, with *count* entered as the state label within each node. (The reset-directed arcs are not shown, nor are the arcs that return to the current state if *enable* is not asserted.) The STG can be used to develop an explicit state machine with two cyclic behaviors, one defining the next-state/output combinational logic, and the other synchronizing the state transitions. Note that this approach to the design of the machine can become unwieldy, because the size of the graph increases with the width of the datapath. It is often true that the number of states of the datapath registers is enormous compared to the number of states of the control unit. Partitioning the design eliminates the need to consider the state of the datapath registers, except to generate status signals that are fed back to the control unit.

The preceding three views of a 4-bit binary counter illustrate how partitioning a sequential machine into a datapath and a controller can reduce the size of the state that needs to be considered in the design and simplify the control unit of the machine. In this example, the implicit state machine has the simplest description; it suppresses structural detail, leaving it to the synthesis tool. The partitioned machine has the most structural detail, a simple controller, and a datapath register whose state does not influence the design; the STG-based approach required a detailed STG and led to a state machine with 16 states, because the state of the machine was the state of the register holding *count*. The relative complexity of the Verilog models and the synthesized hardware are tradeoffs between the equivalent, but alternative machines.[2]

[2]See Problems 10 and 11 at the end of this chapter.

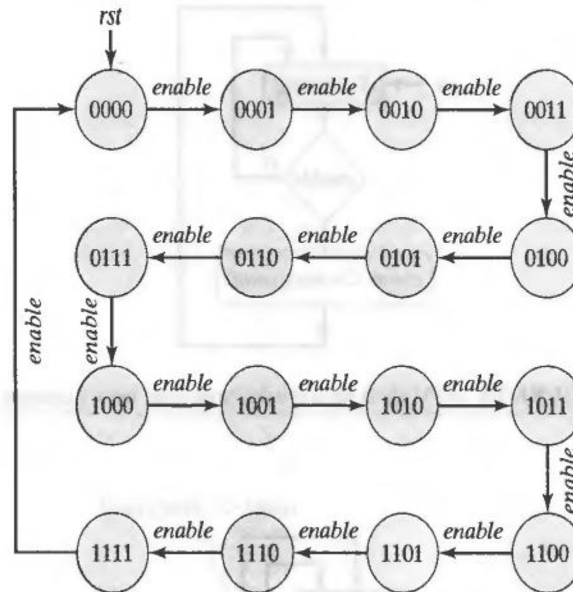


FIGURE 7-4 Simplified STG for a synchronous 4-bit binary counter.

Another view of the binary counter is based on the counter's *activity*. The machine described by the ASM chart in Figure 7-5, *Binary_Counter_ASM*, has one state, *S_running*.[3] At every clock, *enable* is tested and a transition is made back to *S_running*; if *enable* is asserted, a conditional register operation that increments the counter is executed concurrently with the state transition. Even though the ASM chart is simple, it describes the activity of more complex counters and other single-cycle machines that have a different relationship governing the register operations. For example, the function *next_count* could describe a Johnson counter.[4] Note that the description does not require an explicit state register because it remains in the same state. In fact, it reduces to the implicit state machine described above.

A fifth approach to designing the binary counter partitions the machine into a control unit and a datapath unit, but designs a register transfer level (RTL) behavioral model for the datapath unit, rather than a structural model (as we did in Figure 7-3). This style separates the design of the control unit from the design (and synthesis) of the datapath unit and simplifies the description of the datapath unit. It separates the unit that determines *what* happens from the unit that determines *when* it happens. This style might seem like overkill for this counter (and it is), but the style is critical to successful design and synthesis of more complex machines, and is easily implemented in Verilog. Figure 7-6 shows an ASMD chart for *Binary_Counter_Part_RTL*, a counter partitioned into a datapath unit and a control unit, with signal *enable_DP* linking the controller to the datapath. The model's control units passes *enable* through to the datapath unit. The state machine of the controller consists of only the pass-through logic.

[3]This machine is actually the same as *Binary_Counter_Imp* mentioned above.
[4]See Problem 18 in Chapter 5.

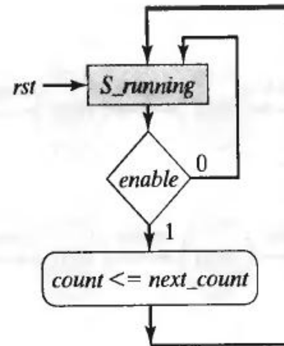


FIGURE 7-5 ASM chart for a synchronous 4-bit binary counter.

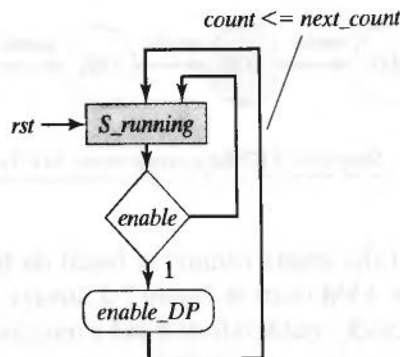


FIGURE 7-6 ASMD chart for a datapath unit, a synchronous 4-bit binary counter, controlled by a state machine.

Example 7.1

The Verilog description *Binary_Counter_Part_RTL* has two nested modules: *Control_Unit*, and *Datapath_Unit*. The datapath has been described with flexibility to implement codes for other counters, independently of the control unit. The simulation results in Figure 7-7 show that counting begins at the first rising edge of *clk* after *enable* is asserted, that counting continues while *enable* is asserted, and that *enable_DB* replicates *enable*.

```
module Binary_Counter_Part_RTL (count, enable, clk, rst);
parameter               size = 4;
output  [size -1: 0]    count;
input                   enable;
input                   clk, rst;
wire                    enable_DP;

Control_Unit  M0 (enable_DP, enable, clk, rst);
Datapath_Unit M1 (count, enable_DP, clk, rst);
endmodule
```

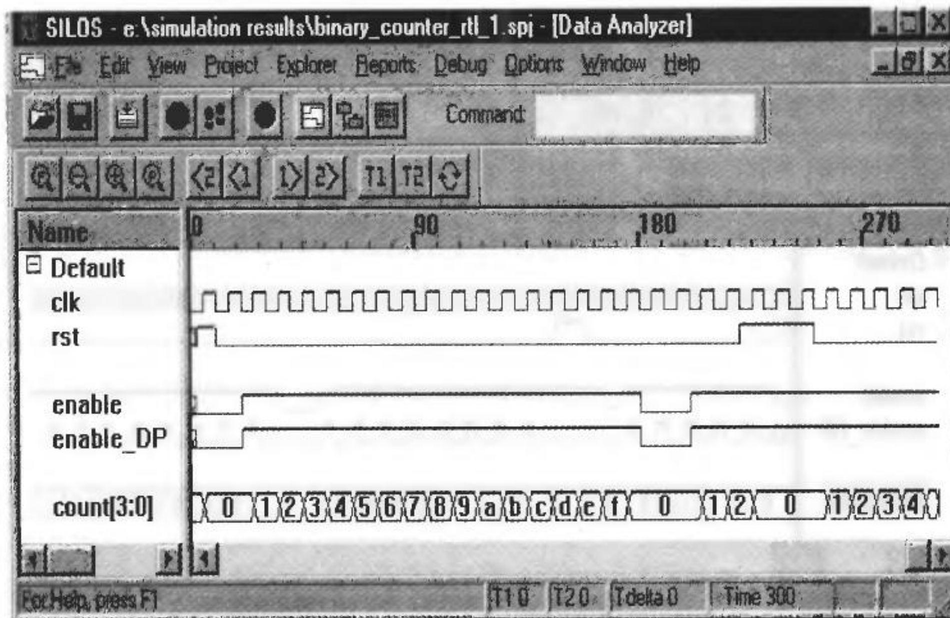


FIGURE 7-7 Simulation results for *Binary_Counter_RTL*, a synchronous 4-bit binary counter controlled by a state machine.

```
module Control_Unit (enable_DP, enable, clk, rst);
 output       enable_DP;
 input        enable;
 input        clk, rst;              // Not needed
 wire         enable_DP = enable;    // pass through

endmodule

module Datapath_Unit (count, enable, clk, rst);
 parameter    size = 4;
 output       [size-1: 0] count;
 input        enable;
 input        clk, rst;
 reg          count;
 wire         [size-1: 0] next_count;

always @ (posedge clk)
 if (rst == 1) count <= 0;
  else if (enable == 1) count <= next_count(count);

function      [size-1: 0]      next_count;
 input        [size-1: 0]      count;
 begin
  next_count = count + 1;
 end
endfunction
endmodule
```

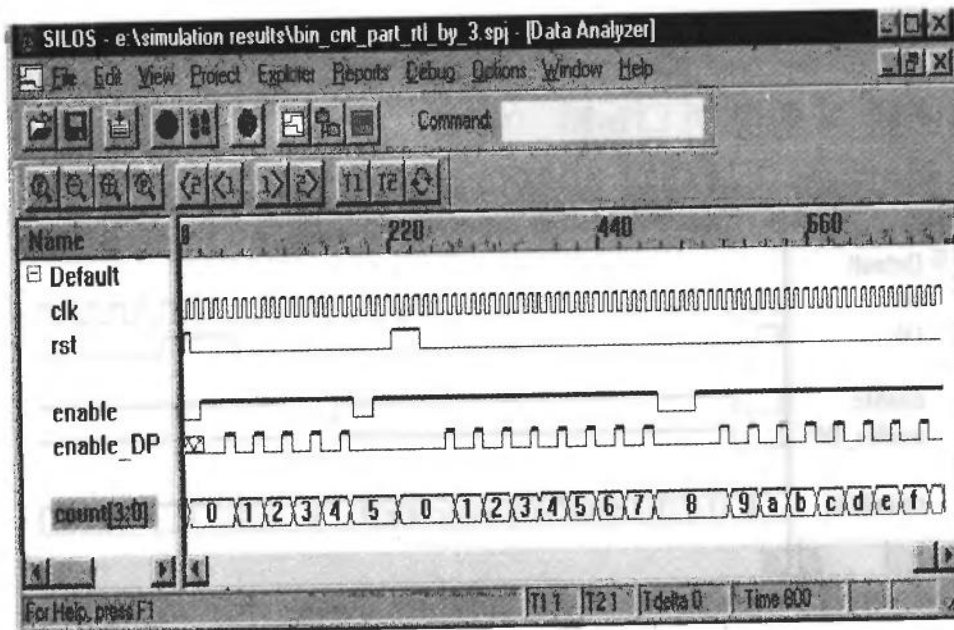


FIGURE 7-8 Simulation results for *Binary_Counter_Part_RTL_by_3* with a control unit to increment the datapath counter every third cycle.

Next, we will redesign the counter to form *Binary_Counter_Part_RTL_by_3*, which increments its count every third clock cycle. Only the control unit must change. One approach is to model the control unit by the implicit Moore machine shown below. The simulated activity of the machine is shown in Figure 7-8. *Caution*: The machine has an interesting feature that calls for a more careful approach.[5]

```
module Control_Unit_by_3  (enable_DP, enable, clk, rst);
 output        enable_DP;
 input         enable;
 input         clk, rst;           // Not needed

 reg           enable_DP;

 always begin: Cycle_by_3
  @ (posedge clk) enable_DP  <= 0;
  if ((rst == 1) || (enable != 1)) disable Cycle_by_3; else
   @ (posedge clk)
    if ((rst == 1) || (enable != 1)) disable Cycle_by_3; else
     @ (posedge clk)
      if ((rst == 1) || (enable != 1)) disable Cycle_by_3;
       else enable_DP <= 1;
 end // Cycle_by_3
endmodule
```

End of Example 7.1

[5]See Problem 12 at the end of this chapter.

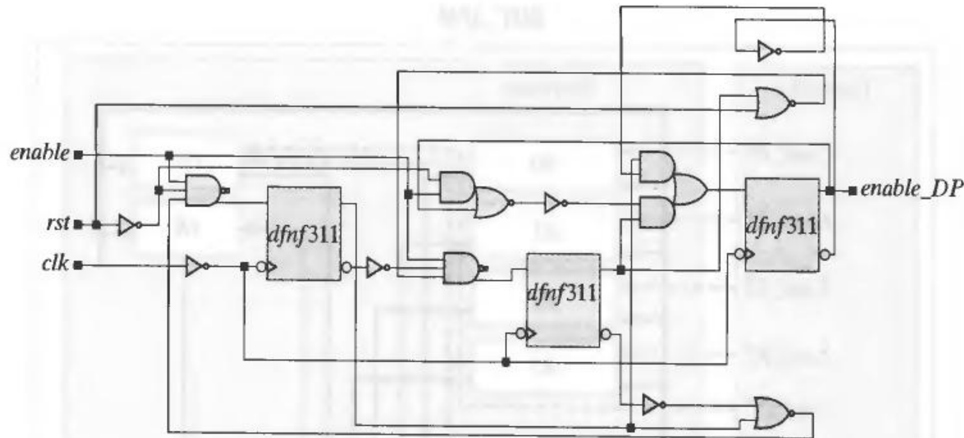


FIGURE 7-9 Circuit synthesized for *Control_Unit_by_3*, the control unit of *Binary_Counter_Part_RTL_by_3*, a 4-bit binary counter partitioned into a control unit and a datapath unit, with the counter incrementing every third clock cycle.

The Verilog module *Binary_Counter_Part_RTL_by_3*, with the modified control unit, is synthesizable. We anticipate that two flip-flops will be needed to implement the implicit-state machine of the control unit, because the state evolves through three embedded clock cycles. One flip-flop will be needed to register *enable_DP*, and four flip-flops will be needed to implement the register holding *count* in the datapath unit. The synthesis results in Figure 7-9 confirm this use of resources for the control unit. However, the postsynthesis behavior of *Binary_Counter_Part_RTL_by_3* is problematic.[6]

7.3 Design and Synthesis of a RISC Stored-Program Machine

Reduced instruction-set computers (RISC) are designed to have a small set of instructions that execute in short clock cycles, with a small number of cycles per instruction. RISC machines are optimized to achieve efficient pipelining of their instruction streams [2]. In this section we will model a simple RISC machine. Our companion website (www.prenhall.com/ciletti) includes the machine's source code and an assembler that can be used to develop programs for student projects. The machine also serves as a starting point for developing architectural variants and a more robust instruction set.

Designers make high-level tradeoffs in selecting an architecture that serves an application. Once an architecture has been selected, a circuit that has sufficient performance (speed) must be synthesized. Hardware description languages (HDLs) play a key role in this process by modeling the system and serving as a descriptive medium that can be used by a synthesis tool.

As an example, the overall architecture of a simple RISC is shown in Figure 7-10. *RISC_SPM* is a stored-program RISC-architecture machine [3, 4]—its instructions are contained in a program stored in memory.

[6]See Problem 12 at the end of this chapter.

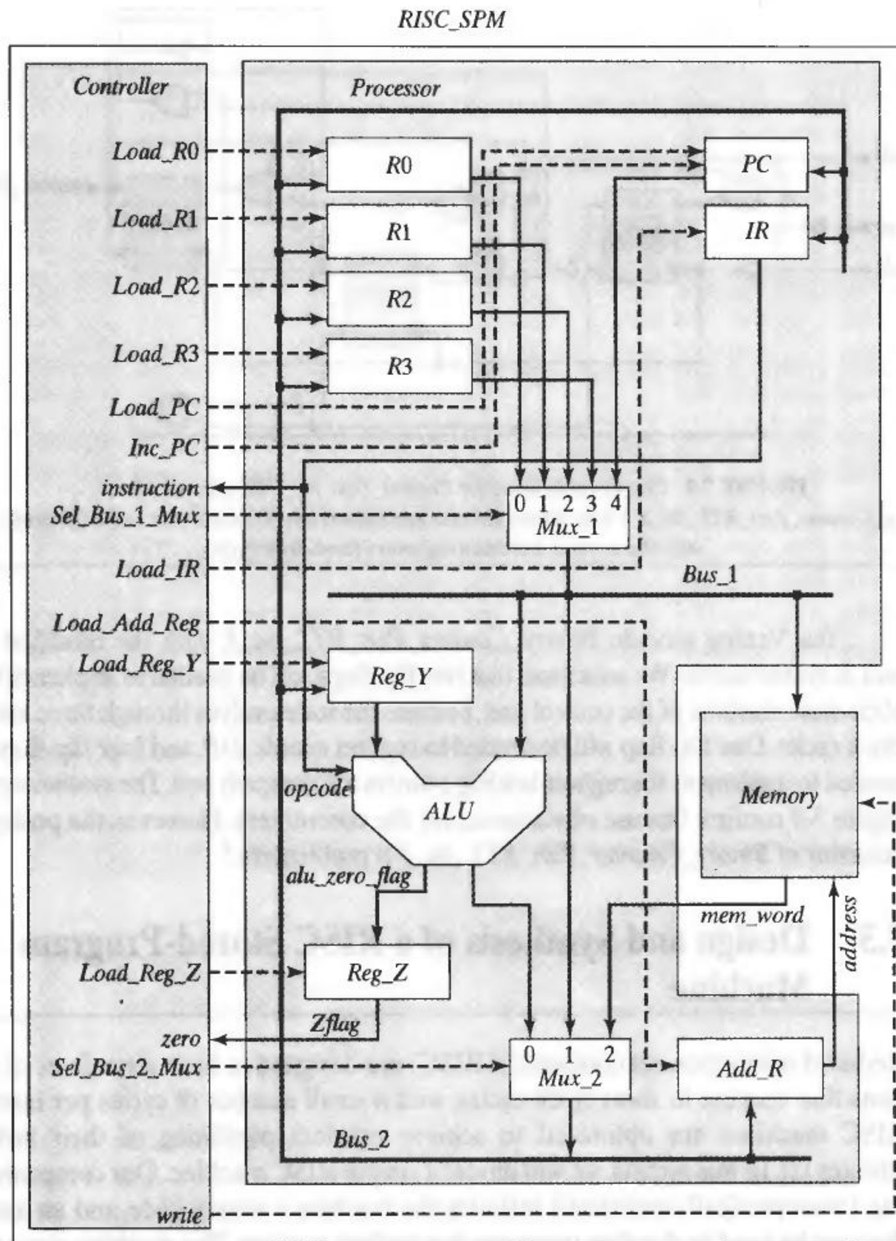


FIGURE 7-10 Architecture of *RISC_SPM*, an RISC stored-program machine (SPM).

The machine consists of three functional units: a processor, a controller, and memory. Program instructions and data are stored in memory. In program-directed operation, instructions are fetched synchronously from memory, decoded, and executed to (1) operate on data within the arithmetic and logic unit (*ALU*), (2) change the contents of storage registers, (3) change the contents of the program counter (*PC*), instruction register (*IR*) and the address register (*ADD_R*), (4) change the contents of memory, (5) retrieve data and instructions from memory, and (6) control the movement of data on the system busses. The instruction register contains the instruction that is currently

being executed; the program counter contains the address of the next instruction to be executed; and the address register holds the address of the memory location that will be addressed next by a read or write operation.

7.3.1 RISC SPM: Processor

The processor includes registers, datapaths, control lines, and an ALU capable of performing arithmetic and logic operations on its operands, subject to the opcode held in the instruction register. A multiplexer, *Mux_1*, determines the source of data that is bound for *Bus_1*, and a second mux, *Mux_2*, determines the source of data bound for *Bus_2*. The input datapaths to *Mux_1* are from four internal general-purpose registers (*R0, R1, R2, R3*), and from the PC. The contents of *Bus_1* can be steered to the ALU, to memory, or to *Bus_2* (via *Mux_2*). The input datapaths to *Mux_2* are from the *ALU*, *Mux_1*, and the memory unit. Thus, an instruction can be fetched from memory, placed on *Bus_2*, and loaded into the instruction register. A word of data can be fetched from memory, and steered to a general-purpose register or to the operand register (*Reg_Y*) prior to an operation of the ALU. The result of an *ALU* operation can be placed on *Bus_2*, loaded into a register, and subsequently transferred to memory. A dedicated register (*Reg_Z*) holds a flag indicating that the result of an *ALU* operation is 0.[7]

7.3.2 RISC SPM: ALU

For the purposes of this example, the ALU has two operand datapaths, *data_1* and *data_2*, and its instruction set is limited to the following instructions:

Instruction	Action
ADD	Adds the datapaths to form *data_1* + *data_2*
SUB	Subtracts the datapaths to form *data_1* − *data_2*
AND	Takes the bitwise-and of the datapaths, *data_1* & *data_2*
NOT	Takes the bitwise Boolean complement of *data_1*

7.3.3 RISC SPM: Controller

The timing of all activity is determined by the controller. The controller must steer data to the proper destination, according to the instruction being executed. Thus, the design of the controller is strongly dependent on the specification of the machine's ALU and datapath resources and the clocking scheme available. In this example, a single clock will be used, and execution of an instruction is initiated on a single edge of the clock

[7]This can be used to monitor a loop index.

(e.g., the rising edge). The controller monitors the state of the processing unit and the instruction to be executed and determines the value of the control signals. The controller's input signals are the instruction word and the zero flag from the *ALU*. The signals produced by the controller are identified as follows:

Control Signal	Action
Load_Add_Reg	Loads the address register
Load _PC	Loads *Bus_2* to the program counter
Load_IR	Loads *Bus_2* to the instruction register
Inc_PC	Increments the program counter
Sel_Bus_1_Mux	Selects among the *Program_Counter, R0, R1, R2*, and *R3* to drive *Bus_1*
Sel_Bus_2_Mux	Selects among *Alu_out, Bus_1*, and memory to drive *Bus_2*
Load_R0	Loads general-purpose register *R0*
Load_R1	Loads general-purpose register *R1*
Load_R2	Loads general-purpose register *R2*
Load_R3	Loads general-purpose register *R3*
Load_Reg_Y	Loads *Bus_2* to the register *Reg_Y*
Load Reg_Z	Stores output of *ALU* in register *Reg_Z*
write	Loads *Bus_1* into the *SRAM* memory at the location specified by the address register

The control unit (1) determines when to load registers, (2) selects the path of data through the multiplexers, (3) determines when data should be written to memory, and (4) controls the three-state busses in the architecture.

7.3.4 RISC SPM: Instruction Set

The machine is controlled by a machine language program consisting of a set of instructions stored in memory. So, in addition to depending on the machine's architecture, the design of the controller depends on the processor's instruction set (i.e., the instructions that can be executed by a program). A machine language program consists of a stored sequence of 8-bit words (bytes). The format of an instruction of *RISC_SPM* can be long or short, depending on the operation.

Short instructions have the format shown in Figure 7-11(a). Each short instruction requires 1 byte of memory. The word has a 4-bit opcode, a 2-bit source register address, and a 2-bit destination register address. A long instruction requires 2 bytes of memory. The first word of a long instruction contains a 4-bit opcode. The remaining 4 bits of the word can be used to specify addresses of a pair of source and destination registers, depending on the instruction. The second word contains the address of the memory word that holds an operand required by the instruction. Figure 7-11(b) shows the 2-byte format of a long instruction.

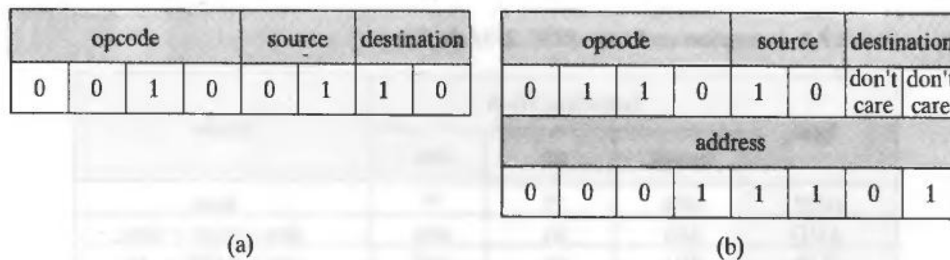


FIGURE 7-11 Instruction format of (a) a short instruction, and (b) a long instruction.

The instruction mnemonics and their actions are listed below.

Single-Byte Instruction	Action
NOP	No operation is performed; all registers retain their values. The addresses of the source and destination register are don't-cares, they have no effect.
ADD	Adds the contents of the source and destination registers and stores the result into the destination register.
AND	Forms the bitwise-and of the contents of the source and destination registers and stores the result into the destination register.
NOT	Forms the bitwise complement of the content of the source register and stores the result into the destination register.
SUB	Subtracts the content of the source register from the destination register and stores the result into the destination register.

Two-Byte Instruction	Action
RD	Fetches a memory word from the location specified by the second byte and loads the result into the destination register. The source register bits are don't-cares (i.e., unused).
WR	Writes the contents of the source register to the word in memory specified by the address held in the second byte. The destination register bits are don't-cares (i.e., unused).
BR	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction. The source and destination bits are don't-cares (i.e., unused).
BRZ	Branches the activity flow by loading the program counter with the word at the location (address) specified by the second byte of the instruction if the zero flag register is asserted.

The *RISC_SPM* instruction set is summarized in Table 7-1.

The program counter holds the address of the next instruction to be executed. When the external reset is asserted, the program counter is loaded with 0, indicating that the bottom of memory holds the next instruction that will be fetched. Under the action of the clock, for single-cycle instructions, the instruction at the address in the program counter is loaded into the instruction register and the program counter is incremented. An instruction decoder determines the resulting action on the datapaths and the ALU. A long instruction is held in 2 bytes, and an additional clock cycle is required to

TABLE 7-1 Instruction set for the *RISC_SPM* machine.

Instr	**Instruction Word**			**Action**
	opcode	**src**	**dest**	
NOP	0000	??	??	none
ADD	0001	src	dest	dest <= src + dest
SUB	0010	src	dest	dest <= dest − src
AND	0011	src	dest	dest <= src && dest
NOT	0100	src	dest	dest <= ~ src
RD*	0101	??	dest	dest <= memory [Add_R]
WR*	0110	src	??	memory[Add_R] < = src
BR*	0111	??	??	PC <= memory[Add_R]
BRZ*	1000	??	??	PC <= memory [Add_R]
HALT	1111	??	??	Halts execution until reset

* Requires a second word of data; ? denotes a don't-care.

execute the instruction. In the second cycle of execution, the second byte is fetched from memory at the address held in the program counter, then the instruction is completed. Intermediate contents of the ALU may be meaningless when two-cycle operations are being executed.

7.3.5 RISC SPM: Controller Design

The machine's controller will be designed as an FSM. Its states must be specified, given the architecture, instruction set, and clocking scheme used in the design. This can be accomplished by identifying what steps must occur to execute each instruction. We will use an ASM chart to describe the activity within the machine, *RISC_SPM*, and to present a clear picture of how the machine operates under the command of its instructions.

The machine has three phases of operation: *fetch*, *decode*, and *execute*. Fetching retrieves an instruction from memory, decoding decodes the instruction, manipulates datapaths, and loads registers; execution generates the results of the instruction. The fetch phase will require two clock cycles—one to load the address register and one to retrieve the addressed word from memory. The decode phase is accomplished in one cycle. The execution phase may require zero, one, or two more cycles, depending on the instruction. The *NOT* instruction can execute in the same cycle that the instruction is decoded; single-byte instructions, such as *ADD*, take one cycle to execute, during which the results of the operation are loaded into the destination register. The source register can be loaded during the decode phase. The execution phase of a 2-byte instruction will take two cycles: (for example *RD*), one to load the address register with the second byte, and one to retrieve the word from the memory location addressed by

the second byte and load it into the destination register. The controller for *RISC_SPM* has the 11 states listed below, with the control actions that must occur in each state.

State	Action
S_idle	State entered after reset is asserted. No action.
S_fet1	Load the address register with the contents of the program counter. (*Note*: *PC* is initialized to the starting address by the reset action.) The state is entered at the first active clock after reset is de-asserted, and is revisited after a *NOP* instruction is decoded.
S_fet2	Load the instruction register with the word addressed by the address register, and increment the program counter to point to the next location in memory, in anticipation of the next instruction or data fetch.
S_dec	Decode the instruction register and assert signals to control datapaths and register transfers.
S_ex1	Execute the *ALU* operation for a single-byte instruction, conditionally assert the zero flag, and load the destination register.
S_rd1	Load the address register with the second byte of a *RD* instruction, and increment the *PC*.
S_rd2	Load the destination register with the memory word addressed by the byte loaded in *S_rd1*.
S_wr1	Load the address register with the second byte of a *WR* instruction, and increment the *PC*.
S_wr2	Load the destination register with the memory word addressed by the byte loaded in *S_wr1*.
S_br1	Load the address register with the second byte of a *BR* instruction, and increment the *PC*.
S_br2	Load the program counter with the memory word addressed by the byte loaded in *S_br1*.
S_halt	Default state to trap failure to decode a valid instruction.

The partitioned ASM chart for the controller of *RISC_SPM* is shown in Figure 7-12, with the states numbered for clarity. Once the ASM charts have been built, the designer can write the Verilog description of the entire machine, for the given architectural partition. This process unfolds in stages. First, the functional units are declared according to the partition of the machine. Then their ports and variables are declared and checked for syntax. Then the individual units are described, debugged, and verified. The last step is to integrate the design and verify that it has correct functionality.

The top-level Verilog module *RISC_SPM* integrates the modules of the architecture of Figure 7-10 and will be presented first. Three modules are instantiated: *Processing_Unit, Control_Unit*, and *Memory_Unit*, with instance names *M0_Processor, M1_Controller*, and *M2_Mem*, respectively. The parameters declared at this level of the hierarchy size the datapaths between the three structural/functional units.

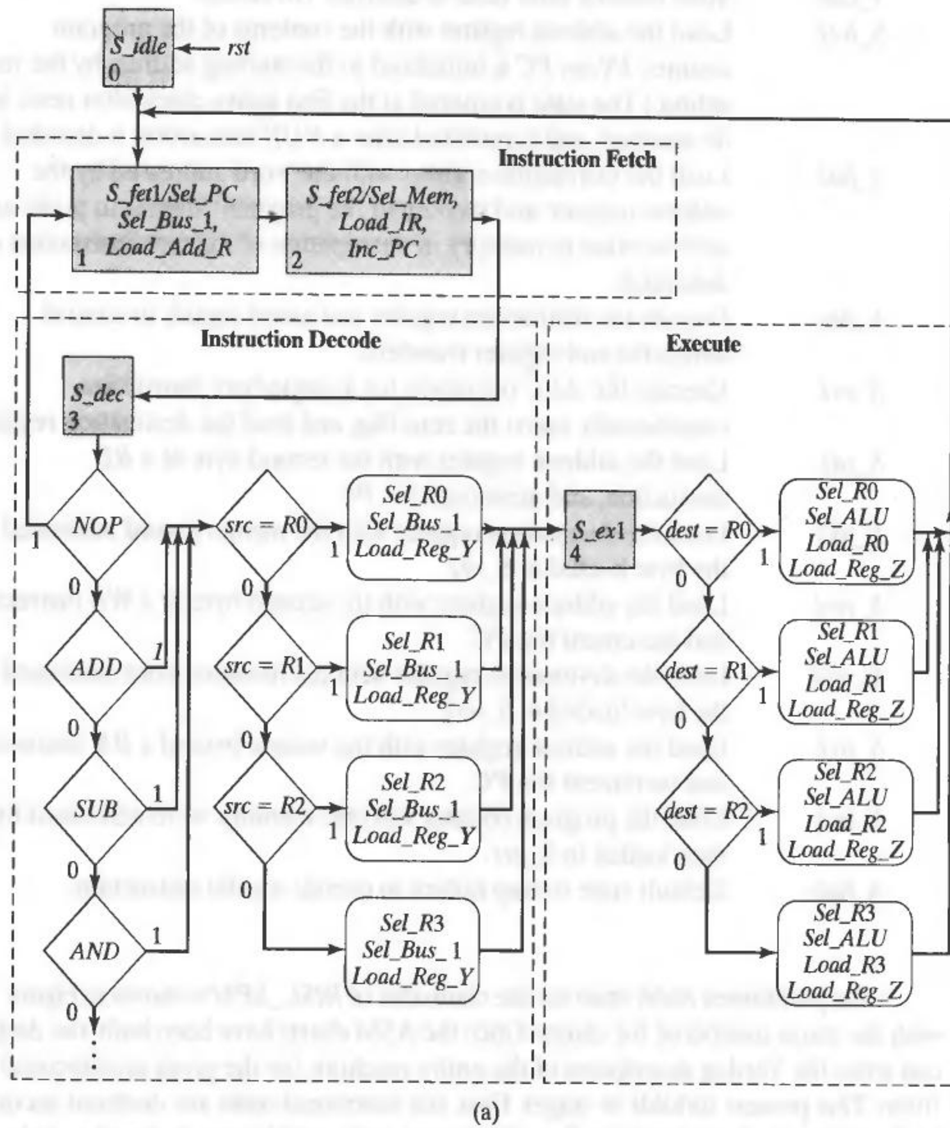
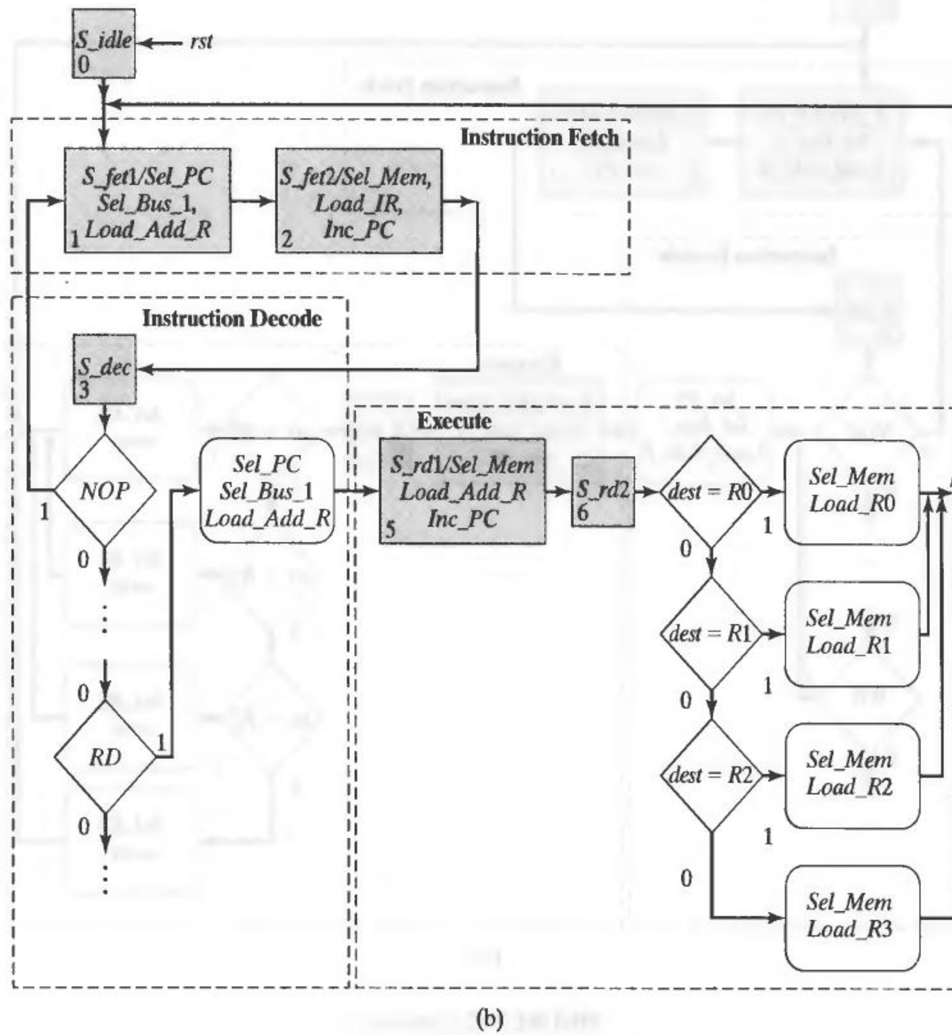
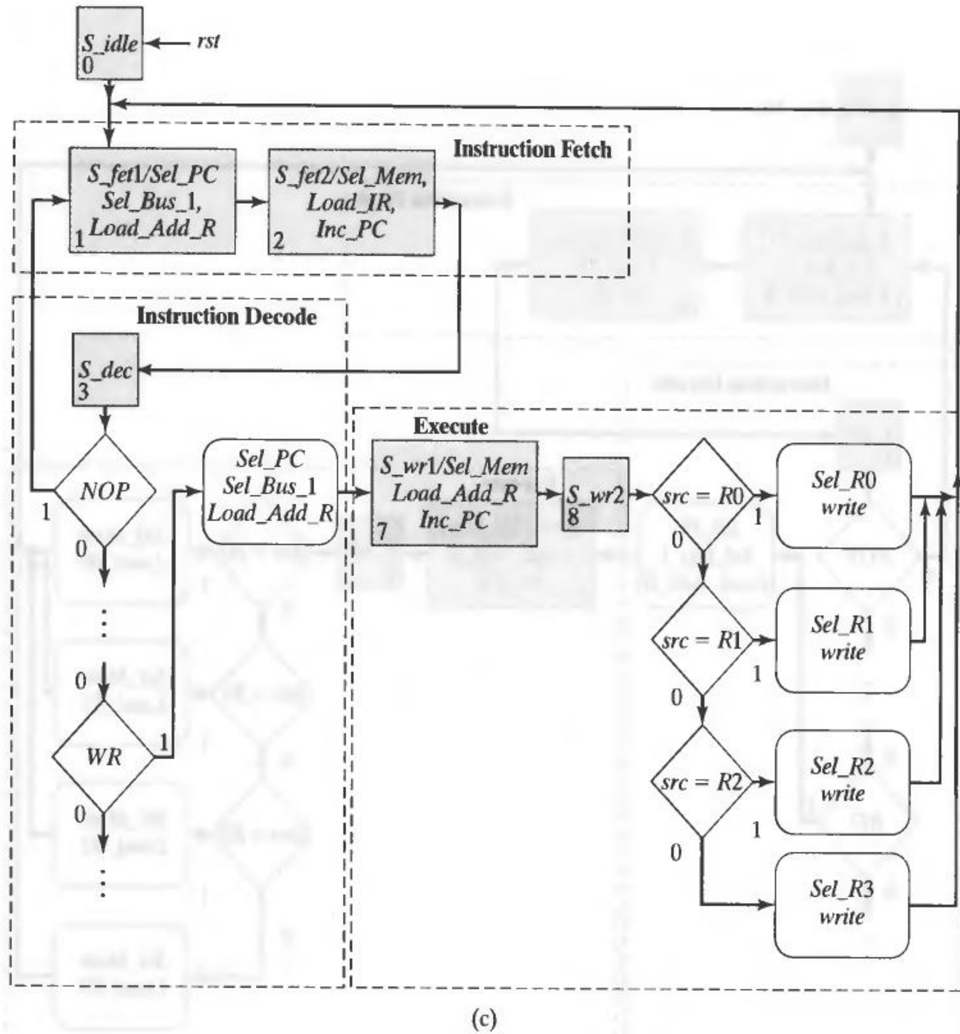


FIGURE 7-12 ASM charts for the controller of a processor that implements the *RISC_SPM* instruction set: (a) *NOP, ADD, SUB, AND*, (b) *RD*, (c) *WR*, (d) *BR, BRZ*, and (e) *NOT*.



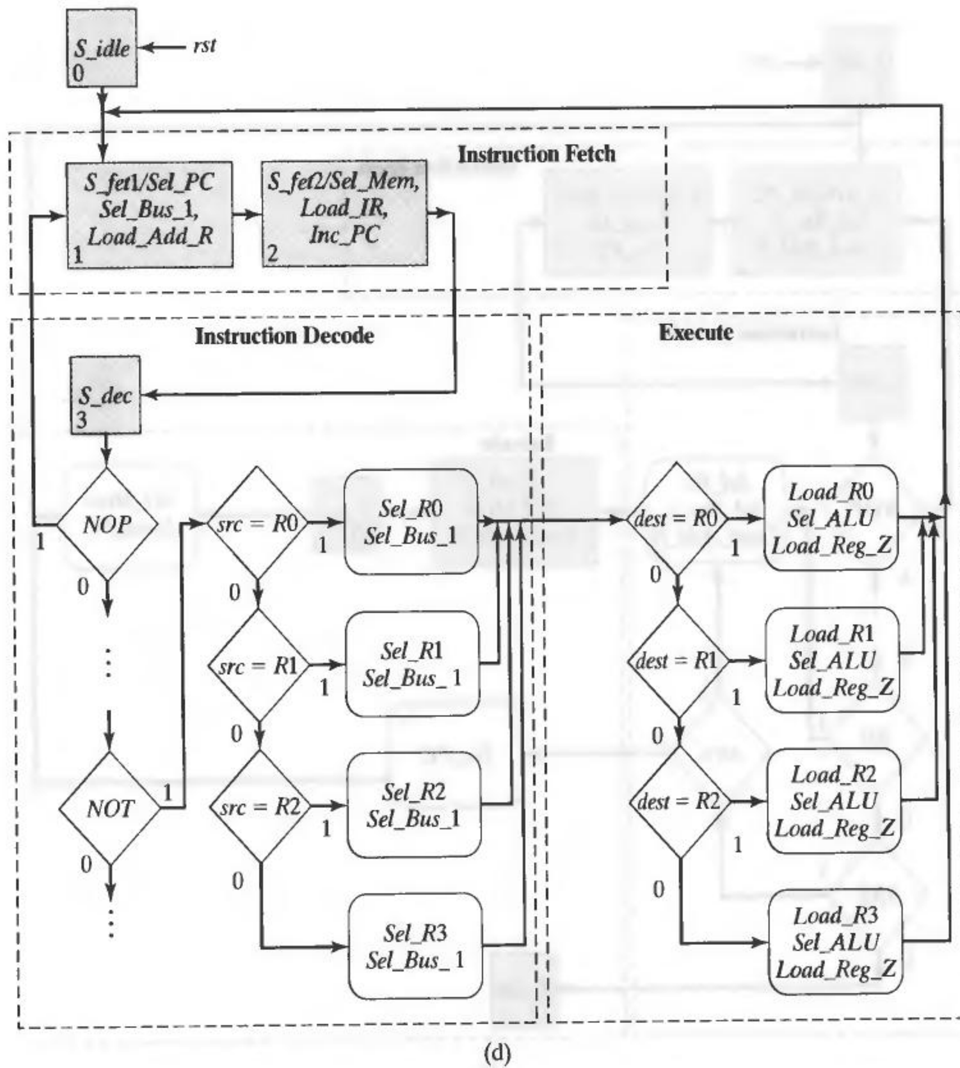
(b)

FIGURE 7-12 Continued



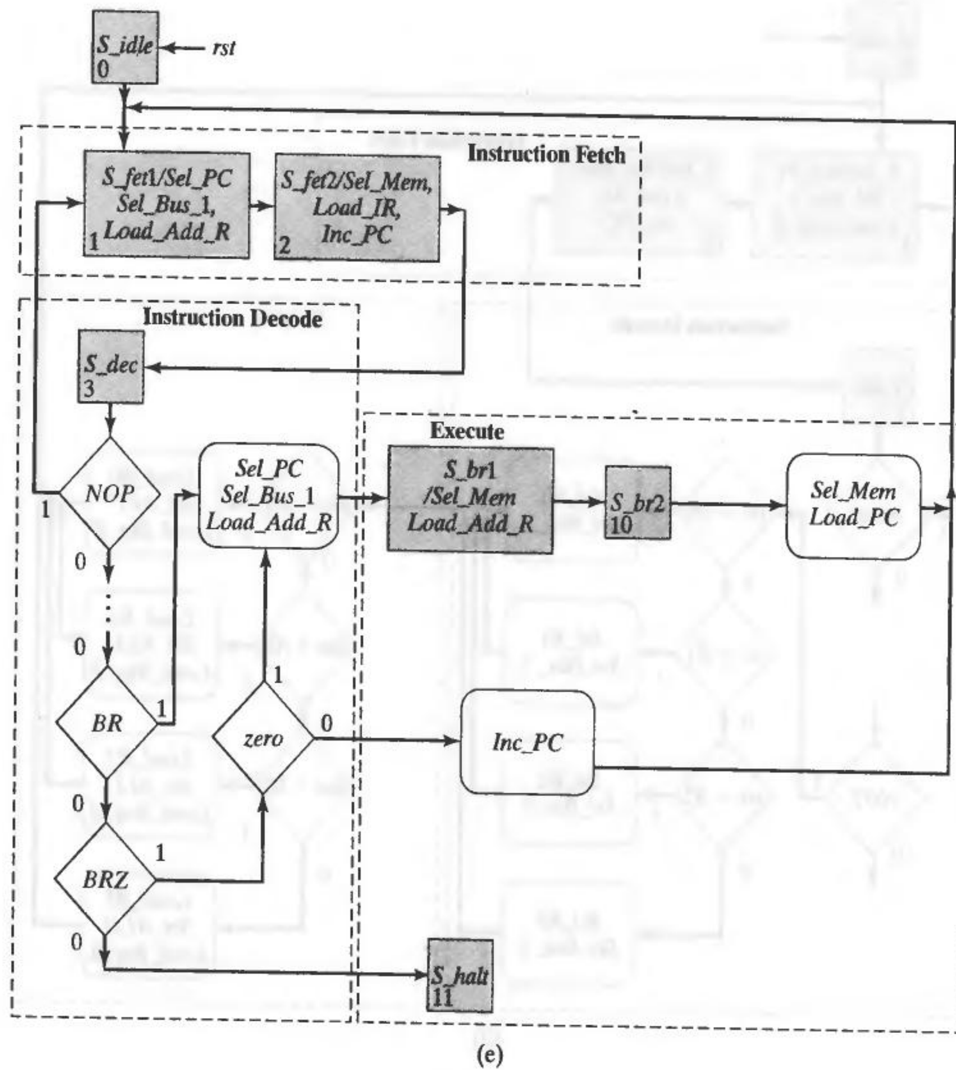
(c)

FIGURE 7-12 Continued



(d)

FIGURE 7-12 Continued



(e)

FIGURE 7-12 Continued

```
module RISC_SPM (clk, rst);
  parameter word_size = 8;
  parameter Sel1_size = 3;
  parameter Sel2_size = 2;
  wire [Sel1_size-1: 0] Sel_Bus_1_Mux;
  wire [Sel2_size-1: 0] Sel_Bus_2_Mux;
  input clk, rst;

  // Data Nets
  wire zero;
  wire [word_size-1: 0] instruction, address, Bus_1, mem_word;

  // Control Nets
  wire Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR;
  wire Load_Add_R, Load_Reg_Y, Load_Reg_Z;
  wire write;

  Processing_Unit M0_Processor
   (instruction, zero, address, Bus_1, mem_word, Load_R0, Load_R1,
   Load_R2, Load_R3, Load_PC, Inc_PC, Sel_Bus_1_Mux, Load_IR,
   Load_Add_R, Load_Reg_Y, Load_Reg_Z,  Sel_Bus_2_Mux, clk, rst);

  Control_Unit M1_Controller (Load_R0, Load_R1, Load_R2, Load_R3, Load_PC,
  Inc_PC, Sel_Bus_1_Mux, Sel_Bus_2_Mux , Load_IR, Load_Add_R,
  Load_Reg_Y, Load_Reg_Z, write, instruction, zero, clk, rst);

  Memory_Unit M2_MEM (
   .data_out(mem_word),
   .data_in(Bus_1),
   .address(address),
   .clk(clk),
   .write(write) );
endmodule
```

The Verilog model of the machine's processor will describe the architecture, register operations, and datapath operations that are represented by the functional units shown in Figure 7-10. The processor instantiates several other modules, which must be declared too.

```
module Processing_Unit (instruction, Zflag, address, Bus_1, mem_word,
  Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Sel_Bus_1_Mux,
  Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z,
  Sel_Bus_2_Mux, clk, rst);

  parameter word_size = 8;
  parameter op_size = 4;
  parameter Sel1_size = 3;
  parameter Sel2_size = 2;

  output [word_size-1: 0]  instruction, address, Bus_1;
  output                   Zflag;
```

```
input [word_size-1: 0]   mem_word;
input                    Load_R0, Load_R1, Load_R2, Load_R3, Load_PC,
                         Inc_PC;
input [Sel1_size-1: 0]   Sel_Bus_1_Mux;
input [Sel2_size-1: 0]   Sel_Bus_2_Mux;
input                    Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z;
input                    clk, rst;

wire                     Load_R0, Load_R1, Load_R2, Load_R3;
wire [word_size-1: 0]    Bus_2;
wire [word_size-1: 0]    R0_out, R1_out, R2_out, R3_out;
wire [word_size-1: 0]    PC_count, Y_value, alu_out;
wire                     alu_zero_flag;
wire [op_size-1 : 0]     opcode = instruction [word_size-1: word_size-op_size];

Register_Unit            R0      (R0_out, Bus_2, Load_R0, clk, rst);
Register_Unit            R1      (R1_out, Bus_2, Load_R1, clk, rst);
Register_Unit            R2      (R2_out, Bus_2, Load_R2, clk, rst);
Register_Unit            R3      (R3_out, Bus_2, Load_R3, clk, rst);
Register_Unit            Reg_Y   (Y_value, Bus_2, Load_Reg_Y, clk, rst);
D_flop                   Reg_Z   (Zflag, alu_zero_flag, Load_Reg_Z, clk, rst);
Address_Register         Add_R   (address, Bus_2, Load_Add_R, clk, rst);
Instruction_Register     IR      (instruction, Bus_2, Load_IR, clk, rst);
Program_Counter          PC      (PC_count, Bus_2, Load_PC, Inc_PC, clk, rst);
Multiplexer_5ch          Mux_1   (Bus_1, R0_out, R1_out, R2_out, R3_out,
                                  PC_count, Sel_Bus_1_Mux);
Multiplexer_3ch          Mux_2   (Bus_2, alu_out, Bus_1, mem_word,
                                  Sel_Bus_2_Mux);
Alu_RISC                 ALU     (alu_zero_flag, alu_out, Y_value, Bus_1,
                                  opcode);
endmodule

module Register_Unit (data_out, data_in, load, clk, rst);
 parameter                word_size = 8;
 output [word_size-1: 0]  data_out;
 input  [word_size-1: 0]  data_in;
 input                    load;
 input                    clk, rst;
 reg                      data_out;

 always @ (posedge clk or negedge rst)
   if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module D_flop (data_out, data_in, load, clk, rst);
 output         data_out;
 input          data_in;
 input          load;
 input          clk, rst;
 reg            data_out;
```

```
always @ (posedge clk or negedge rst)
  if (rst == 0) data_out <= 0; else if (load == 1)data_out <= data_in;
endmodule

module Address_Register (data_out, data_in, load, clk, rst);
 parameter word_size = 8;
 output  [word_size-1: 0]   data_out;
 input   [word_size-1: 0]   data_in;
 input                      load, clk, rst;
 reg                        data_out;
 always @ (posedge clk or negedge rst)
   if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module Instruction_Register (data_out, data_in, load, clk, rst);
 parameter word_size = 8;
 output [word_size-1: 0]  data_out;
 input  [word_size-1: 0]  data_in;
 input                    load;
 input                    clk, rst;
 reg                      data_out;
 always @ (posedge clk or negedge rst)
   if (rst == 0) data_out <= 0; else if (load) data_out <= data_in;
endmodule

module Program_Counter (count, data_in, Load_PC, Inc_PC, clk, rst);
 parameter word_size = 8;
 output [word_size-1: 0]   count;
 input  [word_size-1: 0]   data_in;
 input                     Load_PC, Inc_PC;
 input                     clk, rst;
 reg                       count;
 always @ (posedge clk or negedge rst)
  if (rst == 0) count <= 0; else if (Load_PC) count <= data_in; else if  (Inc_PC)
  count <= count +1;
endmodule

module Multiplexer_5ch (mux_out, data_a, data_b, data_c, data_d, data_e, sel);
 parameter word_size = 8;
 output  [word_size-1: 0]   mux_out;
 input   [word_size-1: 0]   data_a, data_b, data_c, data_d, data_e;
 input   [2: 0] sel;

 assign  mux_out = (sel == 0)     ? data_a: (sel == 1)
                                  ? data_b : (sel == 2)
                                  ? data_c: (sel == 3)
                                  ? data_d : (sel == 4)
                                  ? data_e : 'bx;

 endmodule
```

```
module Multiplexer_3ch (mux_out, data_a, data_b, data_c, sel);
 parameter    word_size = 8;
  output      [word_size-1: 0]   mux_out;
  input       [word_size-1: 0]   data_a, data_b, data_c;
  input       [1: 0] sel;

  assign mux_out = (sel == 0) ? data_a: (sel == 1) ? data_b : (sel == 2) ? data_c: 'bx;
endmodule
```

The ALU is modeled as combinational logic described by a level-sensitive cyclic behavior that is activated whenever the datapaths or the select bus change. Parameters are used to make the description more readable and to reduce the likelihood of a coding error.

```
/*ALU Instruction           Action
ADD                         Adds the datapaths to form data_1 + data_2.
SUB                         Subtracts the datapaths to form data_1 - data_2.
AND                         Takes the bitwise-and of the datapaths, data_1 & data_2.
NOT                         Takes the bitwise Boolean complement of data_1.
*/
// Note: the carries are ignored in this model.

module Alu_RISC (alu_zero_flag, alu_out, data_1, data_2, sel);
  parameter word_size = 8;
  parameter op_size = 4;
  // Opcodes
  parameter NOP           = 4'b0000;
  parameter ADD           = 4'b0001;
  parameter SUB           = 4'b0010;
  parameter AND           = 4'b0011;
  parameter NOT           = 4'b0100;
  parameter RD            = 4'b0101;
  parameter WR            = 4'b0110;
  parameter BR            = 4'b0111;
  parameter BRZ           = 4'b1000;
  output                  alu_zero_flag;
  output [word_size-1: 0] alu_out;
  input  [word_size-1: 0] data_1, data_2;
  input  [op_size-1: 0]   sel;
  reg                     alu_out;

  assign alu_zero_flag = ~|alu_out;
  always @ (sel or data_1 or data_2)
    case (sel)
     NOP:        alu_out = 0;
     ADD:        alu_out = data_1 + data_2; // Reg_Y + Bus_1
     SUB:        alu_out = data_2 - data_1;
     AND:        alu_out = data_1 & data_2;
     NOT:        alu_out = ~ data_2;          // Gets data from Bus_1
     default     alu_out = 0;
    endcase
endmodule
```

The control unit is rather large, but its design has a simple form, and its development follows directly from the ASM charts in Figure 7-12. First, declarations are made for the ports and variables needed to support the description. Then the datapath multiplexers are described with nested continuous assignments using the conditional (*?...:*) operator. Two cyclic behaviors are used: a level-sensitive behavior describes the combinational logic of the outputs and the next state, and an edge-sensitive behavior synchronizes the clock transitions.

```
module Control_Unit (
  Load_R0, Load_R1,
  Load_R2, Load_R3,
  Load_PC, Inc_PC,
  Sel_Bus_1_Mux, Sel_Bus_2_Mux,
  Load_IR, Load_Add_R, Load_Reg_Y, Load_Reg_Z,
  write, instruction, zero, clk, rst);

  parameter word_size = 8, op_size = 4, state_size = 4;
  parameter src_size = 2, dest_size = 2, Sel1_size = 3, Sel2_size = 2;
  // State Codes
  parameter S_idle = 0, S_fet1 = 1, S_fet2 = 2, S_dec = 3;
  parameter  S_ex1 = 4, S_rd1 = 5, S_rd2 = 6;
  parameter S_wr1 = 7, S_wr2 = 8, S_br1 = 9, S_br2 = 10, S_halt = 11;
  // Opcodes
  parameter NOP = 0, ADD = 1, SUB = 2, AND = 3, NOT = 4;
  parameter RD  = 5, WR = 6,  BR = 7, BRZ = 8;
  // Source and Destination Codes
  parameter R0 = 0, R1 = 1, R2 = 2, R3 = 3;

  output Load_R0, Load_R1, Load_R2, Load_R3;
  output Load_PC, Inc_PC;
  output [Sel1_size-1: 0] Sel_Bus_1_Mux;
  output Load_IR, Load_Add_R;
  output Load_Reg_Y, Load_Reg_Z;
  output [Sel2_size-1: 0] Sel_Bus_2_Mux;
  output write;
  input [word_size-1: 0] instruction;
  input zero;
  input clk, rst;

  reg [state_size-1: 0] state, next_state;
  reg Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC;
  reg Load_IR, Load_Add_R, Load_Reg_Y;
  reg Sel_ALU, Sel_Bus_1, Sel_Mem;
  reg Sel_R0, Sel_R1, Sel_R2, Sel_R3, Sel_PC;
  reg Load_Reg_Z, write;
  reg err_flag;

  wire [op_size-1: 0] opcode = instruction [word_size-1: word_size - op_size];
  wire [src_size-1: 0] src = instruction [src_size + dest_size -1: dest_size];
  wire [dest_size-1: 0] dest = instruction [dest_size -1: 0];
```

```
// Mux selectors
assign Sel_Bus_1_Mux[Sel1_size-1: 0] = Sel_R0 ? 0:
                                     Sel_R1 ? 1:
                                     Sel_R2 ? 2:
                                     Sel_R3 ? 3:
                                     Sel_PC ? 4: 3'bx;  // 3-bits, sized number
assign Sel_Bus_2_Mux[Sel2_size-1: 0] = Sel_ALU ? 0:
                                     Sel_Bus_1 ? 1:
                                     Sel_Mem ? 2: 2'bx;

always @ (posedge clk or negedge rst) begin: State_transitions
 if (rst == 0) state <= S_idle; else state <= next_state; end

/* always @ (state or instruction or zero) begin: Output_and_next_state

Note: The above event control expression leads to incorrect operation. The state tran-
sition causes the activity to be evaluated once, then the resulting instruction change
causes it to be evaluated again, but with the residual value of opcode. On the second
pass the value seen is the value opcode had before the state change, which results in
Sel_PC = 0 in state 3, which will cause a return to state 1 at the next clock. Finally,
opcode is changed, but this does not trigger a re-evaluation because it is not in the
event control expression. So, the caution is to be sure to use opcode in the event con-
trol expression. That way, the final execution of the behavior uses the value of opcode
that results from the state change, and leads to the correct value of Sel_PC.
*/

always @ (state or opcode or src or dest or zero) begin: Output_and_next_state
 Sel_R0 = 0;   Sel_R1 = 0;   Sel_R2 = 0;   Sel_R3 = 0;     Sel_PC = 0;
 Load_R0 = 0; Load_R1 = 0;  Load_R2 = 0;  Load_R3 = 0;    Load_PC = 0;

 Load_IR = 0;  Load_Add_R = 0; Load_Reg_Y = 0; Load_Reg_Z = 0;
 Inc_PC = 0;
 Sel_Bus_1 = 0;
 Sel_ALU = 0;
 Sel_Mem = 0;
 write = 0;
 err_flag = 0;  // Used for de-bug in simulation
 next_state = state;

 case (state)  S_idle:          next_state = S_fet1;
               S_fet1:          begin
                                 next_state = S_fet2;
                                 Sel_PC = 1;
                                 Sel_Bus_1 = 1;
                                 Load_Add_R = 1;
                                end
               S_fet2:          begin
                                 next_state = S_dec;
                                 Sel_Mem = 1;
                                 Load_IR = 1;
                                 Inc_PC = 1;
                                end
```

```
S_dec:          case (opcode)
                 NOP: next_state = S_fet1;
                 ADD, SUB, AND: begin
                  next_state = S_ex1;
                  Sel_Bus_1 = 1;
                  Load_Reg_Y = 1;
                  case (src)
                   R0:          Sel_R0 = 1;
                   R1:          Sel_R1 = 1;
                   R2:          Sel_R2 = 1;
                   R3:          Sel_R3 = 1;
                   default      err_flag = 1;
                  endcase
                 end // ADD, SUB, AND

                 NOT: begin
                  next_state = S_fet1;
                  Load_Reg_Z = 1;
                  Sel_Bus_1 = 1;
                  Sel_ALU = 1;
                  case (src)
                   R0:          Sel_R0 = 1;
                   R1:          Sel_R1 = 1;
                   R2:          Sel_R2 = 1;
                   R3:          Sel_R3 = 1;
                   default      err_flag = 1;
                  endcase
                  case (dest)
                   R0:          Load_R0 = 1;
                   R1:          Load_R1 = 1;
                   R2:          Load_R2 = 1;
                   R3:          Load_R3 = 1;
                   default      err_flag = 1;
                  endcase
                 end // NOT

                 RD: begin
                  next_state = S_rd1;
                  Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                 end // RD

                 WR: begin
                  next_state = S_wr1;
                  Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                 end // WR

                 BR: begin
                  next_state = S_br1;
                  Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                 end // BR
```

```
                BRZ: if (zero == 1) begin
                  next_state = S_br1;
                  Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
                end // BRZ
                else begin
                  next_state = S_fet1;
                  Inc_PC = 1;
                end
                default : next_state = S_halt;
              endcase // (opcode)

  S_ex1:      begin
                next_state = S_fet1;
                Load_Reg_Z = 1;
                Sel_ALU = 1;
                case (dest)
                  R0: begin Sel_R0 = 1; Load_R0 = 1; end
                  R1: begin Sel_R1 = 1; Load_R1 = 1; end
                  R2: begin Sel_R2 = 1; Load_R2 = 1; end
                  R3: begin Sel_R3 = 1; Load_R3 = 1; end
                  default : err_flag = 1;
               endcase
              end

  S_rd1:      begin
                next_state = S_rd2;
                Sel_Mem = 1;
                Load_Add_R = 1;
                Inc_PC = 1;
              end

  S_wr1:      begin
                next_state = S_wr2;
                Sel_Mem = 1;
                Load_Add_R = 1;
                Inc_PC = 1;
              end

  S_rd2:      begin
                next_state = S_fet1;
                Sel_Mem = 1;
                case (dest)
                  R0:          Load_R0 = 1;
                  R1:          Load_R1 = 1;
                  R2:          Load_R2 = 1;
                  R3:          Load_R3 = 1;
                  default      err_flag = 1;
                endcase
              end
```

```
                         S_wr2:      begin
                                       next_state = S_fet1;
                                       write = 1;
                                       case (src)
                                         R0:          Sel_R0 = 1;
                                         R1:          Sel_R1 = 1;
                                         R2:          Sel_R2 = 1;
                                         R3:          Sel_R3 = 1;
                                        default       err_flag = 1;
                                       endcase
                                     end

                         S_br1:      begin next_state = S_br2; Sel_Mem = 1;
                                     Load_Add_R = 1; end
                         S_br2:      begin next_state = S_fet1; Sel_Mem = 1;
                                     Load_PC = 1; end
                         S_halt:     next_state = S_halt;
                         default:    next_state = S_idle;
   endcase
  end
endmodule
```

For simplicity, the memory unit of the machine is modeled as an array of D-type flip-flops.

```
module Memory_Unit (data_out, data_in, address, clk, write);
 parameter word_size = 8;
 parameter memory_size = 256;
 output [word_size-1: 0] data_out;
 input [word_size-1: 0] data_in;
 input [word_size-1: 0] address;
 input clk, write;
 reg [word_size-1: 0] memory [memory_size-1: 0];

 assign data_out = memory[address];

 always @ (posedge clk)
  if (write) memory[address] <= data_in;
endmodule
```

7.3.6 RISC SPM: Program Execution

A testbench for verifying that *RISC_SPM* executes a stored program[8] is given below. *test_RISC_SPM* defines probes to display individual words in memory, uses a one-shot

[8] An assembler for the machine is located at the website for this book, and can be used to generate programs for use in embedded applications of the processor.

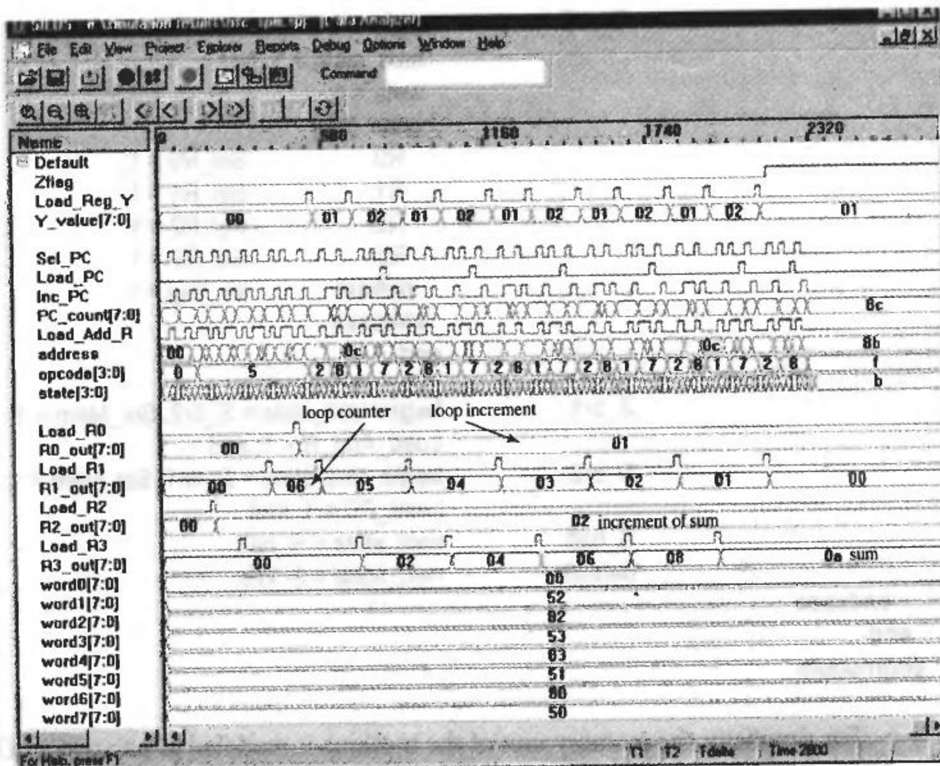


FIGURE 7-13 Simulation results produced by executing a stored program with *RISC_SPM*.

(***initial***) behavior to flush memory, and loads a small program and data into separate areas of memory. The program (1) reads memory and loads the data into the registers of the processor, (2) executes subtraction to decrement a loop counter, (3) adds register contents while executing the loop, and (4) branches to a halt when the loop index is 0. The results of executing the program are displayed in Figure 7-13.

```
module test_RISC_SPM ();
  reg rst;
  wire clk;
  parameter word_size = 8;
  reg [8: 0] k;

  Clock_Unit M1 (clk);
  RISC_SPM M2 (clk, rst);

// define probes
  wire [word_size-1: 0] word0, word1, word2, word3, word4, word5, word6;
  wire [word_size-1: 0] word7, word8, word9, word10, word11, word12, word13;
  wire [word_size-1: 0] word14;

  wire [word_size-1: 0] word128, word129, word130, word131, word132, word255;
  wire [word_size-1: 0] word133, word134, word135, word136, word137;
```

```
wire [word_size-1: 0] word138, word139, word140;
assign word0 = M2.M2_SRAM.memory[0];
assign word1 = M2.M2_SRAM.memory[1];
assign word2 = M2.M2_SRAM.memory[2];
assign word3 = M2.M2_SRAM.memory[3];
assign word4 = M2.M2_SRAM.memory[4];
assign word5 = M2.M2_SRAM.memory[5];
assign word6 = M2.M2_SRAM.memory[6];
assign word7 = M2.M2_SRAM.memory[7];
assign word8 = M2.M2_SRAM.memory[8];
assign word9 = M2.M2_SRAM.memory[9];
assign word10 = M2.M2_SRAM.memory[10];
assign word11 = M2.M2_SRAM.memory[11];
assign word12 = M2.M2_SRAM.memory[12];
assign word13 = M2.M2_SRAM.memory[13];
assign word14 = M2.M2_SRAM.memory[14];

assign word128 = M2.M2_SRAM.memory[128];
assign word129 = M2.M2_SRAM.memory[129];
assign word130 = M2.M2_SRAM.memory[130];
assign word131 = M2.M2_SRAM.memory[131];
assign word132 = M2.M2_SRAM.memory[132];
assign word133 = M2.M2_SRAM.memory[133];
assign word134 = M2.M2_SRAM.memory[134];
assign word135 = M2.M2_SRAM.memory[135];
assign word136 = M2.M2_SRAM.memory[136];
assign word137 = M2.M2_SRAM.memory[137];
assign word138 = M2.M2_SRAM.memory[138];
assign word139 = M2.M2_SRAM.memory[139];
assign word140 = M2.M2_SRAM.memory[140];

assign word255 = M2.M2_SRAM.memory[255];

initial #2800 $finish;

Flush Memory

initial begin: Flush_Memory
  #2 rst = 0; for (k=0; k<=255; k=k+1)M2.M2_SRAM.memory[k] = 0; #10 rst = 1;
end

initial begin: Load_program
  #5
                          // opcode_src_dest
  M2.M2_SRAM.memory[0] = 8'b0000_00_00;       // NOP
  M2.M2_SRAM.memory[1] = 8'b0101_00_10;       // Read 130 to R2
  M2.M2_SRAM.memory[2] = 130;
```

```
M2.M2_SRAM.memory[3] = 8'b0101_00_11;      // Read 131 to R3
M2.M2_SRAM.memory[4] = 131;
M2.M2_SRAM.memory[5] = 8'b0101_00_01;      // Read 128 to R1
M2.M2_SRAM.memory[6] = 128;
M2.M2_SRAM.memory[7] = 8'b0101_00_00;      // Read 129 to R0
M2.M2_SRAM.memory[8] = 129;

M2.M2_SRAM.memory[9] = 8'b0010_00_01;      // Sub R1-R0 to R1

M2.M2_SRAM.memory[10] = 8'b1000_00_00;     // BRZ
M2.M2_SRAM.memory[11] = 134;               // Holds address for BRZ

M2.M2_SRAM.memory[12] = 8'b0001_10_11;     // Add R2+R3 to R3
M2.M2_SRAM.memory[13] = 8'b0111_00_11;     // BR
M2.M2_SRAM.memory[14] = 140;
// Load data
M2.M2_SRAM.memory[128] = 6;
M2.M2_SRAM.memory[129] = 1;
M2.M2_SRAM.memory[130] = 2;
M2.M2_SRAM.memory[131] = 0;
M2.M2_SRAM.memory[134] = 139;
//M2.M2_SRAM.memory[135] = 0;
M2.M2_SRAM.memory[139] = 8'b1111_00_00;    // HALT
M2.M2_SRAM.memory[140] = 9;                // Recycle
end
endmodule
```

7.4 Design Example: UART

Systems that exchange information and interact via serial data channels use modems as interfaces between the host machines/devices and the channel, as shown in Figure 7-14. For example, a modem allows a computer to connect to a telephone line and communicate with a receiving computer through its modem [2, 5]. The host machine stores information in a parallel word format, but transmits and receives data in a serial, single-bit, format. A modem is also called a UART, or *universal asynchronous receiver and*

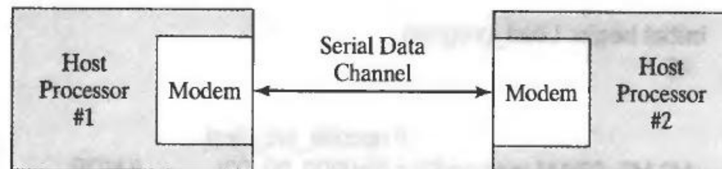


FIGURE 7-14 Processor/modem communication over a serial channel.

Stop Bit	Parity Bit	Data Bit 7	Data Bit 6	Data Bit 5	Data Bit 4	Data Bit 3	Data Bit 2	Data Bit 1	Data Bit 0	Start Bit

FIGURE 7-15 Data format for ASCII text transmitted by a UART.

transmitter, indicating that the device has the capability to both receive and transmit serial data. This design example will address the modeling and synthesis of a UART's transmitter and receiver.

For this discussion, a UART exchanges text data in an American Standard Code for Information Interchange (ASCII) format in which each alphabetical character is encoded by 7 bits and augmented by a parity bit that can be used for error detection. For transmission, the modem wraps this 8-bit subword with a *start-bit* in the least significant bit (LSB), and a *stop-bit* in the most significant bit (MSB), resulting in the 10-bit word format shown in Figure 7-15. The first 9 data bits are transmitted in sequence, beginning with the start-bit, with each bit being asserted at the serial line for one cycle of the modem clock. The stop-bit may assert for more than one clock.

7.4.1 UART Operation

The UART transmitter is always part of larger environment in which a host processor controls transmission by fetching a data word in parallel format and directing the UART to transmit it in a serial format. Likewise, the receiver must detect transmission, receive the data in serial format, strip off the start- and stop-bits, and store the data word in a parallel format. The receiver's job is more complex, because the clock used to send the inbound data is not available at the remote receiver. The receiver must regenerate the clock locally, using the receiving machine's clock rather than the clock of the transmitting machine.

The simplified architecture of a UART presented in Figure 7-16 shows the signals used by a host processor to control the UART and to move data to and from a data bus in the host machine. Details of the host machine are not shown.

7.4.2 UART Transmitter

The input–output signals of the transmitter are shown in the high-level block diagram in Figure 7-17. The input signals are provided by the host processor, and the output signals control the movement of data in the UART. The architecture of the transmitter will consist of a controller, a data register (*XMT_datareg*), a data shift register (*XMT_shftreg*), and a status register (*bit_count*) to count the bits that are transmitted. The status register will be included with the datapath unit.

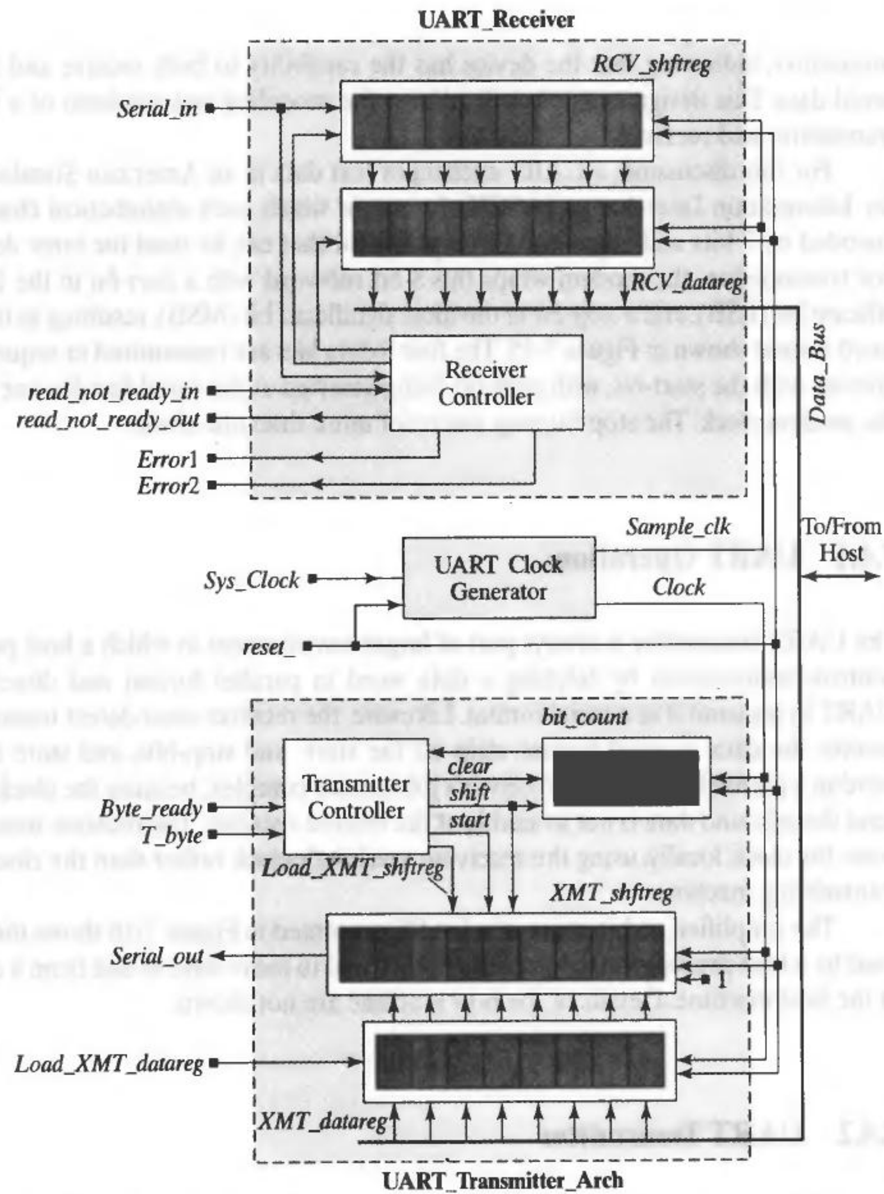


FIGURE 7-16 Block diagram of a UART.

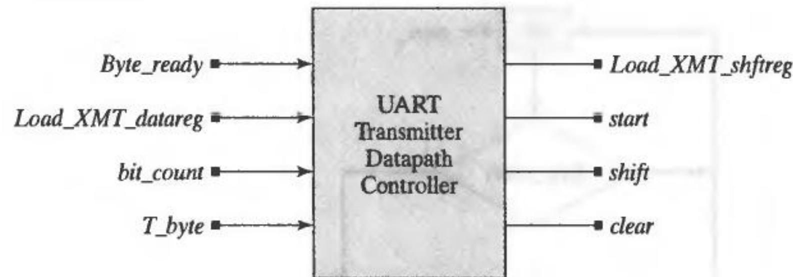


FIGURE 7-17 Interface signals of a state machine controller for a UART transmitter.

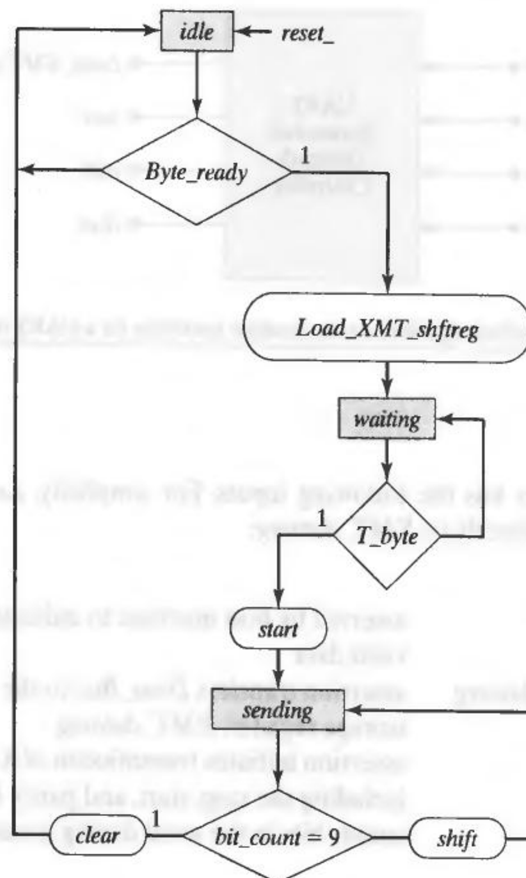
The controller has the following inputs. For simplicity, *Load_XMT_datareg* is shown connected directly to *XMT_datareg*:

Byte_ready	asserted by host machine to indicate that *Data_Bus* has valid data
Load_XMT_datareg	assertion transfers *Data_Bus* to the transmitter data storage register, *XMT_datareg*
T_byte	assertion initiates transmission of a byte of data, including the stop, start, and parity bits
bit_count	counts bits in the word during transmission

The state machine of the controller forms the following output signals that control the datapath of the transmitter:

Load_XMT_shftreg	assertion loads the contents of *XMT_data_reg* into *XMT_shftreg*
start	signals the start of transmission
shift	directs *XMT_shftreg* to shift by one bit towards the LSB and to backfill with a stop bit (1).
clear	clears *bit_count*

The ASM chart of the state machine controlling the transmitter is shown in Figure 7.18. The machine has three states: *idle, waiting*, and *sending*. When *reset_* is asserted, the machine asynchronously enters *idle, bit_count* is flushed, *XMT_shftreg* is loaded with 1s, and the control signals *clear, Load_XMT_shftreg, shift*, and *start* are driven to 0. In *idle*, if an active edge of *Clock* occurs while *Load_XMT_data_reg* is asserted by the external host, the contents of *Data_Bus* will transfer to *XMT_data_reg*. (This action is not part of the ASM chart because it occurs independently of the state of the machine.) The machine remains in *idle* until *start* is asserted.



Note: Only the branch corresponding to a true decision is annotated at a decision diamond; signals that are not shown explicitly asserted are de-asserted. Conditional assertions are indicated by the name of the asserted signal

FIGURE 7-18 ASM chart for the state machine controller for the UART transmitter.

When *Byte_ready* is asserted, *Load_XMT_shftreg* is asserted and *next_state* is driven to *waiting*. The assertion of *Load_XMT_shftreg* indicates that *XMT_datareg* now contains data that can be transferred to the internal shift register. At the next active edge of *Clock*, with *Load_XMT_shftreg* asserted, three activities occur: (1) *state* transfers from *idle* to *waiting*, (2) the contents of *XMT_datareg* are loaded into the leftmost bits of *XMT_shftreg*, a (*word_size* + 1)-bit shift register whose LSB signals the start and stop of transmission, and (3) the LSB of *XMT_shftreg* is reloaded with 1, the stop-bit. The machine remains in *waiting* until the external processor asserts *T_byte*.

At the next active edge of *Clock*, with *T_byte* asserted, *state* enters *sending*, and the LSB of *XMT_shftreg* is set to 0 to signal the start of transmission. At the same time,

shift is driven to 1, and *next_state* retains the state code corresponding to *sending*. At subsequent active edges of *Clock*, with *shift* asserted, *state* remains in *sending* and the contents of *XMT_shftreg* are shifted toward the LSB, which drives the external serial channel. As the data shifts occur, 1s are back-filled in *XMT_shftreg*, and *bit_count* is incremented. With *state* in *sending, shift* asserts while *bit_count* is less than 9. The machine increments *bit_count* after each movement of data, and when *bit_count* reaches 9 *clear* asserts, indicating that all of the bits of the augmented word have been shifted to the serial output. At the next active edge of *Clock*, the machine returns to *idle*.

The control signals produced by the state machine induce state-dependent register transfers in the data path. The activity of the primary inputs (*Byte_ready, Load_XMT_datareg*, and *T_byte*), and the signals from the controller (*Load_XMT_shftreg, start, shift, clear*) are shown in Figure 7-19, along with the movement of data in the datapath registers. The contents of the registers are shown at successive edges of clock, with a time axis going from the top of the figure toward the bottom. Transitions of the active edge of *clock* occur between the successive rows displaying contents of *XMT_datareg*. The bits of the transmitted signal are shown in the sequence in which they are transmitted, with the rightmost cell of *XMT_shftreg* holding the bit that is transmitted at the serial interface at each step. The state of the machine is shown, and the state transitions and datapath register transitions that occur on the rising edges of *clock* are shown in the register boxes. The values of the control signals that cause the register transitions are also shown. The displayed values of the control signals are those they held immediately before the active edge of *Clock*; and which cause the register transfers that are shown. The sequence of output bits is also shown, with 1s being pushed into the MSB of *XMT_shftreg* under the action of *shift*. The sequence of output bits of the transmitted signal are shown as a word at each time step, with the understanding that *the LSB of the word is the first bit that was transmitted*, and the MSB of the word is the most recent bit that was transmitted at the serial interface.

The Verilog description, *UART_Transmitter_Arch*, of the architecture for the transmitter has three cyclic behaviors—a level-sensitive behavior describing the combinational logic for next state and outputs of the controller, an edge-sensitive behavior to synchronize the state transitions of the controller, and another edge-sensitive behavior to synchronize the register transfers of the datapath registers. For simplicity, we include the entire description in a single Verilog module, rather than impose architectural boundaries around the datapath and the controller (see Figure 7-20). The module can be partitioned and synthesized into the individual functional units.

Some simulation results are shown in Figure 7-21 and Figure 7-22 for an 8-bit data word. The waveforms produced by the simulator have been annotated to indicate significant features of the transmitter's behavior. First, observe the values of the signals immediately after *reset_* is asserted. The state is *idle*. Note that *Data_Bus* initially contains the value $a7_h$ (1010_0111_2), a value specified by the testbench used for simulation. With *Byte_ready* not yet asserted, and with *Load_XMT_datareg* asserted, the *Data_Bus* is loaded into *XMT_datareg*. The machine remains in *idle* until *Byte_ready* is asserted. When *Byte_ready* asserts, then *Load_XMT_shftreg* asserts. This causes the

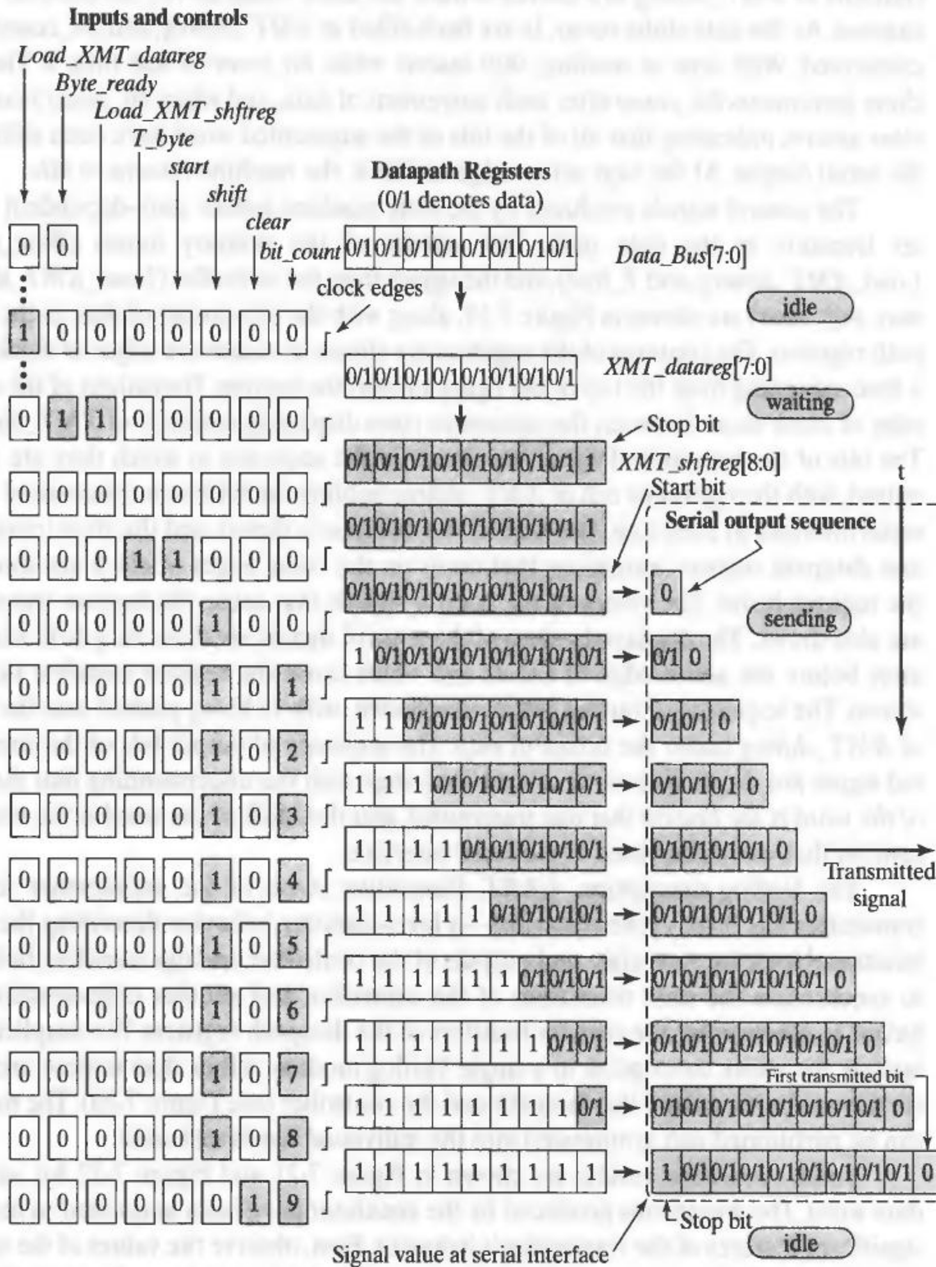


FIGURE 7-19 Control signals and dataflow in an 8-bit UART transmitter.

```
module UART_Transmitter_Arch
  (Serial_out, Data_Bus, Byte_ready, Load_XMT_datareg, T_byte, Clock, reset_);
  parameter     word_size = 8;                     // Size of data word, e.g., 8 bits
  parameter     one_hot_count = 3;                 // Number of one-hot states
  parameter     state_count = one_hot_count;       // Number of bits in state register
  parameter     size_bit_count = 3;                // Size of the bit counter, e.g., 4
                                                   // Must count to word_size + 1
  parameter     idle = 3'b001;                     // one-hot state encoding
  parameter     waiting = 3'b010;
  parameter     sending = 3'b100;
  parameter     all_ones = 9'b1_1111_1111;         // Word + 1 extra bit

  output                      Serial_out           // Serial output to data channel
  input  [word_size - 1:0]    Data_Bus;            // Host data bus containing data word
  input                       Byte_ready;          // Used by host to signal ready
  input                       Load_XMT_datareg;    // Used by host to load the data register
  input                       T_byte;              // Used by host to signal the start of transmission
  input                       Clock;               // Bit clock of the transmitter
  input                       reset_;              // Resets internal registers, loads the
                                                   // XMT_shftreg with ones

  reg  [word_size - 1:0]      XMT_datareg;         // Transmit Data Register
  reg  [word_size:0]          XMT_shftreg;         // Transmit Shift Register: {data, start bit}
  reg                         Load_XMT_shftreg;    // Flag to load the XMT_shftreg
  reg  [state_count - 1:0]    state, next_state;   // State machine controller
  reg  [size_bit_count:0]     bit_count;           // Counts the bits that are transmitted
  reg                         clear;               // Clears bit_count after last bit is sent
  reg                         shift;               // Causes shift of data in XMT_shftreg
  reg                         start;               // Signals start of transmission

  assign Serial_out = XMT_shftreg[0];              // LSB of shift register

  always @ (state or Byte_ready or bit_count or T_byte) begin: Output_and_next_state
    Load_XMT_shftreg = 0;
    clear = 0;
    shift = 0;
    start = 0;
    next_state = state;
    case (state)
      idle:        if (Byte_ready == 1) begin
                     Load_XMT_shftreg = 1;
                     next_state = waiting;
                   end

      waiting:     if (T_byte == 1) begin
                     start = 1;
                     next_state = sending;
                   end

      sending:     if (bit_count != word_size + 1)
                     shift = 1;
                   else begin
                     clear = 1;
                     next_state = idle;
                   end

      default:     next_state = idle;
    endcase
  end
```

FIGURE 7-20 Verilog Description of the UART transmitter.

```
always @ (posedge Clock or negedge reset_) begin: State_Transitions
  if (reset_ == 0) state <= idle; else state <= next_state; end

always @ (posedge Clock or negedge reset_) begin: Register_Transfers
  if (reset_ == 0) begin
    XMT_shftreg <= all_ones;
    bit_count <= 0;
  end
  else begin
    if (Load_XMT_datareg == 1)
      XMT_datareg <= Data_Bus;                      // Get the data bus

    if (Load_XMT_shftreg == 1)
      XMT_shftreg <= {XMT_datareg, 1'b1};           // Load shift reg,
                                                    // insert stop bit
    if (start == 1)
      XMT_shftreg[0] <= 0;                          // Signal start of transmission

    if (clear == 1) bit_count <= 0;
    else if (shift == 1) bit_count <= bit_count + 1;

    if (shift == 1)
      XMT_shftreg <= {1'b1, XMT_shftreg[word_size:1]};   // Shift right, fill with 1's
    end
  end
endmodule
```

FIGURE 7-20 Continued

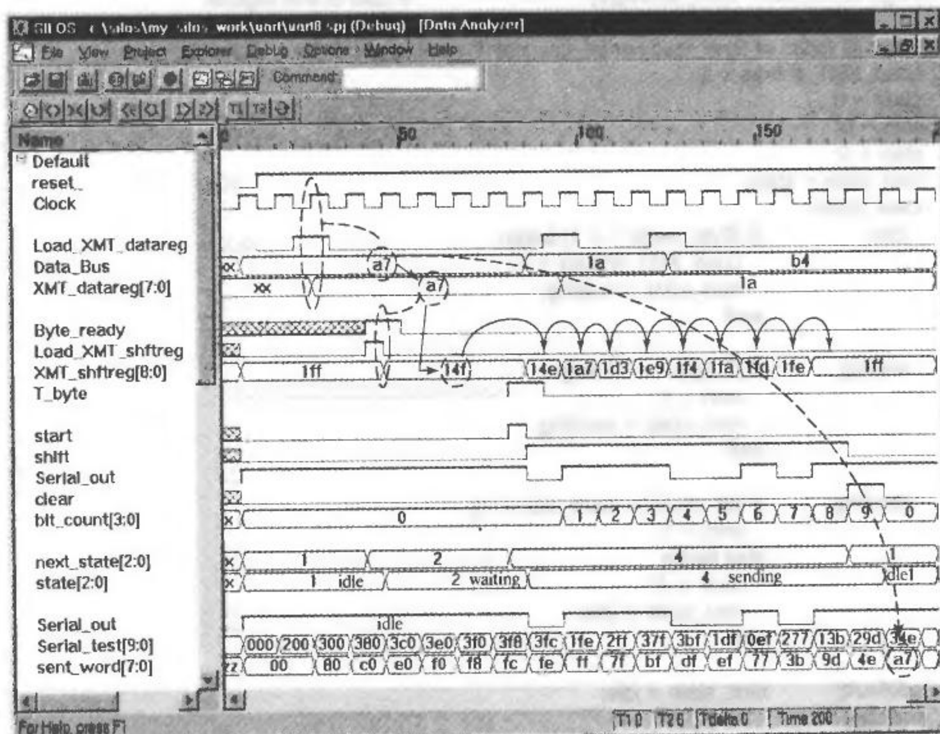


FIGURE 7-21 Annotated simulation results for the 8-bit UART transmitter.

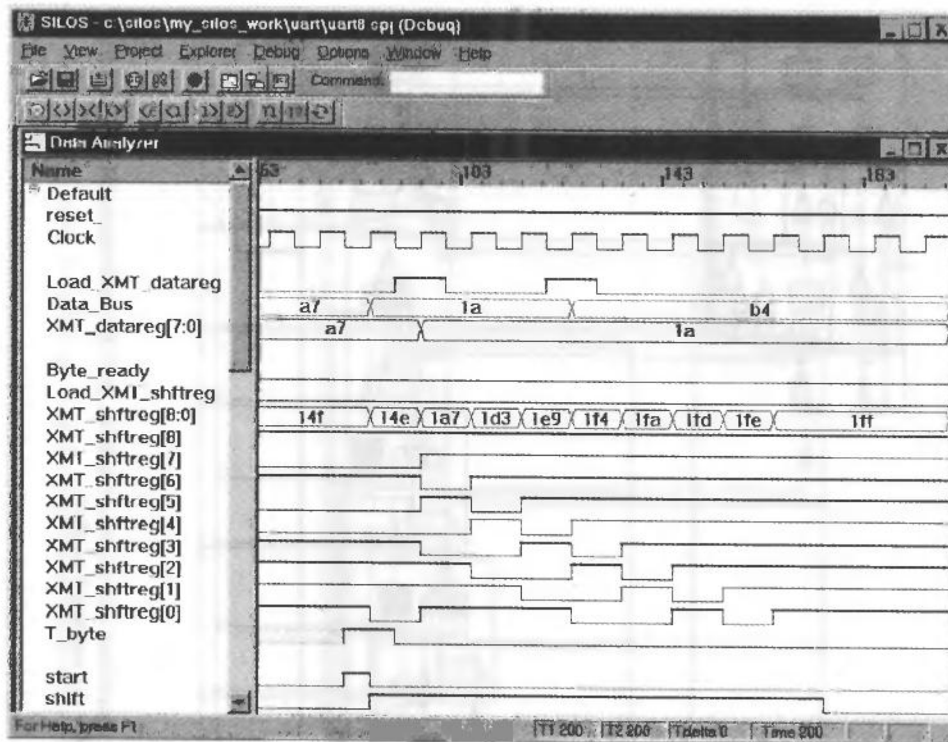


FIGURE 7-22 Data movement through *XMT_shftreg*.

state to change to *waiting* at the next active edge of *Clock*. The 9-bit *XMT_shftreg* is now loaded with the value $\{a7_h, 1\} = 1_0100_1111_2 = 14f_h$. Note that the LSB of *XMT_shftreg* is loaded with a 1. The machine remains in *waiting* until *T_byte* is asserted. The assertion of *T_byte* asserts *start*. The machine enters *sending* at the active edge of *Clock* immediately after the host processor's assertion of *T_byte*, and the LSB of *XMT_shftreg* is loaded with a 0. The 9-bit word in *XMT_shftreg* becomes $1_0100_1110_2 = 14e_h$. The 0 in the LSB signals the start of transmission. Figure 7-21 has been annotated to show the movement of data through *XMT_shftreg*. Note that 1s are filled behind, as the word shifts to the right. At the active edge of the clock after *bit_count* reaches 9 (for an 8-bit word), *clear* asserts, *bit_count* is flushed, and the machine returns to *idle*.

For diagnostic purposes, the testbench includes a 10-bit shift register that receives *Serial_out* (by hierarchical dereferencing). The eight innermost bits of this register are displayed in Figure 7-21 as *sent_word[7: 0]* (skipping the start-bit and the stop-bit) to reveal the correct transmission of data, $a7_h$. The bit sequence of *Serial_out* likewise has this value. This is evident at the active edge of the clock after the assertion of *clear*. The movement of data through *XMT_shftreg* is shown in Figure 7-22.

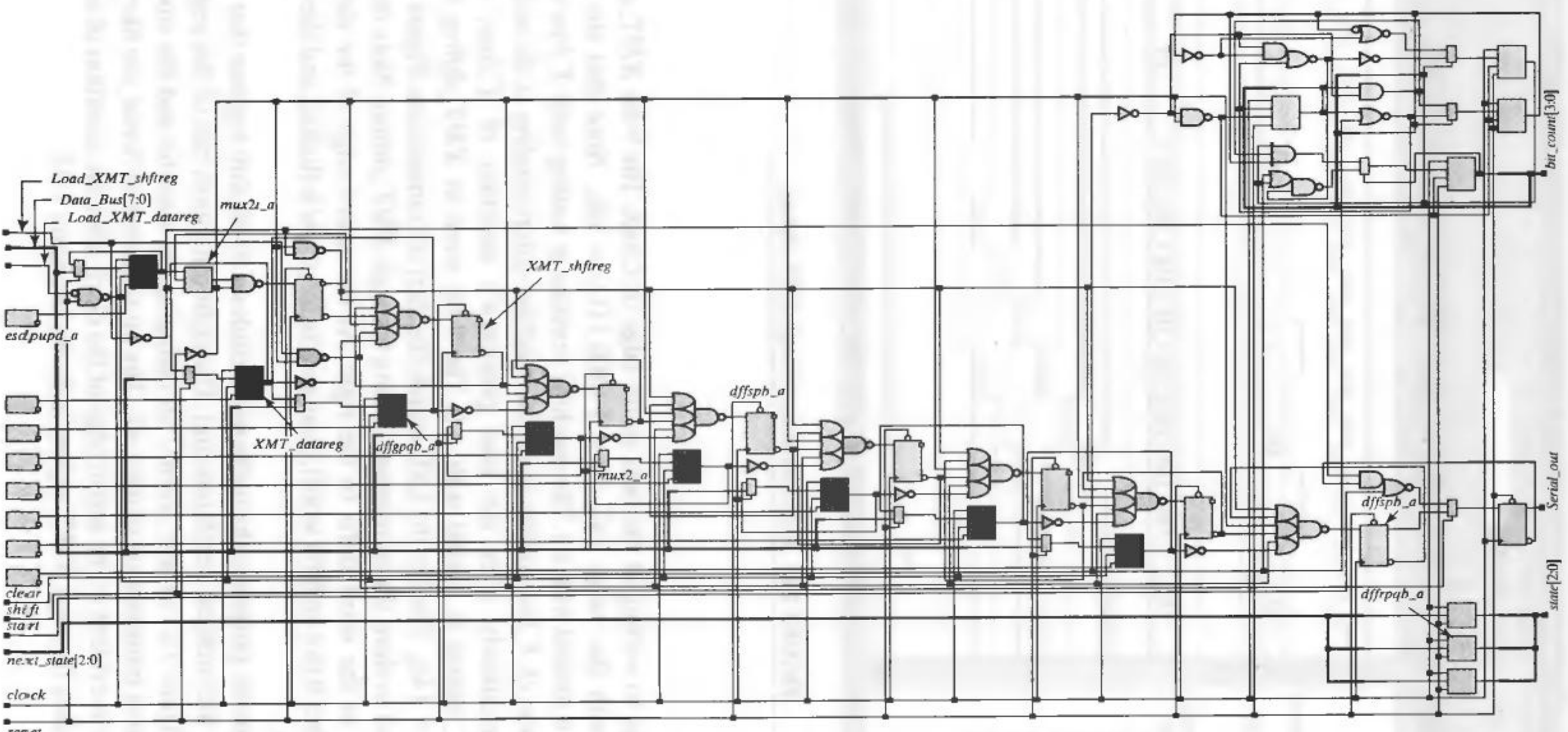


FIGURE 7-23 *UART_transmitter.* (a) logic synthesized to implement the state transitions and register transfers, and (b) combinational logic forming the next state and control signals for the register transfer.

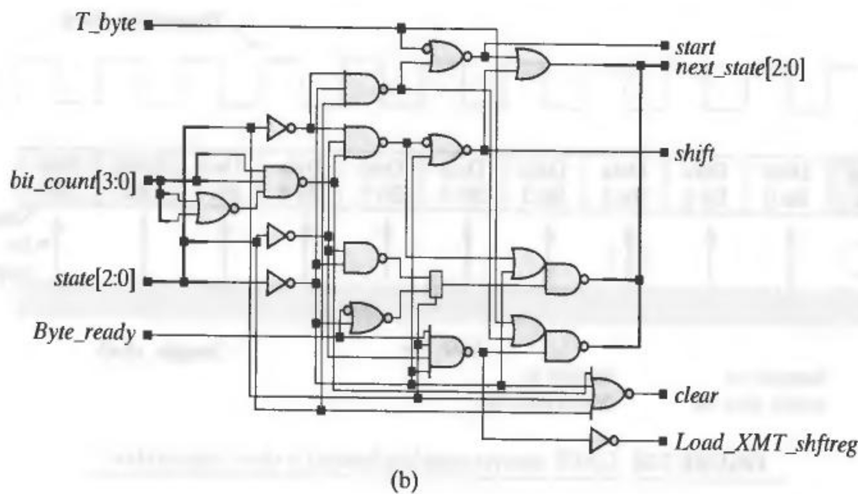


FIGURE 7-23 Continued

The circuit synthesized from *UART_transmitter* is shown in Figure 7-23. The Verilog model synthesizes as a unit, but for illustration and discussion the description was partitioned and synthesized in two parts, one for the state transitions and register transfers, and another for the combinational logic forming the next state and the control signals for the register transfers. The part governing the state transitions and register transfers consists of an 8-bit register holding *XMT_datareg*, a 9-bit shift register holding *XMT_shftreg*, and a bit counter. The circuit uses *dffrgpqb_a*, a D-type flip-flop with a rising-edge clock, asynchronous active-low reset, and internal gated data of the external data or the output, and *dffspb_a*, a D-type flip-flop with rising-edge clock, and asynchronous active-low set. The shift registers have been highlighted in Figure 7-23.

7.4.3 UART Receiver

The UART receiver has the task of receiving the serial bit stream of data, removing the start-bit, and transferring the data in a parallel format to a storage register connected to the host data bus. The data arrives at a standard bit rate, but it is not necessarily synchronized with the internal clock at the host of the receiver, and the transmitter's clock is not available to the receiver. This issue of synchronization is resolved by generating a *local* clock at a higher frequency and using it to sample the received data in a manner that preserves the integrity of the data. In the scheme used here, the data, assumed to be in a 10-bit format, will be sampled at a rate determined by *Sample_clock*, which is

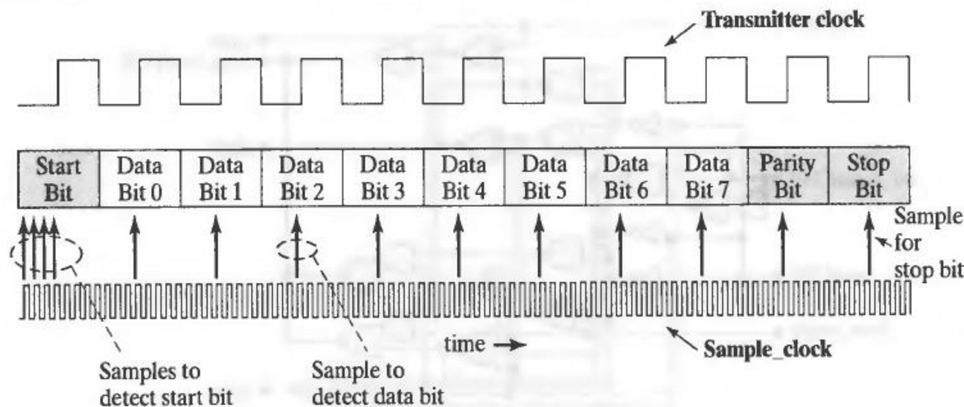


FIGURE 7-24 UART receiver sampling format for clock regeneration.

generated at the receiver's host. The cycles of *Sample_clock* will be counted to ensure that the data are sampled in the middle of a bit time, as shown in Figure 7-24. The sampling algorithm must (1) verify that a start bit has been received, (2) generate samples from 8 bits of the data, and (3) load the data onto the local bus.

Although a higher sampling frequency could be used, the frequency of *Sample_clock* in this example is 8 times the (known) frequency of the bit clock that transmitted the data. This ensures that a slight misalignment between the leading edge of a cycle of *Sample_clock* and the arrival of the start-bit will not compromise the sampling scheme, because the sample will still be taken within the interval of time corresponding to a transmitted bit. The arrival of a start-bit will be determined by successive samples of value 0 after the input data goes low. Then three additional samples will be taken to confirm that a valid start-bit has arrived. Thereafter, 8 successive bits will be sampled at approximately the center of their bit times. Under worst-case conditions of misalignment, the sample is taken a full cycle of *Sample_clock* ahead of the actual center of the bit time, which is a tolerable skew.

The high-level block diagram in Figure 7-25 shows the input–output signals of a state-machine controller that will interface with the host processor and direct the receiver's sampling scheme.

The state machine has the following inputs:

read_not_ready_in	signals that the host is not ready to receive data
Serial_in	serial bit stream received by the unit
*reset_*	active low reset
Sample_counter	counts the samples of a bit
Bit_counter	counts the bits that have been sampled

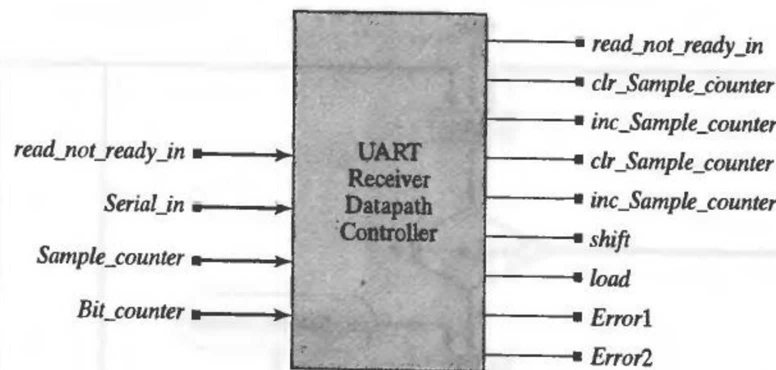


FIGURE 7-25 Interface signals of a state-machine controller for the UART receiver.

The state machine produces the following outputs:

read_not_ready_out	signals that the receiver has received 8 bits
inc_Sample_counter	increments *Sample_counter*
clr_Sample_counter	clears *Sample_counter*
inc_Bit_counter	increments *Bit_counter*
clr_Bit_counter	clears *Bit_counter*
shift	causes *RCV_shftreg* to shift towards the LSB
load	causes *RCV_shftreg* to transfer data to *RCV_datareg*
Error1	asserts if host is not ready to receive data after last bit has been sampled
Error2	asserts if the stop-bit is missing

The ASM chart of a state machine controller for the receiver is shown in Figure 7-26. The machine has three states: *idle, starting*, and *receiving*. Transitions between states are synchronized by *Sample_clk*. Assertion of an asynchronous active-low reset puts the machine in the *idle* state. It remains there until *Serial_in* is low, then makes a transition to *starting*. In *starting*, the machine samples *Serial_in* to determine whether the first bit is a valid start-bit (it must be 0). Depending on the sampled values, *inc_Sample_counter* and *clr_Sample_counter* may be asserted to increment or clear the counter at the next active edge of *Sample_clock*. If the next three samples of *Serial_in* are 0, the machine concludes that the start-bit is valid and goes to the state *receiving*. *Sample_counter* is cleared on the transition to *receiving*. In this state, eight successive samples are taken (one for each bit of the byte, at each active edge of *Sample_clk*), with *inc_Sample_counter* asserted. Then *Bit_counter* is incremented. If the sampled bit is not the last (parity) bit, *inc_Bit_counter* and *shift* are asserted. The assertion of *shift* will cause the sample value to be loaded into the MSB of *RCV_shftreg*, the receiver shift register, and will shift the 7 leftmost bits of the register toward the LSB.

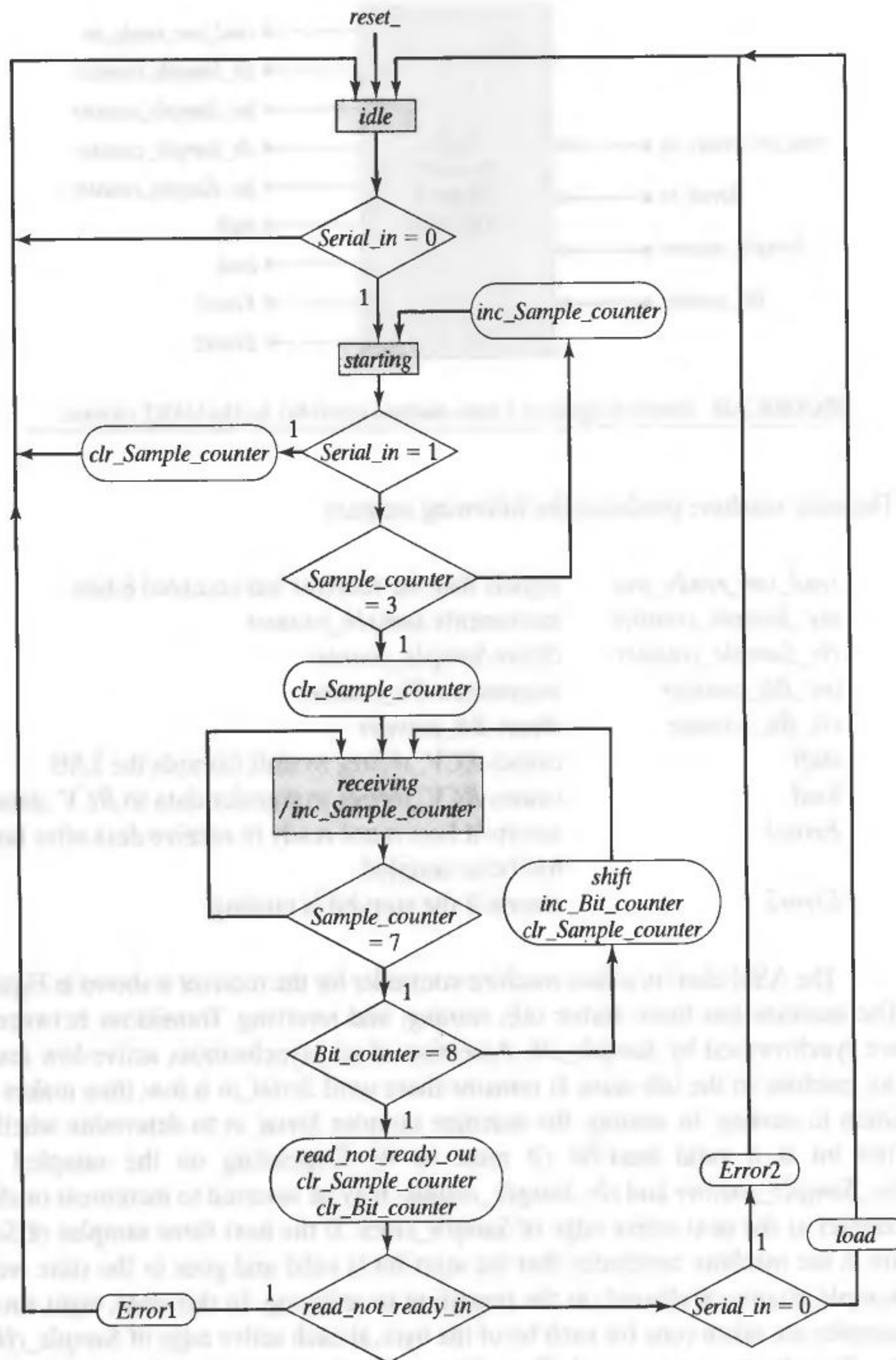


FIGURE 7-26 UART receiver ASM chart.

After the last bit has been sampled, the machine will assert *read_not_ready_out*, a handshake output signal to the processor, and clear the bit counter. At this time, the machine also checks the integrity of the data and the status of the host processor. If *read_not_ready_in* is asserted, the host processor is not ready to receive the data (*Error1*). If a stop-bit is not the next bit (detected by *Serial_in* = 0), there is an error in the format of the received data (*Error2*). Otherwise, *load* is asserted to cause the contents of the shift register to be transferred as a parallel word to *RCV_datareg*, a data register in the host machine, with a direct connection to *data_bus*.

The Verilog description of the 8-bit UART receiver is given in Figure 7-27. The description follows directly from the ASM chart in Figure 7-26.

The simulation results in Figure 7-28 are annotated to show functional features of the waveforms. The received data word is $b5_h = 1011_0101_2$. The reception sequence is from LSB to MSB, and the data move through the inbound shift register from MSB to LSB. The data word is preceded by a start-bit and followed by a stop-bit. With *reset_* having a value of 0, the state is *idle* and the counters are cleared. At the first active edge of *Sample_clock* after the reset condition is de-asserted, with *Serial_in* having a value of 0, the controller's state enters *starting* to determine whether a start-bit is being received. Three more samples of *serial_in* are taken, and after a total of four samples have been found to be 0, the *Sample_counter* is cleared and the state enters *receiving*. After the eighth sample, *shift* is asserted. The sample at the next active edge of the clock is shifted into the MSB of *RCV_shftreg*. The value of *RCV_shftreg* becomes $80_h = 1000_0000_2$. The sampling cycle repeats again, and a value of 0 is sampled and loaded into *RCV_shftreg*, changing the contents of the register to $0100_0000_2 = 40_h$.

The end of the sampling cycle of the received word is shown in Figure 7-29. After the last data bit is sampled, the machine samples once more to detect the stop bit. In the absence of an error, the contents of *RCV_shftreg* will be loaded into *RCV_datareg*. In this example, the value $b5_h$ is finally loaded from *RCV_shftreg* into *RCV_datareg*. Other tests can be conducted to completely verify the functionality of the receiver.

The Verilog description of the partitioned receiver in Figure 7-30 synthesizes into the circuit shown in Figure 7-31. Although the entire description synthesizes as a single unit, the structure of the synthesized result is revealed more easily by partitioning the description into the asynchronous (combinational) and synchronous (state transition) parts shown below. Note that the ports of the parent modules of a partition must be sized properly to accommodate vector ports in the child modules. Otherwise, the ports will be treated as default scalars in the scope of the parent module.

The state-transition part of the synthesized circuit includes *RCV_shftreg*, the shift register receiving *Serial_in*, and *RCV_datareg* (the 8-bit register holding the received word), *Sample_counter, Bit_counter* (the 4-bit counter that determines when all of the bits have been received), and the state register for the machine. Two types of flip-flops are used: the four-input *dffrgpqb_a* and the three-input *dffrpqb_a*. The former is a D-type flip-flop with internal gated data between the external datapath and the output, a rising clock, and an asynchronous active-low reset; the latter is a D-type flip-flop with data, rising clock, and asynchronous active-low reset. Only the state register uses the *dffrpqb_a* flip-flop.

```
module UART8_Receiver
(RCV_datareg, read_not_ready_out, Error1 ,Error2, Serial_in, read_not_ready_in, Sample_clk, reset_);
// Sample_clk is 8x Bit_clk

  parameter   word_size          = 8;
  parameter   half_word          = word_size/2;
  parameter   Num_counter_bits   = 4;              // Must hold count of word_size
  parameter   Num_state_bits     = 2;              // Number of bits in state
  parameter   idle               = 2'b00;
  parameter   starting           = 2'b01;
  parameter   receiving          = 2'b10;

  output      [word_size - 1:0]  RCV_datareg;
  output                         read_not_ready_out,
                                 Error1, Error2;

  input       Serial_in,
              Sample_clk,
              reset_,
              read_not_ready_in,

  reg                                RCV_datareg;
  reg         [word_size - 1:0]      RCV_shftreg;
  reg         [Num_counter_bits - 1:0]  Sample_counter;
  reg         [Num_counter_bits:0]   Bit_counter;
  reg         [Num_state_bits - 1:0] state, next_state;
  reg                                inc_Bit_counter, clr_Bit_counter;
  reg                                inc_Sample_counter, clr_Sample_counter;
  reg                                shift, load, read_not_ready_out;
  reg                                Error1, Error2;

//Combinational logic for next state and conditional outputs

  always @ (state or Serial_in or read_not_ready_in or Sample_counter or Bit_counter) begin
   read_not_ready_out = 0;
   clr_Sample_counter = 0;
   clr_Bit_counter = 0;
   inc_Sample_counter = 0;
   inc_Bit_counter = 0;
   shift = 0;
   Error1 = 0;
   Error2 = 0;
   load = 0;
   next_state = state;

   case (state)
    idle:        if (Serial_in == 0) next_state = starting;

    starting:    if (Serial_in == 1) begin
                   next_state = idle;
                   clr_Sample_counter = 1;
                 end else

                 if (Sample_counter == half_word - 1) begin
                   next_state = receiving;
                   clr_Sample_counter = 1;
                 end else inc_Sample_counter = 1;
```

FIGURE 7-27 Verilog description of UART8_Receiver, an 8-bit UART receiver.

```
receiving:      if (Sample_counter < word_ size - 1) inc_Sample_counter = 1;
                else begin
                  clr_Sample_counter = 1;
                  if (Bit_counter != word_size) begin
                    shift = 1;
                    inc_Bit_counter = 1;
                  end
                  else begin
                    next_state = idle;
                    read_not_ready_out = 1;
                    clr_Bit_counter = 1;
                    if (read_not_ready_in == 1) Error1 = 1;
                    else if (Serial_in == 0) Error2 = 1;
                    else load = 1;
                  end
                end
      default: next_state = idle;

  endcase
end

// state_transitions_and_register_transfers

  always @ (posedge Sample_clk) begin
    if (reset_ == 0) begin                          // synchronous reset_
      state <= idle;
      Sample_counter <= 0;
      Bit_counter <= 0;
      RCV_datareg <= 0;
      RCV_shftreg <= 0;
    end
    else begin
      state <= next_state;

      if (clr_Sample_counter == 1) Sample_counter <= 0;
      else if (inc_Sample_counter == 1) Sample_counter <= Sample_counter + 1;

      if (clr_Bit_counter == 1) Bit_counter <= 0;
      else if (inc_Bit_counter == 1) Bit_counter <= Bit_counter + 1;
      if (shift == 1 ) RCV_shftreg <= {Serial_in, RCV_shftreg[word_size - 1:1]};
      if (load == 1) RCV_datareg <= RCV_shftreg;
    end
  end
endmodule
```

FIGURE 7-27 Continued

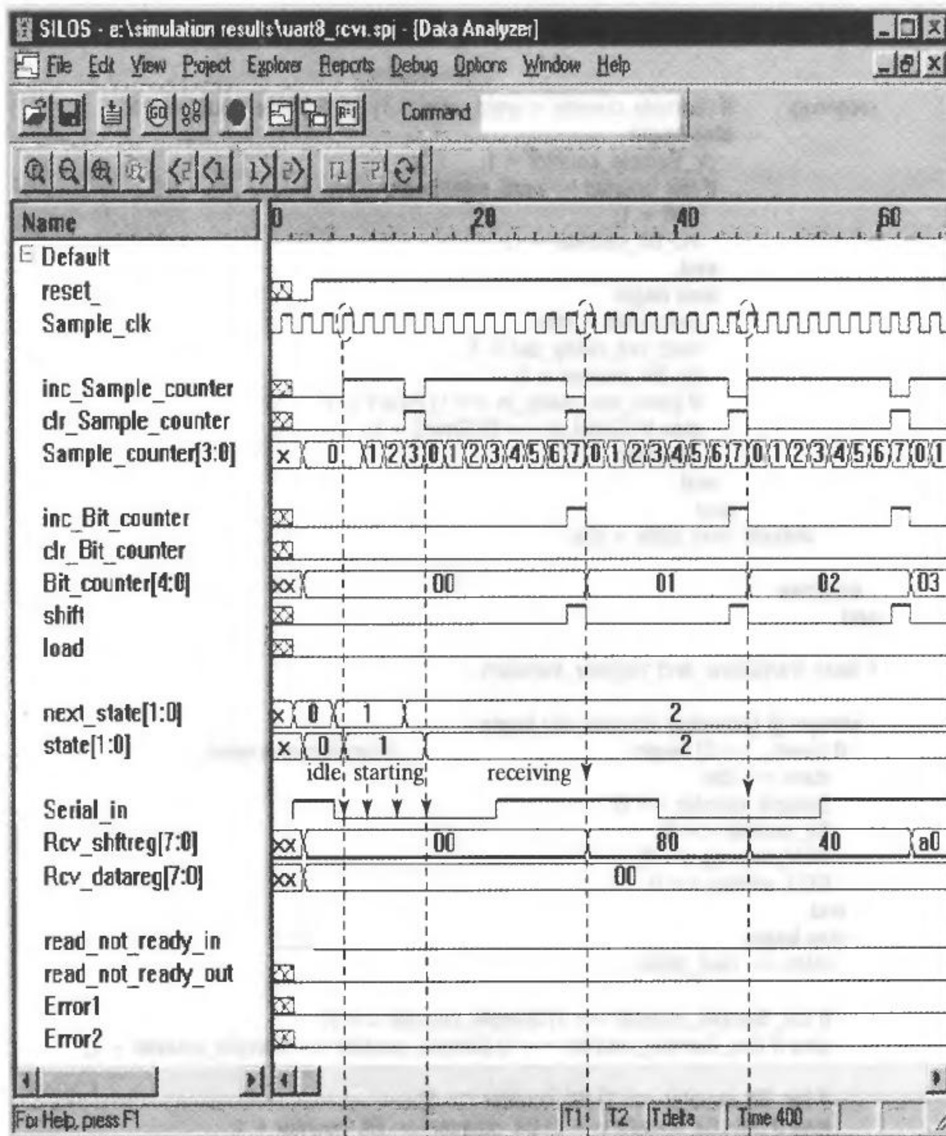
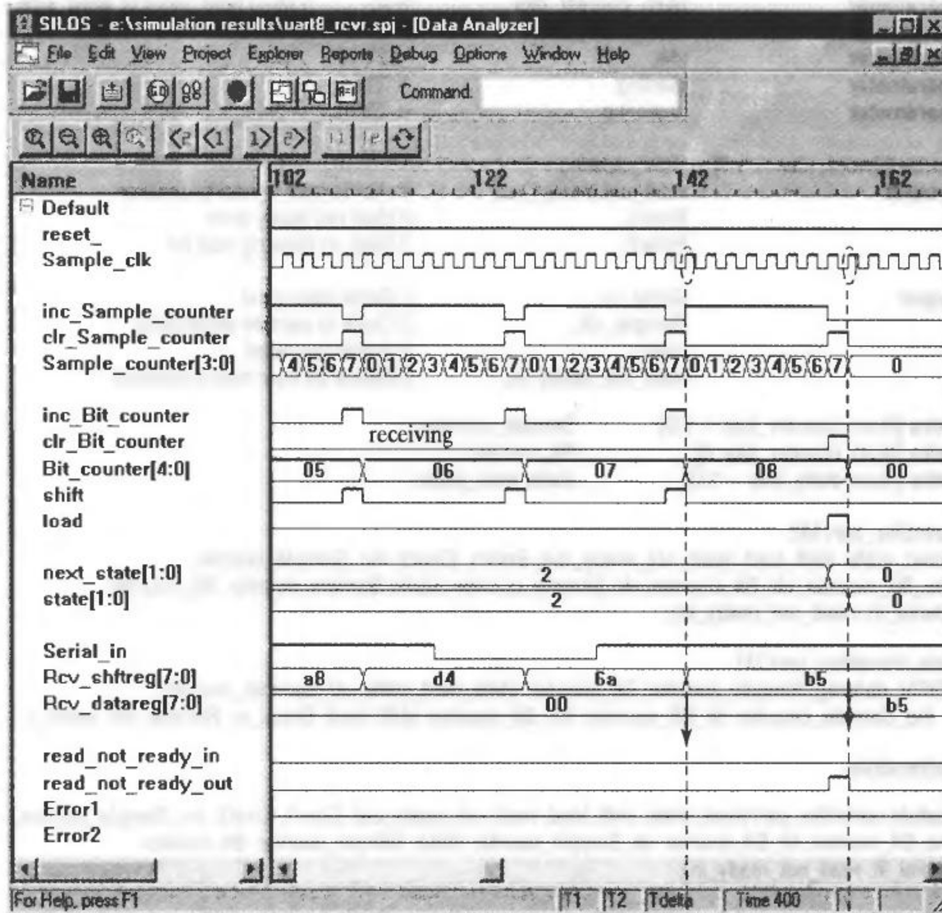


FIGURE 7-28 Annotated simulation results for the UART receiver.



Shift eighth sample into Rcv_shftreg
$6a_h = 0110_1010_2 \rightarrow 1011_0101_2 = b5_h$

Verify stop-bit
Load RCV_data_reg with RCV_shftreg

FIGURE 7-29 Transfer of data word into *RCV_datareg* at the end of sampling.

```
module UART8_rcvr_partition (RCV_datareg, read_not_ready_out, Error1, Error2, Serial_in,
  read_not_ready_in, Sample_clk, reset_);

 // partitioned UART receiver                             // Sample_clk is 8x Bit_clk

 parameter          word_size               = 8;
 parameter          half_word               = word_size / 2;
 parameter          Num_counter_bits        = 4;     // Must hold count of word_size
 parameter          Num_state_bits          = 2;     // Number of bits in state
 parameter          idle                    = 2'b00;
 parameter          starting                = 2'b01;
 parameter          receiving               = 2'b10;

 output [word_size - 1:0]  RCV_datareg;
 output                    read_not_ready_out,         // Handshake to host processor
                           Error1,                     // Host not ready error
                           Error2;                     // Data_in missing stop bit

 input                     Serial_in,                  // Serial data input
                           Sample_clk,                 // Clock to sample serial data
                           reset_,                     // Active-low reset
                           read_not_ready_in;          // Status bit from host processor

 wire [Num_counter_bits - 1:0]        Sample_counter;
 wire [Num_counter_bits: 0]           Bit_counter;
 wire [Num_state_bits - 1:0]          state, next_state;

controller_part M2
 (next_state, shift, load, read_not_ready_out, Error1, Error2, inc_Sample_counter,
 inc_Bit_counter, clr_Bit_counter, clr_Sample_counter, state, Sample_counter, Bit_counter,
 Serial_in, read_not_ready_in);

state_transition_part M1
 (RCV_datareg, Sample_counter, Bit_counter, state, next_state, clr_Sample_counter,
  inc_Sample_counter, clr_Bit_counter, inc_Bit_counter, shift, load, Serial_in, Sample_clk, reset_);

endmodule

module controller_part (next_state, shift, load, read_not_ready_out, Error1, Error2, inc_Sample_counter,
 inc_Bit_counter, clr_Bit_counter, clr_Sample_counter, state, Sample_counter, Bit_counter,
 Serial_in, read_not_ready_in);

 parameter          word_size               = 8;
 parameter          half_word               = word_size / 2;
 parameter          Num_counter_bits        = 4;     // Must hold count of word_size
 parameter          Num_state_bits          = 2;     // Number of bits in state
 parameter          idle                    = 2'b00;
 parameter          starting                = 2'b01;
 parameter          receiving               = 2'b10;
```

FIGURE 7-30 Verilog description of *UART8_rcvr_partition*, an 8-bit UART receiver partitioned into a controller and a datapath.

```
output [Num_state_bits - 1:0]   next_state;
output                          shift, load, inc_Sample_counter;
output                          inc_Bit_counter, clr_Bit_counter, clr_Sample_counter;
output                          read_not_ready_out, Error1, Error2;

input [Num_state_bits - 1:0]    state;
input [Num_counter_bits - 1:0]  Sample_counter;
input [Num_counter_bits: 0]     Bit_counter;
input                           Serial_in, read_not_ready_in;

reg next_state;
reg inc_Sample_counter, inc_Bit_counter, clr_Bit_counter, clr_Sample_counter;
reg shift, load,  read_not_ready_out, Error1, Error2;

always @ (state or Serial_in or read_not_ready_in or Sample_counter or Bit_counter) begin
  read_not_ready_out = 0;         //Combinational logic for next state and conditional outputs
  clr_Sample_counter = 0;
  clr_Bit_counter = 0;
  inc_Sample_counter = 0;
  inc_Bit_counter = 0;
  shift = 0;
  Error1 = 0;
  Error2 = 0;
  load = 0;
  next_state = state;

  case (state)
   idle:         if (Serial_in == 0) next_state = starting;

   starting:     if (Serial_in == 1) begin
                   next_state = idle;
                   clr_Sample_counter = 1;
                 end else

                 if (Sample_counter == half_word - 1) begin
                   next_state = receiving;
                   clr_Sample_counter = 1;
                 end else inc_Sample_counter = 1;

   receiving:    if (Sample_counter < word_size - 1) inc_Sample_counter = 1;
                 else begin
                   clr_Sample_counter = 1;
                   if (Bit_counter != word_size) begin
                     shift = 1;
                     inc_Bit_counter = 1;
                   end
                   else begin
                     next_state = idle;
                     read_not_ready_out = 1;
                     clr_Bit_counter = 1;
                     if (read_not_ready_in == 1) Error1 = 1;
                     else if (Serial_in == 0) Error2 = 1;
                     else load = 1;
                   end
                 end
   default:      next_state = idle;

  endcase
 end
endmodule
```

FIGURE 7-30 Continued

```
module state_transition_part (RCV_datareg, Sample_counter, Bit_counter, state, next_state,
clr_Sample_counter, inc_Sample_counter, clr_Bit_counter, inc_Bit_counter, shift, load, Serial_in,
Sample_clk, reset_);
 parameter              word_size          = 8;
 parameter              half_word          = word_size / 2;
 parameter              Num_counter_bits   = 4;    // Must hold count of word_size
 parameter              Num_state_bits     = 2;    // Number of bits in state
 parameter              idle               = 2'b00;
 parameter              starting = 2'b01;
 parameter              receiving          = 2'b10;

 output [word_size - 1:0]          RCV_datareg;
 output [Num_counter_bits - 1:0]   Sample_counter;
 output [Num_counter_bits: 0]      Bit_counter;
 output [Num_state_bits - 1:0]     state;

 input [Num_state_bits - 1:0]      next_stage;
 input                             Serial_in;
 input                             inc_Sample_counter, inc_Bit_counter;
 input                             clr_Bit_counter, clr_Sample_counter, shift, load;
 input                             Sample_clk, reset_;

 reg                               Sample_counter, Bit_counter;
 reg [word_size - 1:0]             RCV_shftreg, RCV_datareg;
 reg                               state;

// state_transitions_and_datapath_register_transfers

  always @ (posedge Sample_clk) begin
   if (reset_ == 0) begin                    // synchronous reset_
     state <= idle;
     Sample_counter <= 0;
     Bit_counter <= 0;
     RCV_datareg <= 0;
     RCV_shftreg <= 0;
   end
   else begin
     state <= next_state;

     if (clr_Sample_counter == 1) Sample_counter <= 0;
     else if (inc_Sample_counter == 1) Sample_counter <= Sample_counter + 1;

     if (clr_Bit_counter == 1) Bit_counter <= 0;
     else if (inc_Bit_counter == 1) Bit_counter <= Bit_counter + 1;
     if (shift == 1) RCV_shftreg <= (Serial_in, RCV_shftreg[word_size - 1:1]};
     if (load == 1) RCV_datareg <= RCV_shftreg;
   end
  end
endmodule
```

FIGURE 7-30 Continued

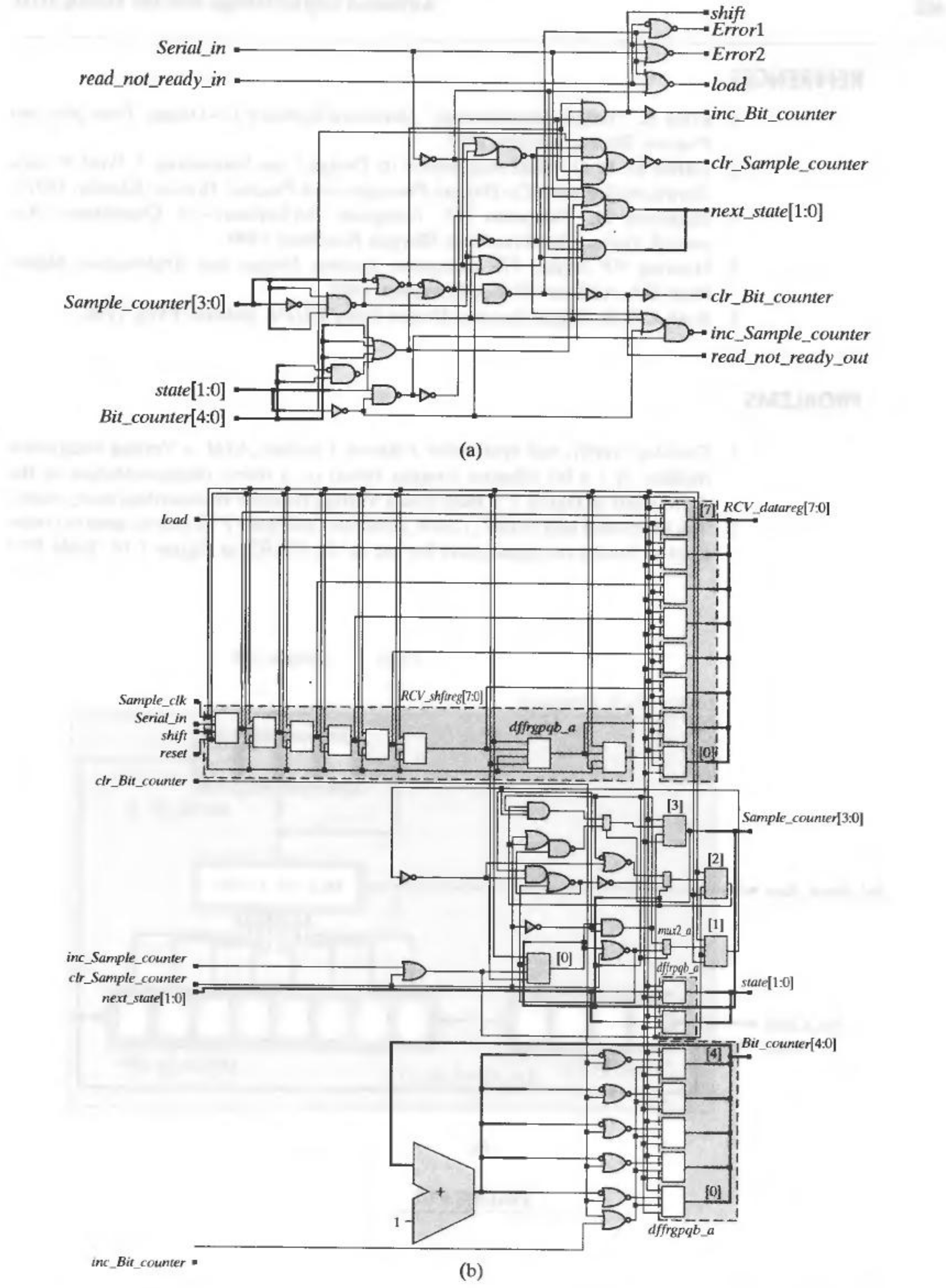


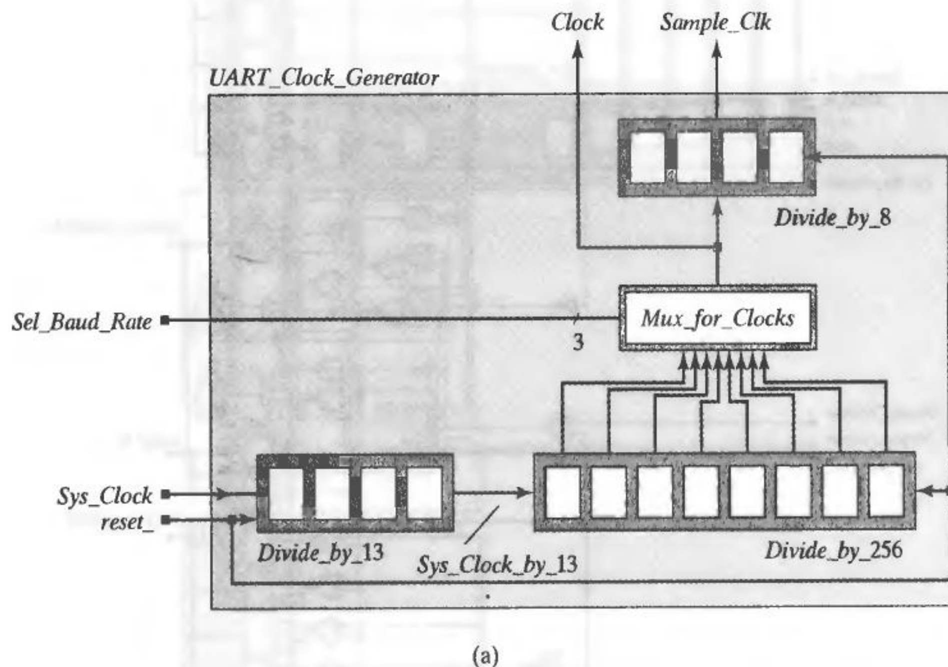
FIGURE 7-31 Circuits synthesized from *UART8_receiver*. (a) state transition and register transfer logic, and (b) combinational logic (forming the next state), output register, and control signals for register transfers.

REFERENCES

1. Ernst R. "Target Architectures." *Hardware/Software Co-Design: Principles and Practice*. Boston: Kluwer, 1997.
2. Gajski D, et al. "Essential Issues in Design," In: Staunstrup J, Wolf W, eds. *Hardware/Software Co-Design: Principles and Practice*. Boston: Kluwer, 1997.
3. Hennessy JL, Patterson DA. *Computer Architecture—A Quantitative Approach*. 2nd ed. San Francisco: Morgan Kaufman, 1996.
4. Heuring VP, Jordan HF. *Computer Systems Design and Architecture*. Menlo Park, CA: Addison-Wesley Longman, 1997.
5. Roth CW, Jr. *Digital Systems Design Using VHDL*. Boston: PWS, 1998.

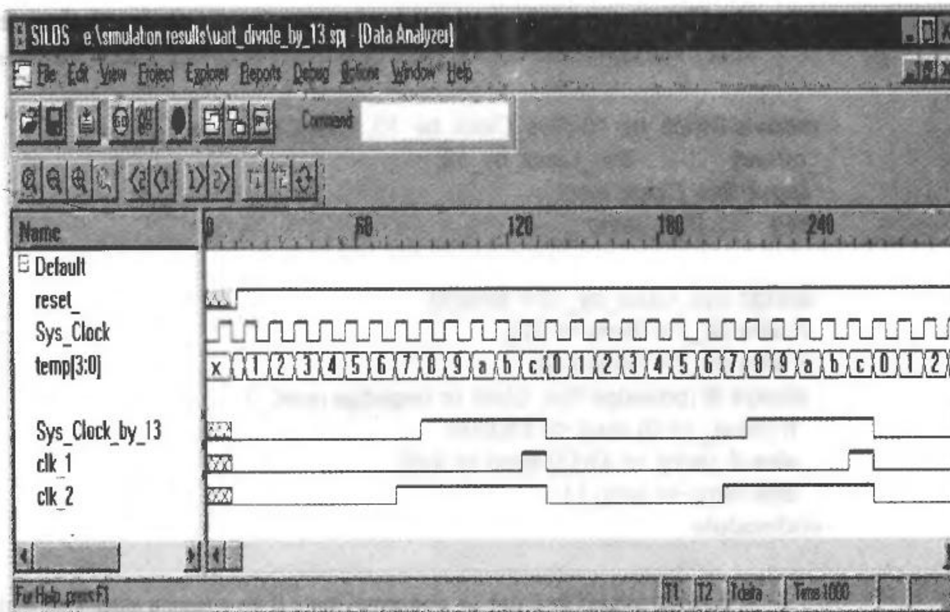
PROBLEMS

1. Develop, verify, and synthesize *Johnson_Counter_ASM*, a Verilog behavioral module of a 4-bit Johnson counter based on a direct implementation of the ASM chart in Figure 7.5. *Hint*: Use a Verilog function to described *next_count*.
2. The functional unit *UART_Clock_Generator* in Figure P7-2 can be used to create a set of baud rate signal pairs for use in the UART in Figure 7-16. Table P7.2



(a)

FIGURE P7-2



(b)

FIGURE P7-2 Continued

shows the pairs that are generated if *Sys_Clock* is an 8-MHz signal. The code for an experimental version of *Divide_by_13* is also given, with *temp* being a 4-bit counter that counts from 0 to 12 before recycling, at $\frac{1}{13}$ the frequency of *Sys_Clock*. The signal *Sys_Clock_by_13* is generated by the condition that register *temp* has a 1 in its MSB, which is true for the last five contiguous cycles. The signal *clk_1* is generated from the condition that *temp* has the value 12; *clk_2* is generated from the condition that *temp* is greater than 6, and produces a more symmetric waveform than the two others.

(a) Synthesize three versions of *Divide_by_13*, one for each of the three possibilities illustrated by the waveforms in Figure P7-2, and compare their circuits.

TABLE P7.2

Sel_Baud_Rate	*Clock*	*Sample_Clock*
000	307,696	38462
001	153,838	19231
010	76,920	9615
011	38,464	4808
100	18,232	2404
101	9,616	1202
110	4,808	601
111	2,404	300.5

(b) Choose one of the methods for forming *Sys_Clock_by_13*, then develop, verify, and synthesize the complete description of *UART_Clock_Generator*.

```
module Divide_by_13 (Sys_Clock_by_13, Sys_Clock, reset_);
 output          Sys_Clock_by_13;
 input Sys_Clock, reset_;
 reg   [3: 0]    temp;

 assign Sys_Clock_by_13 = temp[3];
 // wire clk_1 = (temp == 12);
 // wire clk_2 = (temp > 6);
 always @ (posedge Sys_Clock or negedge reset_ )
  if (reset_ == 0) temp <= 4'b0000;
  else if  (temp == 4'd12) temp <= 4'd0;
  else temp <= temp +1;
endmodule
```

3. A counter is said to enter an abnormal state if its enters a state that is not explicitly decoded in its next-state function. A self-correcting counter has the ability to recover from an abnormal state. The key is to choose default assignments that ensure recovery. Develop and verify a Verilog model of a self-correcting 4-bit Johnson counter using the next state 0001_2 if the current state is 0--0, with "-" denoting a don't-care. Demonstrate that the counter is self-correcting, and synthesize the circuit.
4. The text below describes a fragment of code taken from a Verilog model of a counter. When *mode* is *ring2* the counter is to act like a ring counter, but move two adjacent bits at a time. The simulation results in Figure P7-4 shows that the counter enters an unknown count for one clock, and then fails to count correctly. Find the cause of the error.

```
module Counter8_Prog (count, enable, mode, direction, enable,  clk, reset);
output [7: 0]    count;
input  [1: 0]    mode;
input            enable, direction;
input            clk, reset;
reg              count;
parameter        start_count      = 1;

// Mode of count
parameter              binary   = 0;
parameter              ring1    = 1;
parameter              ring2    = 2;
//parameter            spiral   = 2;
parameter              jump2    = 3;
```

```
// Direction of count
parameter           left      = 0;
parameter           right     = 1;
parameter           up        = 0;
parameter           down      = 1;

always @ (posedge clk or posedge reset)
 if (reset ==1) count <= start_count;
 else if (enable ==1)
  case (mode)
   ring1:      count <= ring1_count      (count, direction);
   ring2:      count <= ring2_count      (count, direction);
   jump2:      count <= jump2_count      (count, direction);
   default:    count <= binary_count     (count, direction);
  endcase
```

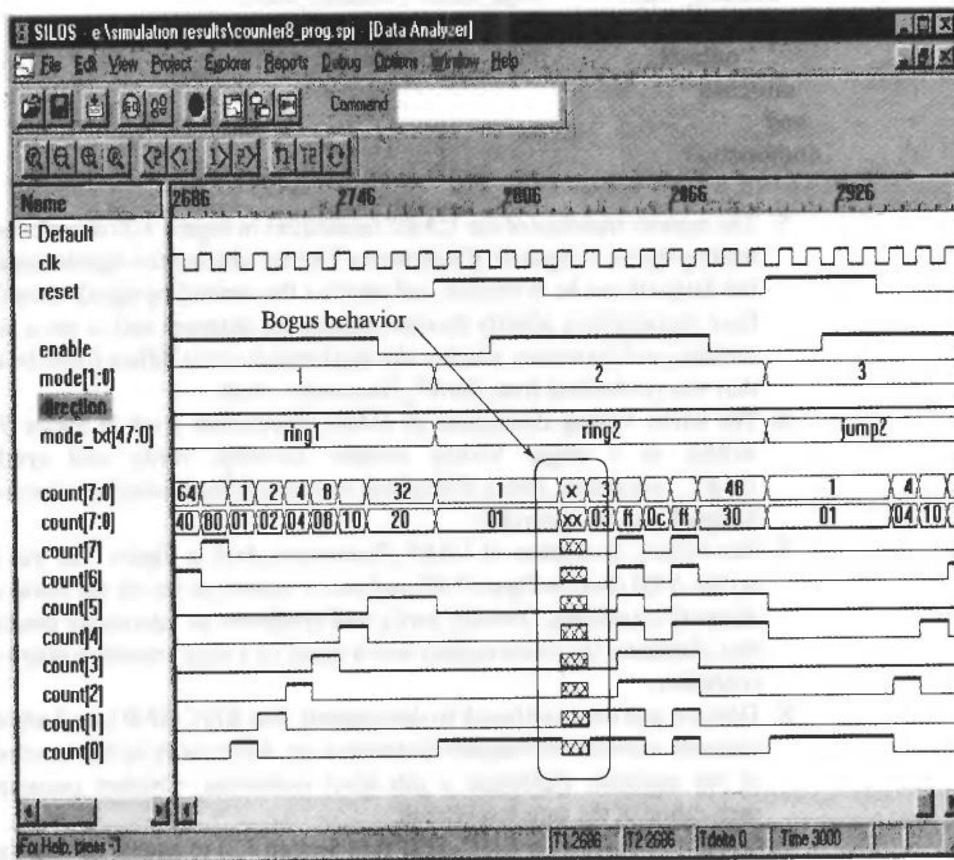


FIGURE P7-4

```
function [7: 0]  ring2_count;
 input [7: 0]    count;
 input           direction;
 begin
  ring2_count = start_count; //8'b0000_0011;
  if (direction == left)
  case (count)
   start_count:         ring2_count = 8'b1100_0000;
   8'b1100_0000:        ring2_count = 8'b0000_0011;
   default:             ring2_count = 8'b1111_1111;
  endcase
  else
  case (count)
   start_count:         ring2_count = 8'b0000_0011;
   8'b1100_0000:        ring2_count = 8'b0011_0000;
   8'b0011_0000:        ring2_count = 8'b0000_1100;
   8'b0000_1100:        ring2_count = 8'b0000_0011;
   8'b0000_0011:        ring2_count = 8'b1100_0000;
      default:          ring2_count = 8'b1111_1111;
  endcase
 end
endfunction
```

5. The register transfers of the UART transmitter in Figure 7-20 decode the controlling signals in separate ***if*** statements. Discuss whether the signals controlling the datapath can be in conflict, and whether the controlling signals should have been decoded by a priority decoder. Modify the datapath unit to use a priority decoder, and determine whether the synthesized circuit differs from the circuit that was synthesized from *UART_Transmitter_Arch*.
6. The entire Verilog description of *UART_Transmitter_Arch* in Figure 7-20 is written as a single Verilog module. Develop, verify, and synthesize *UART_Transmitter_Part*, a description with hierarchical boundaries around the datapath and the controller.
7. The Verilog description of *UART_Transmitter_Arch* in Figure 7-20 was based on the ASM chart in Figure 7-18, and uses a counter to record the status of the datapath transitions. Develop, verify, and synthesize an alternative description that eliminates the status register and is based on a state-transition graph of the controller.
8. Develop and use a testbench to demonstrate that *RISC_SPM* (see Section 7.3) correctly executes its complete instruction set. After verifying the functionality of the machine, synthesize a gate-level realization. Conduct postsynthesis verification of the gate-level circuit.
9. Modify *ALU_RISC* in *RISC_SPM* (see Section 7.3) to handle carries. Synthesize the new machine.
10. (a) Develop and verify *Binary_Counter_Arch*, a Verilog model with the hierarchical partition, module names, and port structure (*count, enable, clock, rst*) shown in Figure 7-3, with *count* having a parameterized width. Also, develop and verify the nested modules *Control_Unit* and *Datapath_Unit_Arch*, as well as

the child modules, *Mux* and *Count_Register*. (b) Synthesize *Binary_Counter_Arch* and *Binary_Counter_Behav_imp* (an implicit-state machine behavioral model) for a 4-bit wide datapath. (c) Compare the synthesized circuits. (d) Compare the results of simulating the counters, using a common testbench.

11. Develop, verify, and synthesize *Binary_Counter_STG*, using the STG in Figure 7-4 as a guide. Compare to the results of Problem 10.
12. The implicit Moore machine with a modified control unit in *Binary_Counter_Part_RTL_by_3* (see Example 7.1) increments the counter every third clock, but takes one extra cycle to recover from a reset condition (i.e., the first increment of the counter occurs at the fourth clock edge after reset is de-asserted). Using a testbench, verify this claim. Explain why this behavior occurs, then develop and verify an alternative (partitioned) machine that will recover from a reset condition after three clock edges and increment thereafter on every third edge.
13. Modify the datapath and control units in *Binary_Counter_Part_RTL* (see Example 7.1) to implement *Johnson_Counter_RTL_by_3*, a Johnson counter that increments every third edge of the external clock. Synthesize the model and verify the functionality of the synthesized circuit.
14. If the function *next_count* in *Binary_Counter_Part_RTL* (see Example 7.1) is modified to have the nonblocking assignment *next_count* <= *count* + *1*, the machine behaves incorrectly, as shown by the simulation results in Figure P7-14. Under what conditions will a simulator assign ***x*** to a variable? Explain why *count* is driven to ***x***.

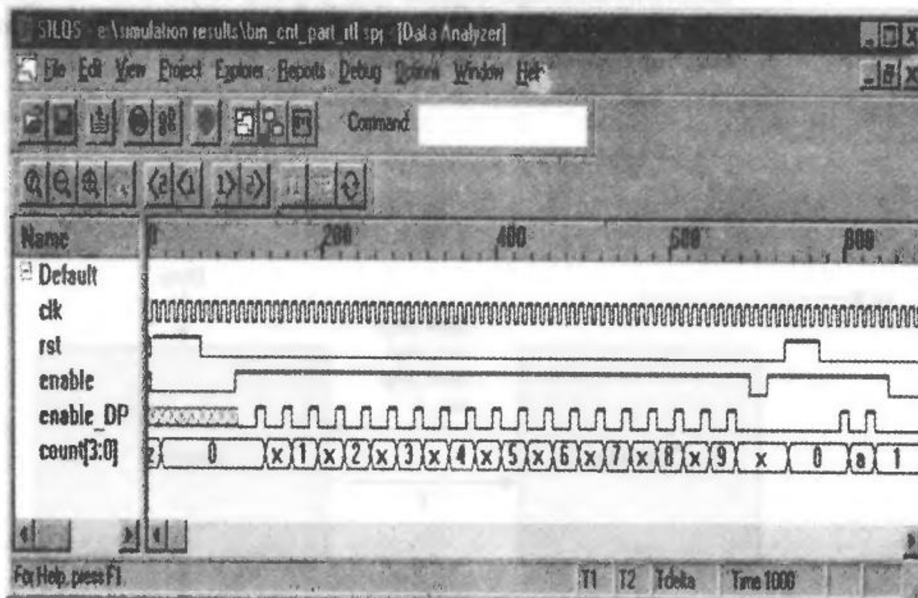


FIGURE P7-14

15. Design a sequential machine that finds the size of the largest gap between two successive 1s in a 16-bit word. Partition the design into a state machine controller and a datapath. The datapath accepts the 16-bit word and produces an output

word whose value is the binary equivalent of the gap size. The interface between the controller and the datapath is shown in Figure P7-15(a).

The ASMD chart in Figure P7-15(b) partially describes the controller. Using the register operations that are annotated on the chart, develop the complete ASMD chart showing the assertions of the output signals of the controller. The data path has a bit counter (*k*), a storage register (*tmp*), and a gap register (*Gap*). Note that the machines are interacting, because the bit count register of the datapath is fed back to the controller.

The machine sequences through the bits of the data word, beginning at the LSB. The register *tmp* holds the current value of the gap. The register *Gap* holds the largest gap that has been found as the search evolves. The datapath control signals have the following functionality:

flush_tmp	empties the *tmp* register
incr_tmp	increments the *tmp* register
store_tmp	loads *Gap* with *tmp*
incr_k	increments the bit counter

(a) Using the attached shell file and the completed ASM chart, write a Verilog model of the machine. The shell includes all signal declarations that will be needed, and includes partially declared cyclic behaviors that (1) implement the state transitions, (2) generate the next state and outputs, and (3) control the datapath registers. The internal architecture of the datapath is not shown, but is implicit in the model. (b) Using the testbench below (also available at the web site), verify the design. Organize your graphical output to list the signals in the order shown in Figure P7-15(c). (*Note*: Data are shown in hex and binary format; *Gap* is shown in decimal format.)

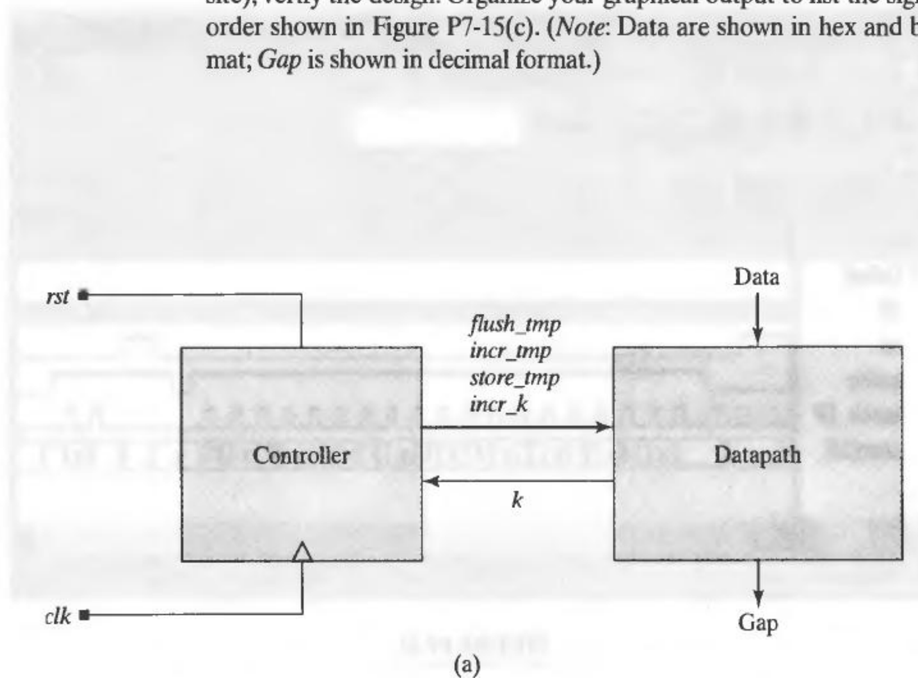
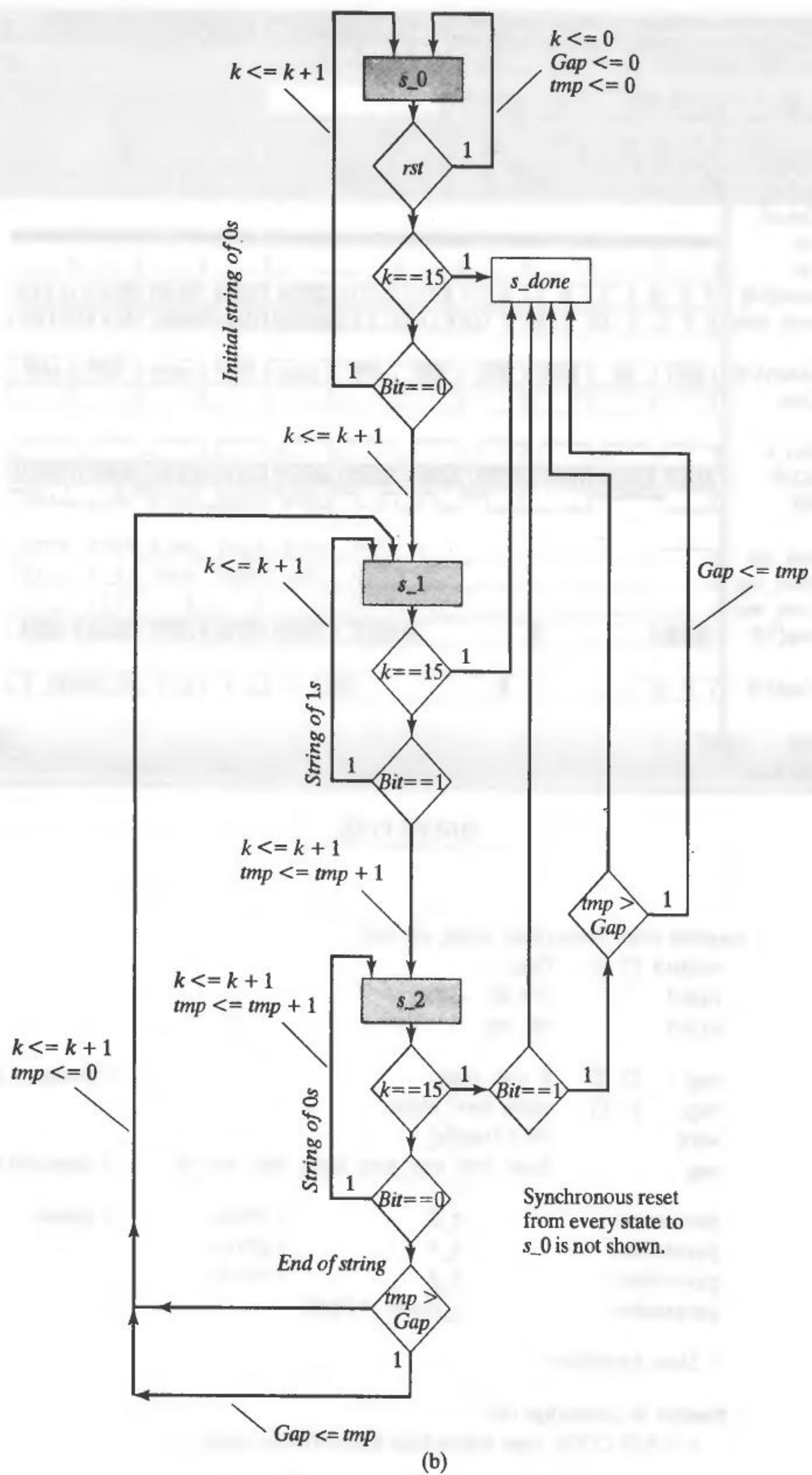


FIGURE P7-15a



(b)

FIGURE P7-15b

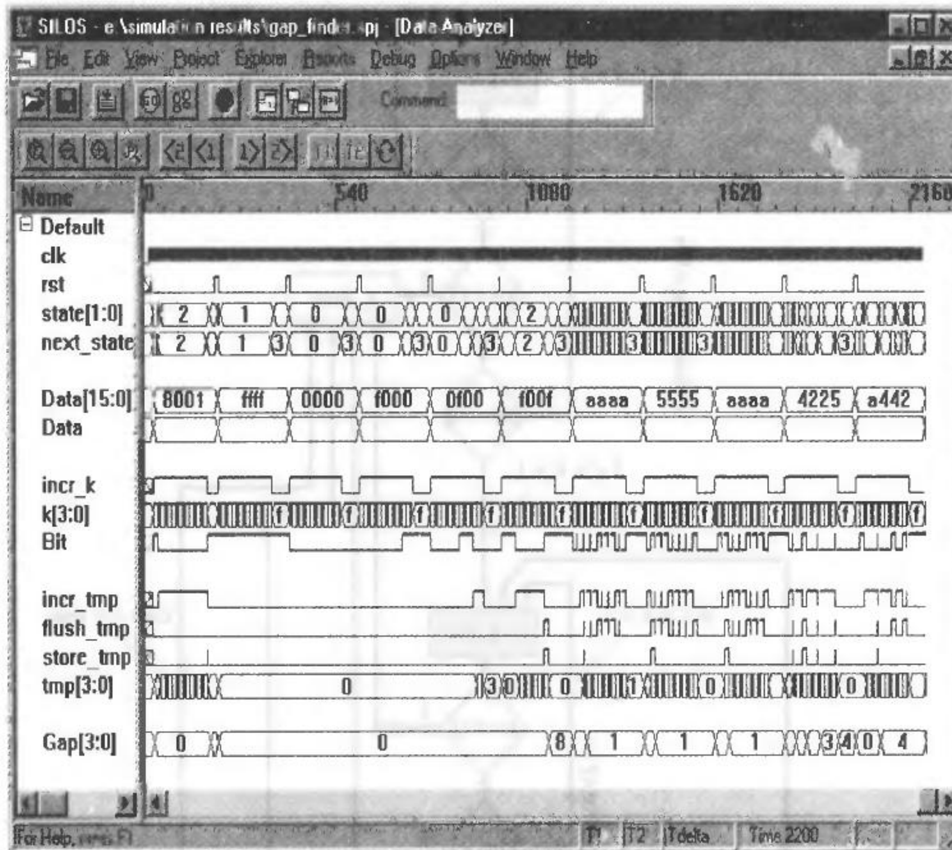


FIGURE P7-15c

```
module Gap_finder(Gap, Data, clk, rst);
  output [3: 0]      Gap;
  input              [15: 0]    Data;
  input              clk, rst;

  reg      [3: 0]    k, tmp, Gap;                               // datapath registers
  reg      [1: 0]    state, next_state;
  wire               Bit = Data[k];
  reg                flush_tmp, incr_tmp, store_tmp, incr_k;    // datapath controls

  parameter              s_0           = 2'b00;                 // states
  parameter              s_1           = 2'b01;
  parameter              s_2           = 2'b10;
  parameter              s_done  = 2'b11;

  // State transitions

 always @ (posedge clk)
   // YOUR CODE (use active-high synchronous reset)
```

```
// Combinational logic for next state and outputs

always @ (state or Bit or k) begin // Must have k to see 16'hffff
                                          // Try simulating without it too
  next_state = state;                     // Remain until conditions are met
  incr_k = 0;                             // Set all variables to 0 on entry
  incr_tmp = 0;                           // to avoid bogus latches
  store_tmp = 0;
  flush_tmp = 0;

  case (state)
       // YOUR CODE for next state and outputs to control the datapath
 endcase
end

   // Edge-sensitive behavior (i.e.,  synchronized) for datapath operations
   always @ (posedge clk)
    if (rst == 1) begin k <= 0; Gap <= 0; tmp <= 0; end
    else begin
       // YOUR CODE for register operations controlled by the state machine

 end
endmodule

module annotate_Gap_finder ();             // Annotate the clock parameters
 defparam t_Gap_finder.M2.Latency = 10;
 defparam t_Gap_finder.M2.Offset = 5;
 defparam t_Gap_finder.M2.Pulse_Width = 5;
endmodule

  module t_Gap_finder ();
   reg    [15: 0]   Data;
   reg              rst;
   wire   [3: 0]    Gap;

   Gap_finder      M1 (Gap, Data, clk, rst);
   Clock_Prog      M2 (clk);

   initial #2200 $finish;

   initial begin     // expect Gap = 14
     #20 rst = 1;
     #5 Data = 16'b1000_0000_0000_0001;
     #5 rst = 0;
   end

   initial begin   // expect Gap = 0
     #200 rst = 1;
     #5 Data = 16'hffff;
     #5 rst = 0;
   end
```

```
initial begin    // expect gap = 0
  #400 rst = 1;
  #5 Data = 16'h0000;
  #5 rst = 0;
end
initial begin    // expect gap = 0
  #600 rst = 1;
  #5 Data = 16'hf000;
  #5 rst = 0;
end
initial begin    // expect gap = 0
  #800 rst = 1;
  #5 Data = 16'h0f00;
  #5 rst = 0;
end
initial begin    // expect gap = 8
  #1000 rst = 1;
  #5 Data = 16'hf00f;
  #5 rst = 0;
end
initial begin    // expect Gap = 0
  #1200 rst = 1;
  #5 Data = 16'haaaa;
  #5 rst = 0;
end
initial begin    // expect gap = 1
  #1400 rst = 1;
  #5 Data = 16'h5555;
  #5 rst = 0;
end
initial begin    // expect Gap = 4 (decreasing gap size)
  #1600 rst = 1;
  #5 Data = 16'b0100_0010_0010_0101;
  #5 rst = 0;
end
initial begin    // expect Gap = 4 (increasing gap size)
  #1800 rst = 1;
  #5 Data = 16'b1010_0100_0100_0010;
  #5 rst = 0;
end
endmodule;
```

20. Develop, verify, and synthesize a frequency divider with a programmable divisor for the base frequency, and a programmable duty cycle.
21. Develop, verify, and synthesize a Verilog model of a decoder that will decode a 16-bit address to determine in which of eight 8-k segments of a 64-k memory the word resides.

22. Describe the differences between the circuits that will be synthesized from the following Verilog cyclic behaviors:

```
always @ (a or b or c or d) y = a + b + c + d;
always @ (a or b or c or d) y = (a + b) + (c + d);
```

23. The instruction set of *RISC_SPM* is limited, and might not serve a particular application very well. Develop *RISC_SPM_e* and an enhanced version of *RISC_SPM* with additional instructions that would be useful if it is to be an embedded processor within a vending machine that is to accept currency, make change, and dispense coffee in response to selections made by the customer. The allowed selections are identified in Figure P7-23. The machine is to assert signals that (1) control dispensing units that blend the coffee according to the customer's choices, (2) accept currency and dispense change, and (3) send messages to a display panel.

Size	Venti	Grande	Tall
Coffee	Normal	Decaff	Espresso
Flavor	Hazelnut	Vanilla	Raspberry
Creamer Type	Half-n-Half	Whole	Skim
Creamer Amount	Heavy	Medium	Tall
Sweetener Type	Sugar	Artificial	Light
Sweetener Amount	Heavy	Medium	Equal

FIGURE P7-23