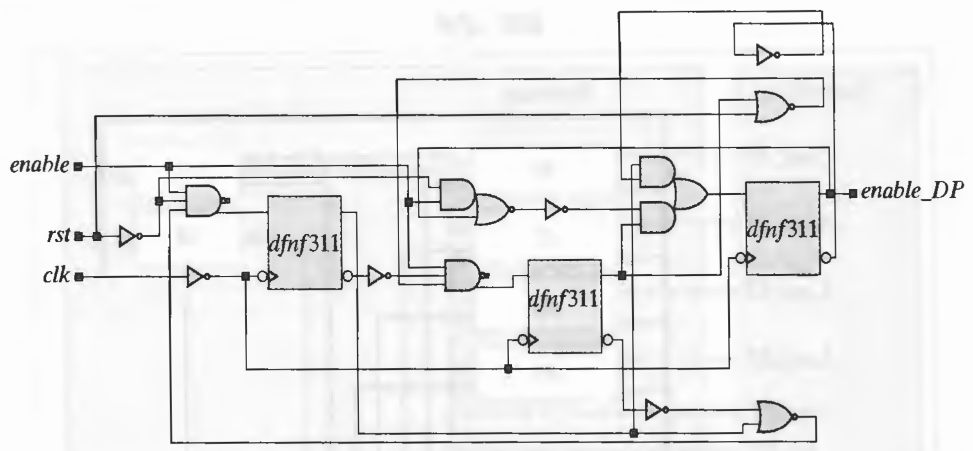


FIGURE 7-9 Circuit synthesized for *Control_Unit_by_3*, the control unit of *Binary_Counter_Part_RTL_by_3*, a 4-bit binary counter partitioned into a control unit and a datapath unit, with the counter incrementing every third clock cycle.

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.

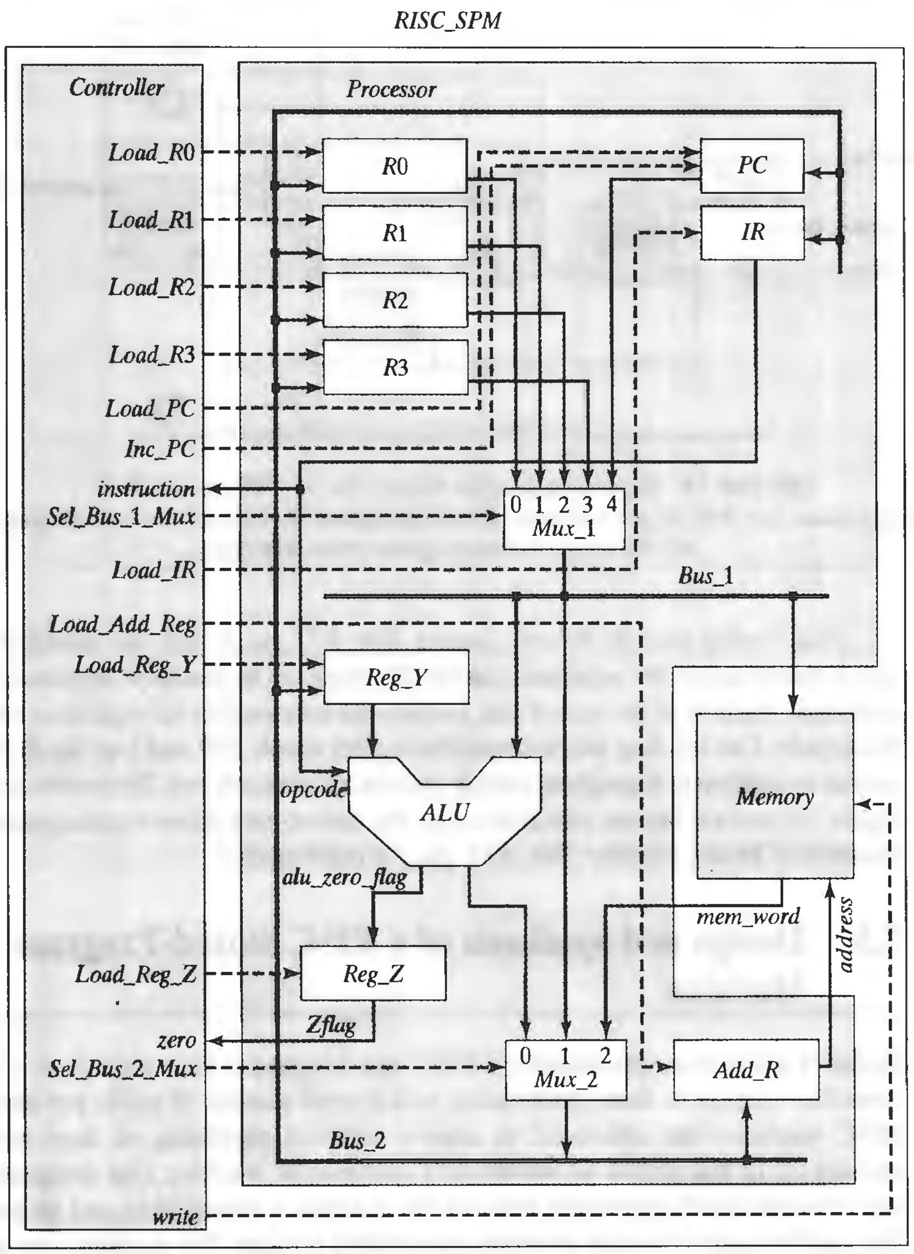


FIGURE 7-10 Architecture of *RISC_SPM*, an RISC stored-program machine (SPM).

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.

opcode				source		destination	
0	0	1	0	0	1	1	0

(a)

opcode				source		destination	
0	1	1	0	1	0	don't care	don't care
address							
0	0	0	1	1	1	0	1

(b)

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

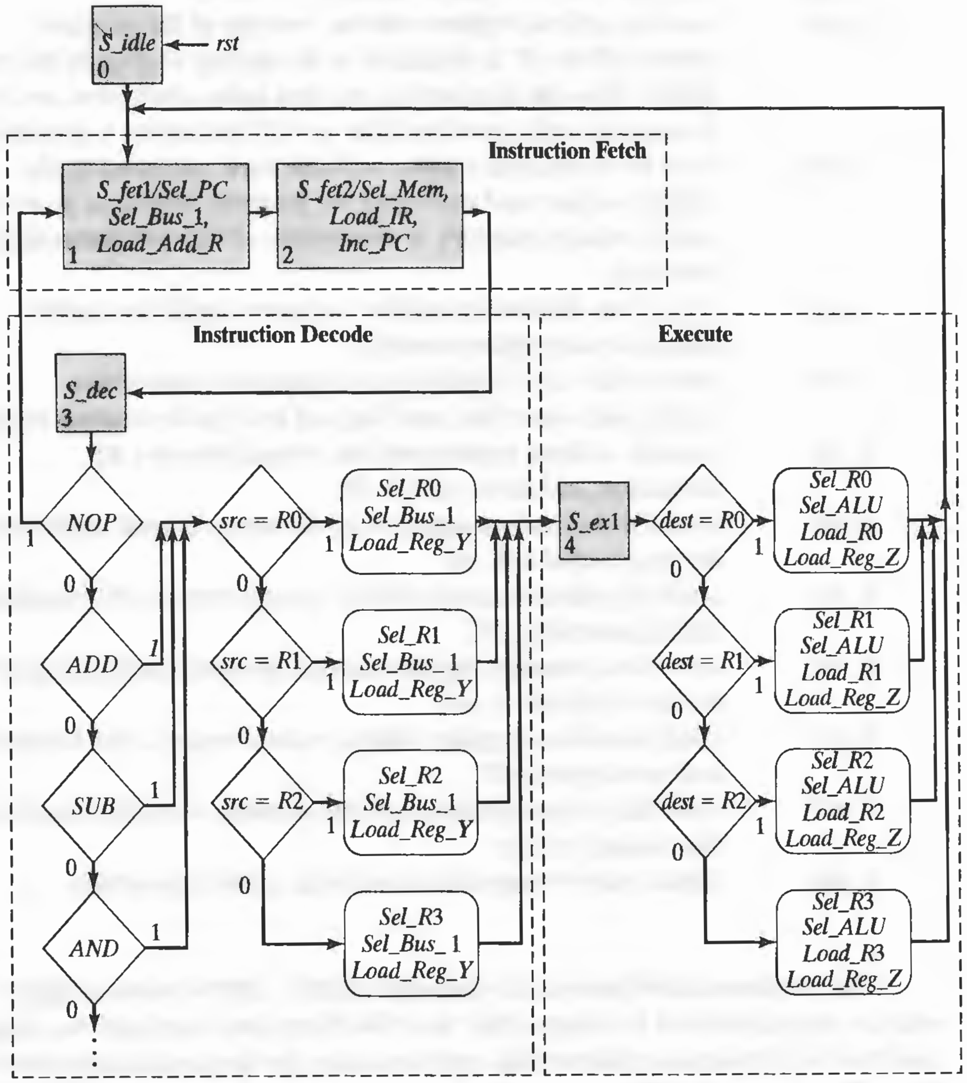
The machine's controller will be designed as an FSM. Its states must be specified, given the architecture, instruction set, and clocking scheme used in the design. This can be accomplished by identifying what steps must occur to execute each instruction. We will use an ASM chart to describe the activity 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.

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.

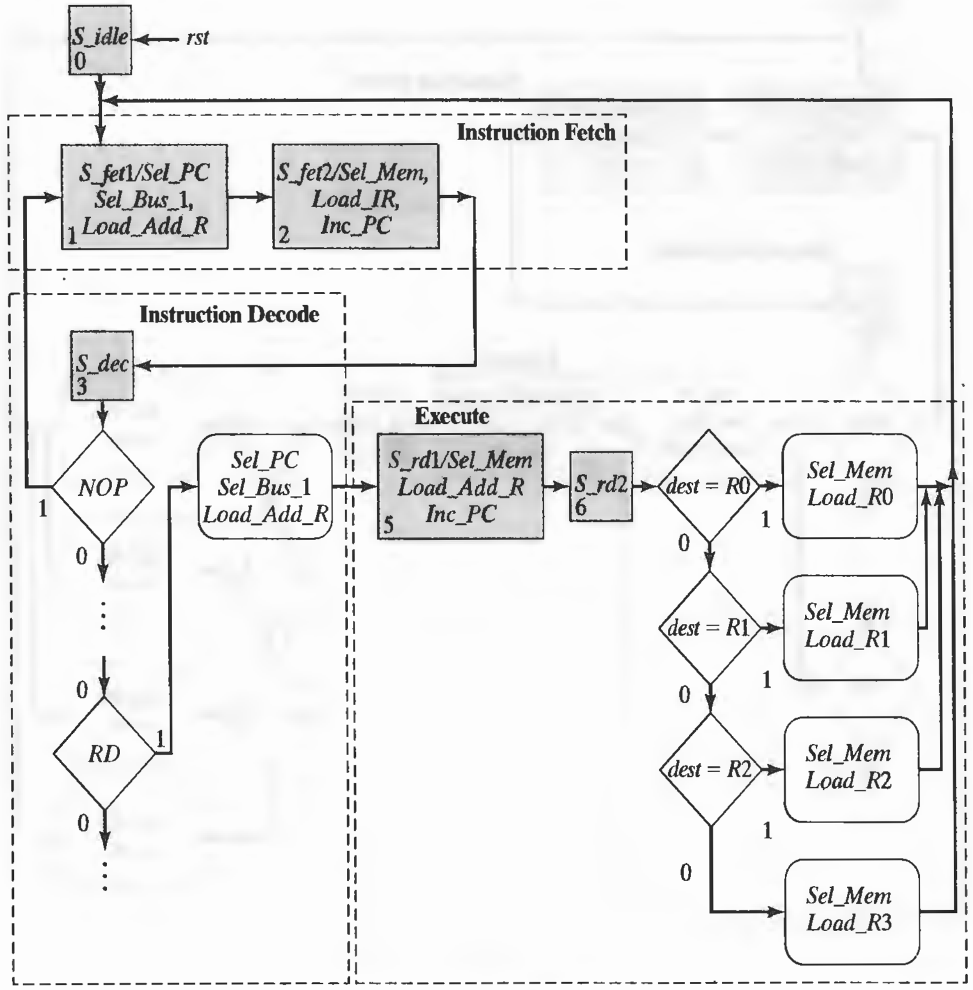
The partitioned ASM chart for the controller of *RISC_SPM* is shown in Figure 7-12, with the states numbered for clarity. Once the ASM charts have been built, the designer can write the Verilog description of the entire machine, for the given architectural partition. This process unfolds in stages. First, 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.


S_idle
0
rst
Instruction Fetch
S_fet1/Sel_PC
Sel_Bus_1,
Load_Add_R
1
S_fet2/Sel_Mem,
Load_IR,
Inc_PC
2
Instruction Decode
S_dec
3
NOP
ADD
SUB
AND
src = R0
src = R1
src = R2
Sel_R0 Sel_Bus_1 Load_Reg_Y
Sel_R1 Sel_Bus_1 Load_Reg_Y
Sel_R2 Sel_Bus_1 Load_Reg_Y
Sel_R3 Sel_Bus_1 Load_Reg_Y
Execute
S_ex1
4
dest = R0
dest = R1
dest = R2
Sel_R0 Sel_ALU Load_R0 Load_Reg_Z
Sel_R1 Sel_ALU Load_R1 Load_Reg_Z
Sel_R2 Sel_ALU Load_R2 Load_Reg_Z
Sel_R3 Sel_ALU Load_R3 Load_Reg_Z
0
1

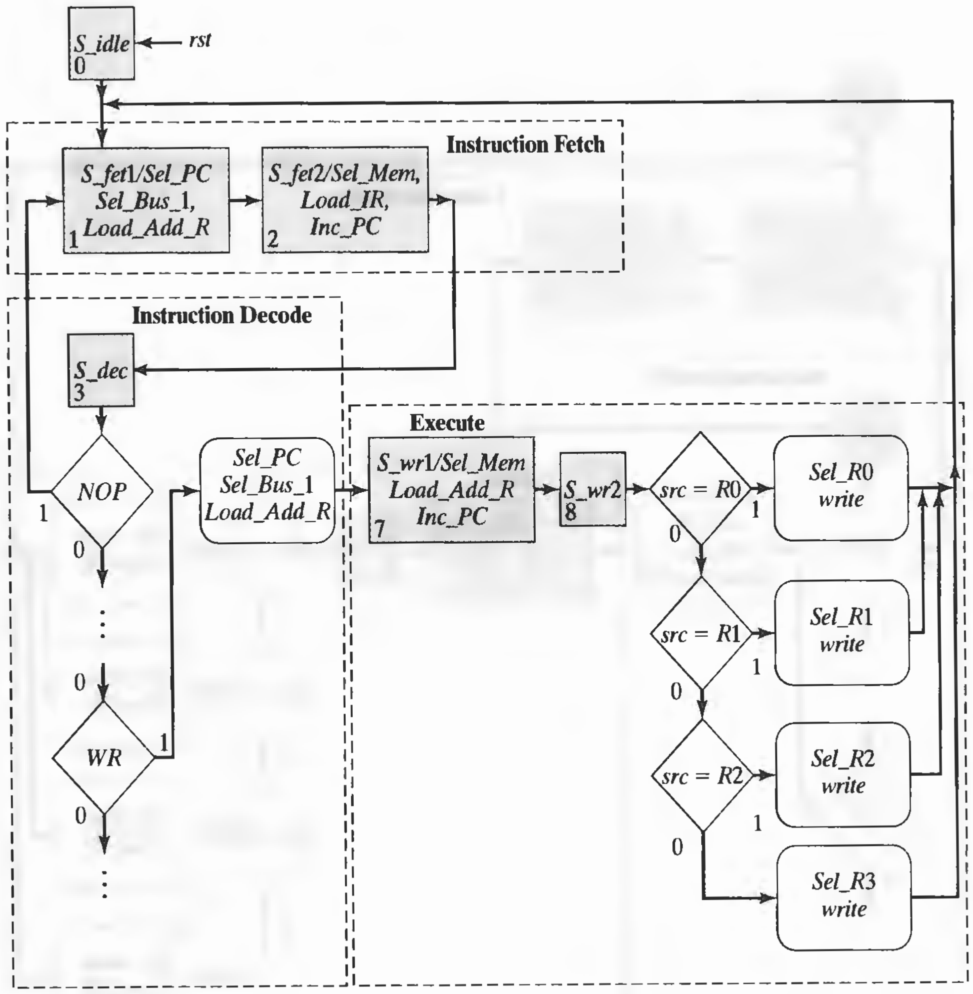
(a)

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*.



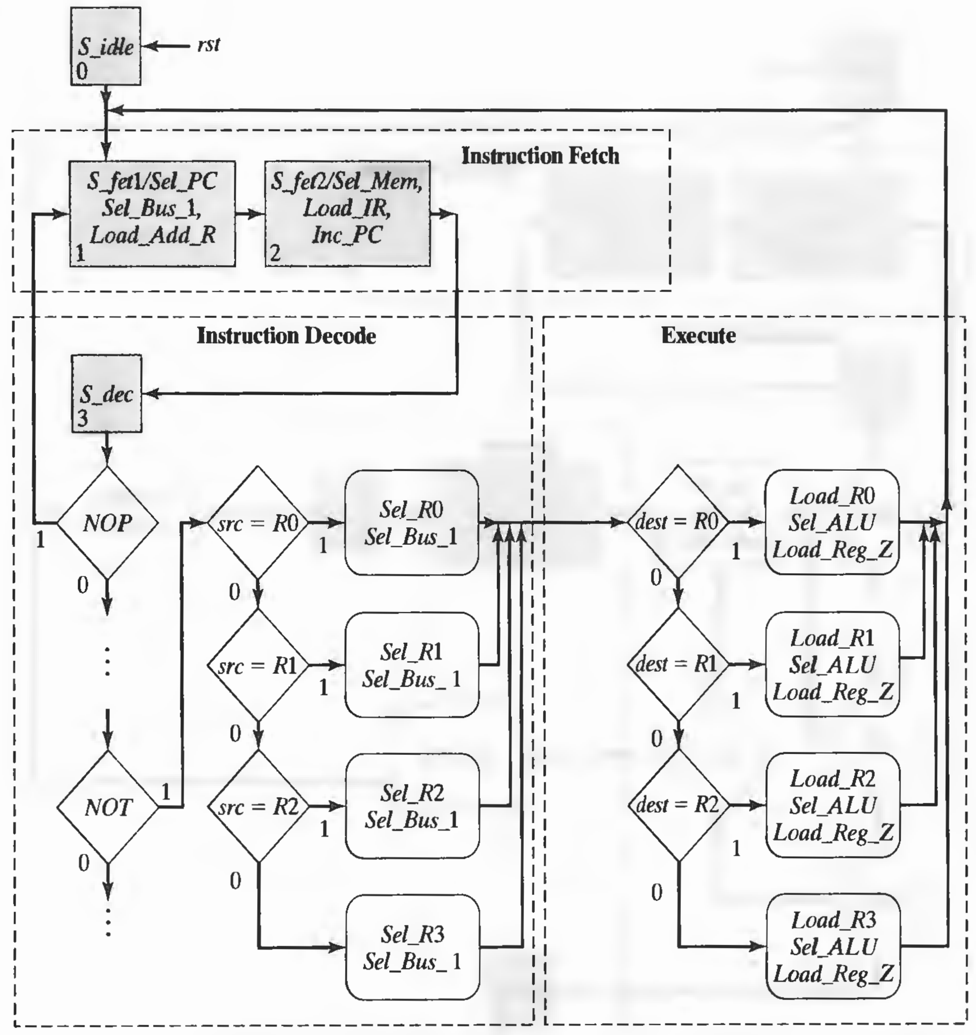
(b)

FIGURE 7-12 Continued



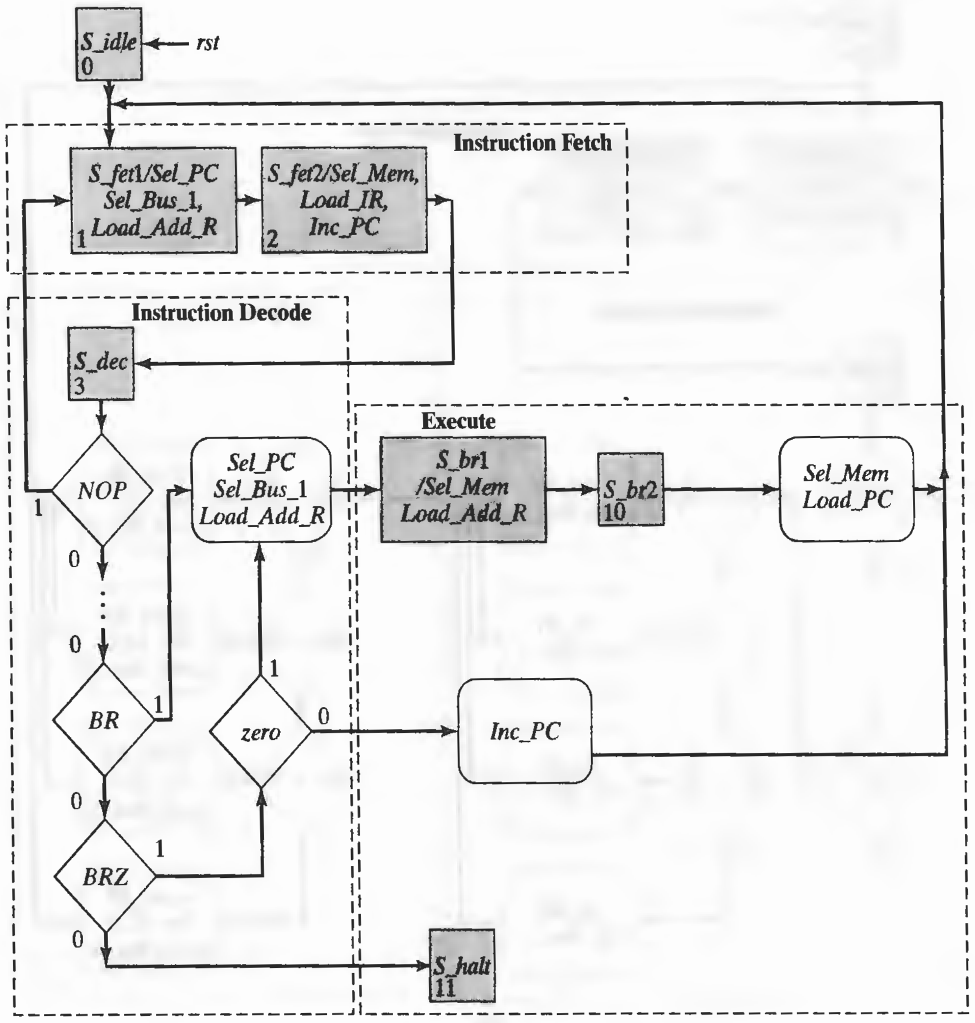
(c)

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 transition 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 control 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.

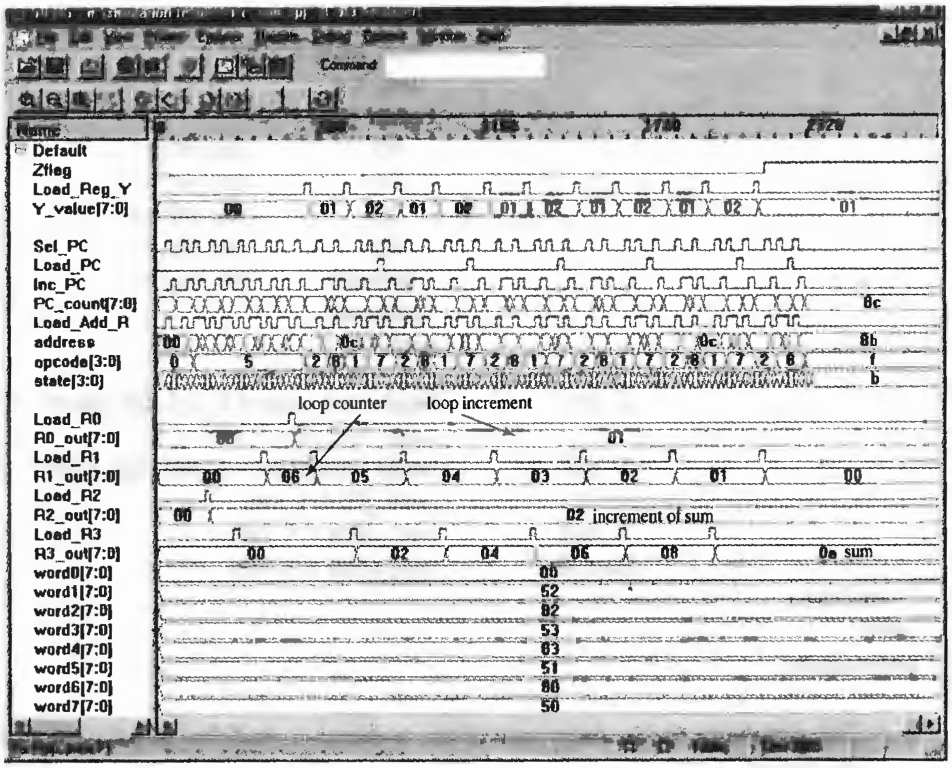


FIGURE 7-13 Simulation results produced by executing a stored program with *RISC_SPM.*

(*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*

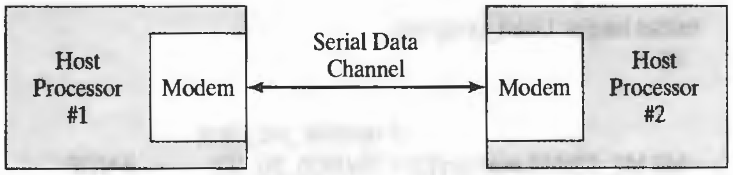


FIGURE 7-14 Processor/modem communication over a serial channel.