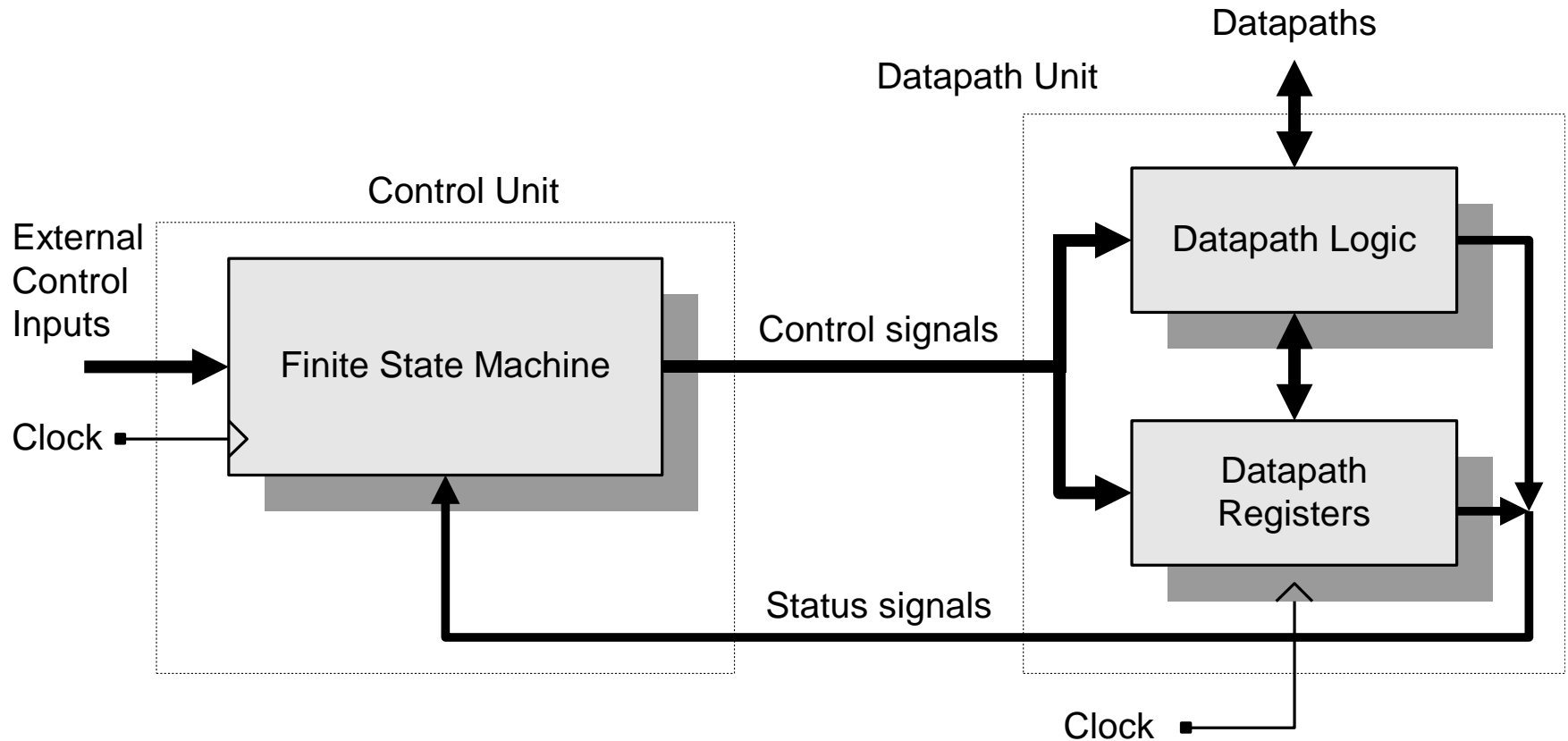


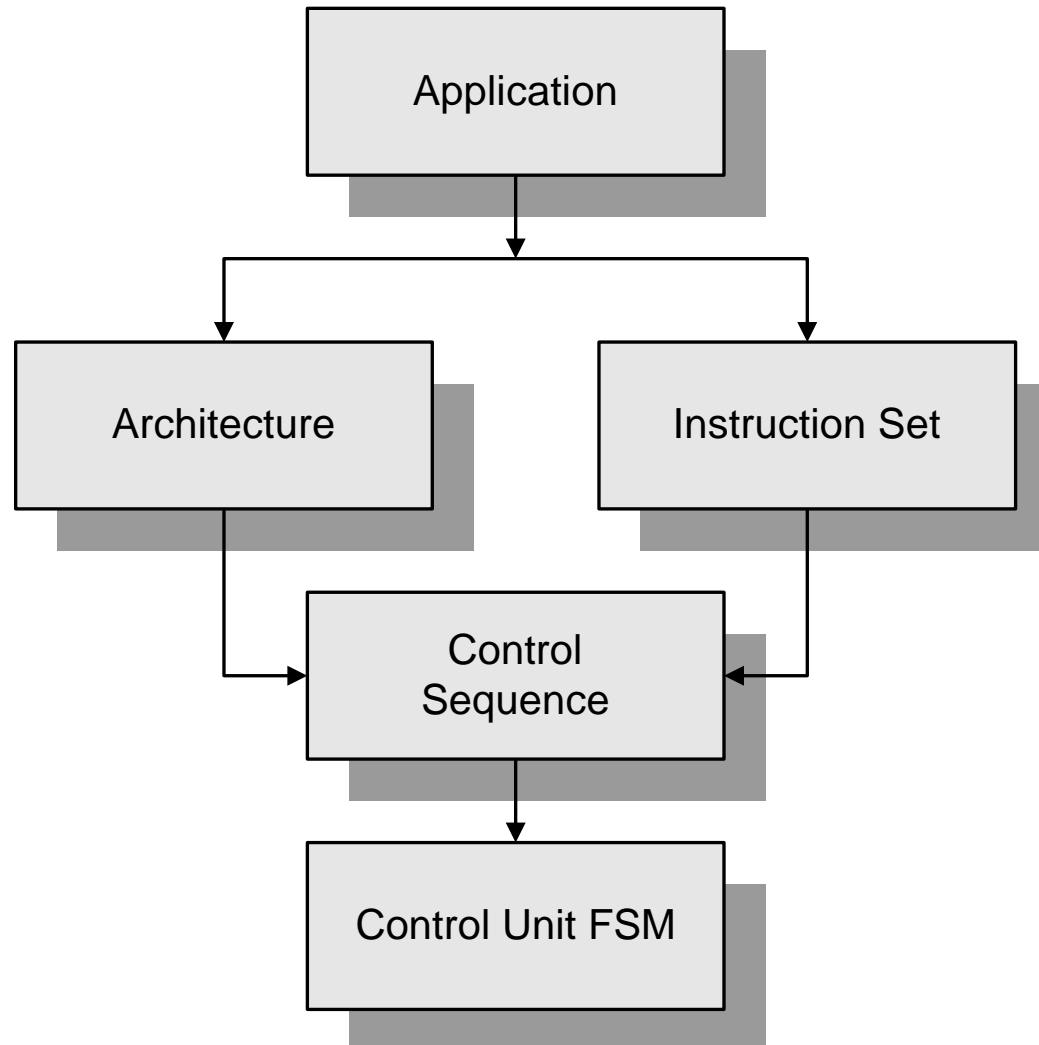
CPU Design Example

Prof. Soo-Ik Chae

Partitioned Sequential Machine



Application-Driven Architecture



RISC ALU

- ❖ Digital circuits are designed to perform arithmetic and/or logic operations, or in general a mixture of them
 - Arithmetic operations : ADD, SUB, MUL, DIV etc.
 - Logic operations : NOT, AND, OR etc.
- ❖ A mathematical problem usually can be decomposed into a number of arithmetic and/or logic operations which may be executed serially or in parallel. However, to save the cost of hardware circuits, usually only one copy of circuit is built for each specific arithmetic or logic function. As a result, the mathematical problem is rearranged into a sequential order of the combinations of these operations.
 - E.g. : partition $D = A + B + C$ into $D \leftarrow A + B$ followed by $D \leftarrow D + C$
- ❖ These arithmetic or logic circuits are collected together to form an arithmetic and logic unit (ALU), and a set of systematic methods are created for this ALU to invoke the functions built in the ALU
- ❖ These methods are collectively referred to as the instruction set, which may also involve memory access and/or data movement

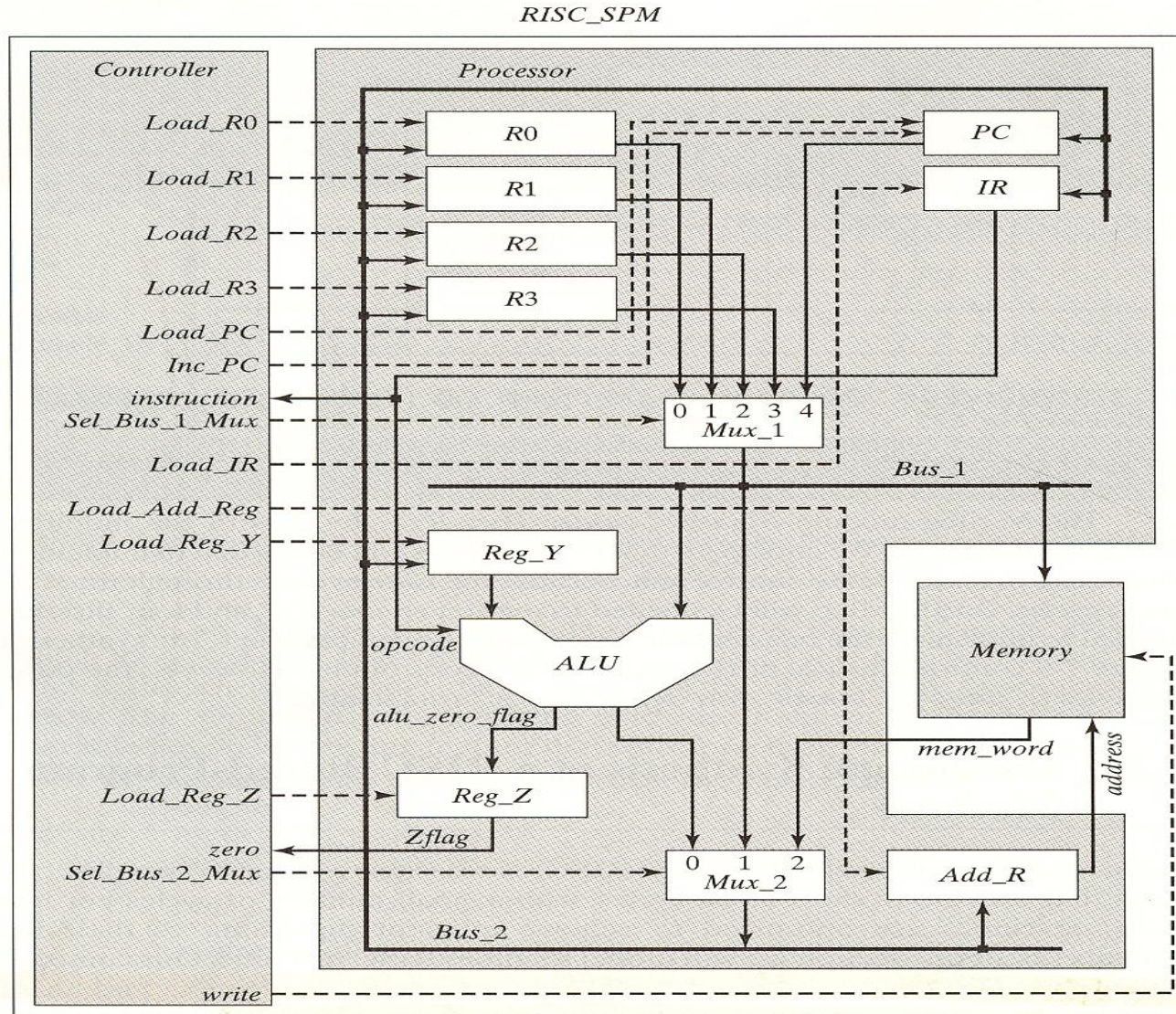
RISC vs CISC

- ❖ To support the functions of ALU, a number of data elements such as registers and counters are also necessary to help bring data from memory/register to the inputs of ALU and store the output of ALU to memory or registers
- ❖ The entire set of supporting elements plus the ALU as well as its accompanying instruction set is referred to as the central processing unit (CPU)
- ❖ Central processing unit (CPU)
 - As it is suggested by its name, all arithmetic operations are executed by this unit one instruction a time.
 - There are two types of design methodologies for the instruction sets of CPU
 - Reduced instruction-set computers (RISC), which features
 - A small number of instructions that execute in short cycles
 - A small number of cycles per instruction
 - Complex instruction-set computers (CISC)

RISC Stored Program Machine (SPM)

- ❖ A RISC store-program machine (SPM) consists of three functional units : a processor, a controller and memory
- ❖ Program instructions and data are stored in memory
- ❖ Instructions are fetched from memory synchronously, decoded and executed to
 - Operate on data with ALU
 - Change the contents of storage registers
 - Change the content of the program counter (PC), instruction register (IR) and the address register (ADD_R)
 - Change the content of memory
 - Retrieve data and instructions from memory
 - Control the movement of data on the system busses

RISC Stored Program Machine (SPM)



Control Functions

- ❖ Functions of the control unit:
 - Determine when to load registers
 - Select the path of data through the multiplexers
 - Determine when data should be written to memory
 - Control the three-state busses in the architecture.

Control Signals

Control Signal

Action

Load_Add_R

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

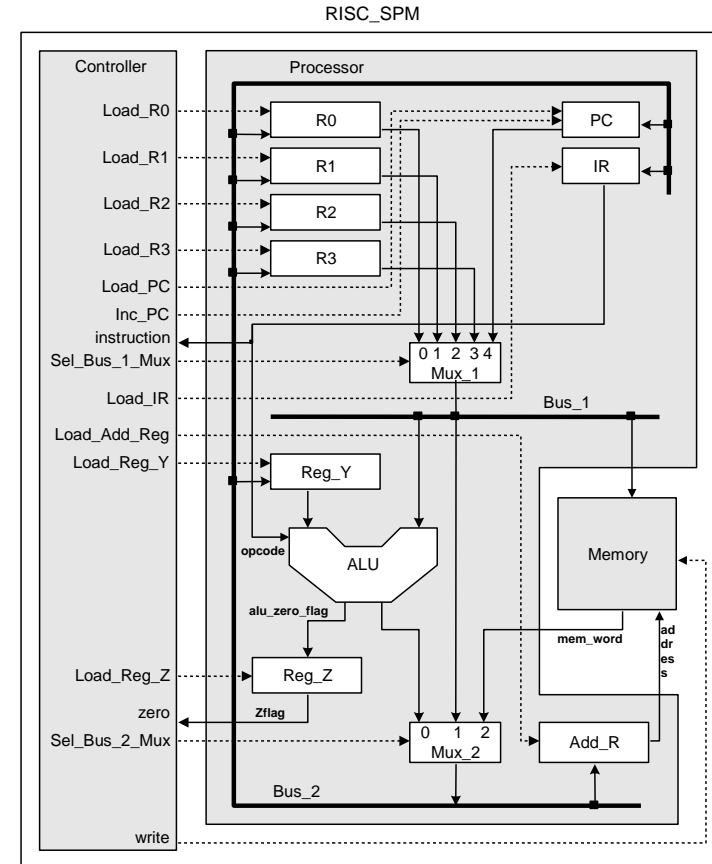
Loads *Bus_2* to the register *Reg_Y*

Load_Reg_Z

Stores output of *ALU* in register *Reg_Z*

write

Loads *Bus_1* into the *SRAM* memory



RISC SPM: Instruction Set [1]

- ❖ The design of the controller depends on the processor's instruction set.
- ❖ RISC SPM has two types of instructions.
- ❖ Short Instruction – 8 bits (basic arithmetic)

opcode				source		destination	
0	0	1	0	0	1	1	0

- ❖ Long Instruction – 16 bits (accessing memory)

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

RISC SPM: Instruction Set [2]

❖ Instruction Set

■ Single-Byte instruction

- NOP
- ADD : $\text{Dest} \leftarrow \text{Source} + \text{Des}$
- AND : $\text{Dest} \leftarrow \text{Source} \& \text{Des.}$
- NOT : $\text{Dest} \leftarrow \text{Source}$
- SUB : $\text{Dest} \leftarrow \text{Source} - \text{Des}$

Opcode				Source		Dest.	

■ Two-Byte instruction

- RD : $\text{Dest} \leftarrow \text{Memory}$
- WR : $\text{Memory} \leftarrow \text{Source}$
- BR : $\text{PC} \leftarrow \text{Address}$
- BRZ : $\text{PC} \leftarrow \text{Address}$ if zero flag == 0

Opcode				Source		Dest.	
						X	X
Address							

RISC SPM: Instruction Set [3]

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

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

Controller Design

- ❖ Three phases of operation: *fetch*, *decode*, and *execute*.
 - Fetching: Retrieves an instruction from memory (2 clock cycles)
 - Decoding: Decodes the instruction, manipulates datapaths, and loads registers (1 cycle)
 - Execution: Generates the results of the instruction (0, 1, or 2 cycles)

Controller States [1]

S_idle State entered after reset is asserted. No action.

S_fet1 Load the Add_R with the contents of the PC

S_fet2 Load the IR with the word addressed by the Add_R,
Increment the PC

S_dec Decode the IR
Assert signals to control datapaths and register transfers.

S_ex1 Execute the *ALU* operation for a single-byte instruction,
Conditionally assert the zero flag, Load the destination register

Controller States [2]

S_rd1 Load Add_R with the second byte of an RD instruction
 Increment the PC.

S_rd2 Load the destination register with memory[Add_R]

S_wr1 Load Add_R with the second byte of a WR instruction,
 Increment the PC.

S_wr2 Write memory[Add_R] with the source register

S_br1 Load Add_R with the second byte of a BR instruction
 Increment the PC.

S_br2 Load the PC with the memory[Add_R]

S_halt Default state to trap failure to decode a valid instruction

Which states are similar?

❖ RISC-SPM controller : The machine has three phases of operation: fetch, decode and execute

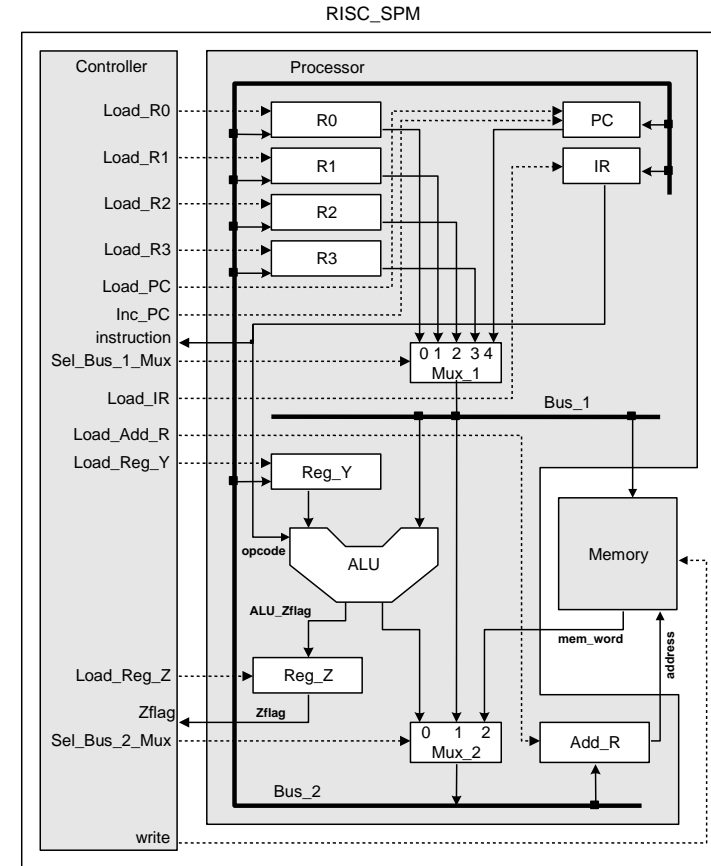
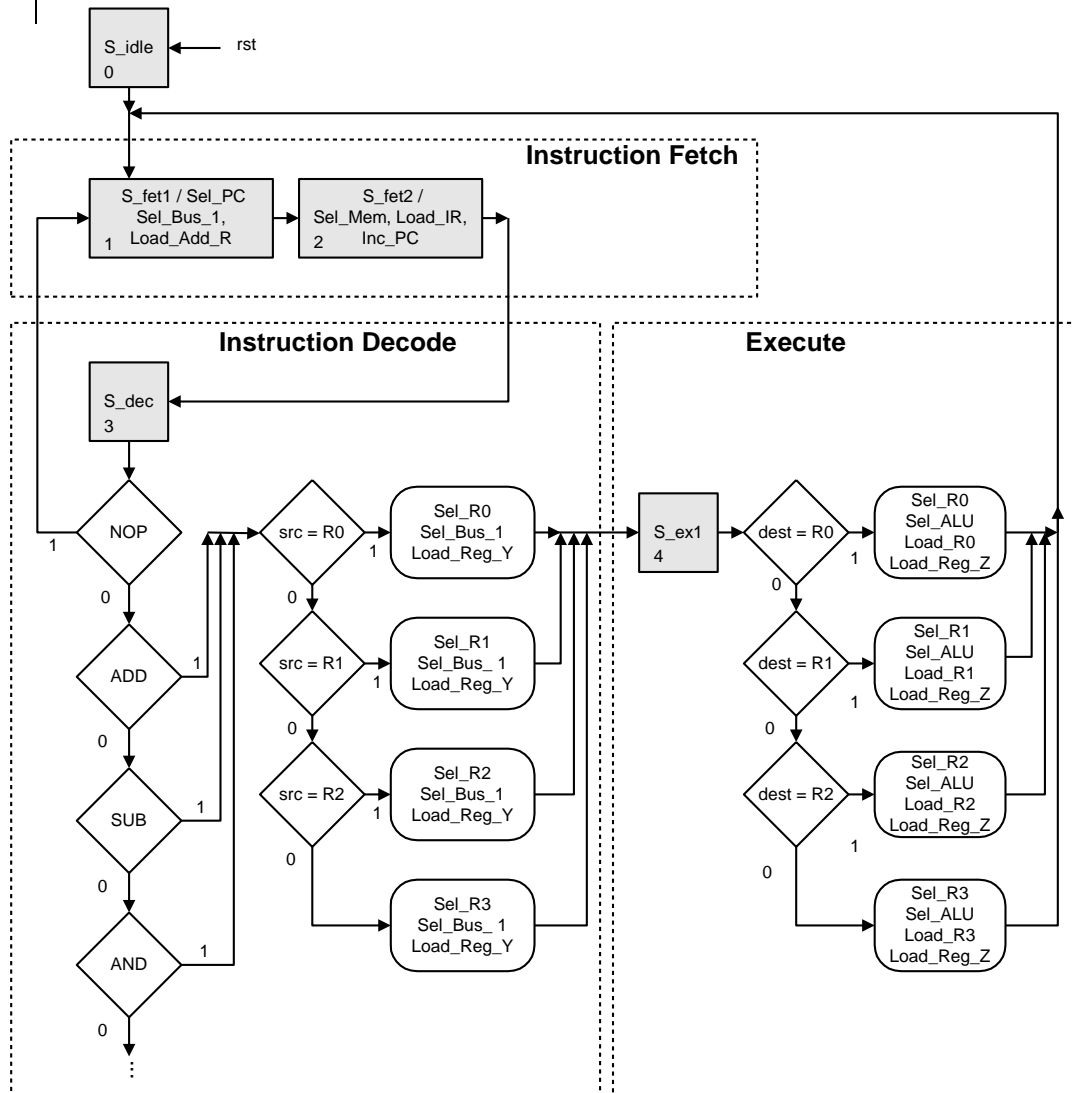
■ A total 11 states

- S_idle: initial state
- S_fet1: $\text{addr reg} \leftarrow \text{PC}$, S_fet2: $\text{load instruction} \leftarrow \text{mem}[\text{addr}]$
- S_dec: decode the instr and assert control signals to datapath
- S_ex1: execute the ALU for single-byte instr and load dest reg
- S_rd1: $\text{addr reg} \leftarrow \text{2nd byte of a RD instruction}$ and $\text{PC}++$
- S_rd2: $\text{dest} \leftarrow \text{mem}[\text{S_rd1}]$
- S_wr1: $\text{addr reg} \leftarrow \text{2nd byte of a WR instruction}$ and $\text{PC}++$
- S_wr2: $\text{mem}[\text{S_wr1}] \leftarrow \text{source}$
- S_br1: $\text{addr reg} \leftarrow \text{2nd byte of a BR instruction}$ and $\text{PC}++$
- S_br2: $\text{PC} \leftarrow \text{mem}[\text{S_br1}]$
- S_halt : trap failure to decode a valid instruction

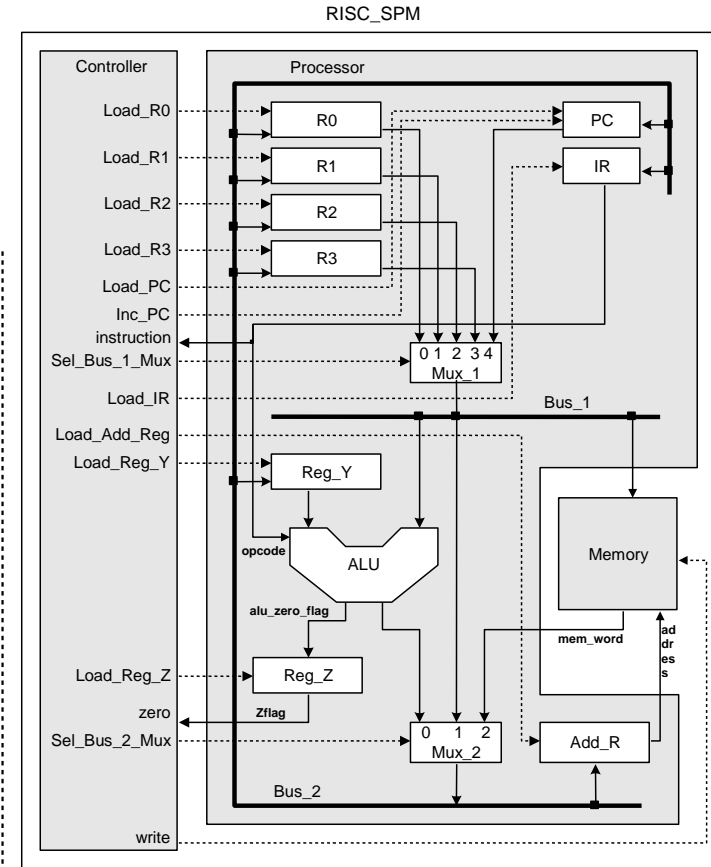
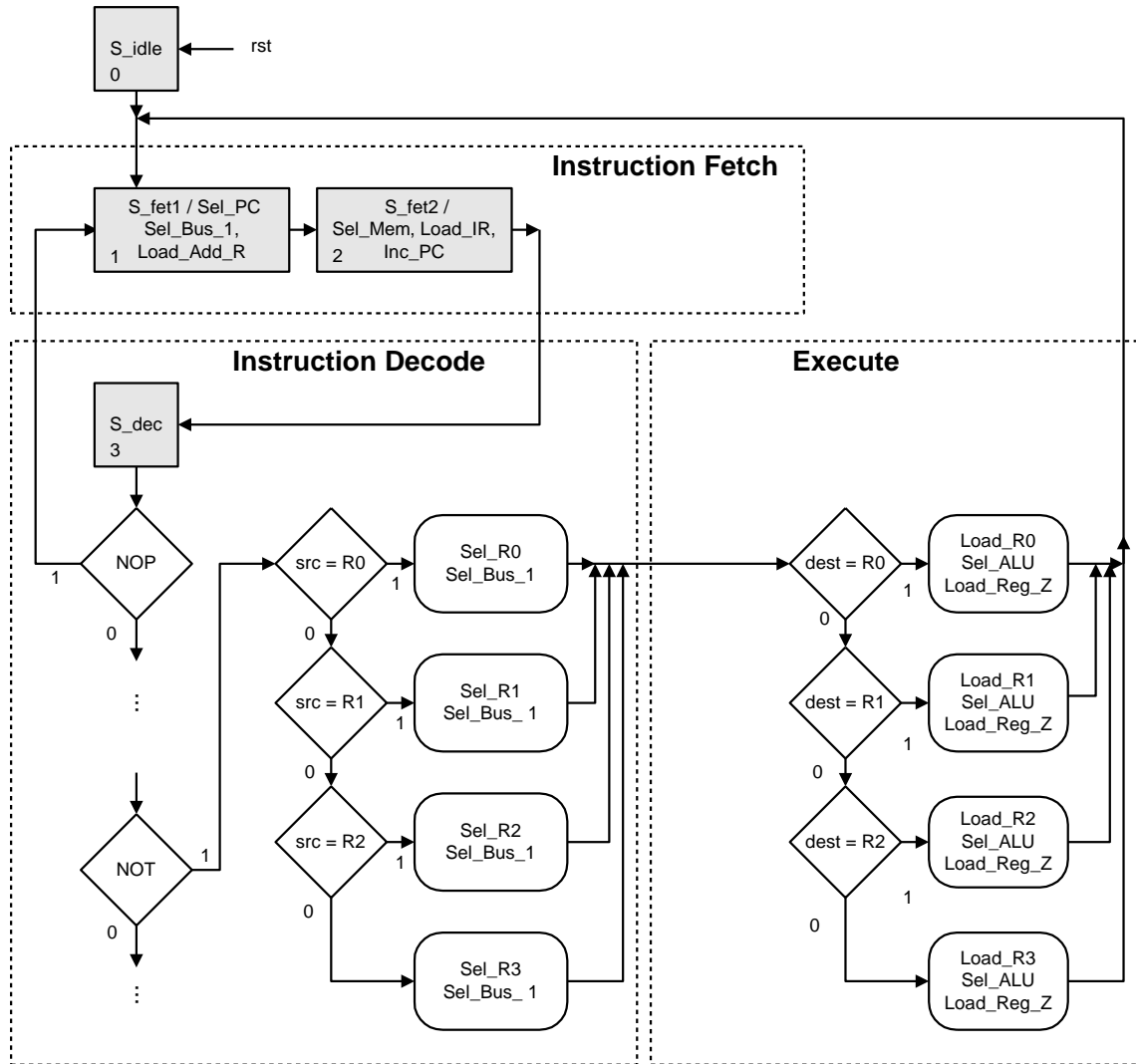
Instruction Sequence

- ❖ Fetch instruction from memory
- ❖ Decode instruction and fetch operands
- ❖ Execute instruction
 - ALU operations
 - Update storage registers
 - Update program counter (PC)
 - Update the instruction register (IR)
 - Update the address register (ADD_R)
 - Update memory
 - Control datapaths

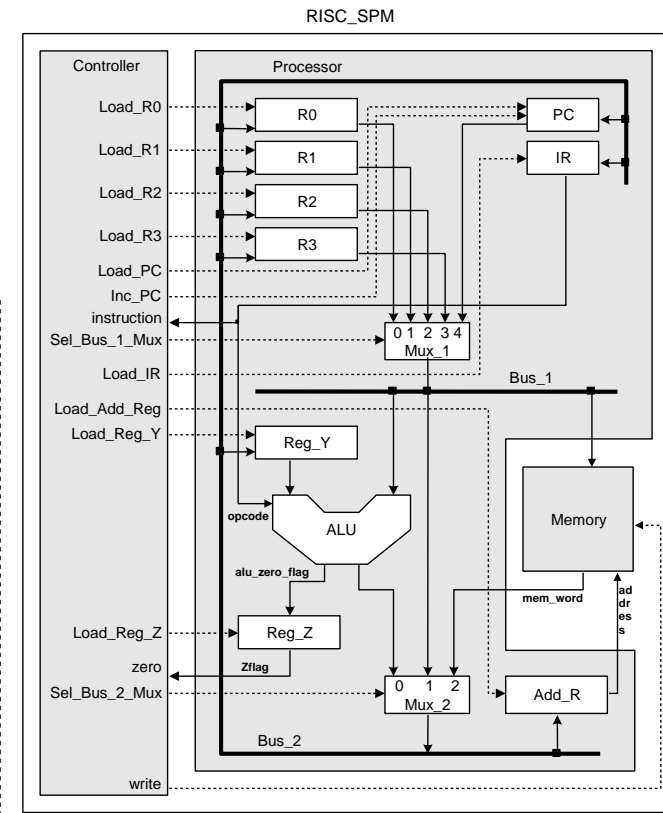
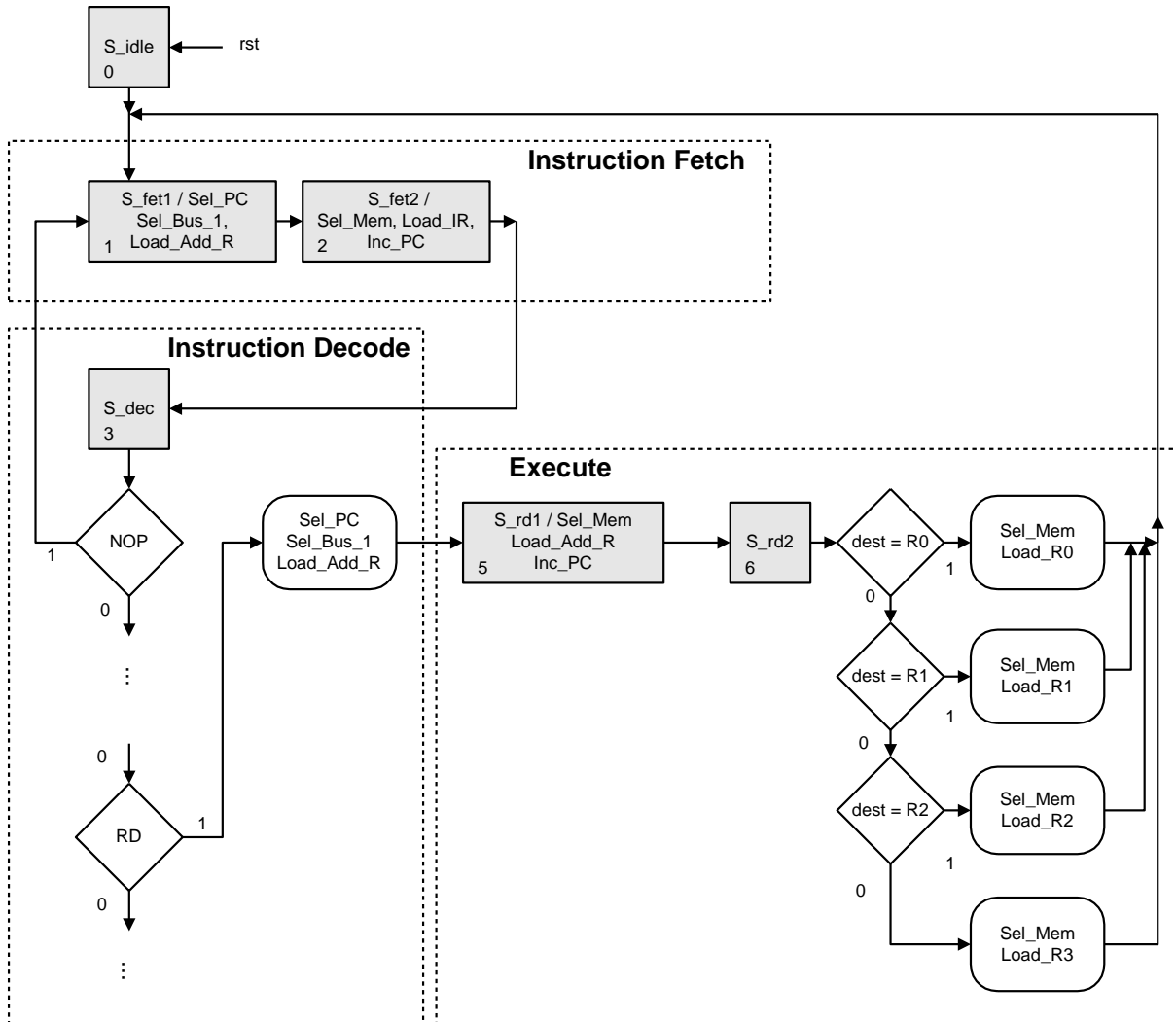
Controller ASM: NOP/ADD/SUB/AND



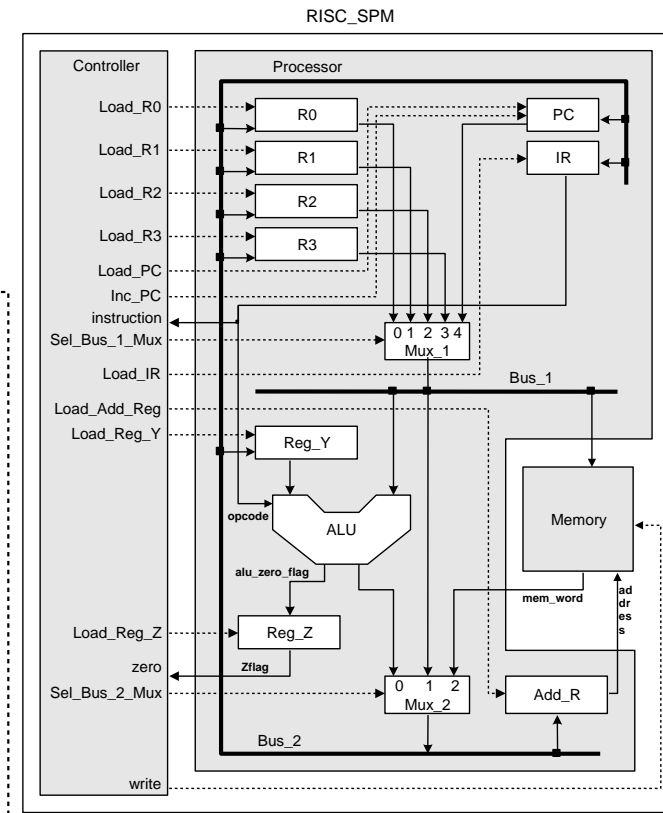
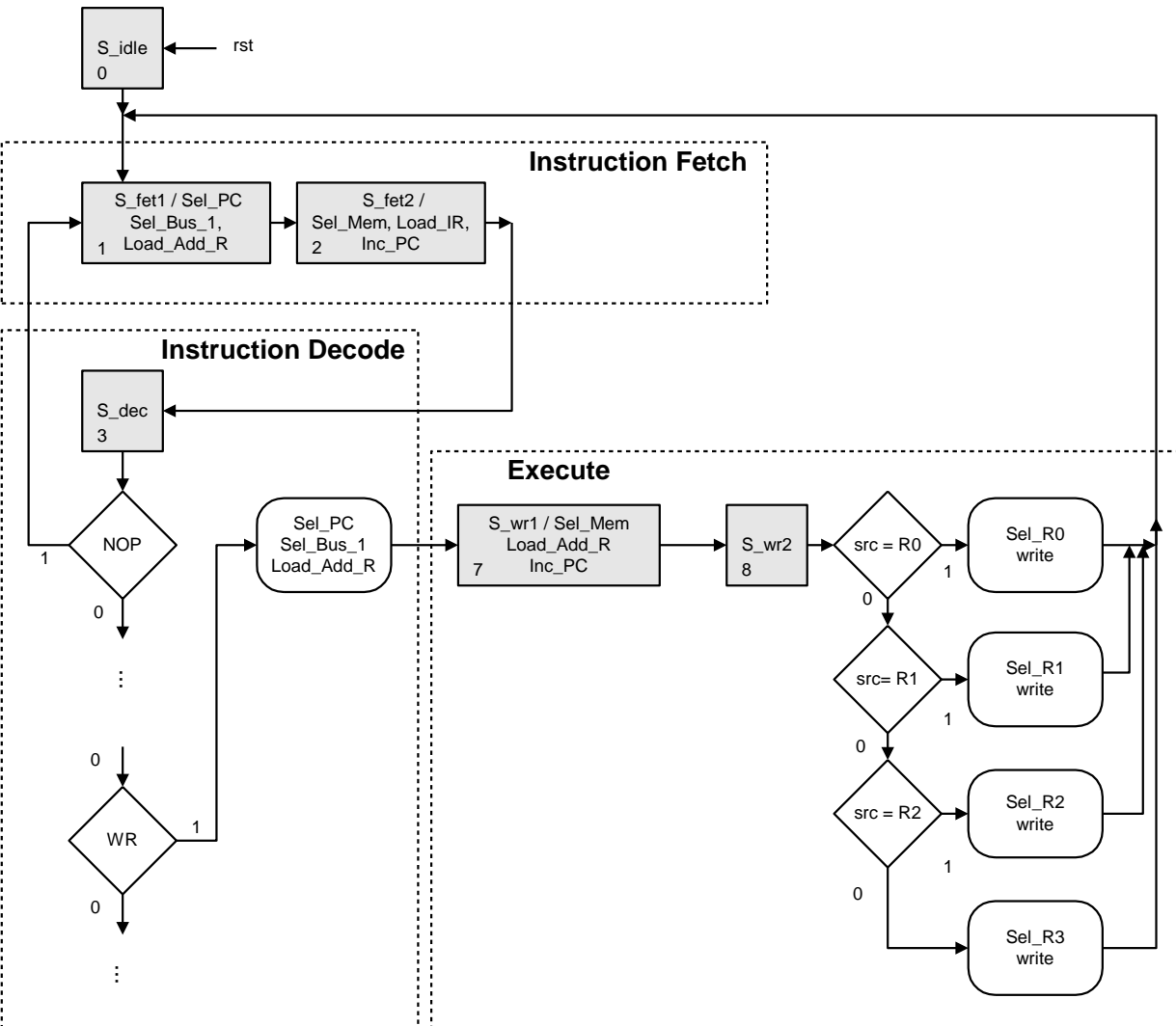
Controller ASM: NOT



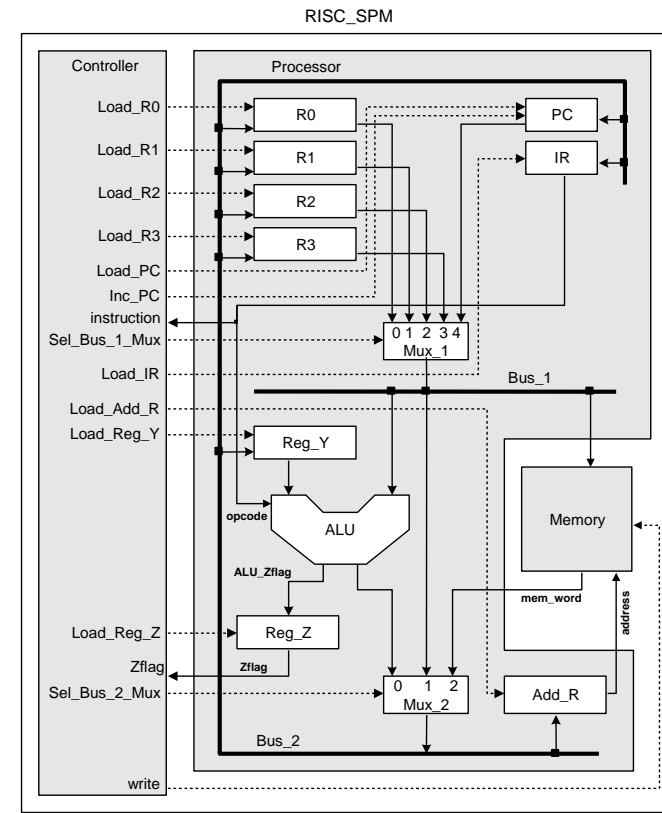
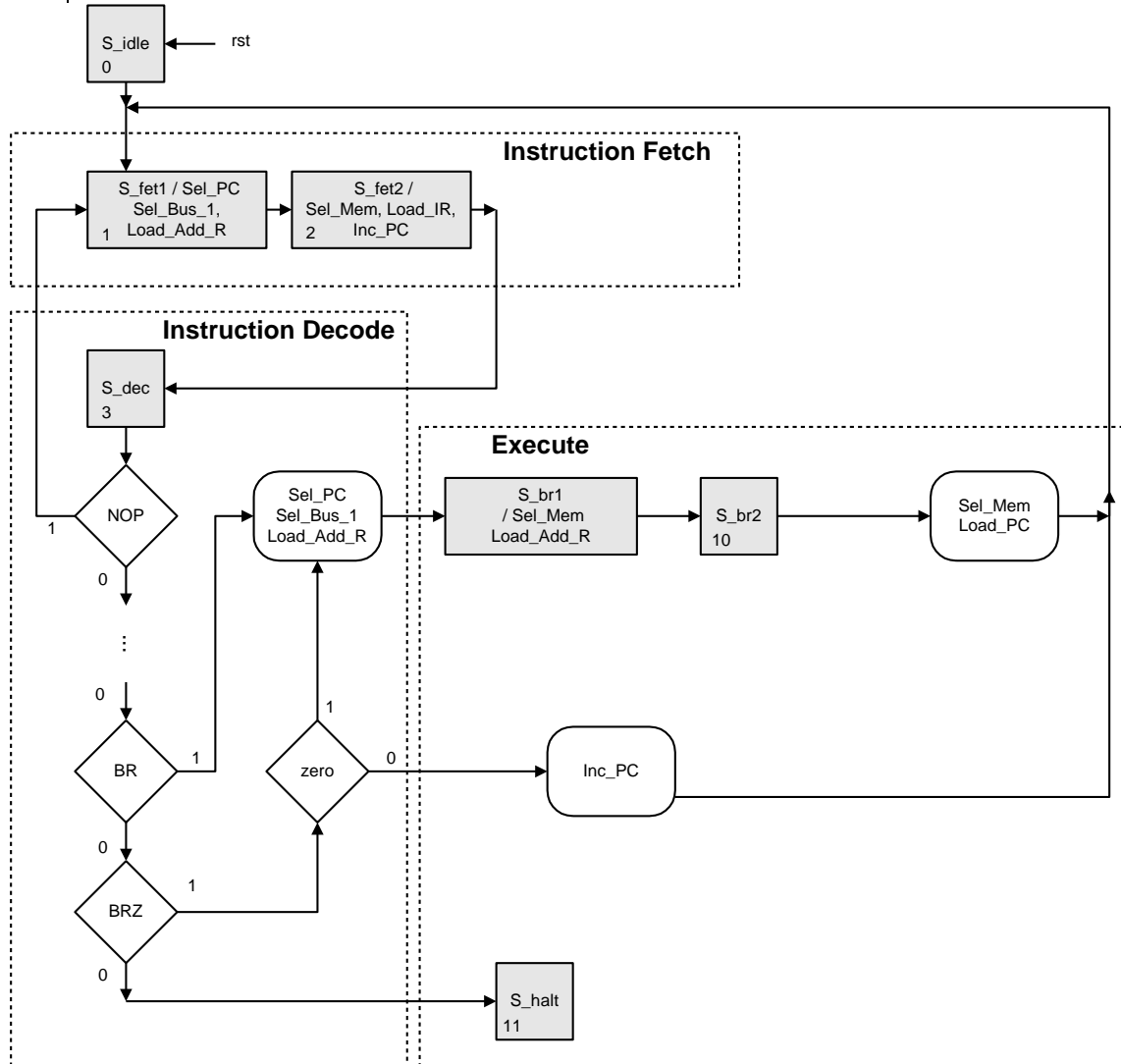
Controller ASM: RD



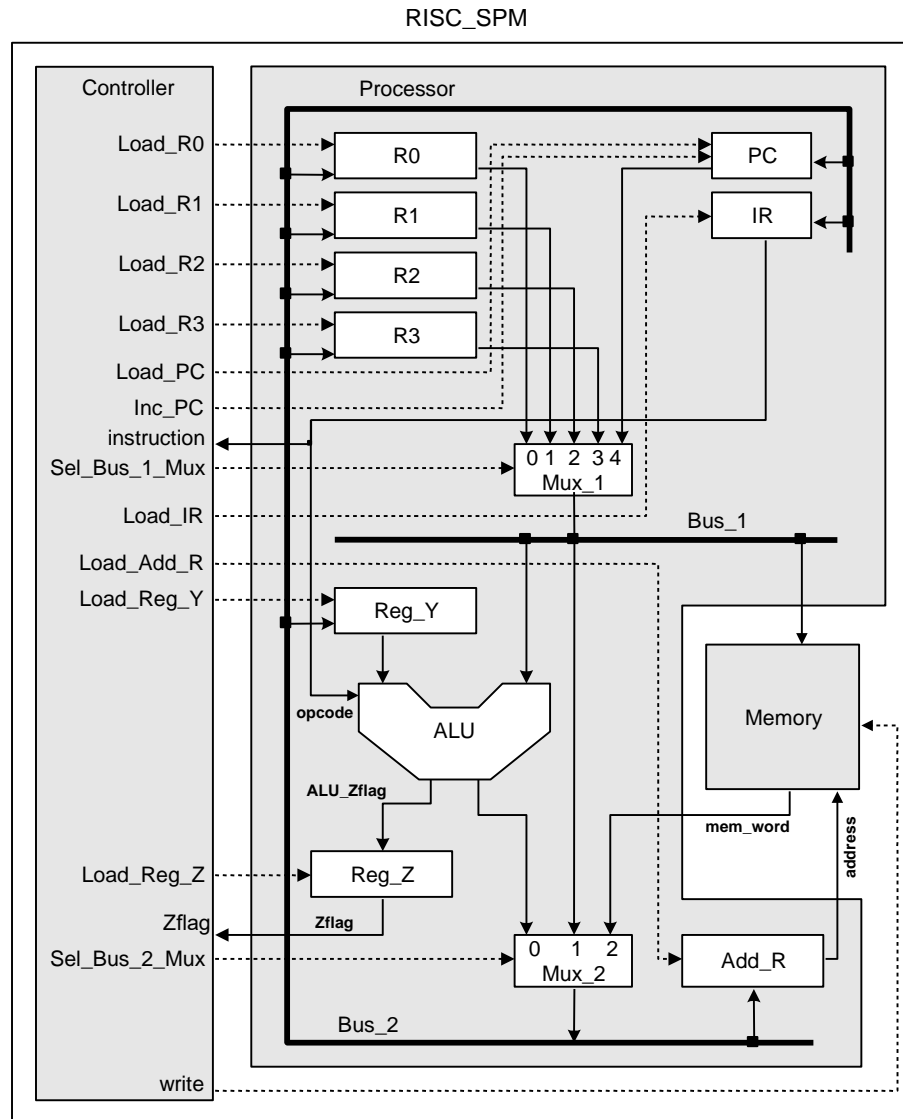
Controller ASM: WR



Controller ASM: BR/BRZ



RISC Stored Program Machine (SPM)



RISC-SPM [1]

```
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;
```

RISC-SPM [2]

```
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
```

```
/* For parameters that occur in multiple modules, include them in a separate file (e.g., definitions.v)
   and then use `include definitions.v */
```

```
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;

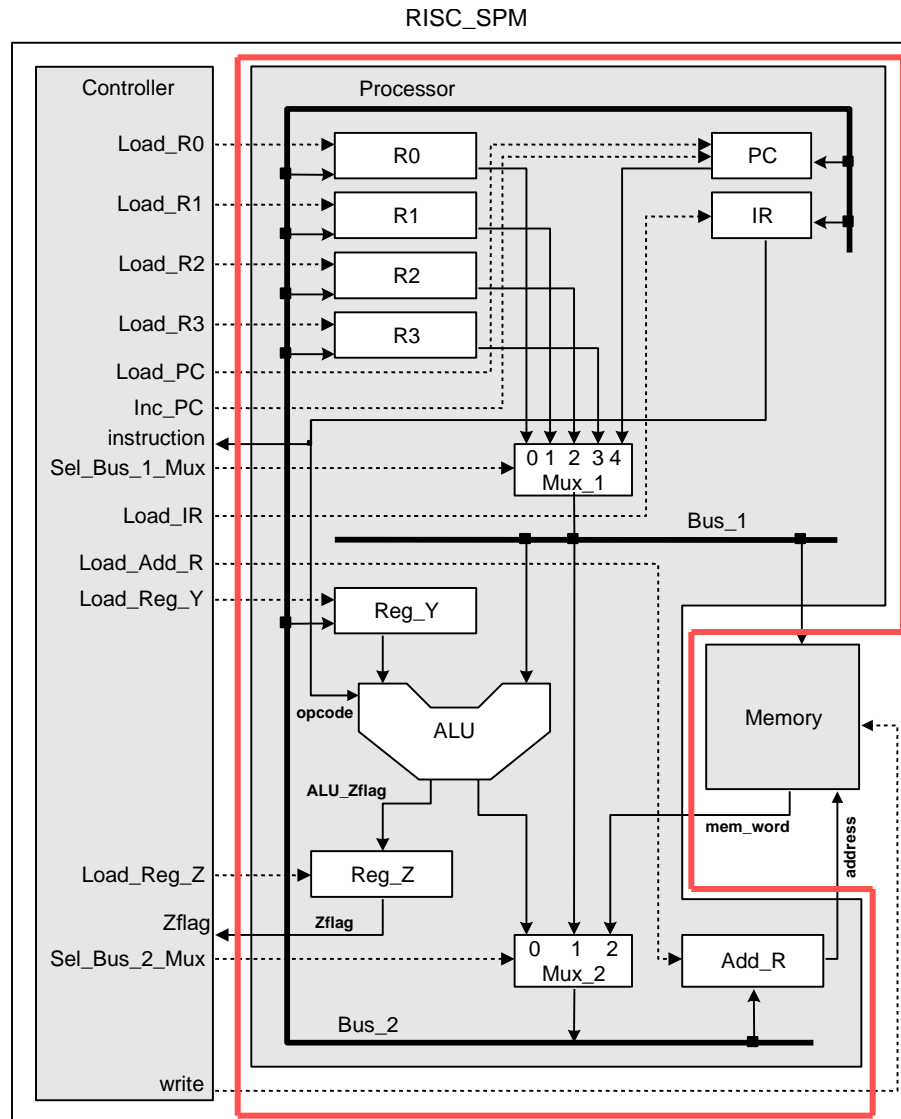
  // Control Nets
  wire Load_R0, Load_R1, Load_R2, Load_R3, Load_PC, Inc_PC, Load_IR;
  wire Load_Add_R, Load_Reg_Y, Load_Reg_Z;
  wire write;

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

Processing unit of the RISC SPM



17 signals except
rst and clk

Processing Unit [1]

```

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;

```

Processing Unit [2]

```

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);

```


Processing Unit [3]

```

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

```

Register unit, D_flop, and address register

```

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 [word_size-1: 0] 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;

```

Register unit, D_flop, and address register

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

Instruction register

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

Program Counter

```
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
```

5-Input Multiplexor

```
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

/* Implementation for Multiplexer_3ch is similar */
What would happen if we left out 'bx ?
```

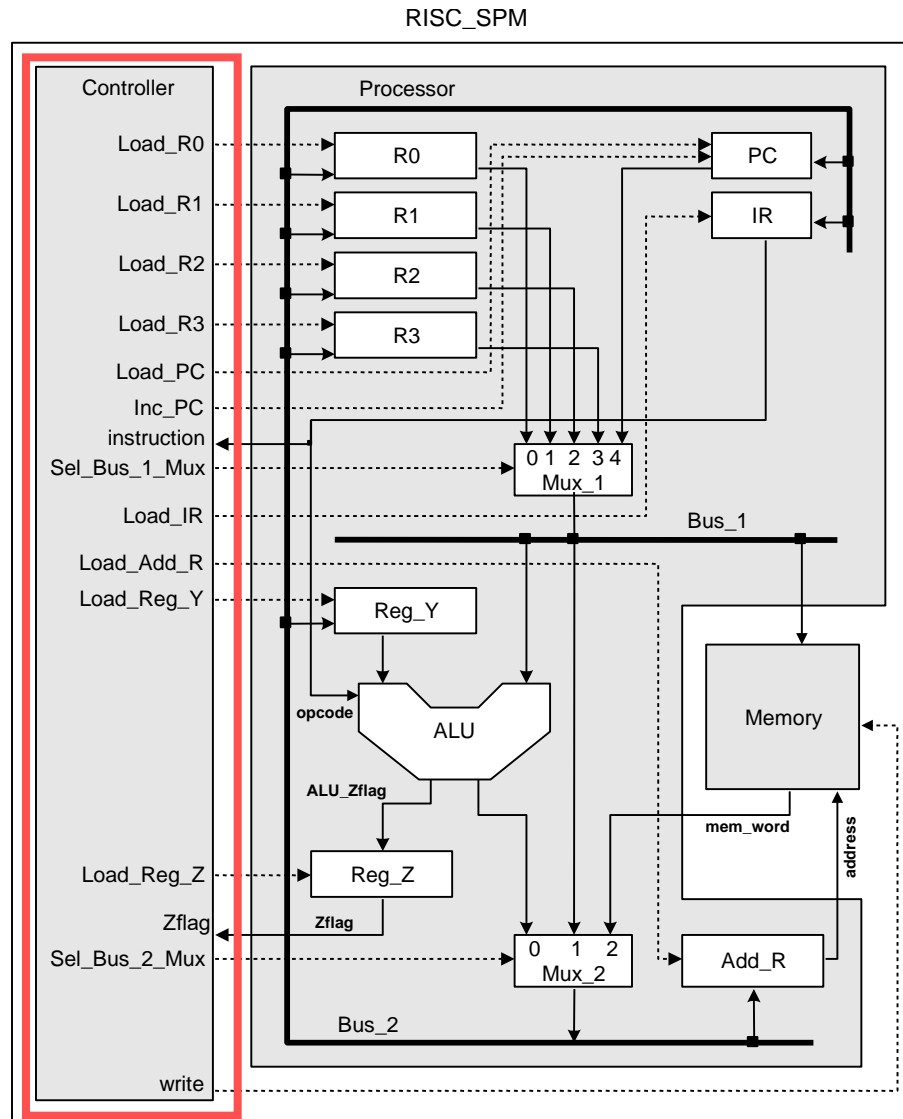
ALU

```

module Alu_RISC (alu_zero_flag, alu_out, data_1, data_2, sel);
  parameter word_size = 8, op_size = 4;
  // Opcodes
  parameter NOP = 4'b0000, ADD = 4'b0001, SUB = 4'b0010, AND = 4'b0011,
    NOT = 4'b0100, RD = 4'b0101, WR = 4'b0110, BR = 4'b0111, 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;
      SUB: alu_out = data_2 - data_1;
      AND: alu_out = data_1 & data_2;
      NOT: alu_out = ~ data_2;
      default: alu_out = 0;
    endcase
endmodule

```


Control unit of the RISC SPM



Control Unit [1]

```
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 Register Codes  
    parameter R0 = 0, R1 = 1, R2 = 2, R3 = 3;  
    // See textbook for output, input, and reg declarations
```

Control Unit [2]

```
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];
```

Control Unit [3]

// 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;

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** opcode **or** src **or** dest **or** zero)

begin: Output_and_next_state

 // set default values for control signals

 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;

Control Unit [4]

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_fet1

S_fet2: **begin** next_state = S_dec; Sel_Mem = 1; Load_IR = 1;
Inc_PC = 1; **end** // S_fet2

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

Control Unit [5]

```
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
```

Control Unit [6]

```

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
            else begin next_state = S_fet1; Inc PC = 1; end // BRZ
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_ex1

```

Control Unit [7]

```

S_rd1: begin next_state = S_rd2; Sel_Mem = 1; Load_Add_R = 1;
        Inc_PC = 1; end // S_rd1
S_wr1: begin next_state = S_wr2; Sel_Mem = 1; Load_Add_R = 1;
        Inc_PC = 1; end // S_wr1
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_rd2
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_wr2
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 //state case and always block
endmodule

```


Memory Unit

```
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  
  
/* How would a more realistic memory differ? */
```

Questions on RISC SPM

- ❖ Why not include memory as part of the processing unit?
- ❖ How could the design be simplified?

Testing RISC SPM

- ❖ Clear the memory
- ❖ Load the memory with a simple program and data
- ❖ Execute simple program
 - Reads values from memory into registers
 - Perform subtraction to decrement a loop counter
 - Add register contents while executing the loop
 - Branch to a halt when the loop index is 0
- ❖ Probe memory locations and control signals to ensure correct execution

Testing RISC SPM Verilog [1]

```
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, ..., word14;           // instructions  
  wire [word_size-1: 0] word128, word129, ..., word140;      // data  
  assign word0 = M2.M2_SRAM.memory[0];                      // words 1 to 13  
  assign word14 = M2.M2_SRAM.memory[14];  
  assign word128 = M2.M2_SRAM.memory[128];                  // words 129 to 139  
  assign word140 = M2.M2_SRAM.memory[140];
```

Testing RISC SPM Verilog [2]

```

initial #2800 $finish;           // set end point for simulation
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 //
    Flush_Memory
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 Mem[130] to R2
    M2.M2_SRAM.memory[2] = 130;                     // R2 = 2
    M2.M2_SRAM.memory[3] = 8'b0101_00_11;           // Read Mem[131] to R3
    M2.M2_SRAM.memory[4] = 131;                     // R3 = 0
    M2.M2_SRAM.memory[5] = 8'b0101_00_01;           // Read Mem[128] to R1
    M2.M2_SRAM.memory[6] = 128;                     // R1 = 6
    M2.M2_SRAM.memory[7] = 8'b0101_00_00;           // Read Mem[129] to R0
    M2.M2_SRAM.memory[8] = 129;                     // R0 = 1

```

Testing RISC SPM Verilog [3]

```
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 (139)
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;                // Holds address for BR (9)
// 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;                // Repeat Loop
end
endmodule
```