ASSIGNMENT -7
Advanced Digital Logic Design

BODDU MOURYA CHANDRA
1002022108

Contents

1) Design Requirements :

1)Simulating RISC_SPM code from section 7.3 of Ciletti.

2)Incorporating Branch On Overflow instruction(BRO) in RISC_SPM

3)Incorporating Multiply(MULT) Instruction in RISC_SPM

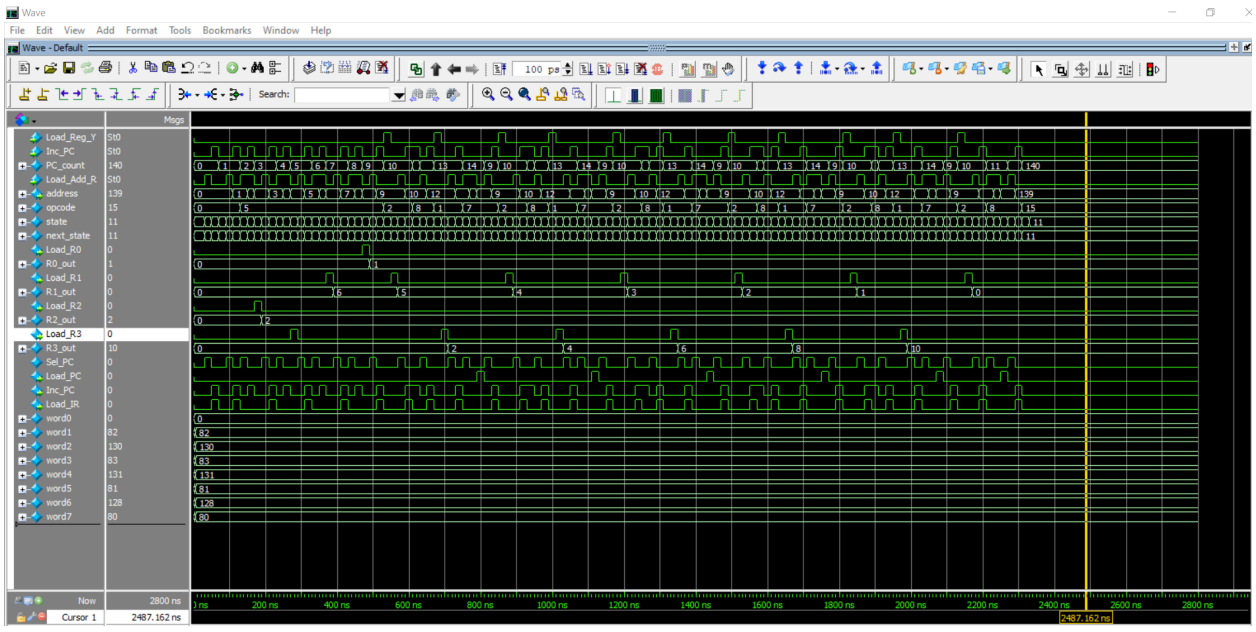
2) Test Bench Results :

Q1) TEXT BOOK EXAMPLE

Program:

```
// 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[139] = 8'b1111_00_00;         // HALT
M2.M2_SRAM.memory[140] = 9;                 //  Recycle
```

Modelsim Result :

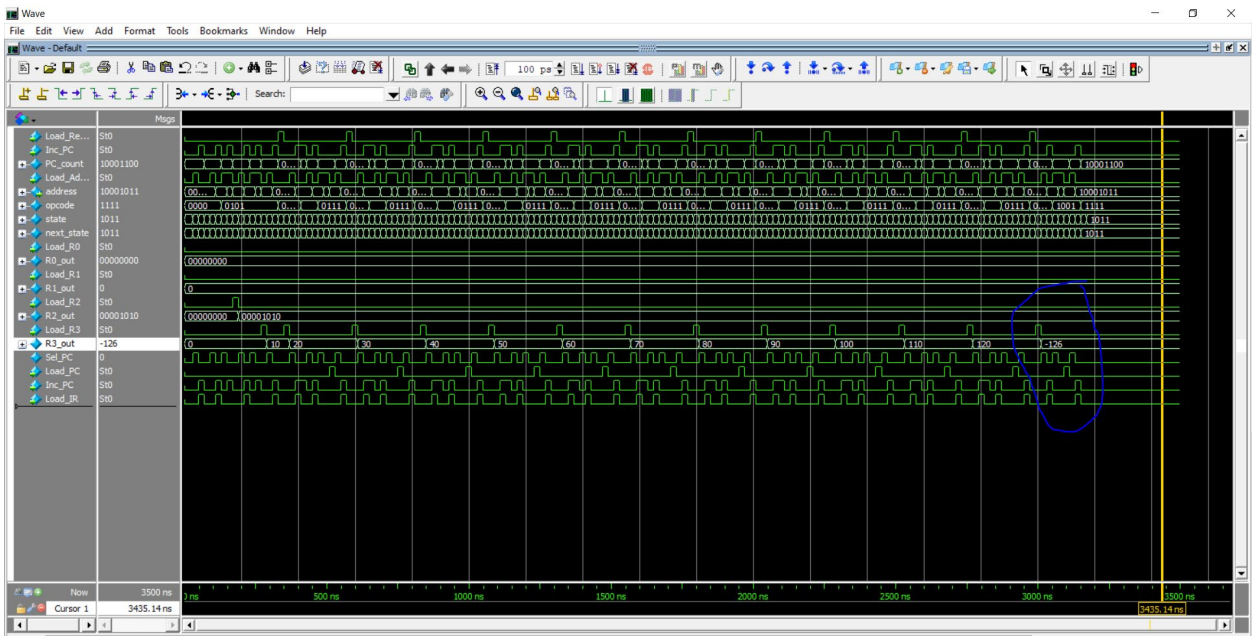


Q2)BRO INSTRUCTION

Program :

```
// Add the contents of R2 and R3 untill the R3 hits Overflow
#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'b0001_10_11;       // Add R2+R3 to R3
M2.M2_SRAM.memory[6] = 8'b1001_00_11;       // BRO
M2.M2_SRAM.memory[7] = 134;
M2.M2_SRAM.memory[8] = 8'b0111_00_11;       // BR
M2.M2_SRAM.memory[9] = 140;

// Load data
M2.M2_SRAM.memory[130] = 10;
M2.M2_SRAM.memory[131] = 10;
M2.M2_SRAM.memory[134] = 139;
M2.M2_SRAM.memory[139] = 8'b1111_00_00;         // HALT
M2.M2_SRAM.memory[140] = 5;                 //  Recycle
```

Modelsim Result :

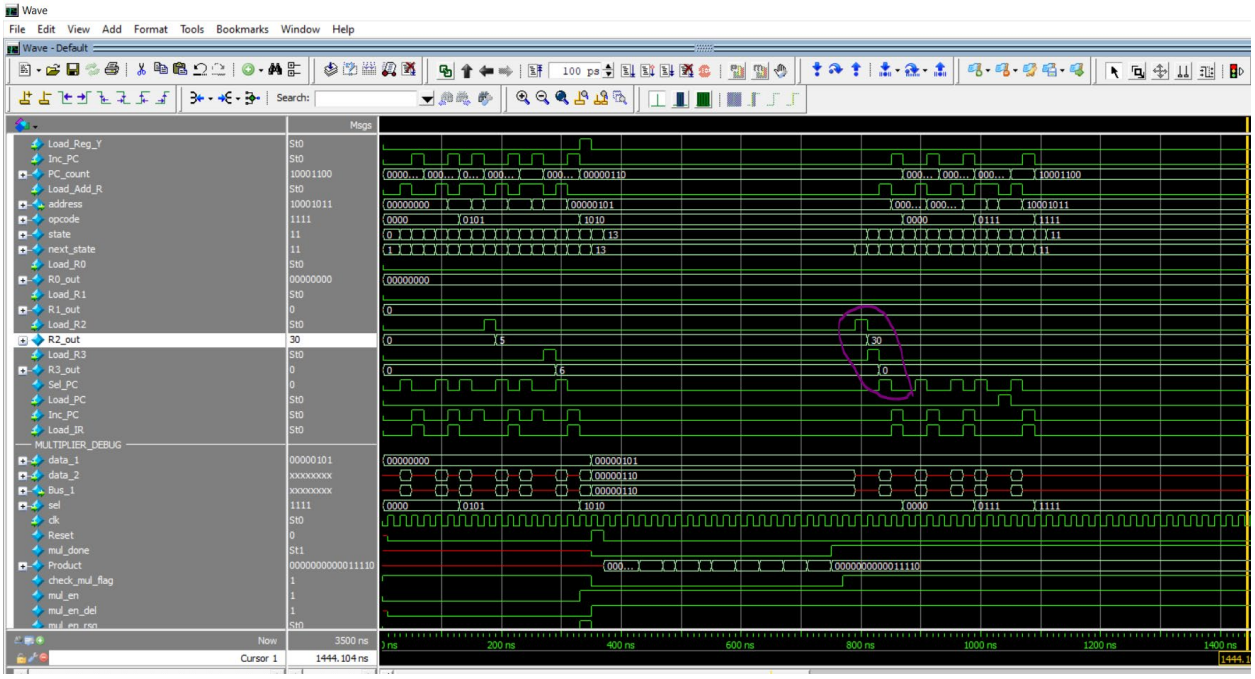


Q3)MUL INSTRUCTION

Program:

```
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'b1010_10_11;       // Multiply R2 and R3 and load R2 with Product_LSB and load R3 with product_MSB
M2.M2_SRAM.memory[8] = 8'b0111_00_11;       // BR
M2.M2_SRAM.memory[9] = 134;

// Load data
M2.M2_SRAM.memory[130] = 5;
M2.M2_SRAM.memory[131] = 6;
M2.M2_SRAM.memory[134] = 139;
M2.M2_SRAM.memory[139] = 8'b1111_00_00;         // HALT
M2.M2_SRAM.memory[140] = 5;                 //  Recycle
```

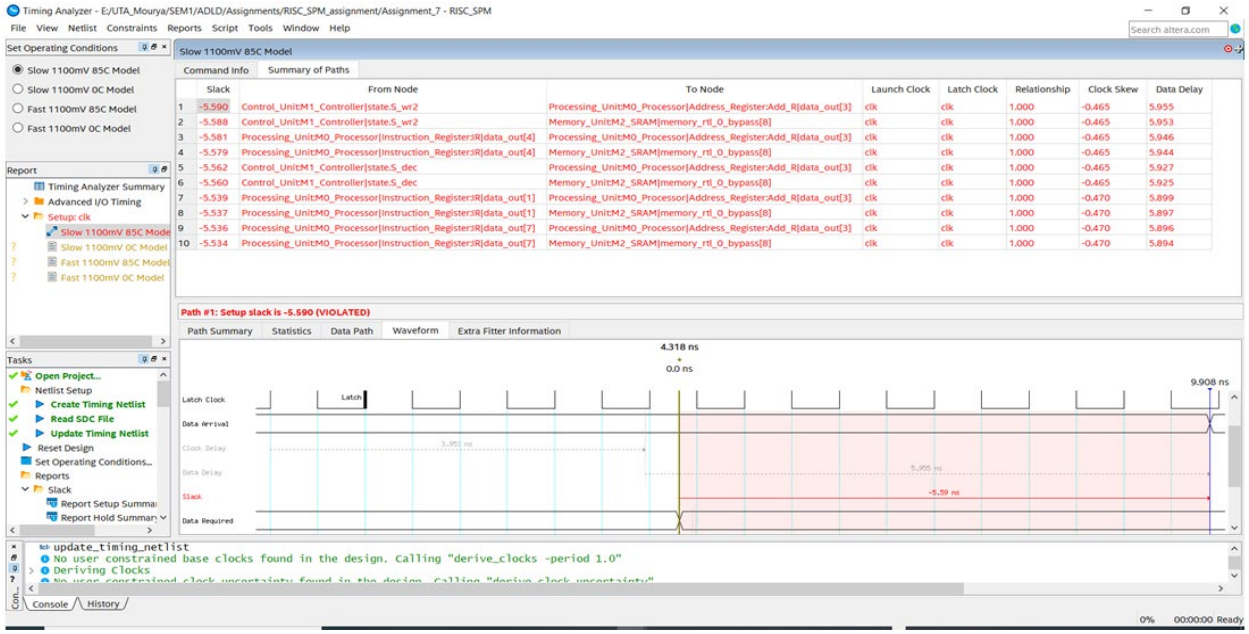
Modelsim Result :



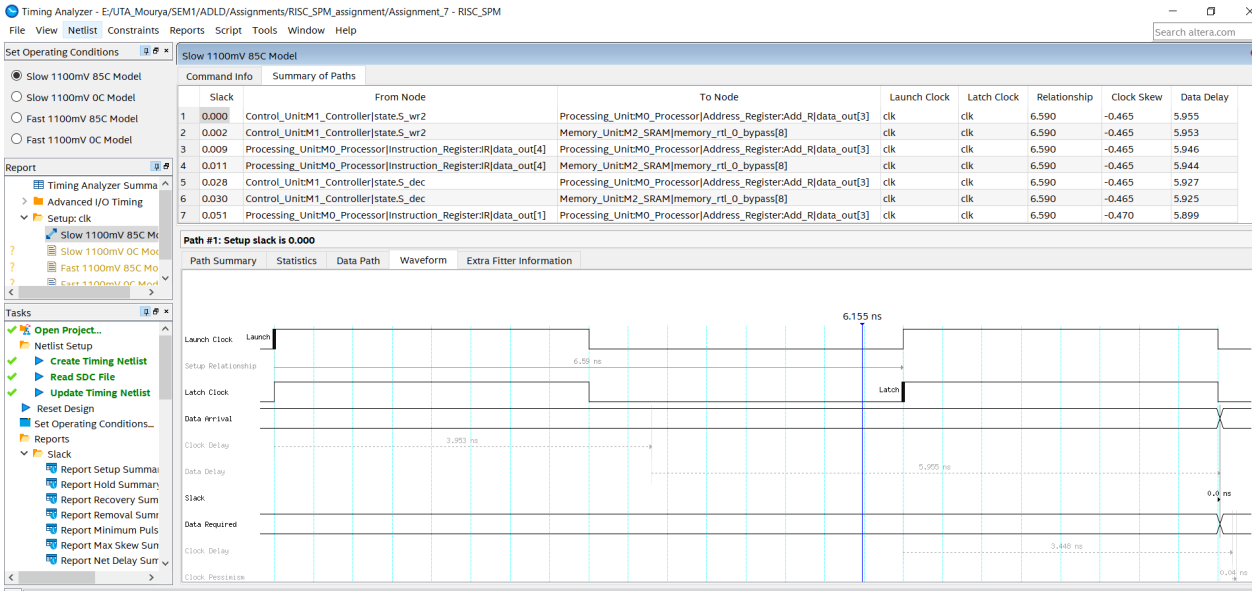
3) Timing Analysis:

Q1) TEXT BOOK EXAMPLE

- The below picture shows the timing analysis with 1ns clock .
- Slack = -5.590 ns

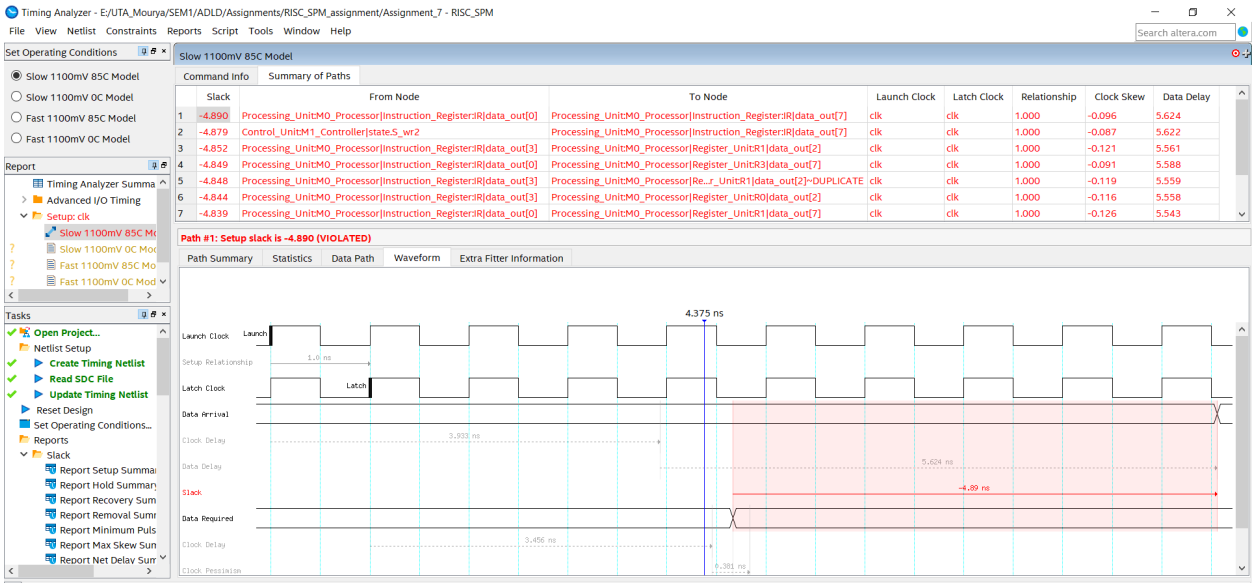


- The below picture shows the timing analysis with 6.590 ns clock period.
- Slack = 0 ns ,MAX_Frequency = 1/6.590 GHz =151.7 MHz

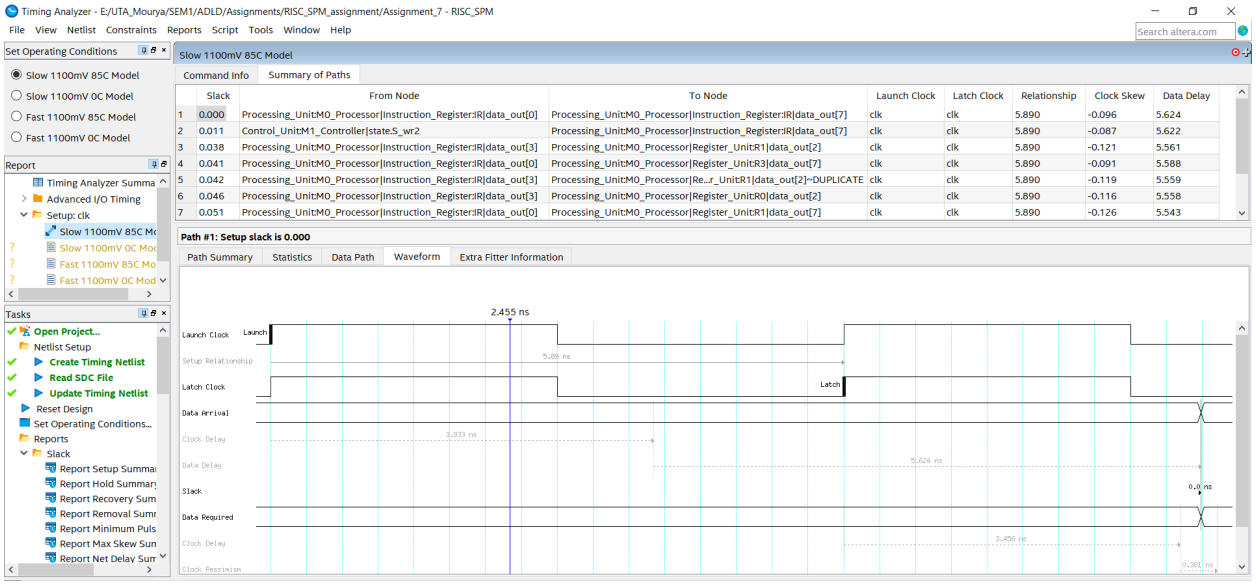


Q2) BRO INSTRUCTION INCLUDED

- The below picture shows the timing analysis with 1ns clock .
- Slack = -4.890 ns

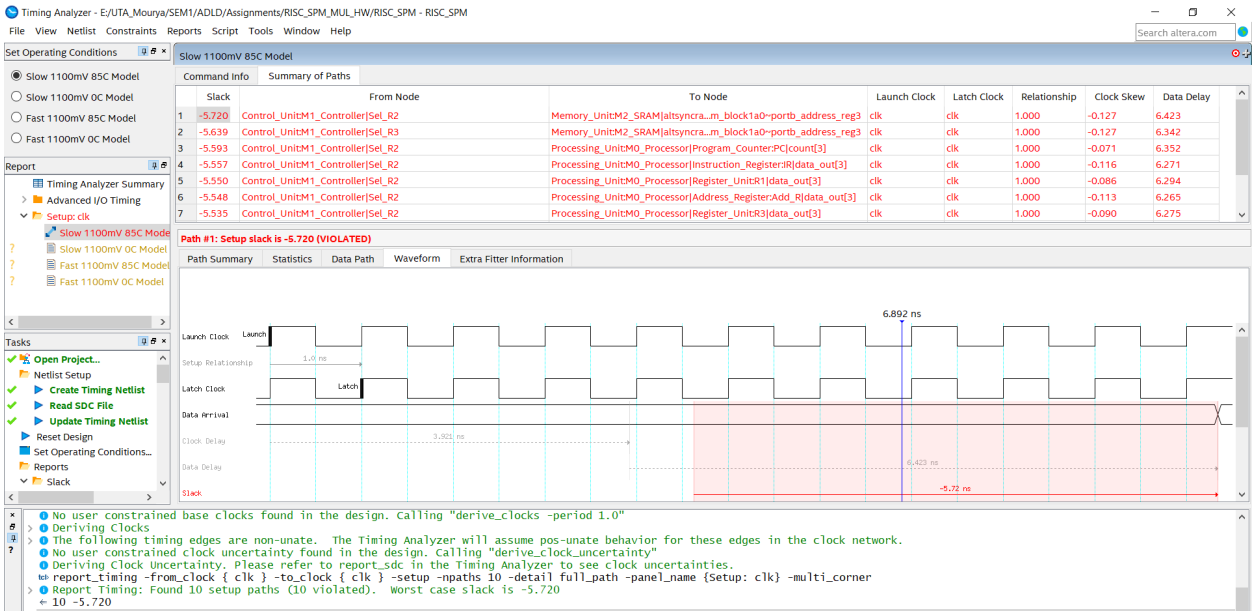


- The below picture shows the timing analysis with 5.890 ns clock period.
- Slack = 0 ns ,MAX_Frequency = 1/5.890 GHz =169.7 MHz

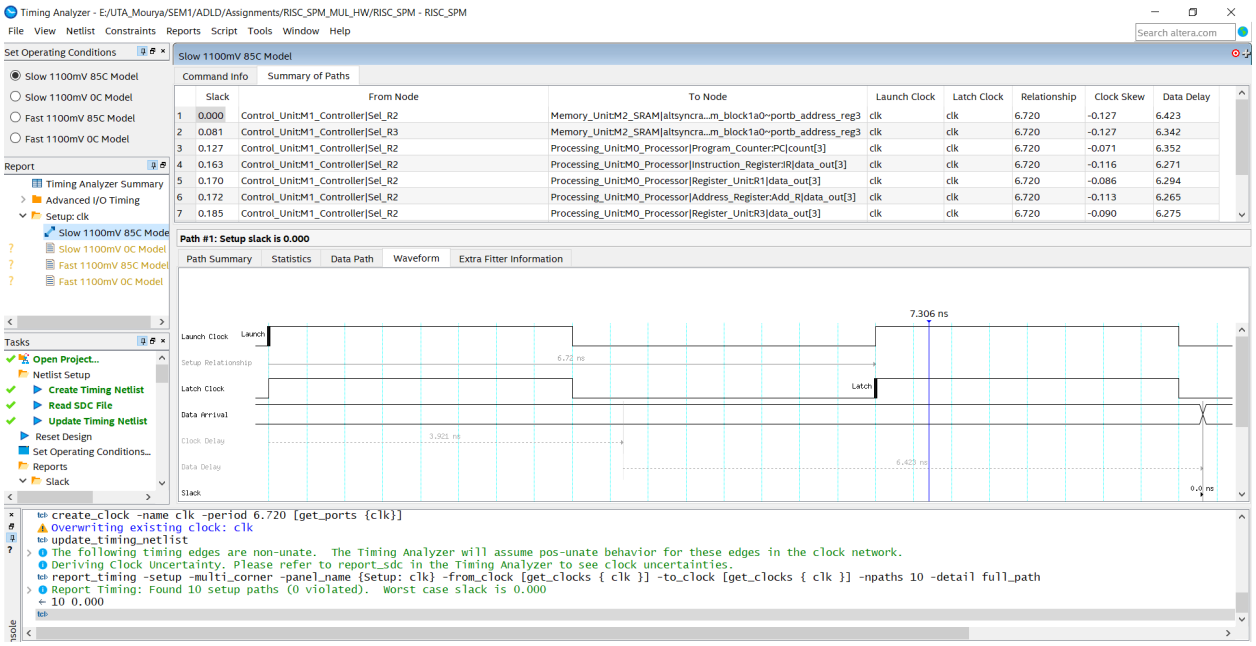


Q3)MULT INSTRUCTION INCLUDED

- The below picture shows the timing analysis with 1ns clock .
- Slack = -5.720 ns

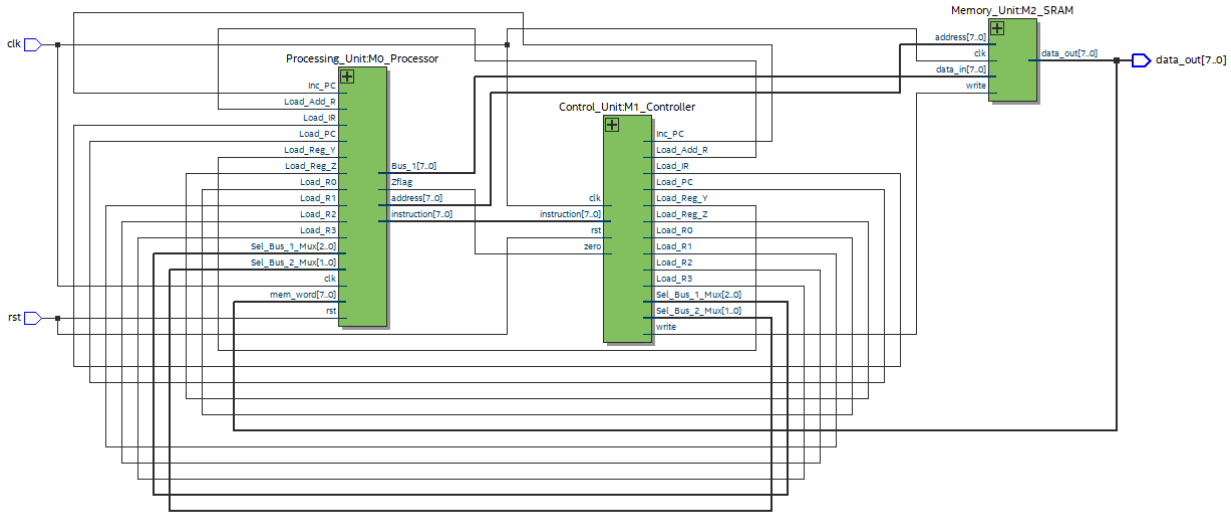


- The below picture shows the timing analysis with 6.590 ns clock period.
- Slack = 0 ns ,MAX_Frequency = 1/6.720 GHz =148.8 MHz

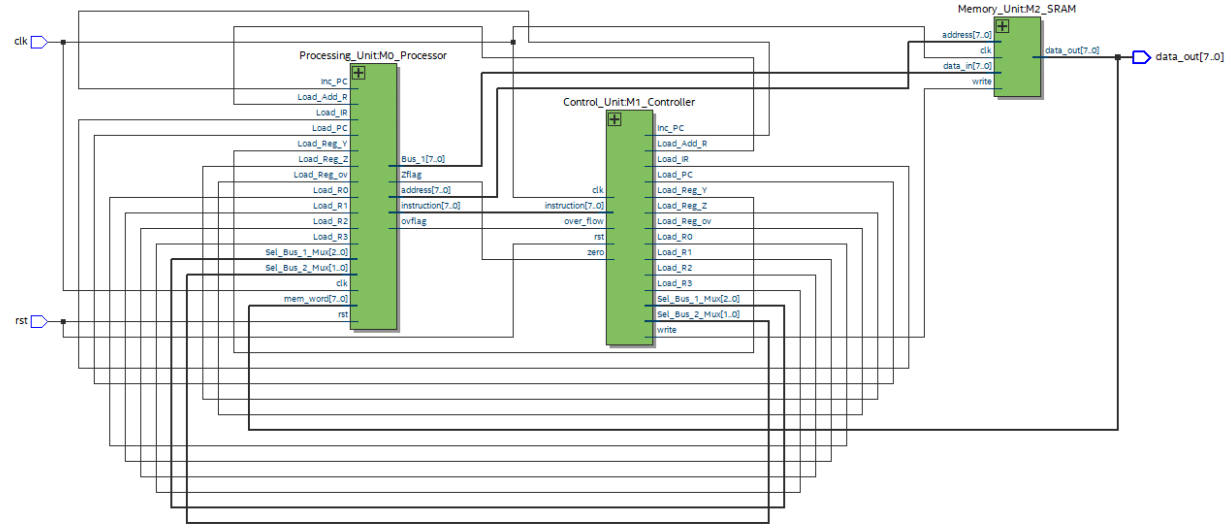


4)RTL Diagrams:

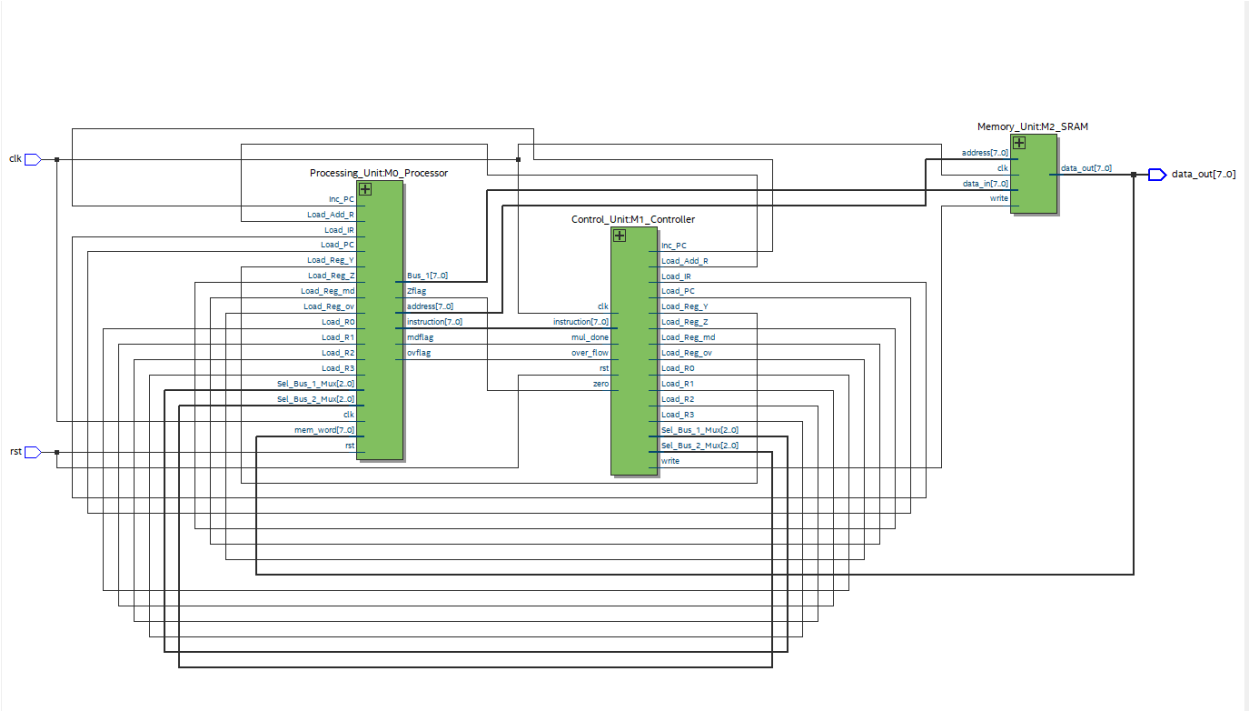
Q1) TEXT BOOK EXAMPLE



Q2)BRO INSTRUCTION



Q3)MULT INSTRUCTION



5)Verilog Codes :

1)RISC_SPM:

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

  input clk, rst;
  output [word_size-1: 0] data_out;

  assign data_out = mem_word  ;

  // Data Nets
  wire zero,over_flow,mul_done;
  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,Load_Reg_ov,Load_Reg_md;
  wire write;

  Processing_Unit M0_Processor (instruction,zero, over_flow,mul_done,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, Load_Reg_ov, Load_Reg_md,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,
Load_Reg_ov,Load_Reg_md,
    write, instruction, zero,over_flow,mul_done, clk, rst);

  Memory_Unit M2_SRAM (
    .data_out(mem_word),
    .data_in(Bus_1),
    .address(address),
    .clk(clk),
    .write(write) );

endmodule
```

2) Processing_Unit:

```
module Processing_Unit (instruction, Zflag,ovflag, mdflag,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,
Load_Reg_ov,Load_Reg_md,
 Sel_Bus_2_Mux, clk, rst);

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

 output [word_size-1: 0]        instruction, address, Bus_1              ;
 output                                  mdflag                  ;
 output                                  ovflag                  ;
 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,Load_Reg_ov,Load_Reg_md;
 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,mul_LSB_byte,mul_MSB_byte       ;
 wire                                    alu_zero_flag                   ;
 wire                                    alu_overflow_flag               ;
 wire                                    alu_mul_done_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_md          (mdflag, alu_mul_done_flag, Load_Reg_md, clk, rst);
 D_flop                  Reg_ov (ovflag, alu_overflow_flag, Load_Reg_ov, 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_5ch   Mux_2(Bus_2, alu_out, Bus_1, mem_word,mul_LSB_byte,mul_MSB_byte,
Sel_Bus_2_Mux)                    ;
  Alu_RISC            ALU   (alu_mul_done_flag,alu_overflow_flag,alu_zero_flag,
alu_out,mul_LSB_byte,mul_MSB_byte, Y_value, Bus_1, opcode,clk)                    ;

Endmodule
```

3) ALU:

```
module Alu_RISC (alu_mul_done_flag,alu_overflow_flag,alu_zero_flag,
alu_out,mul_LSB_byte,mul_MSB_byte, data_1, data_2, sel,clk);
 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                                          ;
 parameter BRO        = 4'b1001                                          ;
 parameter MULT       = 4'b1010                                          ;

 output                                      alu_overflow_flag          ;
 output                                      alu_zero_flag              ;
 output                                      alu_mul_done_flag          ;
 output       [word_size-1: 0]       alu_out                      ;
 output       [word_size-1: 0]       mul_LSB_byte                 ;
 output       [word_size-1: 0]       mul_MSB_byte                 ;
 input  [word_size-1: 0]       data_1, data_2               ;
 input  [op_size-1: 0]         sel                          ;
 input                                       clk                                    ;
 reg          [word_size-1: 0]       alu_out                      ;
 reg          [word_size-1: 0]       Multiplicand                 ;
 reg          [word_size-1: 0]       Multiplier                   ;
 reg                                         Reset                              ;
 wire                                        mul_done                           ;
 reg                                         mul_en                             ;
 reg                                         mul_en_del                         ;
 wire                                        mul_en_rsg                         ;
 wire    [(2*word_size)-1: 0]          Product                      ;
 reg                    check_mul_flag;

//ZERO FLAG
 assign  alu_zero_flag = ~|alu_out;

//OVERFLOW FLAG
 assign  alu_overflow_flag       =       (data_1[word_size - 1] & data_2[word_size - 1] &
~alu_out[word_size - 1])|(~data_1[word_size - 1] & ~data_2[word_size - 1] & alu_out[word_size - 1])
          ;

//MUL_DONE_FLAG
```

```
assign alu_mul_done_flag        =       mul_done        ;

initial
begin
        check_mul_flag  <=      1'b1    ;
        mul_en                  <=      1'b0    ;
end

        always @ (sel or data_1 or data_2)
        begin
                case  (sel)
                 NOP:   begin alu_out = 0;    end
                 ADD:   begin alu_out = data_1 + data_2;  mul_en = 0    ; end // Reg_Y + Bus_1
                 SUB:   begin alu_out = data_2 - data_1;mul_en = 0      ; end
                 AND:   begin alu_out = data_1 & data_2;mul_en = 0      ; end
                 NOT:   begin alu_out = ~ data_2; mul_en = 0    ; end     // Gets data from Bus_1
                 MULT: mul_en = 1       ;
                 default:       alu_out = 0       ;
                endcase
        end

        always @(posedge clk)
        begin
                mul_en_del      <=      mul_en ;
        end

        assign mul_en_rsg       = mul_en & ~(mul_en_del)        ;

        always @(posedge clk)
        begin
                if(mul_en_rsg == 1'b1 && check_mul_flag == 1'b1)
                begin
                        Reset                   <=      1'b1    ;
                        check_mul_flag          <=      1'b0    ;
                end
                else if(mul_done == 1'b1)
                begin
                        check_mul_flag          <=      1'b1    ;
                        Reset                   <=      1'b0    ;
                end
                else
                        Reset                   <=      1'b0    ;
        end


        assign mul_LSB_byte     =       Product[7:0]    ;
        assign mul_MSB_byte     =       Product[15:8]   ;
```

```
        Eight_Bit_Multiplier_RISC MULTIPLIER
        (
                .Clock(clk),
                .Reset(Reset)    ,
                .Multiplicand(data_1)   ,
                .Multiplier(data_2)      ,
                .Product(Product)        ,
                .mul_done(mul_done)
        );

Endmodule
```

4) Eight_Bit_Multiplier_RISC:

```
//Multiplier. Verilog behavioral model.
module Eight_Bit_Multiplier_RISC
	(
		input			Clock,
		input			Reset	,
		input	[7:0]		Multiplicand	,
		input	[7:0]		Multiplier	,
		output	[15:0]		Product		,
		output			mul_done
	);

	reg	[7:0]	RegQ	;	// Q register
	reg	[16:0]	RegA	;// A register
	reg	[16:0]	RegM	;	// M register
	reg	[2:0]	Count	;	//	3-bit iteration counter

	wire	C0, Start, Add, Shift, sub, sub_flag		;

	assign	Product = {RegA[7:0],RegQ}			;

	// 3-bit counter for #iterations
	always @(posedge Clock)
	if (Start == 1)
		Count <= 3'b00 ;	// clear in Start state
	else if (Shift == 1)
		Count <= Count + 1 ;		// increment in Shift state

	assign C0	= Count[2] & Count[1] & Count[0]	;	// detect count = 7

	// Multiplicand register (load only)
	always @(posedge Clock)
		if (Start == 1)
			RegM <= {{8{Multiplicand[7]}},Multiplicand}				;

	// Multiplier register (load, shift)
	always @(posedge Clock)
	if (Start == 1)
			RegQ <= Multiplier	;	// load in Start state
	else if (Shift == 1)
		RegQ <= {RegA[0],RegQ[7:1]}	;	// shift in Shift state

	// Accumulator register (clear, load, shift)
	always @(posedge Clock)
	if (Start == 1)
			RegA	<=	16'd0		;
	else if(sub == 1)
```

```
            RegA    <=      RegA - RegM     ;                   //subtract sub stae
        else if (Add == 1)
            RegA <= RegA + RegM             ;           // load in Add state
        else if(Shift == 1)                 // shift in Shift state
            RegA    <= RegA >> 1    ;


        // Instantiate controller module

        MultControl Ctrl
        (
            .Clock(Clock),
            .Reset(Reset),
            .Q0(RegQ[0]),
            .C0(C0),
            .Start(Start),
            .Add(Add)       ,
            .sub(sub)       ,
            .Shift(Shift),
            .Halt(mul_done)
        );

Endmodule
```

5) MultControl:

```
//Multiplier controller. Verilog behavioral model.
module MultControl
	(
		input Clock, Reset, Q0, C0,		//declare inputs
		output Start, Add,sub, Shift, Halt //declare outputs
	);

	reg [5:0] state;		//five states (one hot –one flip-flop per state)

	//one-hot state assignments for five states

	parameter StartS=6'b000001, TestS=6'b000010, AddS=6'b000100, ShiftS=6'b001000,
HaltS=6'b010000,subS=6'b100000	;

	// State transitions on positive edge of Clock or Resets

	always @(posedge Clock, posedge Reset)
	if (Reset==1)
		state <= StartS		;				//enter StartS state on Reset
	else
	//change state on Clock
	case (state)
	StartS:
			state <= TestS		;		// StartS to TestS
	TestS:
		if (Q0 == 1 && C0 == 1)
			state <= subS		;
		else if(Q0 == 1)
			state <= AddS		;				// TestS to AddS if Q0=1
		else
			state <= ShiftS	;				// TestS to ShiftS if Q0=0
	AddS:
			state <= ShiftS	;	// AddS to ShiftS
	subS:
			state <= ShiftS	;	// subS to ShiftS
	ShiftS:
		if (C0)
			state <= HaltS		;		// ShiftS to HaltS if C0=1
		else
			state <= TestS		;				// ShiftS to TestS if
C0=0
	HaltS:
			state <= HaltS		;				// stay in HaltS
	endcase
```

```
        // Moore model - activate one output per state
        assign Start    = state[0]      ;   // Start=1 in state StartS, else 0
        assign Add      = state[2]      ;   // Add=1 in state AddS, else 0
        assign Shift    = state[3]      ;   // Shift=1 in state ShiftS, else 0
        assign Halt     = state[4]      ;   // Halt=1 in state HaltS, else 0

        assign sub              = state[5]      ;

endmodule
```

6) Address_Register:

```
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
```

7) 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
```

8) Register_Unit:

```
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
```

9) D_flop:

```
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
```

10) Multiplexer_5ch:

```
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
```

11) 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    [word_size-1: 0]        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
```

12) Control_Unit:

```
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, Load_Reg_ov,Load_Reg_md,
 write, instruction, zero,over_flow,mul_done, clk, rst);

 parameter word_size = 8, op_size = 4, state_size = 4;
 parameter src_size = 2, dest_size = 2, Sel1_size = 3, Sel2_size = 3;
 // 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,S_ex2 = 12,mul_done_wait = 13,LD_MUL_MSB=14;
 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            ,MULT = 10;
 parameter RD  = 5, WR =  6,  BR =  7, BRZ = 8,BRO = 9    ;
 // 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,Load_Reg_ov,Load_Reg_md;
 output [Sel2_size-1: 0] Sel_Bus_2_Mux;
 output write;
 input [word_size-1: 0] instruction;
 input zero;
 input over_flow;
 input mul_done;
 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,Sel_mul_LSB,Sel_mul_MSB;
 reg Sel_R0, Sel_R1, Sel_R2, Sel_R3, Sel_PC;
 reg Load_Reg_Z, write,Load_Reg_ov,Load_Reg_md;
 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];
 reg  temp = 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:
                                         Sel_mul_LSB ? 3:
                                         Sel_mul_MSB ? 4: 3'bx;

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

  //always @ (posedge clk or state or opcode or zero) begin: Output_and_next_state
  always @ (posedge clk ) begin: Output_and_next_state
  //always @ (posedge clk ) 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;
Load_Reg_ov = 0;Load_Reg_md=0        ;
    Inc_PC                  = 0             ;
    Sel_Bus_1      = 0                      ;
    Sel_ALU        = 0                      ;
    Sel_Mem        = 0                      ;
    Sel_mul_LSB = 0                         ;
    Sel_mul_MSB = 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//Fetching the next instruction without incrementing PC if the condition satis
                        next_state = S_br1;
                        Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
            end // BRZ
else
            begin //If the condition fails then the code in the next address shouldn't be fetch
             next_state = S_fet1;
             Inc_PC = 1;
            end

BRO: if (over_flow == 1)
                        begin
                        next_state = S_br1; //Fetching the next instruction without incrementing PC if
                        Sel_PC = 1; Sel_Bus_1 = 1; Load_Add_R = 1;
            // BRZ
            else //If the condition fails then the code in the next address shouldn't be fetched an
                        begin
                                    next_state = S_fet1;
                                    Inc_PC = 1;
                        end

            MULT: begin
                                                next_state      = S_ex2 ;
                                                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 // MULT

                        default :
                                                next_state = S_halt;
                        endcase  // (opcode)
```

```
S_ex1:          begin
                        next_state = S_fet1;
                        Load_Reg_Z = 1;
                        Load_Reg_ov  =      1;      //BRO
                        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_ex2:          begin
                        next_state    = mul_done_wait      ;
                        //Load_Reg_md       = 1                           ;
                        //temp                          =~temp;
                                case  (dest)
                                        R0: Sel_R0 = 1;
                                        R1: Sel_R1 = 1;
                                        R2: Sel_R2 = 1;
                                        R3: Sel_R3 = 1;
                                        default : err_flag = 1;
                                endcase
                        end


mul_done_wait:begin
                        if(mul_done == 1)
                        begin
                                next_state = LD_MUL_MSB                 ;
                                Load_Reg_Z    =      1                                ;
                                Load_Reg_ov   =      1                                ;
                                Sel_mul_LSB =  1                                ;
                                 case  (src)
                                        R0: Load_R0 = 1                 ;
                                        R1: Load_R1 = 1                 ;
                                        R2: Load_R2 = 1                 ;
                                        R3: Load_R3 = 1                 ;
                                        default : err_flag = 1          ;
                                endcase
                        end
                else
                        begin
                         next_state    = mul_done_wait                           ;
                         Load_Reg_md = 1                                ;
                         temp                   =~temp ;// for debug
```

```
                        case  (dest)
                                R0: Sel_R0 = 1;
                                R1: Sel_R1 = 1;
                                R2: Sel_R2 = 1;
                                R3: Sel_R3 = 1;
                                default : err_flag = 1;
                            endcase
                        end
                        end

LD_MUL_MSB:  begin
                    next_state = S_fet1;
                    Load_Reg_Z      =       1                       ;
                    Load_Reg_ov     =       1                       ;
                    Sel_mul_MSB = 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_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
```

13) 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
```