Small modules

1. sign extend

a. code

```
1  module sign_extend( input  [15:0] sign_ex_in,
2                      output reg [31:0] sign_ex_out);
3
4
5   always@(sign_ex_in)
6  begin
7
8   sign_ex_out = { {16{sign_ex_in[15]}}, sign_ex_in};
9
10  end
11
```

b. TB

```
module sign_extend_tb;

reg  [15:0] sign_ex_in;
wire [31:0] sign_ex_out;

sign_extend extension1 (sign_ex_in, sign_ex_out);

initial begin
        $monitor("Time = %t | sign_ex_in = %b | sign_ex_out = %b",$time, sign_ex_in, sign_ex_out);

        sign_ex_in = 16'b1010101010101010;
        #10;
        sign_ex_in = 16'b0101010101010101;
        #10;
        sign_ex_in = 16'b0000000000000000;
        #10;
        sign_ex_in = 16'b1111111111111111;
        #10;
        sign_ex_in = 16'b0000000011111111;
        #10;
        sign_ex_in = 16'b1111111100000000;
end
initial #100 $finish ;
endmodule
```

c. RTL Viewer

d. Result

```
VSIM 3> run
# Time =                    0 | sign_ex_in = 1010101010101010 | sign_ex_out = 11111111111111111010101010101010
# Time =                   10 | sign_ex_in = 0101010101010101 | sign_ex_out = 00000000000000000101010101010101
# Time =                   20 | sign_ex_in = 0000000000000000 | sign_ex_out = 00000000000000000000000000000000
# Time =                   30 | sign_ex_in = 1111111111111111 | sign_ex_out = 11111111111111111111111111111111
# Time =                   40 | sign_ex_in = 0000000011111111 | sign_ex_out = 00000000000000000000000011111111
# Time =                   50 | sign_ex_in = 1111111100000000 | sign_ex_out = 11111111111111111111111100000000
```

2. parameterized shift_left_twice

a. code

```
module shift_left_twice #(parameter width = 31)
(input       [width:0] shift_in,
 output  reg [31:0] shift_out);

always @(shift_in)
begin
shift_out = shift_in << 2;
end

endmodule
```

b. TB

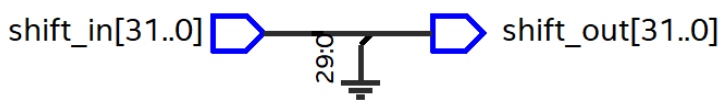
1. when 32-bit input

```
module shift_left_twice_32_bit_tb;

reg  [31:0] shift_in;
wire [31:0] shift_out;

shift_left_twice #(.width(31)) shifter1
(shift_in, shift_out);

initial
begin
     $monitor("time = %t  |  shift_in = %b  | shift_out = %b", $time, shift_in, shift_out);
     shift_in = 32'b10101010101010101010101010101010;
     #10;
     shift_in = 32'b11111111111111111111111111111111;
     #10;
     shift_in = 32'b00000000000000000000000000000000;
     #10;
     shift_in = 32'b00000000000000001111111111111111;
end

initial #150 $finish;

endmodule
```

2. when 26-bit input

```
module shift_left_twice_26_bit_tb;

reg  [25:0] shift_in;
wire [31:0] shift_out;

shift_left_twice #(.width(25)) shifter1
(shift_in, shift_out);

initial
begin
     $monitor("time = %t  |  shift_in = %b  | shift_out = %b", $time, shift_in, shift_out);
     shift_in = 26'b10101010101010101010101010;
     #10;
     shift_in = 26'b11111111111111111111111111;
     #10;
     shift_in = 26'b00000000000000000000000000;
     #10;
     shift_in = 26'b00000000000001111111111111;
end

initial #150 $finish;

endmodule
```

c. RTL viewer



d. Result

```
# Loading work.shift_left_twice_26_bit_tb
# Loading work.shift_left_twice
VSIM 3> run
# time =                    0  |  shift_in = 10101010101010101010101010  | shift_out = 00001010101010101010101010101000
# time =                   10  |  shift_in = 11111111111111111111111111  | shift_out = 00001111111111111111111111111100
# time =                   20  |  shift_in = 00000000000000000000000000  | shift_out = 00000000000000000000000000000000
# time =                   30  |  shift_in = 00000000000001111111111111  | shift_out = 00000000000000001111111111111100
VSIM 4> vsim work.shift_left_twice_32_bit_tb
# vsim work.shift_left_twice_32_bit_tb
# Loading work.shift_left_twice_32_bit_tb
# Loading work.shift_left_twice
VSIM 5> run
# time =                    0  |  shift_in = 10101010101010101010101010101010  | shift_out = 10101010101010101010101010101000
# time =                   10  |  shift_in = 11111111111111111111111111111111  | shift_out = 11111111111111111111111111111100
# time =                   20  |  shift_in = 00000000000000000000000000000000  | shift_out = 00000000000000000000000000000000
# time =                   30  |  shift_in = 00000000000000001111111111111111  | shift_out = 00000000000000011111111111111100
```

3.adder

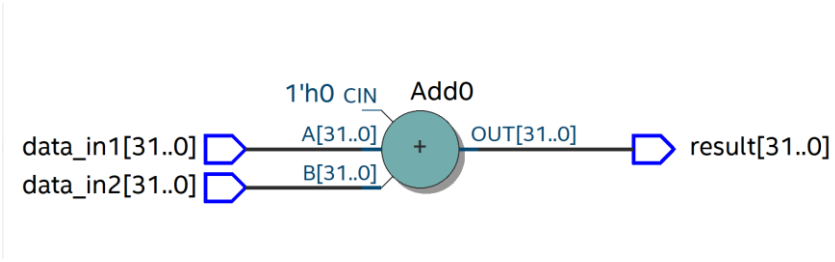
a. code

```
1  module adder( input  [31:0] data_in1, data_in2,
2                output [31:0] result);
3
4   assign result = data_in1 + data_in2;
5
6   endmodule
7
```

b. TB

```
8
9   module adder_tb;
10
11  reg  [31:0] data_in1, data_in2;
12  wire [31:0] result;
13
14  adder add1 (data_in1, data_in2, result);
15
16  initial
17  begin
18      $monitor("time = %t   |  data_in1 = %d  | data_in2 = %d   | result = %d ",$time, data_in1, data_in2, result);
19    // $monitor("time = %t   |  data_in1 = %b  | data_in2 = %b   | result = %b ",$time, data_in1, data_in2, result);
20
21      data_in1 = 10;
22      data_in2 = 5;
23      #10;
24
25      data_in1 = 16;
26      data_in2 = 4;
27      #10;
28
29      data_in1 = 30;
30      data_in2 = 10;
31      #10;
32
33      data_in1 = 32;
34      data_in2 = 8;
35  end
36
37  initial #150 $finish;
38
39  endmodule
```

c. RTL Viewer



d. Result

```
VSIM 4> run
# time =                    0   |  data_in1 =         10  | data_in2 =          5   | result =         15
# time =                   10   |  data_in1 =         16  | data_in2 =          4   | result =         20
# time =                   20   |  data_in1 =         30  | data_in2 =         10   | result =         40
# time =                   30   |  data_in1 =         32  | data_in2 =          8   | result =         40
```

---

4. Mux2_1(parameterized)

a. Code

```
module mux2_1 #(parameter width = 4)
(input [width : 0] in1, in2,
 input sel,
 output reg [width : 0] out_mux
                                );

always @(in1, in2, sel)
begin
case(sel)
     1'b0: out_mux = in1;

     1'b1: out_mux = in2;

endcase
end

endmodule
```

b. TB

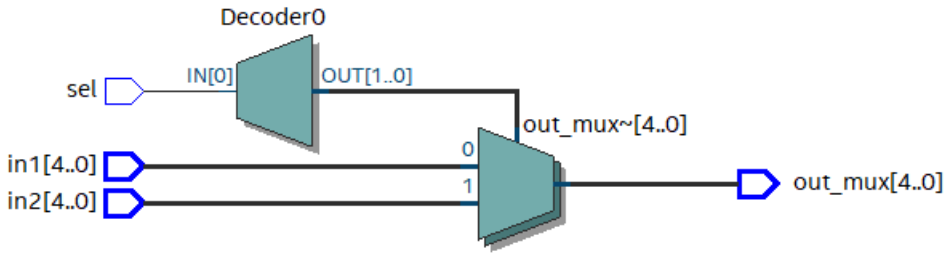
1. 5-bit in/out

```
module mux2_1 #(parameter width = 4)
(input [width : 0] in1, in2,
 input sel,
 output reg [width : 0] out_mux
                                );

always @(in1, in2, sel)
begin
case(sel)
     1'b0: out_mux = in1;

     1'b1: out_mux = in2;

endcase
end

endmodule
```

2. 32-bit in/out

```
//32-bit mux
module mux2_1_tb32;
reg  [31 : 0] in1, in2;
reg sel;
wire [31 : 0] out_mux;

mux2_1 #(31) muxtest ( in1, in2, sel, out_mux);

initial
begin
      $monitor("time = %t  |  in1 = %b  |  in2 = %b  |  sel = %b  |  out_mux =  %b",$time, in1, in2, sel, out_mux);
      in1 = {32{1'b1}};
      in2 = 32'b0;
      sel = 1'b0;
      #10
      in1 = {32{1'b1}};
      in2 = 32'b0;
      sel = 1'b1;

end
initial #150 $finish;

endmodule
```

c. RTL Viewer



d. Result

```
VSIM 3> run
# time =                  0  |  in1 = 11111  |  in2 = 00000  |  sel = 0  |  out_mux =  11111
# time =                 10  |  in1 = 11111  |  in2 = 00000  |  sel = 1  |  out_mux =  00000
VSIM 4> vsim work.mux2_1_tb32
# vsim work.mux2_1_tb32
# Loading work.mux2_1_tb32
# Loading work.mux2_1
VSIM 5> run
# time =                  0  |  in1 = 11111111111111111111111111111111  |  in2 = 00000000000000000000000000000000  |  sel = 0  |  out_mux =  11111111111111111111111111111111
# time =                 10  |  in1 = 11111111111111111111111111111111  |  in2 = 00000000000000000000000000000000  |  sel = 1  |  out_mux =  00000000000000000000000000000000
```

Maim modules

1.Alu

a. Code

```
module Alu(input [31:0] scr_A, scr_B,
           input [2:0] alu_ctr,
           output reg [31:0] result,
           output reg z_flag );

 always @(*)
begin
 result='b0;
 z_flag ='b1;
case(alu_ctr)
     //bitwise and
     3'b000: begin
     result = scr_A & scr_B;
     if(result == 0) begin
        z_flag = 1;
     end else begin
        z_flag = 0;
     end
     end

     //bitwise or
     3'b001: begin
     result = scr_A | scr_B;
     if(result == 0) begin
        z_flag = 1;
     end else begin
        z_flag = 0;
     end
     end
```

```
      //addition
      3'b010: begin
      result = scr_A + scr_B;
      if(result == 0) begin
         z_flag = 1;
      end else begin
         z_flag = 0;
      end
      end

      //subtraction
      3'b100: begin
      result = scr_A - scr_B;
      if(result == 0) begin
         z_flag = 1;
      end else begin
         z_flag = 0;
      end
      end

      //multiplication
      3'b101: begin
      result = scr_A * scr_B;
      if(result == 0) begin
         z_flag = 1;
      end else begin
         z_flag = 0;
      end
      end
```

```
        //set less than (if srca < srcb then aluresult = 1 )
        3'b110: begin
        if(scr_A < scr_B) begin
            result = 1;
        end else begin
            result = 0;
        end
        if(result == 0) begin
            z_flag = 1;
        end else begin
            z_flag = 0;
        end
        end

        default : begin
        result = 0;
        z_flag = 1'b1;
        end


    endcase
    end
    endmodule
```

b. Alu TB

```
module Alu_tb;

reg  [31:0] scr_A, scr_B;
reg  [2:0] alu_ctr;
wire [31:0] result;
wire        z_flag;

Alu Alu_test(scr_A, scr_B, alu_ctr, result, z_flag);
initial
begin
     $monitor("time = %t  | scr_A = %b  | scr_B = %b  | alu_ctr = %b  | result = %b  | z_flag = %b",$time, scr_A, scr_B, alu_ctr, result, z_flag);
    // test all cases for alu_ctr when (scr_a > scr-B)
     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b000;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b001;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b010;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b011;
     #10;
```

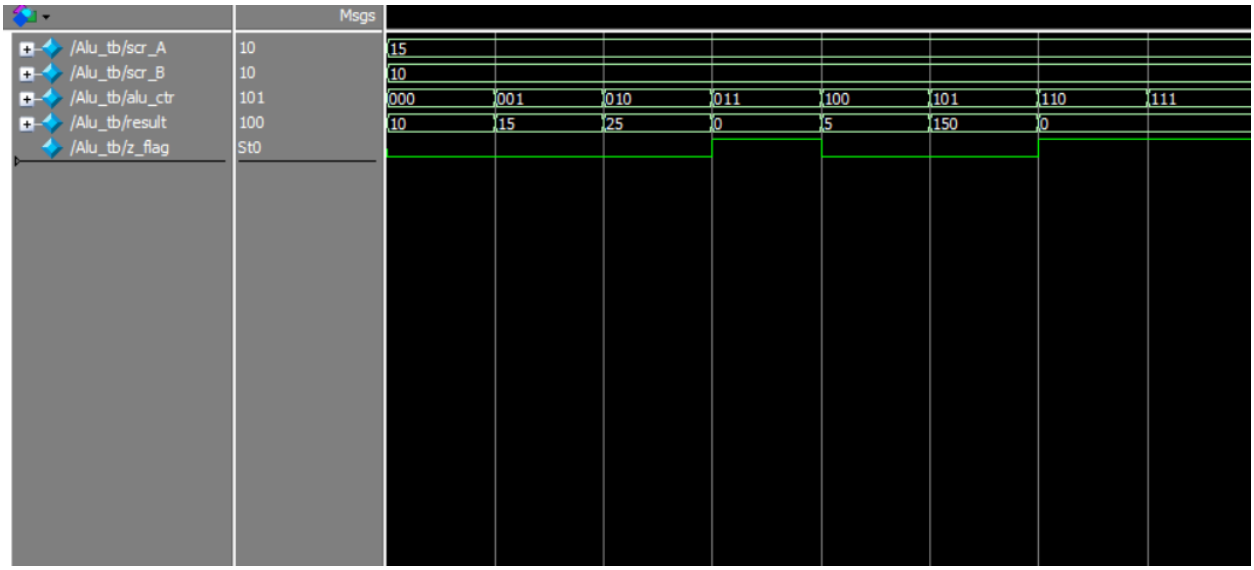
```
     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b100;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b101;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b110;
     #10;

     scr_A = 15;
     scr_B = 10;
     alu_ctr = 3'b111;
     #10;
```

```
//-------------------------------------------------------------------------//
     // test all cases for alu_ctr when (scr_a < scr-B)
     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b000;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b001;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b010;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b011;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b100;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b101;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b110;
     #10;

     scr_A = 15;
     scr_B = 20;
     alu_ctr = 3'b111;
     #10;
''                                                              ''
```

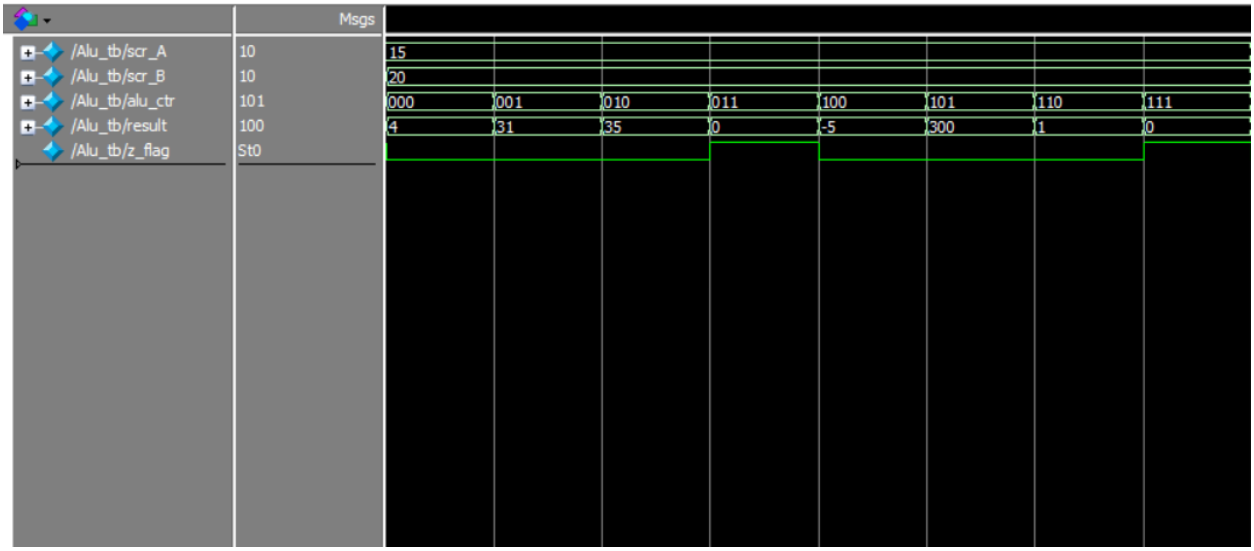
```
//-------------------------------------------------------------------------//
    // test all cases for alu_ctr when (scr_a = scr-B)
     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b000;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b001;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b010;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b011;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b100;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b101;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b110;
     #10;

     scr_A = 10;
     scr_B = 10;
     alu_ctr = 3'b111;
     #10;

end

initial #250 $finish;
endmodule
```

c. Result

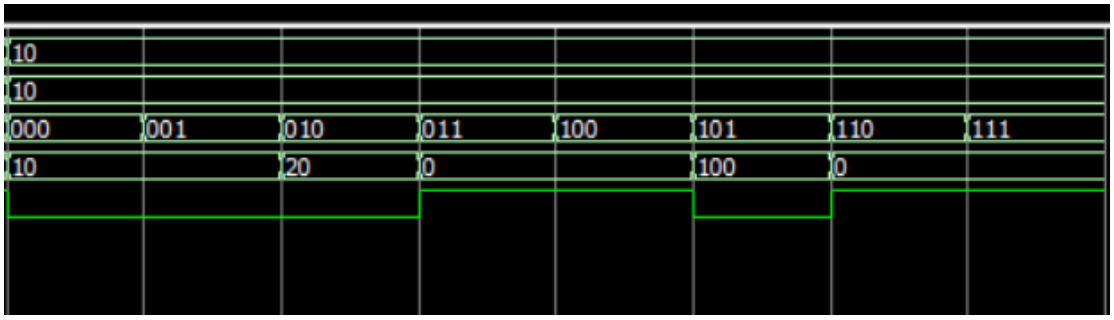
1. for first case of testing



2. 2nd case



3. 3^{ed} case of testing



2. Program Counter

a. Code

```
1
2   module pc(input [31:0] pc_in, // the output of the instraction memory in case of jump & branch
3             input clk, n_rst,
4             input jump, pcsrc,  //control signals to detect branch and jump case
5             output reg [31:0] pc_out);
6    wire [31:0] pcplus4  = pc_out + 4;
7    wire [31:0] sn_ex_temp;
8    sign_extend ex (pc_in[15:0], sn_ex_temp);
9    always@(posedge clk or negedge n_rst)
10  begin
11       if(!n_rst) begin
12          pc_out <= 32'b0;
13       end else begin
14          case({jump, pcsrc})
15
16             2'b00:   pc_out <= pcplus4  ;
17
18             2'b01:   pc_out <= pcplus4  + (sn_ex_temp << 2);
19
20             2'b10:   pc_out <= {pcplus4[31:28],pc_in[25:0],2'b00};
21
22             default: pc_out <= pcplus4  ;
23          endcase
24       end
25  end
26   endmodule
```

b. TB

1. case #1 (Reset)

```
38   module pc_tb;
39   reg  [31:0] pc_in; // instruction
40   reg  clk, n_rst;
41   reg  jump, pcsrc;
42   wire [31:0] pc_out;
43
44   pc pc_test(pc_in, clk, n_rst, jump, pcsrc, pc_out);
45
46   // Clock generation
47   always #5 clk = ~ clk;
48
49       initial begin
50       // Test case 1: n_rst is low, output is expected to be cleared (pc_out = 0)
51       clk = 0;
52       n_rst = 1'b0;                  // Assert reset
53       #10;
54
55       if (pc_out !== 32'b0)
56         $display("Test case 1 failed: pc_out= %b ", pc_out);
57       else
58         $display("Test case 1 passed");
59
```

2. case#2 (R-type) to increase PC by 4

```
61    // Test case 2:n_rst is high & R-type instruction
62      jump   = 0;                                          // No jump
63      pcsrc  = 0;                                          // No branch
64      pc_in  = 32'b00000010001100101010000000000100000;    // add $s0, $s1, $s2
65      n_rst  = 1;                                          // Release reset
66      #10;
67
68      if (pc_out !== 4)
69        $display("Test case 2 failed: pc_out = %b ", pc_out);
70      else
71        $display("Test case 2 passed");
```

3. case#3 (j-type) to make PC jump to address 1028(decimal)

```
73      // Test case 3:n_rst is high and there is jump instraction
74      jump   = 1;                                          // jump
75      pcsrc  = 0;                                          // No branch
76      pc_in  = 32'b00001000000000000000000100000001;       // j 1028
77      n_rst  = 1;                                          // Release reset
78
79      #10;
80
81      if (pc_out !== 1028)
82        $display("Test case 2 failed: pc_out = %b | pc_out<decimal> = %d", pc_out, pc_out);
83      else
84        $display("Test case 2 passed");
```

4. case#4 (beq) to make PC branch at a label at address 1048(decimal)

```
81      if (pc_out !== 1028)
82        $display("Test case 2 failed: pc_out = %b | pc_out<decimal> = %d", pc_out, pc_out);
83      else
84        $display("Test case 2 passed");
85
86      // Test case 4:n_rst is high and there is branch instraction
87
88      jump   = 0;                                          // No jump
89      pcsrc  = 1;                                          // branch
90      pc_in  = 32'b00010001000000000000000000000100;       // beq at a label at 1048
91      n_rst  = 1;                                          // Release reset
92
93      #10;
94
95      if (pc_out !== 1048)
96        $display("Test case 2 failed: pc_out = %b | pc_out<decimal> = %d", pc_out, pc_out);
97      else
98        $display("Test case 2 passed");
```

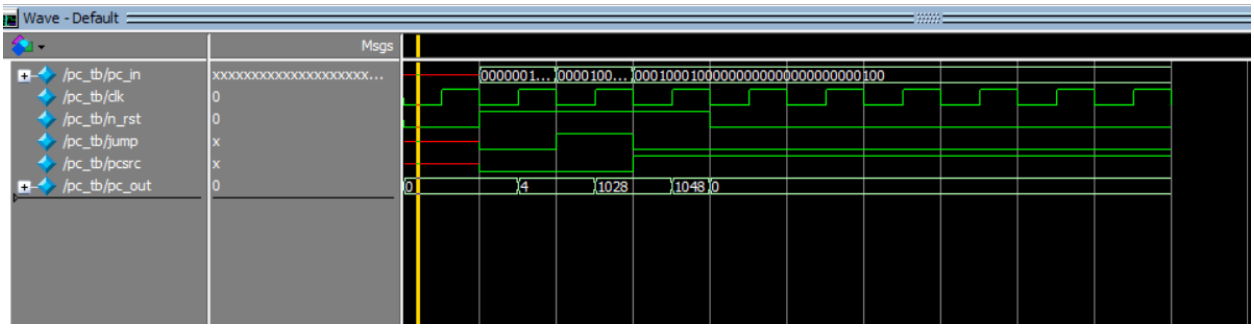
5. case#5 assert the reset to make PC go back to address 0

```
100     // Test case 5: n_rst is low, output is expected to be cleared (pc_out = 0)
101     n_rst = 1'b0;                  // Assert reset
102     #10;
103
104     if (pc_out !== 32'b0)
105       $display("Test case 1 failed: pc_out= %b ", pc_out);
106     else
107       $display("Test case 1 passed");
108
109
110   end
111 initial #1000 $finish;
112 endmodule
```

b. Results

```
# Test case 1 passed
# Test case 2 passed
# Test case 3 passed
# Test case 4 passed
# Test case 5 passed
```

Waves:



---

3. Instruction memory

a. code

```
1  module inst_mem(input  [31:0] address,
2                  output reg[31:0] instruction);
3
4   reg [31:0] inst_array [255:0];
5
6   initial $readmemh("programone.hex",inst_array);
7
8   always @(*)
9  begin
10      instruction = inst_array[address[9:2]];
11 end
12  endmodule
13
```

Programone.hex is a hexadecimal file containing the following machine code

1- Number Factorial: (hex format)

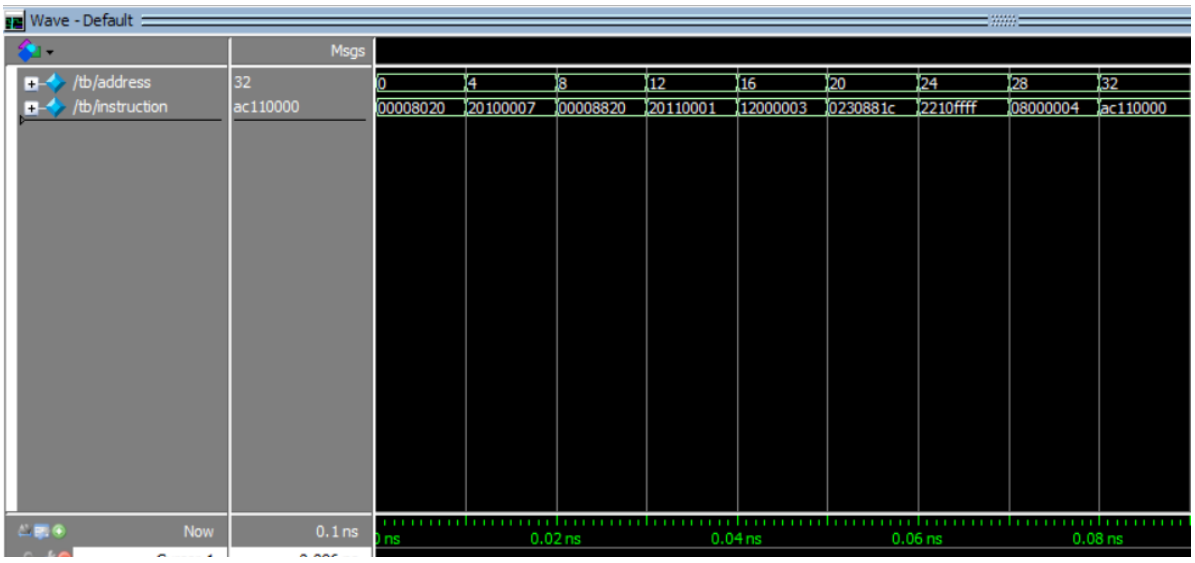
```
00008020
20100007      // change the least two significant decimals with the number
00008820
20110001
12000003
0230881C
2210FFFF
08000004
AC110000
```

b. TB

```
15
16  module tb;
17  reg  [31:0] address;
18  wire [31:0] instruction;
19
20  inst_mem test1(address, instruction);
21
22  initial
23        begin
24        $monitor("time = %t  |  address = %d   |  instruction = %h",$time, address, instruction);
25        address = 0;
26        #10
27        address = 4;
28        #10
29        address = 8;
30        #10
31        address = 12;
32        #10
33        address = 16;
34        #10
35        address = 20;
36        #10
37        address = 24;
38        #10
39        address = 28;
40        #10
41        address = 32;
42        end
43
44  initial #150 $finish;
45  endmodule
46
```

c. Results

```
# time =                    0  |  address =           0   |  instruction = 00008020
# time =                   10  |  address =           4   |  instruction = 20100007
# time =                   20  |  address =           8   |  instruction = 00008820
# time =                   30  |  address =          12   |  instruction = 20110001
# time =                   40  |  address =          16   |  instruction = 12000003
# time =                   50  |  address =          20   |  instruction = 0230881c
# time =                   60  |  address =          24   |  instruction = 2210ffff
# time =                   70  |  address =          28   |  instruction = 08000004
# time =                   80  |  address =          32   |  instruction = ac110000
```



4. Register file

a. Code

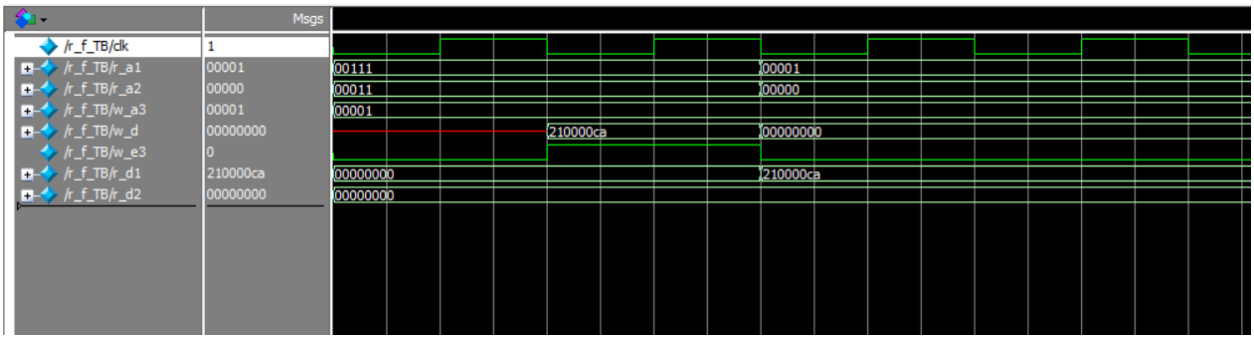
```
module Register_file(input  [4:0]  r_a1, r_a2,          // READ ADDRESSES
                     input  [4:0]  w_a3,                // WRITE ADDRESS
                     input  [31:0] w_d,                 // WRITE DATA
                     input  w_e3,  clk,                 // WRITE ENABLE & CLK
                     output [31:0] r_d1, r_d2);     // READ DATA

reg [31:0] registers [31:0];

integer i;
initial begin
for(i = 0; i < 32; i = i + 1) begin
    registers[i] = 32'b0;
    end
end


// asynchronus data reading
assign r_d1 = registers[r_a1];
assign r_d2 = registers[r_a2];

// synchronus data writing
always @(posedge clk)
begin
     if(w_e3) begin

        if(w_a3 != 0) registers[w_a3] <= w_d ;  // as register zero is always zero

     end else begin
         registers[w_a3] <= registers[w_a3];
     end
end

endmodule
```

b. TB

```
44  module r_f_TB;
45  reg [4:0]  r_a1, r_a2, w_a3;
46  reg [31:0] w_d;
47  reg        w_e3, clk;
48  wire[31:0] r_d1, r_d2;
49
50  Register_file m(r_a1, r_a2, w_a3, w_d, w_e3, clk, r_d1, r_d2);
51  always #10 clk = ~ clk;
52  initial begin
53  $monitor("time = %t  |  read_address_1 = %d  |  read_address_2 = %d  | write_address = %d  |  written_data = %h  | enable = %b  | read_
54   clk  = 0;
55   r_a1 = 7;
56   r_a2 = 3;
57   w_e3 = 0;
58   w_a3 = 1;
59   #20
60
61   w_e3 = 1;
62   w_a3 = 1;
63   w_d  = 32'b00100001000000000000000011001010;
64   #20
65
66   w_e3 = 0;
67   w_d  = 0;
68   r_a1 = 1;
69   r_a2 = 0;
70
71   end
72
73   initial #2000 $finish;
74  endmodule
75
76
```

c. Result (same as expected)

```
add wave -position insertpoint sim:/r_f_TB/*
VSIM 4> run
# time =                  0  |  read_address_1 =  7  |  read_address_2 =  3  | write_address =  1  |  written_data = xxxxxxxx  | enable = 0  | read_data_1 = 00000000  | read_data_2 = 00000000
# time =                 20  |  read_address_1 =  7  |  read_address_2 =  3  | write_address =  1  |  written_data = 210000ca  | enable = 1  | read_data_1 = 00000000  | read_data_2 = 00000000
# time =                 40  |  read_address_1 =  1  |  read_address_2 =  0  | write_address =  1  |  written_data = 00000000  | enable = 0  | read_data_1 = 210000ca  | read_data_2 = 00000000
```



5. Data Memory

a. Code

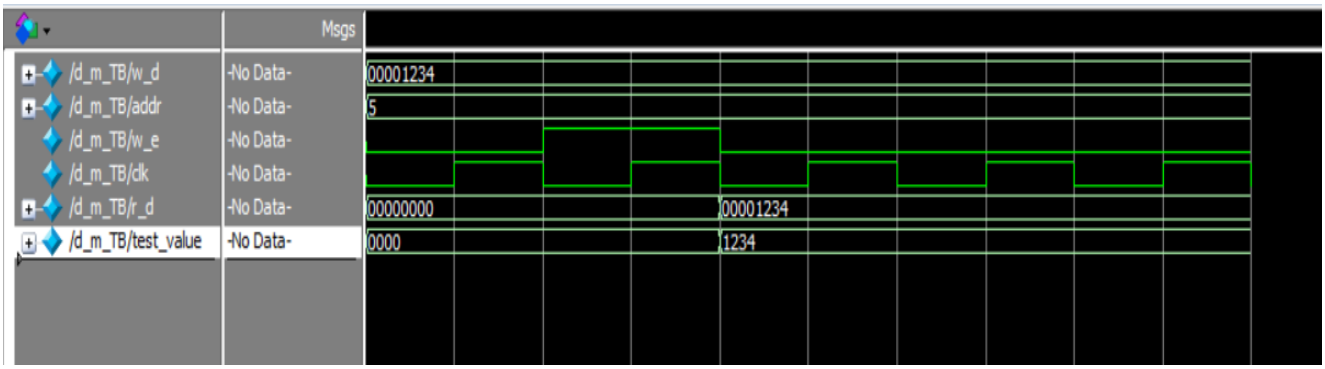
```
1
2   module data_mem(input [31:0] addr, w_d,
3                   input clk, w_e,
4                   output reg[31:0] r_d,
5                   output    [15:0] test_value);
6
7    reg [31:0] data_array [255:0];
8
9    integer i;
10  initial begin
11   r_d = 32'b0;
12  for(i =  0; i < 256; i = i + 1) begin
13      data_array[i] = 32'b0;
14      end
15   end
16
17   always @ (posedge clk)
18  begin
19       if(w_e) begin
20       data_array[addr[9:2]] <= w_d;
21       end else begin
22       data_array[addr[9:2]] <= data_array[addr[9:2]];
23       end
24   end
25
26   always @(*)
27  begin
28      if(~w_e) begin
29       r_d <= data_array[addr[9:2]];
30       end else begin
31       r_d <= r_d;
32       end
33   end
34
35   assign test_value = r_d[15:0];
36
37   endmodule
```

b. TB

```
37
38   module d_m_TB;
39   reg [31:0] w_d, addr;
40   reg        w_e, clk;
41   wire[31:0] r_d;
42   wire [15:0] test_value;
43
44   data_mem d (addr, w_d, clk, w_e, r_d, test_value);
45   always #10 clk = ~ clk;
46
47  initial begin
48   $monitor("time=%t  |  address = %d  | enable = %b  |  written_data = %h| read_data = %
49    clk  = 0;
50    addr = 5;
51    w_e  = 0;
52    w_d  = 32'b00000000000000000000001001000110100;
53    #20
54
55    w_e = 1;
56    #20
57
58    w_e = 0;
59
60    end
61
62   initial #2000 $finish;
63   endmodule
```

c. Result (same as expected)

```
VSIM 3> run
# time=                0  |  address =        5  | enable = 0  |  written_data = 00001234| read_data = 00000000 | test_value = 0000
# time=               20  |  address =        5  | enable = 1  |  written_data = 00001234| read_data = 00000000 | test_value = 0000
# time=               40  |  address =        5  | enable = 0  |  written_data = 00001234| read_data = 00001234 | test_value = 1234
```



---

6. Control unit (which is divided into main decoder and Alu decoder)

a. Top module code

```
1  module ctrl_unit (input  [5:0] op_code, funct,
2                    output   jump, memwrite, regwrite, redest, alusrc, memtoreg, branch,
3                    output   [2:0] alu_ctrl);
4
5  wire [1:0] alu_op;
6  main_decoder d1(op_code, jump, memwrite, regwrite, redest, alusrc, memtoreg, branch,alu_op);
7  alu_decoder  d2(alu_op, funct, alu_ctrl);
8  endmodule
9
```

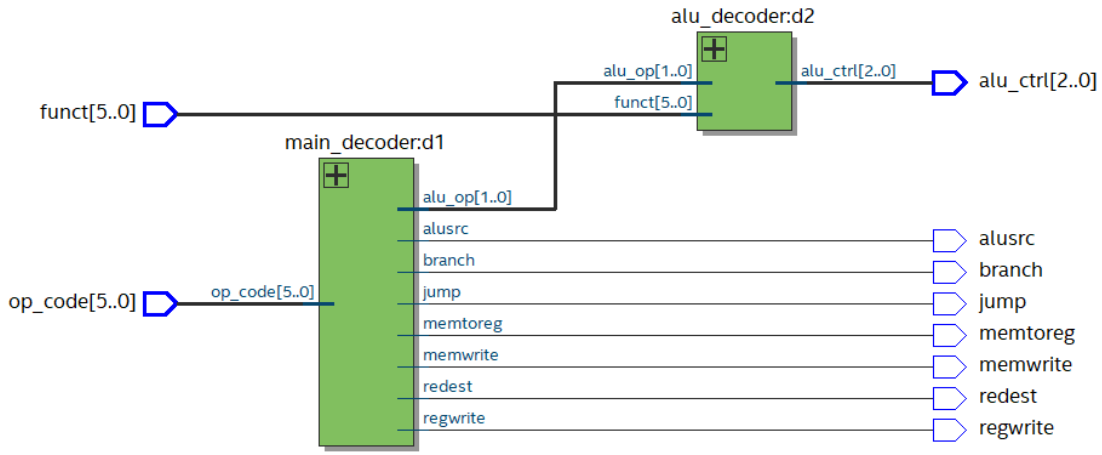
b. main decoder code

```
module main_decoder (input  [5:0] op_code,
                     output  reg jump, memwrite, regwrite, redest, alusrc, memtoreg, branch,
                     output reg [1:0] alu_op);
always @(*) begin
        case(op_code)

            6'b100011: begin
                       jump     = 0;
                       alu_op   = 0;
                       memwrite = 0;
                       regwrite = 1'b1;
                       redest   = 0;
                       alusrc   = 1'b1;
                       memtoreg = 1'b1;
                       branch   = 0;
            end

            6'b101011: begin
                       jump     = 0;
                       alu_op   = 0;
                       memwrite = 1'b1;
                       regwrite = 0;
                       redest   = 0;
                       alusrc   = 1'b1;
                       memtoreg = 1'b1;
                       branch   = 0;
            end

            6'b000000: begin
                       jump     = 0;
                       alu_op   = 2'b10;
                       memwrite = 0;
                       regwrite = 1'b1;
                       redest   = 1'b1;
                       alusrc   = 0;
                       memtoreg = 0;
                       branch   = 0;
            end

             6'b001000: begin
                        jump     = 0;
                        alu_op   = 0;
                        memwrite = 0;
                        regwrite = 1'b1;
                        redest   = 0;
                        alusrc   = 1'b1;
                        memtoreg = 0;
                        branch   = 0;
             end

             6'b000100: begin
                        jump     = 0;
                        alu_op   = 2'b01;
                        memwrite = 0;
                        regwrite = 0;
                        redest   = 0;
                        alusrc   = 0;
                        memtoreg = 0;
                        branch   = 1'b1;
             end

             6'b000010: begin
                        jump     = 1'b1;
                        alu_op   = 0;
                        memwrite = 0;
                        regwrite = 0;
                        redest   = 0;
                        alusrc   = 0;
                        memtoreg = 0;
                        branch   = 0;
             end


            default: begin
                     jump     = 0;
                     alu_op   = 0;
                     memwrite = 0;
                     regwrite = 0;
                     redest   = 0;
                     alusrc   = 0;
                     memtoreg = 0;
                     branch   = 0;
            end
        endcase
end
 endmodule
```

c. Alu Decoder

```
96
97  module alu_decoder(input  [1:0] alu_op,
98                     input  [5:0] funct,
99                     output reg [2:0]alu_ctrl);
100
101 always @(*)
102 begin
103     case(alu_op)
104
105          2'b00: alu_ctrl = 3'b010;
106
107          2'b01: alu_ctrl = 3'b100;
108
109          2'b10: begin
110                 case(funct)
111
112                      6'b100000: alu_ctrl = 3'b010;
113
114                      6'b100010: alu_ctrl = 3'b100;
115
116                      6'b101010: alu_ctrl = 3'b110;
117
118                      6'b011100: alu_ctrl = 3'b101;
119
120                      default  : alu_ctrl = 3'b010;
121                 endcase
122           end
123
124          default: alu_ctrl = 3'b010;
125     endcase
126 end
127 endmodule
```

d. RTL viewer

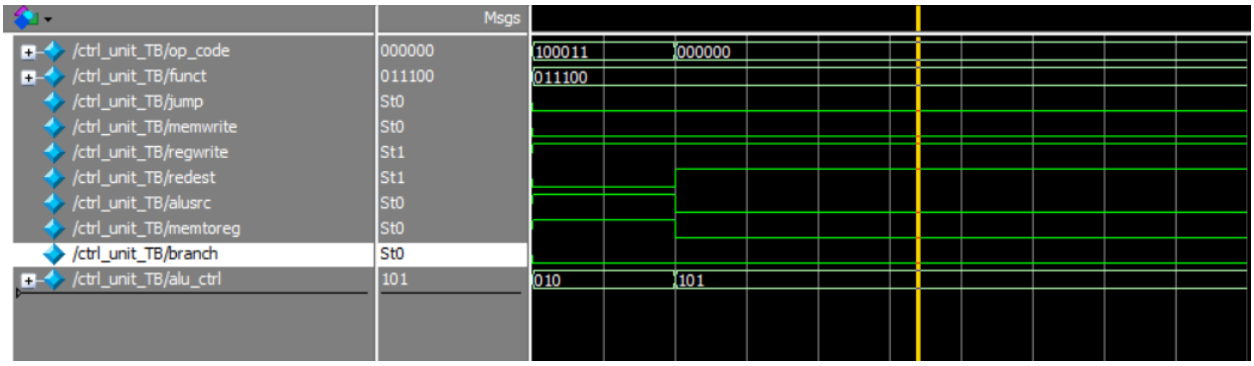


e. TB

```
module ctrl_unit_TB;

reg  [5:0] op_code, funct;
wire jump, memwrite, regwrite, redest, alusrc, memtoreg, branch;
wire [2:0] alu_ctrl;

ctrl_unit b (op_code, funct, jump, memwrite, regwrite, redest, alusrc, memtoreg,

initial begin
$monitor("time=%t| op_code = %b | funct = %b | jump = %b| memwrite = %b| regwrite

 op_code = 6'b100011;
 funct   = 6'b011100;
 #20

 op_code = 6'b000000;
 funct   = 6'b011100;

end

initial #2000 $finish;

endmodule
```

f. Results (same as expected)

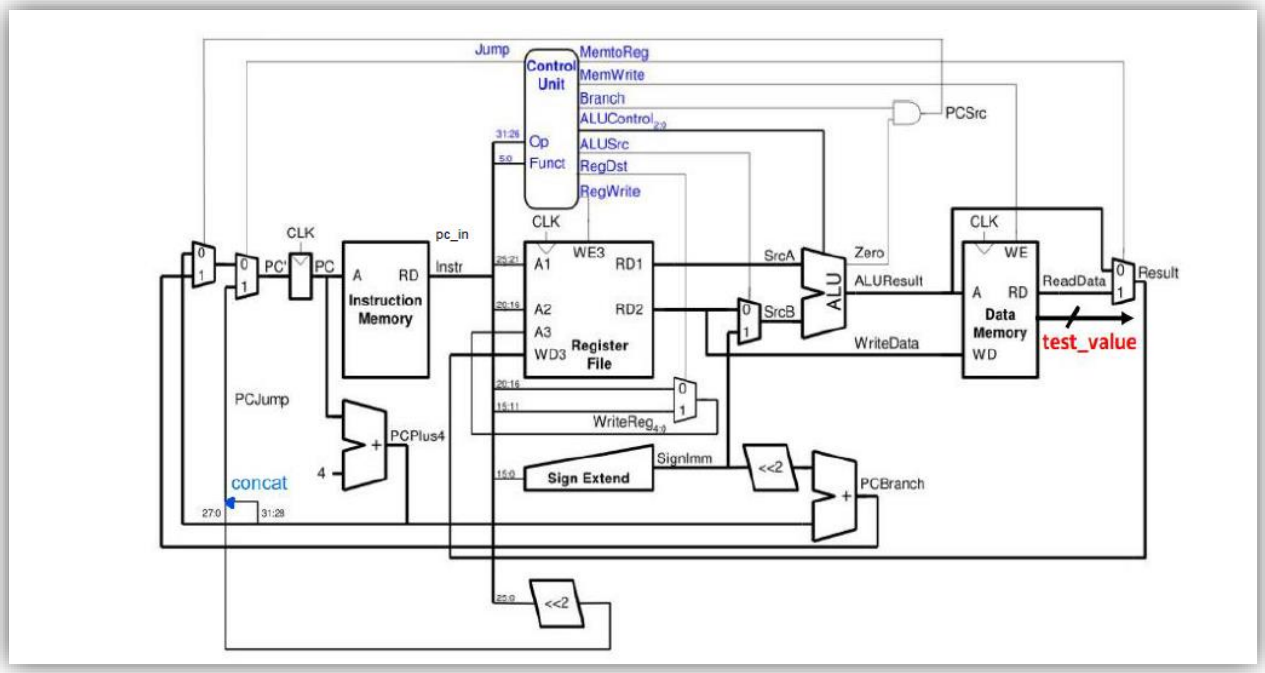
```
# -- Compiling module alu_decoder
# -- Compiling module ctrl_unit_TB
#
# Top level modules:
#       ctrl_unit_TB
#
ModelSim> vsim work.ctrl_unit_TB
# vsim work.ctrl_unit_TB
# Loading work.ctrl_unit_TB
# Loading work.ctrl_unit
# Loading work.main_decoder
# Loading work.alu_decoder
add wave -position insertpoint sim:/ctrl_unit_TB/*
/SIM 4> run
# time=                   0| op_code = 100011 | funct = 011100 | jump = 0| memwrite = 0| regwrite = 1 | redest= 0| alusrc= 1| memtoreg= 1| branch = 0| alu_ctrl = 010
# time=                  20| op_code = 000000 | funct = 011100 | jump = 0| memwrite = 0| regwrite = 1 | redest= 1| alusrc= 0| memtoreg= 0| branch = 0| alu_ctrl = 101

/SIM 5>
```



Top module for single cycle MIPS processor

Using all of the previous blocks (sub-modules)

- According to the following connection between the pre-designed modules, the top-level module is designed as explained in the following code.



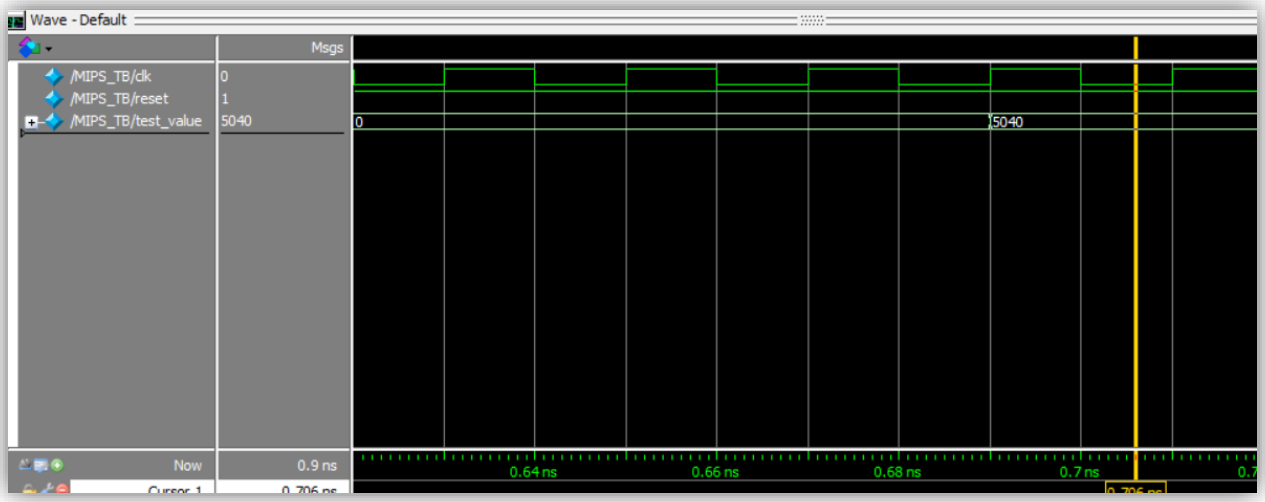
a. Code

```
1  module MIPS(input  clk, reset,
2              output [15:0] test_value);
3
4   wire        jump, memwrite, regwrite, redest, alusrc, memtoreg, branch, pcsrc, z_flag;
5   wire [4:0]  writereg;
6   wire [2:0]  alu_ctrl;
7   wire [31:0] alu_result, pr_c, inst, readdata, signimm, scr_B, r_d1, r_d2, result;
8
9   ctrl_unit      m1  (inst[31:26], inst[5:0], jump, memwrite, regwrite, redest, alusrc, memtoreg
10  mux2_1 #(4)    m2  (inst[20:16], inst[15:11], redest, writereg);
11  Register_file  m3  (inst[25:21], inst[20:16], writereg, result, regwrite,  clk, r_d1, r_d2);
12  sign_extend    m4  (inst[15:0], signimm);
13  mux2_1 #(31)   m5  (r_d2, signimm, alusrc, scr_B);
14  Alu            m6  (r_d1, scr_B, alu_ctrl, alu_result, z_flag );
15  data_mem       m7  (alu_result, r_d2, clk, memwrite, readdata, test_value);
16  mux2_1 #(31)   m8  (alu_result, readdata, memtoreg, result);
17  assign pcsrc = z_flag & branch;
18  pc             m9 (inst, clk, reset, jump, pcsrc, pr_c);
19  inst_mem       m10 (pr_c, inst);
20
21  endmodule
22
```

b. TB

```
23
24  module MIPS_TB;
25  reg  clk, reset;
26  wire [15:0]test_value;
27
28  MIPS mm (clk, reset, test_value);
29  always #10 clk = ~ clk;
30
31  initial begin
32
33  $monitor("time = %t  |  reset = %b |  test_value = %h ",$time, reset, test_value);
34
35  clk = 0;
36  reset = 1'b0;
37  #20;
38
39  reset = 1'b1;
40
41  end
42  endmodule
```

c. Results (7 factorial)



d. Results (CGD)