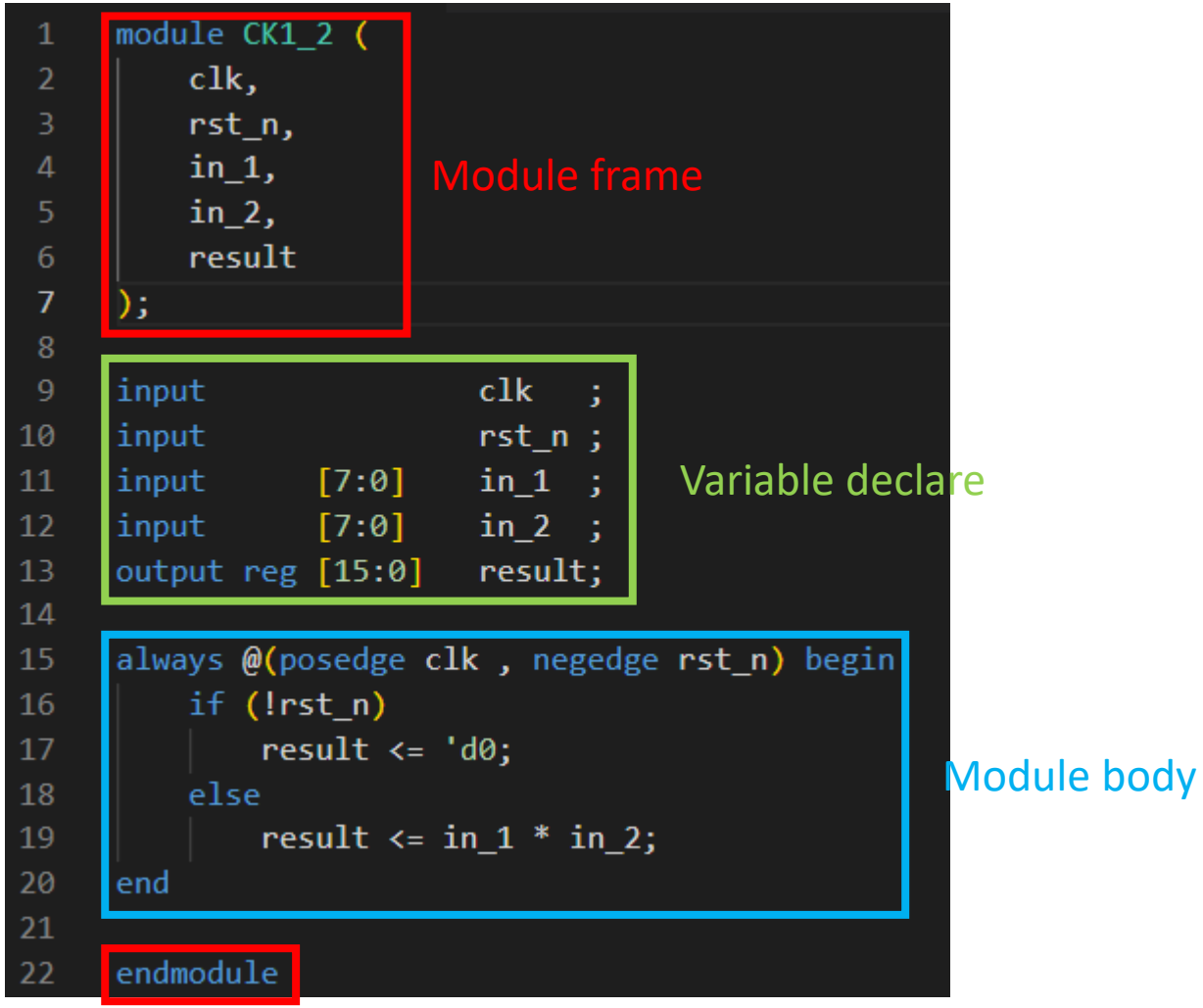
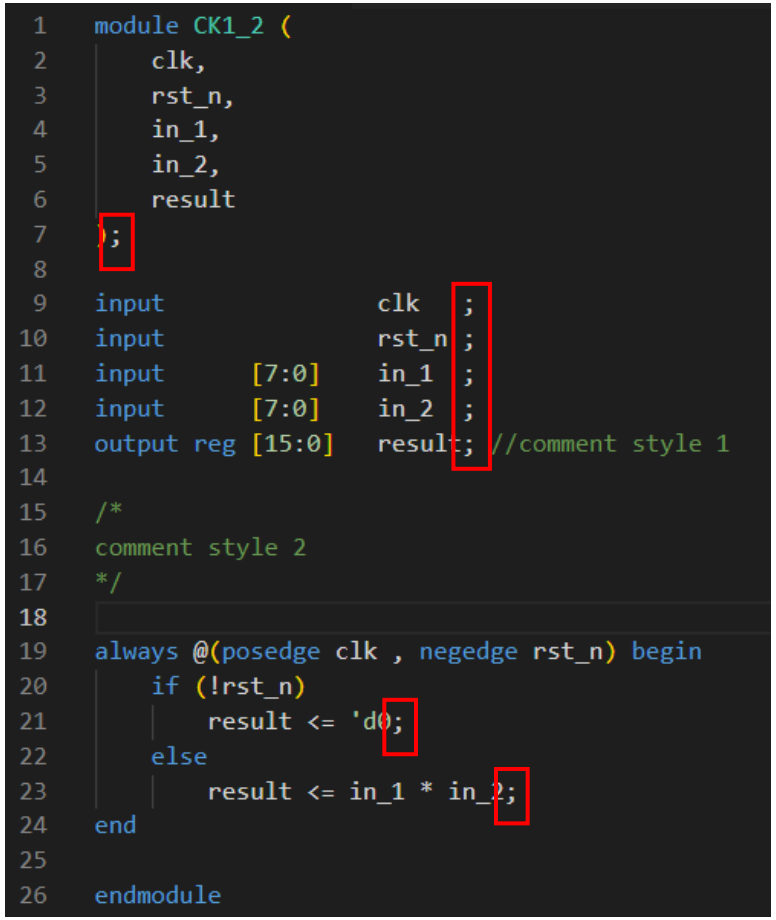
Basic Verilog RTL

Module


module CK1_2 (
clk,
rst_n,
in_1,
in_2,
result
);
Module frame
input clk ;
input rst_n ;
input [7:0] in_1 ;
input [7:0] in_2 ;
output reg [15:0] result;
Variable declare
always @(posedge clk , negedge rst_n) begin
if (!rst_n)
result <= 'd0;
else
result <= in_1 * in_2;
end
Module body
endmodule

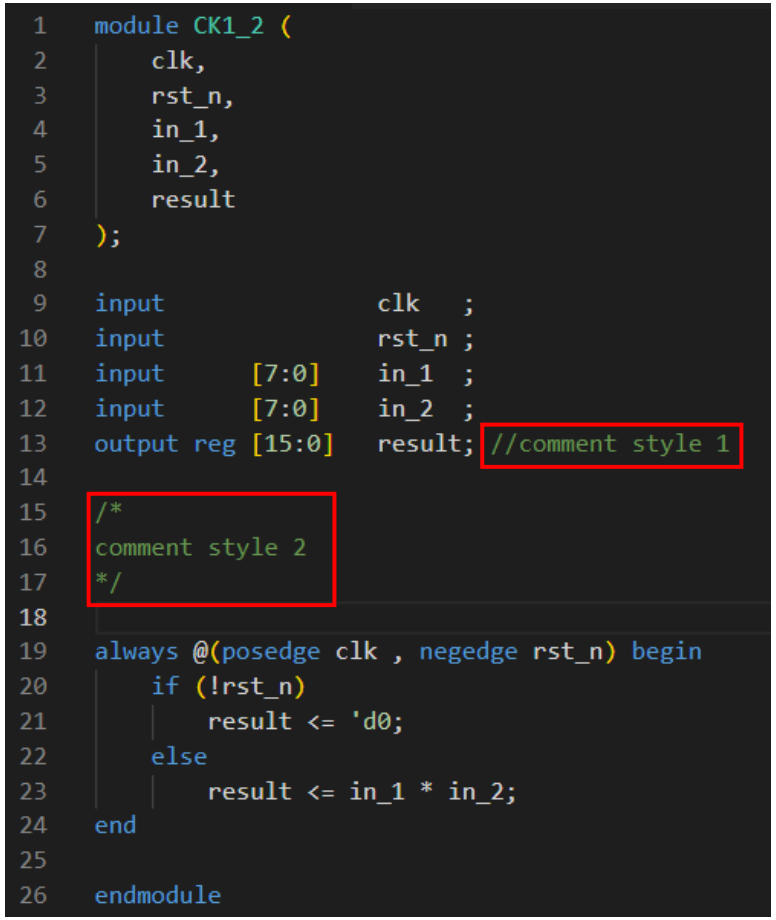
Basic Syntax



```
1   module CK1_2 (
2       clk,
3       rst_n,
4       in_1,
5       in_2,
6       result
7   );
8
9   input               clk   ;
10  input               rst_n ;
11  input       [7:0]   in_1  ;
12  input       [7:0]   in_2  ;
13  output reg [15:0]   result; //comment style 1
14
15  /*
16  comment style 2
17  */
18
19  always @(posedge clk , negedge rst_n) begin
20      if (!rst_n)
21          result <= 'd0;
22      else
23          result <= in_1 * in_2;
24  end
25
26  endmodule
```

- End statements with ;

Basic Syntax



```
1   module CK1_2 (
2       clk,
3       rst_n,
4       in_1,
5       in_2,
6       result
7   );
8
9   input               clk   ;
10  input               rst_n ;
11  input      [7:0]    in_1  ;
12  input      [7:0]    in_2  ;
13  output reg [15:0]   result; //comment style 1
14
15  /*
16  comment style 2
17  */
18
19  always @(posedge clk , negedge rst_n) begin
20      if (!rst_n)
21          result <= 'd0;
22      else
23          result <= in_1 * in_2;
24  end
25
26  endmodule
```

- Two style of comments
  - Single line comment after “//”.
  - Block comment from “/*” to “*/”.

Basic Syntax

```
1   module CK1_2 (
2       clk,
3       rst_n,
4       in_1,
5       in_2,
6       result
7   );
8
9   input               clk   ;
10  input               rst_n ;
11  input       [7:0]   in_1  ;
12  input       [7:0]   in_2  ;
13  output reg [15:0]   result; //comment style 1
14
15  /*
16  comment style 2
17  */
18
19  always @(posedge clk , negedge rst_n) begin
20      if (!rst_n)
21          result <= 'd0;
22      else
23          result <= in_1 * in_2;
24  end
25
26  endmodule
```

- Use begin … end to compound statements.
- {} is used to combine signals in verilog.

Declare variable

```
reg        bit;
reg [15:0] data;                    //    1 data          with          bit-width 16
reg [15:0] array  [15:0];           //   16 data    array with elements' bit-width 16
reg [15:0] matrix [15:0][15:0];     //16*16 data 2d array with elements' bit-width 16

data[7]         //return 7th bit of data.
data[15:8]      //return 8bit signal from data[15] to data[8].
array[3]        //return 3rd element of array.
array[3][5]     //return 5th bit of 3rd element of array.
matrix[7][7]    //return matrix with index [7][7].
```

- <type> [MSB:LSB] <variable>

  Type : reg or wire for RTL.

  [MSB:LSB]: determine bit width.

Data type

```
1   module CK1_3 (
2       clk,
3       rst_n,
4       in_1,
5       in_2,
6       result
7   );
8
9   input               clk;
10  input               rst_n;
11  input       [7:0]   in_1;
12  input       [7:0]   in_2;
13  output reg  [15:0]  result;
14
15  wire        [15:0]  mul_tmp;
16
17  assign mul_tmp = in_1 * in_2;
18
19  always @(posedge clk , negedge rst_n) begin
20      if (!rst_n)
21          result <= 'd0;
22      else
23          result <= mul_tmp;
24  end
25
26  endmodule
```

- wire
  - Continuously driven signal.
  - Use “assign” to assign a value or operations.
  - Only for combinational signals.
  - assign only be used outside always blocks.
  - Default type of output ports is wire.

Data type

```
1   module CK1_3 (
2       clk,
3       rst_n,
4       in_1,
5       in_2,
6       result
7   );
8
9   input               clk;
10  input               rst_n;
11  input       [7:0]   in_1;
12  input       [7:0]   in_2;
13  output reg  [15:0]  result;
14
15  wire        [15:0]  mul_tmp;
16
17  assign mul_tmp = in_1 * in_2;
18
19  always @(posedge clk , negedge rst_n) begin
20      if (!rst_n)
21          result <= 'd0;
22      else
23          result <= mul_tmp;
24  end
25
26  endmodule
```

- reg
  - Default is X (unknown).
  - Assigned in always block.
  - For combinational or sequential signals.
  - Can not be assigned by 2 or more always blocks for RTL design.

Data type	wire	reg
assignment	assign	Always block
Combinational	O	O
Sequential	X	O

Always Block

```
always @(posedge clk , negedge rst_n) begin
    if (!rst_n)
        result <= 'd0;
    else
        result <= mul_tmp;
end
```

```
reg [15:0] result;

always @(in_1, in_2) begin
    result <= in_1 * in_2;
end
```

- Event driven block.
  - Start procedure if event triggered.
- Can be used for combinational or sequential blocks.

Blocking and non-blocking assignment

Blocking assignment (=)

- Execute statements line by line.

```
reg [7:0] A, B;

always @(posedge clk) begin
    A = B;
    B = A;
end
```

- As B = A = B, A and B will stuck at same value.

Non-blocking assignment (<=)

- Execute statements at same time.

```
reg [7:0] A, B;

always @(posedge clk) begin
    A <= B;
    B <= A;
end
```

- Swap A and B every cycle.

Highly recommended for sequential circuits.

Value Representation

```
wire        [15:0] a, b, c;
wire signed [15:0] d;

assign a = 16'b0000000000000000;    // assgin 16bit 0000000000000000 in binary  to a
assign b = 16'h1234;                // assgin 16bit             1234 in heximal to b
assign c = 'b0;                     // 'b0 will automatically extend 16'b0 (bit width of c)
assign d = -'d15;                   // assgin -15 (2's complement)   in decimal to d
```

- Value format:

  Constant: <size>’<base><value>

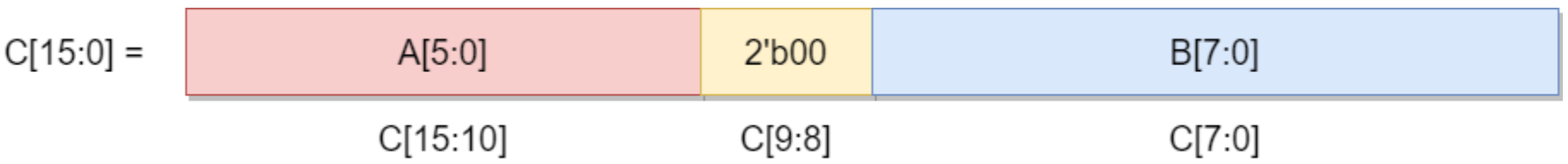
  <size> : bit width.

  <base> : b, o, d, h.

  <value> : value in <base> representation

Bus Concatenation

```
wire [7:0]  A;
wire [7:0]  B;
wire [15:0] C;

assign C = {A[5:0], 2'b0, B};
```



Operations

```
assign C = A + B;   // add
assign C = A - B;   // sub
assign C = A * B;   // mul
assign C = A / B;   // div
assign C = A % B;   // mod
assign C = A > B;   // greater than
assign C = A < B;   // less than
assign C = A & B;   // bit-wise and
assign C = A | B;   // bit-wise or
assign C = A ^ B;   // bit-wise XOR
assign C = A && B;  // logical and
assign C = A || B;  // logical or
assign C = A << B;  // shift left
assign C = A >> B;  // shift right
```

Signed Issue

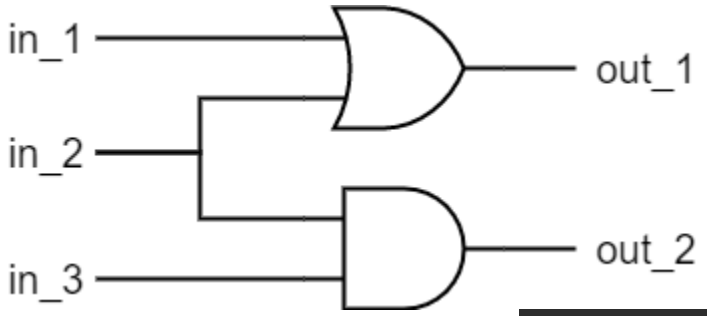
```
wire [7:0] A;
wire [3:0] B;
wire [7:0] C;

assign C = A + B;
```

- B will be zero extended to 8 bits.

```
wire signed [7:0] A;
wire signed [3:0] B;
wire signed [7:0] C;

assign C = A + B;
```

- B will be sign extended to 8 bits.
- Both operands must be signed.

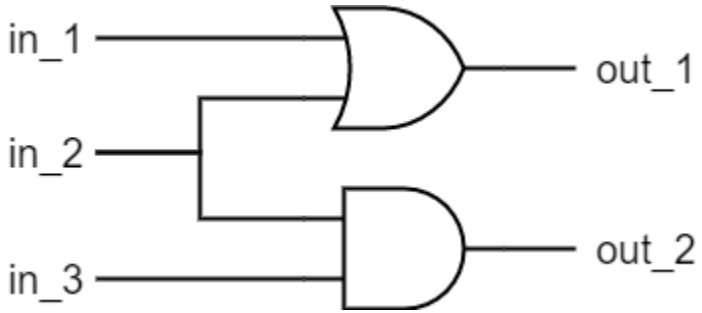
Combinational Circuit




```
1   module CK2_1 (
2       in_1,
3       in_2,
4       in_3,
5       out_1,
6       out_2
7   );
8
9   input  in_1, in_2, in_3;
10  output out_1, out_2;
11
12  assign out_1 = in_1 | in_2;
13  assign out_2 = in_2 & in_3;
14
15  endmodule
```

```
1   module CK2_1 (
2       in_1,
3       in_2,
4       in_3,
5       out_1,
6       out_2
7   );
8
9   input       in_1, in_2, in_3;
10  output reg  out_1, out_2;
11
12  always @(in_1, in_2, in_3) begin
13      out_1 = in_1 | in_2;
14      out_2 = in_2 & in_3;
15  end
16
17  endmodule
```

Combinational Circuit

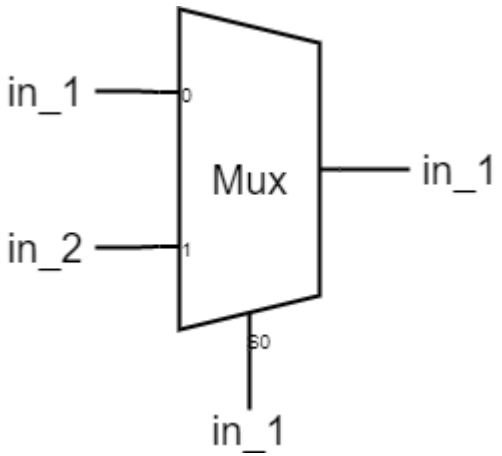



```
1   module CK2_1 (
2       in_1,
3       in_2,
4       in_3,
5       out_1,
6       out_2
7   );
8
9   input       in_1, in_2, in_3;
10  output reg  out_1, out_2;
11
12  always @(in_1, in_2, in_3) begin
13      out_1 = in_1 | in_2;
14      out_2 = in_2 & in_3;
15  end
16
17  endmodule
```

recommended

```
1   module CK2_1 (
2       in_1,
3       in_2,
4       in_3,
5       out_1,
6       out_2
7   );
8
9   input       in_1, in_2, in_3;
10  output reg  out_1, out_2;
11
12  always @(*) begin
13      out_1 = in_1 | in_2;
14      out_2 = in_2 & in_3;
15  end
16
17  endmodule
```

If-else




```
module CK2_2 (
    in_1,
    in_2,
    sel,
    out,
);

input       in_1, in_2, sel;
output reg  out;

always @(*) begin
    if (sel == 'b0)
        out = in_1;
    else
        out = in_2;
end

endmodule
```

```
1   module CK2_3 (
2       in_1,
3       in_2,
4       sel,
5       out,
6   );
7
8   input       in_1, in_2, sel;
9   output      out;
10
11  assign out = (sel == 'b0)? in_1 : in_2;
12
13  endmodule
```

in-line style

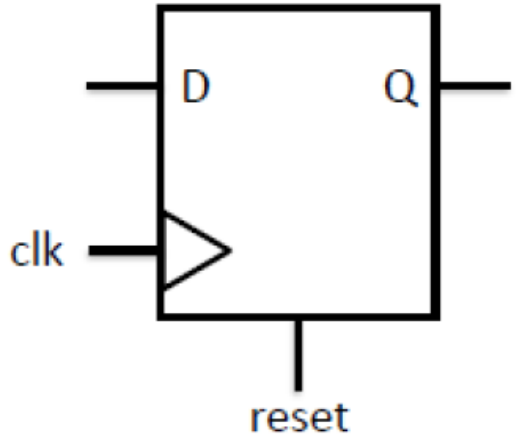
Case

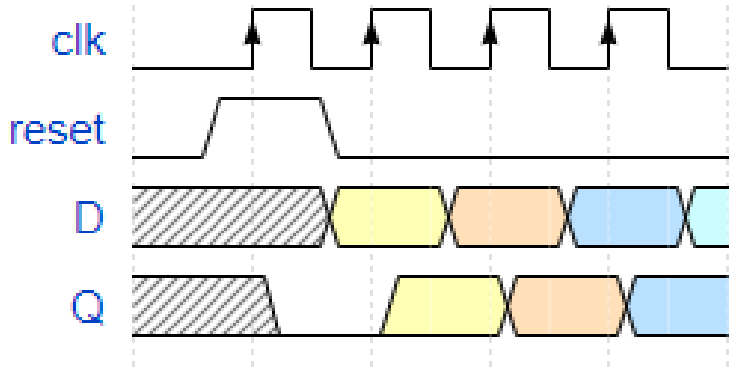
```
1   module CK2_4 (
2       in_1,
3       in_2,
4       in_3,
5       in_4,
6       sel,
7       out,
8   );
9
10  input           in_1, in_2, in_3, in_4;
11  input   [2:0]   sel;
12  output  reg     out;
13
14  always @(*) begin
15      case (sel)
16          'd0 :   out = in_1;
17          'd1 :   out = in_2;
18          'd2 :   out = in_3;
19          'd3 :   out = in_4;
20          default: out = 'b0;
21      endcase
22  end
23
24  endmodule
```

ALU with Case

```
1   module CK2_5 (
2       src1,
3       src2,
4       op,
5       result,
6   );
7
8   input       [7:0]   src1, src2;
9   input       [2:0]   op;
10  output  reg [7:0]   result;
11
12  always @(*) begin
13      case (op)
14          'd0     : result = src1 + src2;
15          'd1     : result = src1 - src2;
16          'd2     : result = src1 & src2;
17          'd3     : result = src1 | src2;
18          default : result = 'b0;
19      endcase
20  end
21
22  endmodule
```

Sequential circuit





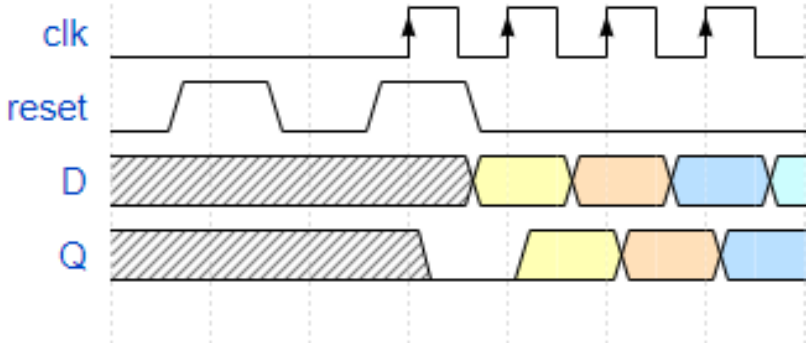

```
reg  Q;

always @(posedge clk) begin
    if (reset)
        Q <= 'b0;
    else
        Q <= D;
end
```

- Active high and Synchronous reset

Synchronous and Asynchronous Reset

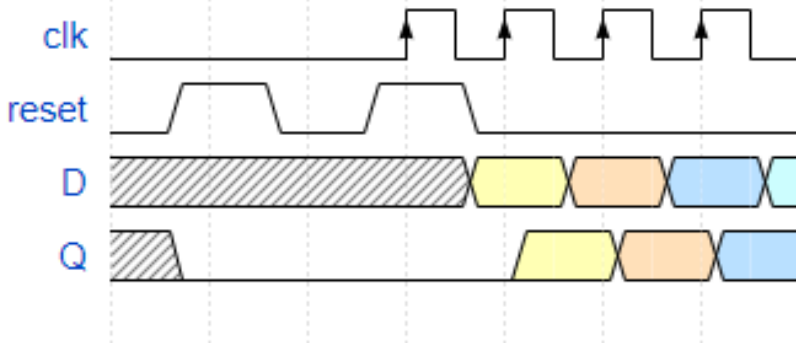
Synchronous & active high

```
reg  Q;

always @(posedge clk) begin
    if (reset)
        Q <= 'b0;
    else
        Q <= D;
end
```

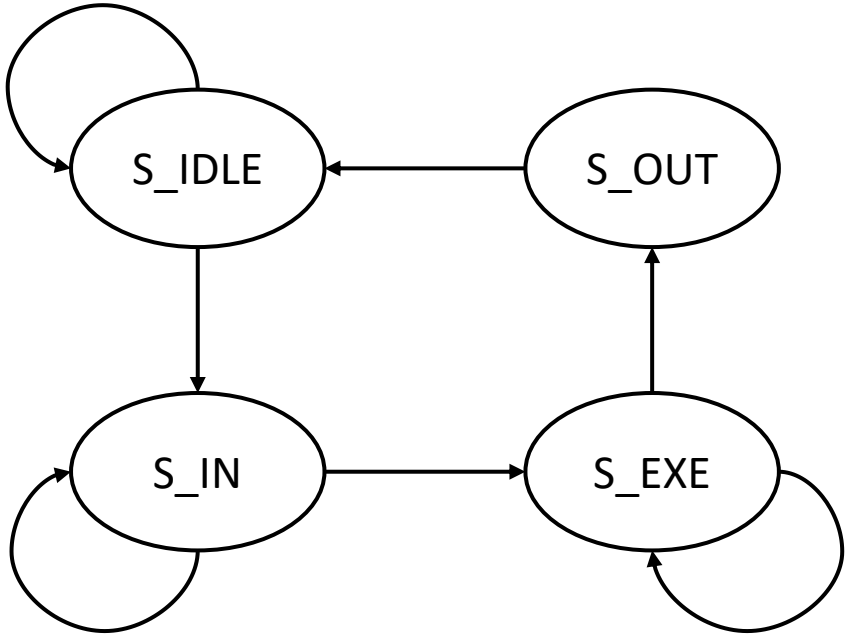


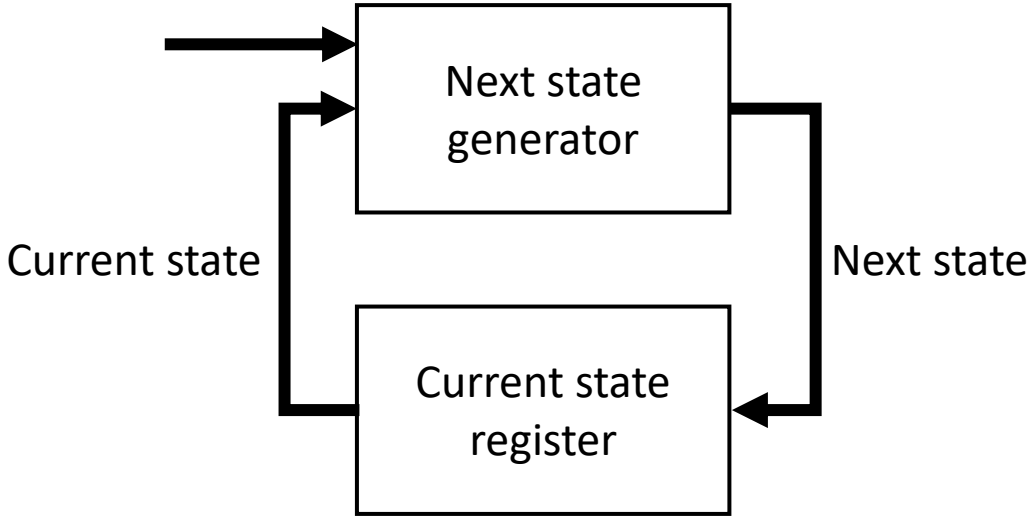
Asynchronous & active high

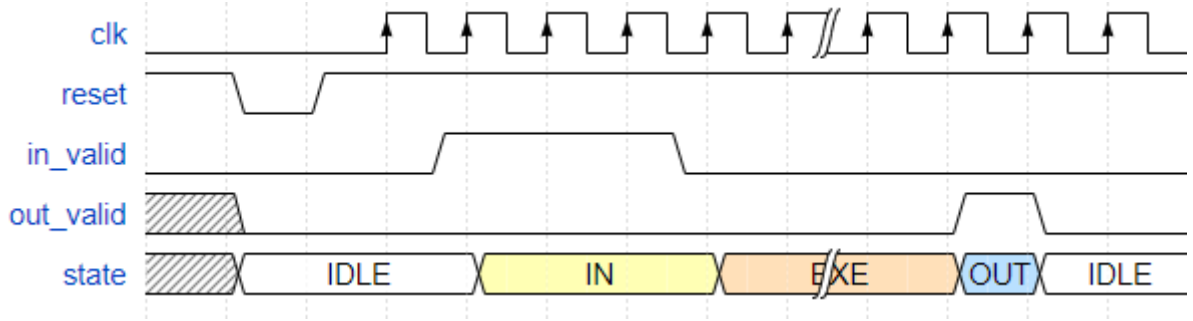
```
reg Q;

always @(posedge clk, posedge reset) begin
    if (reset)
        Q <= 'b0;
    else
        Q <= D;
end
```



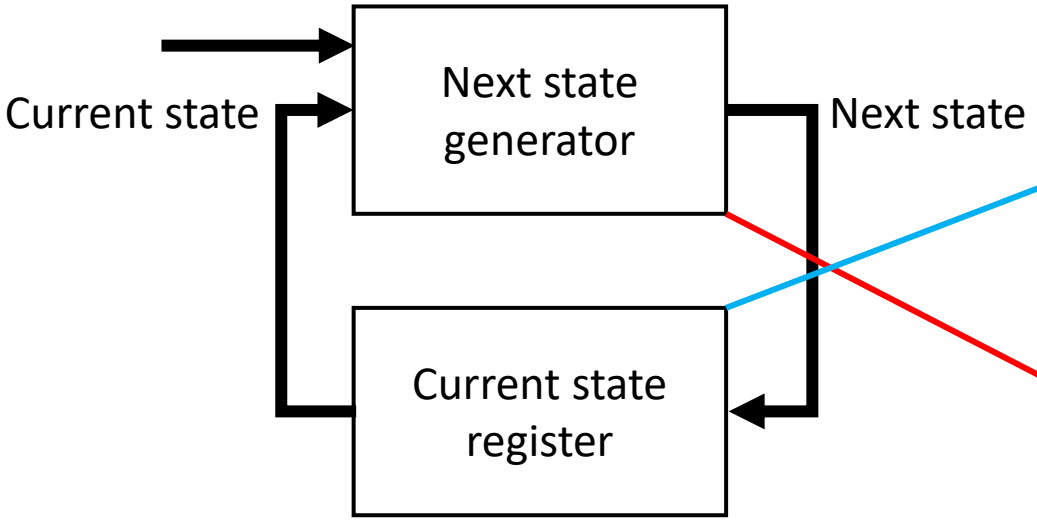
Finite State Machine (FSM)







Finite State Machine (FSM)



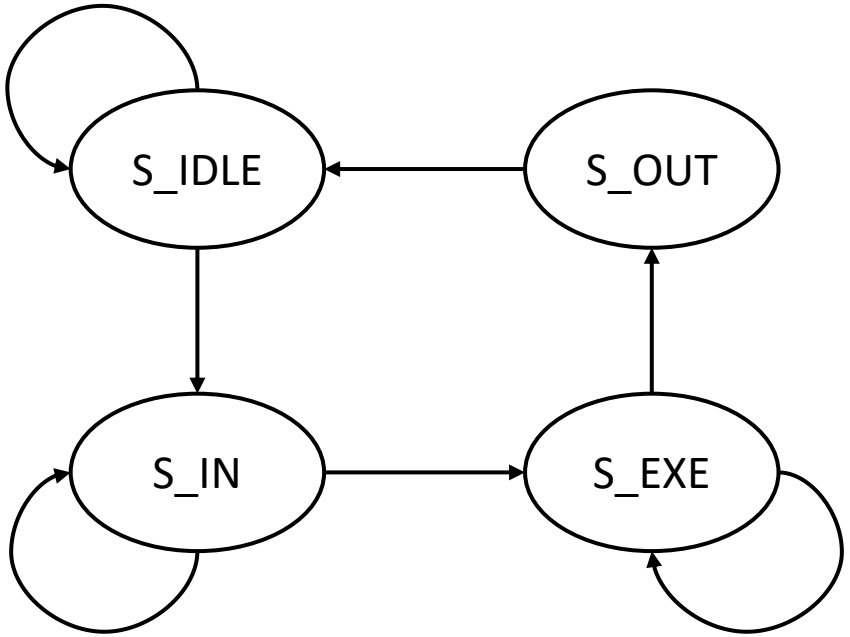

```
localparam S_IDLE   = 'd0;
localparam S_IN     = 'd1;
localparam S_EXE    = 'd2;
localparam S_OUT    = 'd3;

reg [1:0] current_state, next_state;

always @(posedge clk, negedge rst_n) begin
    if (!rst_n)
        current_state <= S_IDLE;
    else
        current_state <= next_state;
end
```

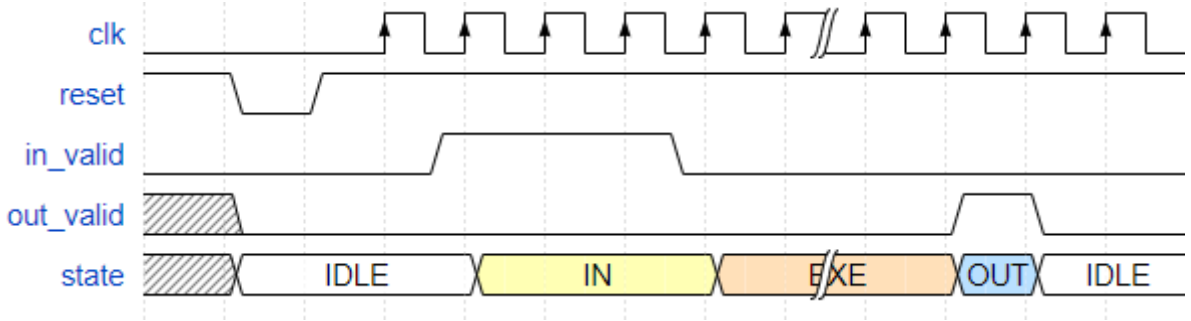
sequential

```
always @(*) begin
    case (current_state)
        S_IDLE  :   if (in_valid)   next_state = S_IN;
                    else            next_state = S_IDLE;
        S_IN    :   if (in_valid)   next_state = S_IN;
                    else            next_state = S_EXE;
        S_EXE   :   if (done)       next_state = S_OUT;
                    else            next_state = S_EXE;
        S_OUT   :                   next_state = S_IDLE;
        default : next_state = S_IDLE;
    endcase
end
```

combinational

Finite State Machine (FSM)






```
localparam S_IDLE   = 'd0;
localparam S_IN     = 'd1;
localparam S_EXE    = 'd2;
localparam S_OUT    = 'd3;

reg [1:0] current_state, next_state;

always @(posedge clk, negedge rst_n) begin
    if (!rst_n)
        current_state <= S_IDLE;
    else
        current_state <= next_state;
end
```

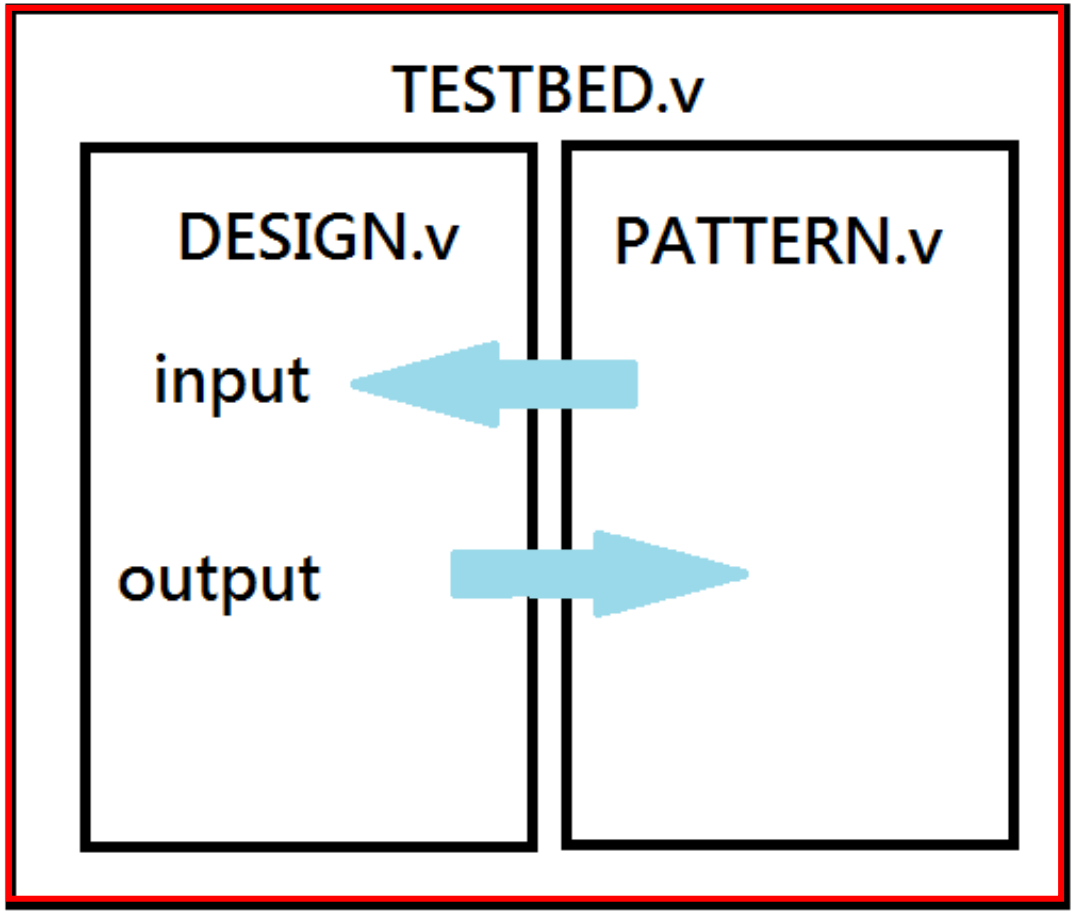
```
always @(*) begin
    case (current_state)
        S_IDLE  :   if (in_valid)   next_state = S_IN;
                    else            next_state = S_IDLE;
        S_IN    :   if (in_valid)   next_state = S_IN;
                    else            next_state = S_EXE;
        S_EXE   :   if (done)       next_state = S_OUT;
                    else            next_state = S_EXE;
        S_OUT   :                   next_state = S_IDLE;
        default : next_state = S_IDLE;
    endcase
end
```

combinational

for loop example

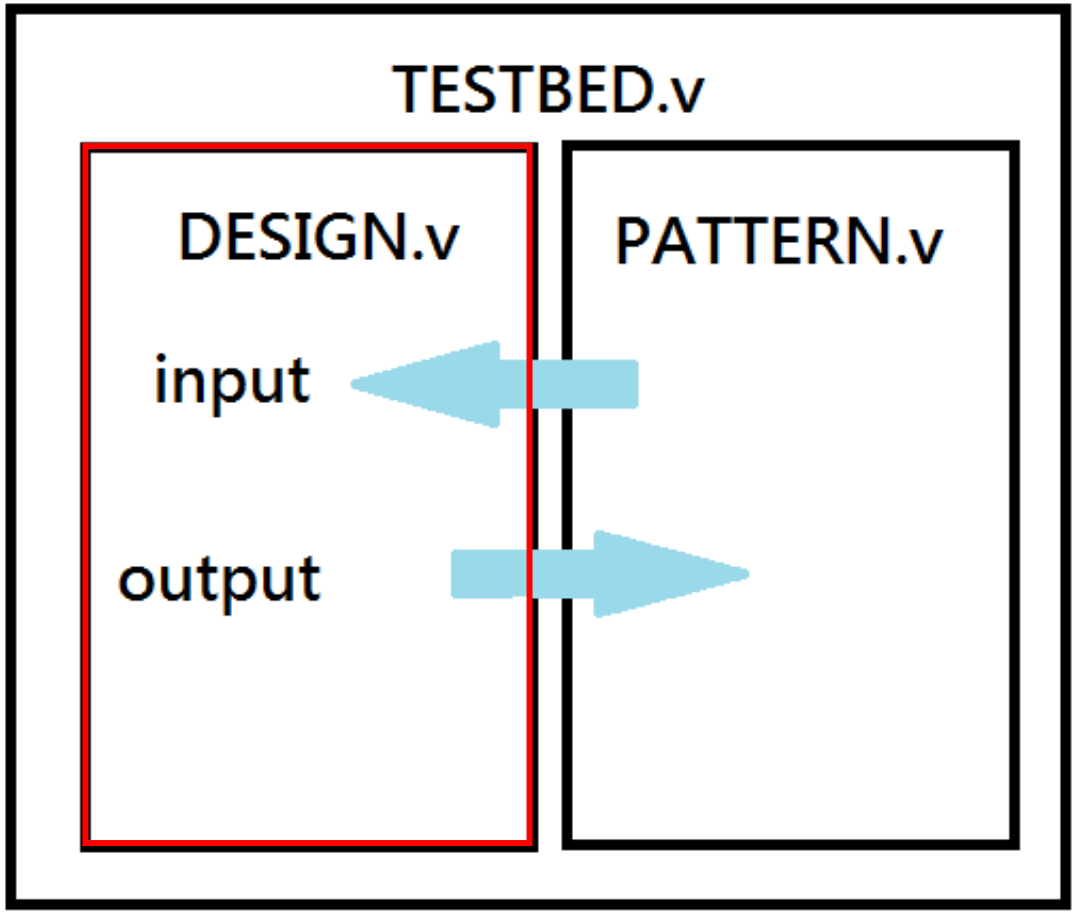
```
integer i;
reg [31:0] tmp[31:0];

always @(posedge clk, negedge rst_n ) begin
    if (!rst_n) begin
        for (i = 0; i < 32; i = i + 1) begin
            tmp[i] <= 'd0;
        end
    end else begin
        /*
        somethine else
        */
    end
end
```

Testbed

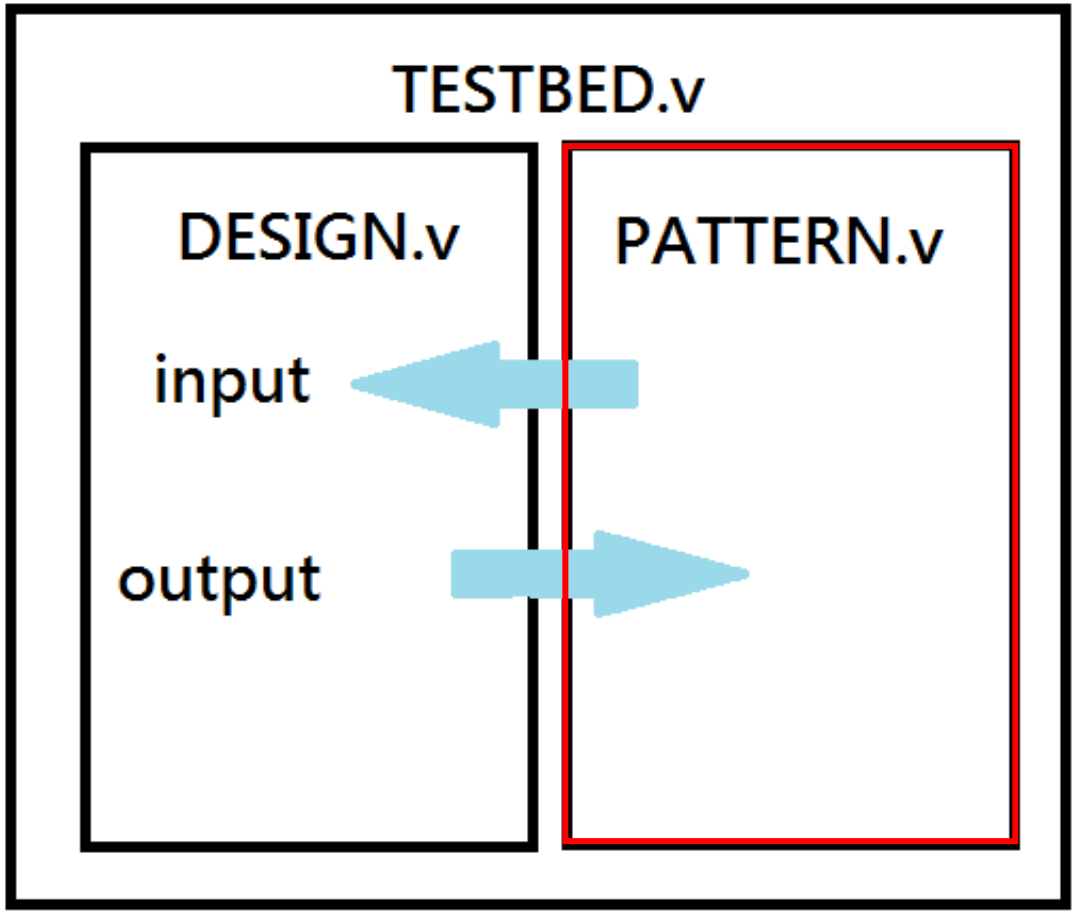



```
module TESTBED;

// wire connection
wire clk,rst_n,in_valid,out_valid,mem_wen;
wire [11:0] mem_addr;
wire [31:0] inst,mem_dout,inst_addr,mem_din;


SP My_SP(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .out_valid(out_valid),
    .mem_wen(mem_wen),
    .inst(inst),
    .mem_dout(mem_dout),
    .inst_addr(inst_addr),
    .mem_addr(mem_addr),
    .mem_din(mem_din)
);

MEM My_MEM(
    .Q(mem_dout),
    .CLK(clk),
    .CEN(1'b0),
    .WEN(mem_wen),
    .A(mem_addr),
    .D(mem_din),
    .OEN(1'b0)
);

PATTERN My_PATTERN(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .out_valid(out_valid),
    .inst(inst)
```

Testbed



Testbed



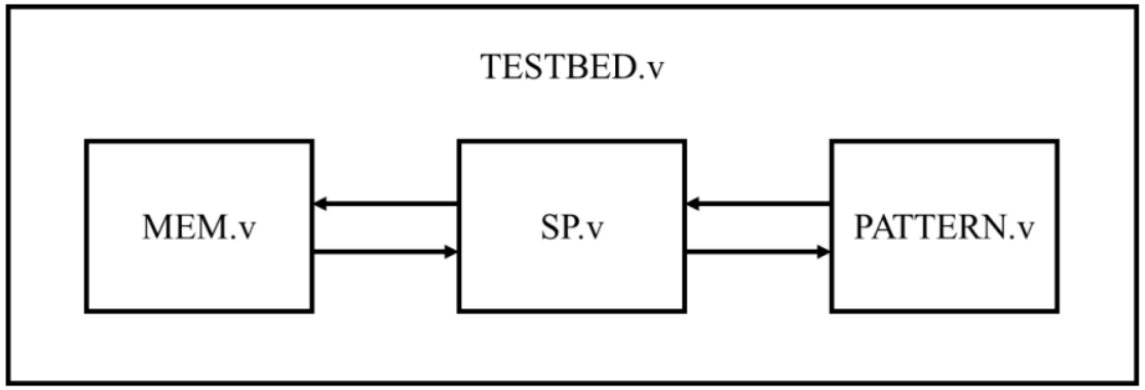
Recommended Text Editor

- VS code with Verilog extension
- Notepad++
- Sublime

Testbench and Pattern

Port declaration

```
module PATTERN(
    // Output Signals
    clk,
    rst_n,
    in_valid,
    inst,
    // Input Signals
    out_valid,
    inst_addr
);

//================================
//   Input and Output Declaration
//================================

output reg clk,rst_n,in_valid;
output reg [31:0] inst;

input wire out_valid;
input wire [31:0] inst_addr;
```




```
SP My_SP(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .out_valid(out_valid),
    .mem_wen(mem_wen),
    .inst(inst),
    .mem_dout(mem_dout),
    .inst_addr(inst_addr),
    .mem_addr(mem_addr),
    .mem_din(mem_din)
);

MEM My_MEM(
    .Q(mem_dout),
    .CLK(clk),
    .CEN(1'b0),
    .WEN(mem_wen),
    .A(mem_addr),
    .D(mem_din),
    .OEN(1'b0)
);

// modify "PATTERN My_PATTERN" to "PATTERN_p My_PATTERN" for pipelined design
PATTERN My_PATTERN(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .out_valid(out_valid),
    .inst(inst),
    .inst_addr(inst_addr)
);
```

Variable declaration

```
//================================================================
// parameters & integer
//================================================================

integer pat_num = 1000, out_max_latency = 10, seed = 64;
integer i, t, pat, out_counter;
integer golden_inst_addr;        // ************** Program Counter ************* //
integer instruction [999:0];     // ******** Instruction (from inst.txt) ******* //
integer opcode, rs, rt, rd, shamt, func, immediate, address;
integer golden_r [31:0];         // ********** Gloden answer for Reg ********** //
integer mem [4095:0];            // ******** Data Memory (from mem.txt) ******** //
```

```
// read data mem & instrction
$readmemh("instruction.txt", instruction);
$readmemh("mem.txt", mem);
```

Initial / task block

```
//================================================
// initial
//================================================

initial begin
    // read data mem & instrction
    $readmemh("instruction.txt", instruction);
    $readmemh("mem.txt", mem);
    // initialize control signal
    rst_n = 1'b1;
    in_valid = 1'b0;
    // initial variable
    golden_inst_addr = 0;
    for(i = 0; i < 32; i = i + 1)begin
        golden_r[i] = 0;
    end
    // inst=X
    inst = 32'bX;
    // reset check task
    reset_check_task;
    // generate random idle clk
    t = $random(seed) % 3 + 1'b1;
    repeat(t) @(negedge clk);
    // main pattern
    for(pat = 0; pat < pat_num; pat = pat + 1)begin
        input_task;
        out_valid_wait_task;
        check_ans_task;
    end
    check_memory_task;
    display_pass_task;
end
```

```
//================================
// task
//================================

// reset check task
task reset_check_task; begin···
end endtask

// input task
task input_task; begin···
end endtask

// out_valid wait task
task out_valid_wait_task; begin
end endtask

// check_ans_task
task check_ans_task; begin···
end endtask

// check_memory_task
task check_memory_task; begin···
end endtask

// display fail task
task display_fail_task; begin···
end endtask

// display pass task
task display_pass_task; begin···
end endtask
```

Input data / Output data check

```
// input task
task input_task; begin
    // inst = ? ,in_valid = 1
    inst = instruction[golden_inst_addr >> 2];
    in_valid = 1'b1;
    @(negedge clk);

    // inst = x ,in_valid = 0
    inst = 32'bX;
    in_valid = 1'b0;
end endtask
```

Input data change at negedge clk

Register value “r[]” From your design

Generate answer from your pattern

```
// check register
for(i=0;i<32;i=i+1)begin
    if(My_SP.r[i] !== golden_r[i])begin
        display_fail_task;
        $display("--------------------------------------------------------------------");
        $display("*                        PATTERN NO.%4d                           *",pat);
        $display("*                   register [%2d]  error                         *",i);
        $display("*          answer should be : %d , your answer is : %d             *",golden_r[i],My_SP.r[i]);
        $display("--------------------------------------------------------------------");
        repeat(2) @(negedge clk);
        $finish;
    end
end
```

Check output data at negedge clk

Function validation

check_ans_task

```
// answer calculate
opcode = instruction[golden_inst_addr_out>>2][31:26];
rs = instruction[golden_inst_addr_out>>2][25:21];
// R-type
// I-type
// PC & jump, beq...etc.
// hint: it's necessary to consider sign extension while calculating
/*


    Complete your function validation here



*/
```

input_task

```
// PC
opcode = instruction[golden_inst_addr_in>>2][31:26];
rs = instruction[golden_inst_addr_in>>2][25:21];
// You can calculate golden_inst_addr_in is (triggered by Jump, beq...etc.) or (PC + 4), as a check for design output inst_addr
// PC & jump, beq...etc.
// hint: it's necessary to consider sign extension while calculating
/*

    Complete your function validation here

*/
```