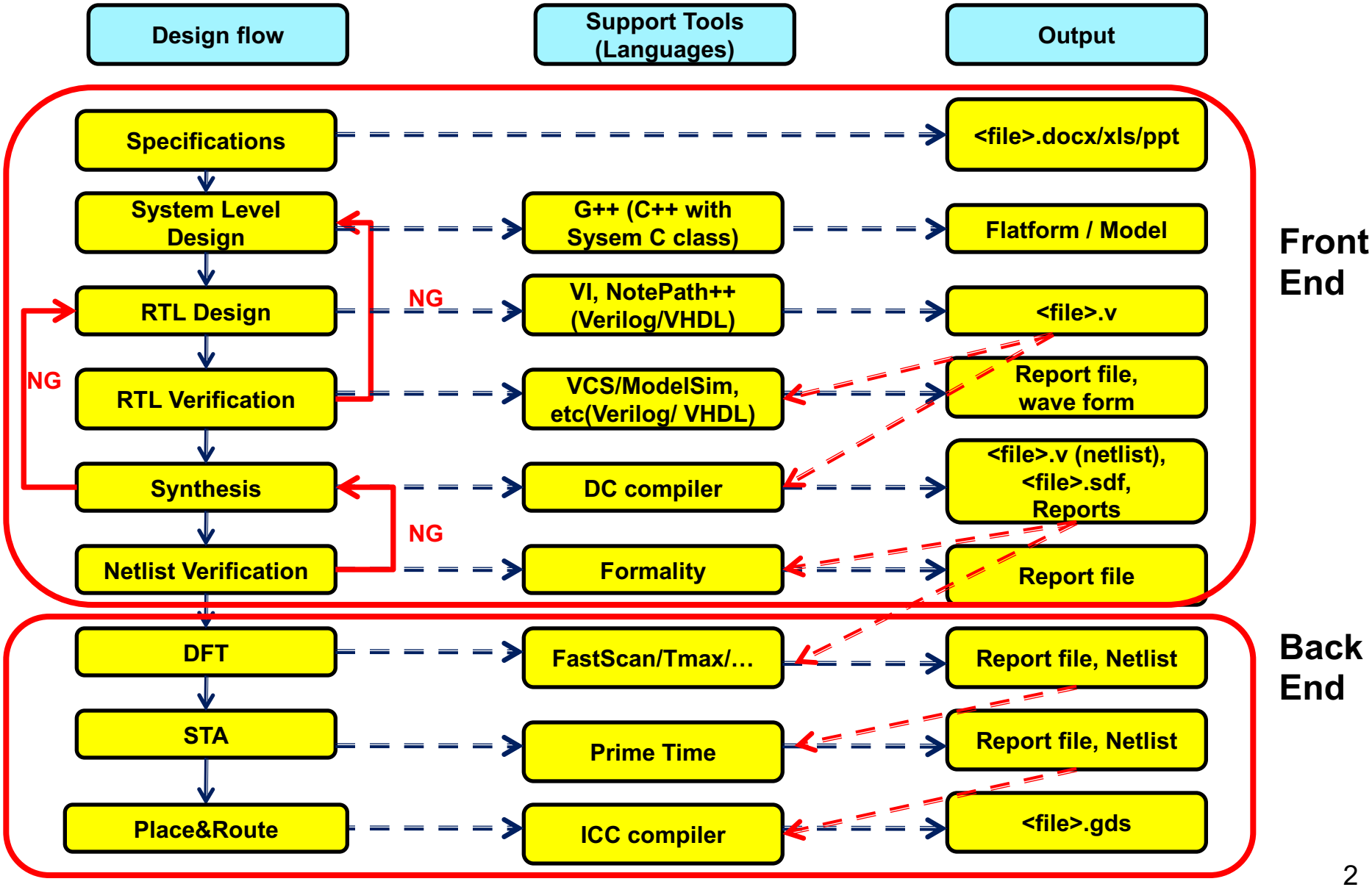
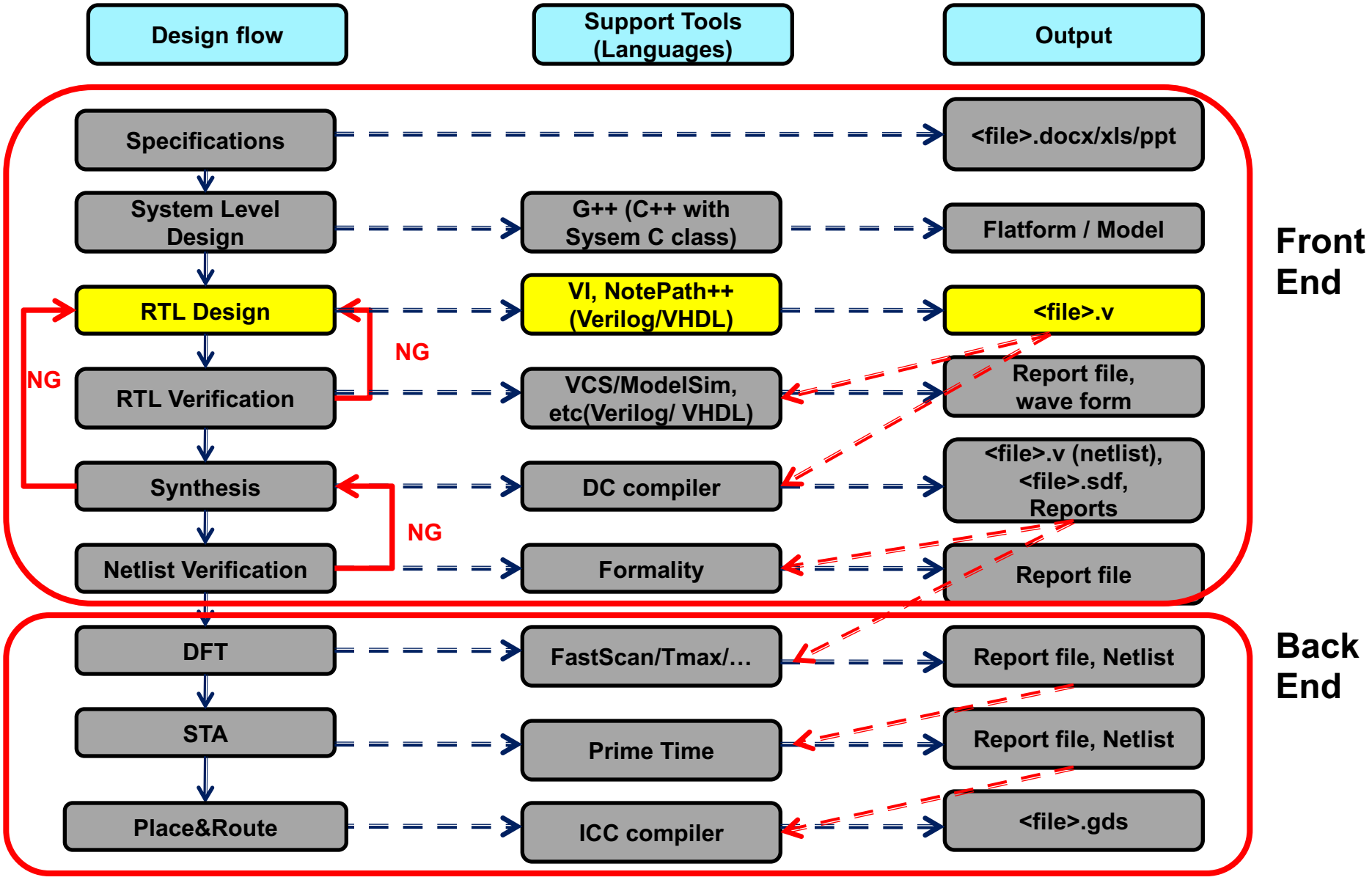
Verilog Basic

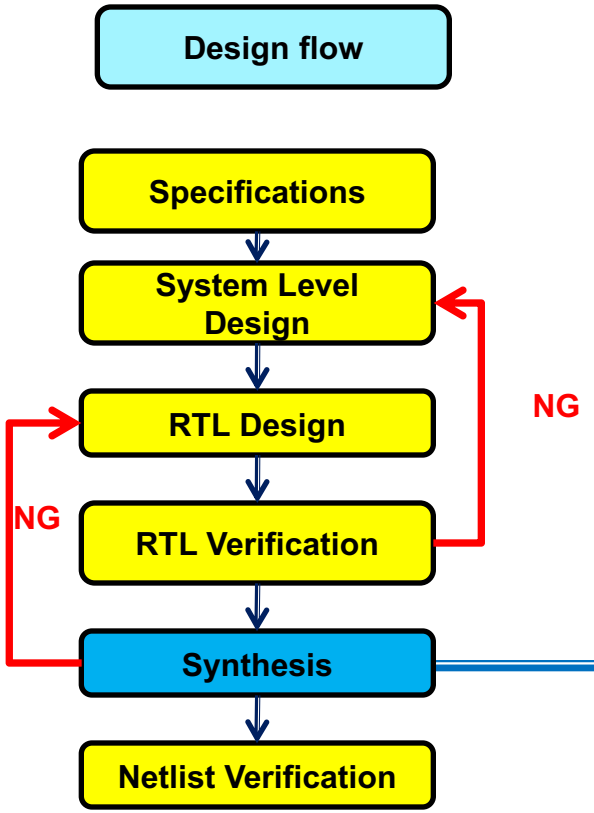
Cell Based Design Flow Review



RTL Design Step



Cell Based Design Flow Review



Can simulate, but cannot synthesize to Gate Level

Tools are not enough excellent to understand your writing

Agenda

- Introduction
- Verilog language with different levels?
- How to write Verilog code well to be able to Synthesize to gate level?

Introduction

❖Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuit.

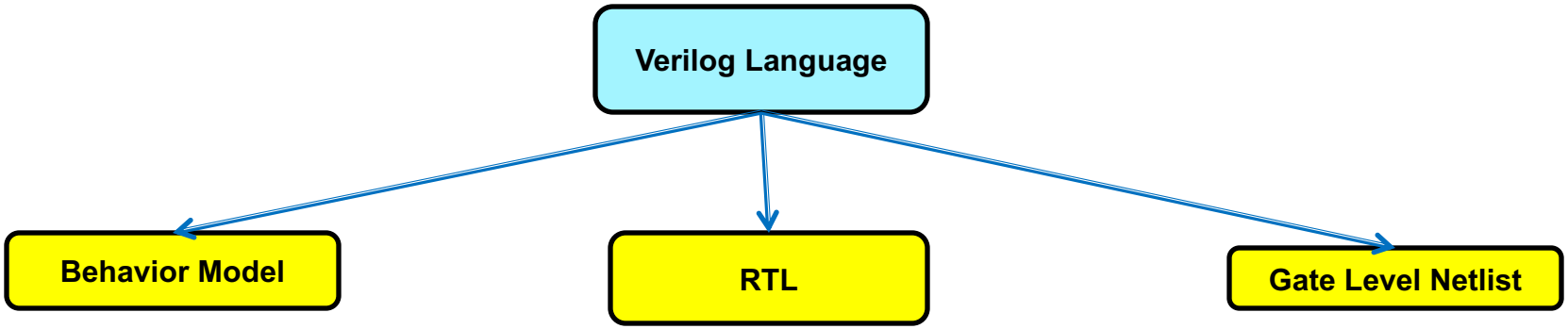
❖Verilog was the first modern hardware description language to be invented. It was created by Phil Moorby and Prabhu Goel during the winter of 1983/1984

❖Versions: Verilog-95, Verilog 2001, Verilog 2005, SystemVerilog

Agenda

- Introduction
- **Verilog language with different level?**
- How to write Verilog code well ?

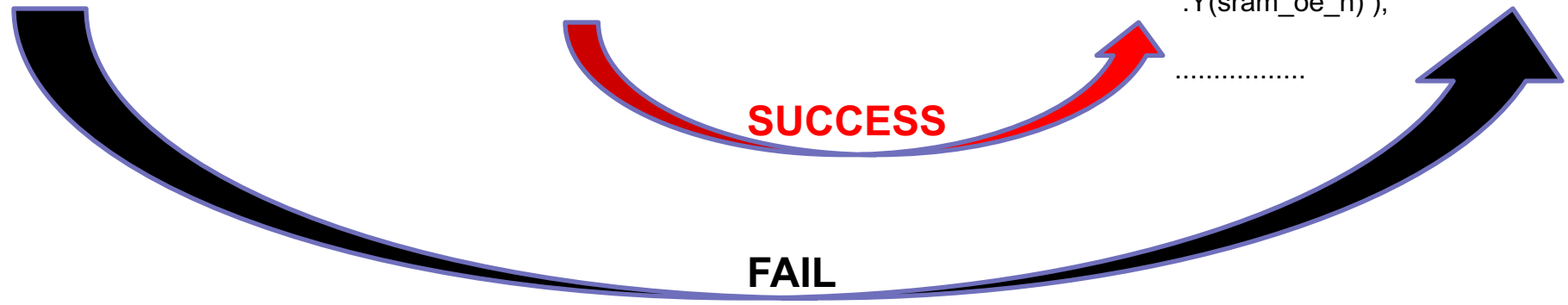
Behavior level



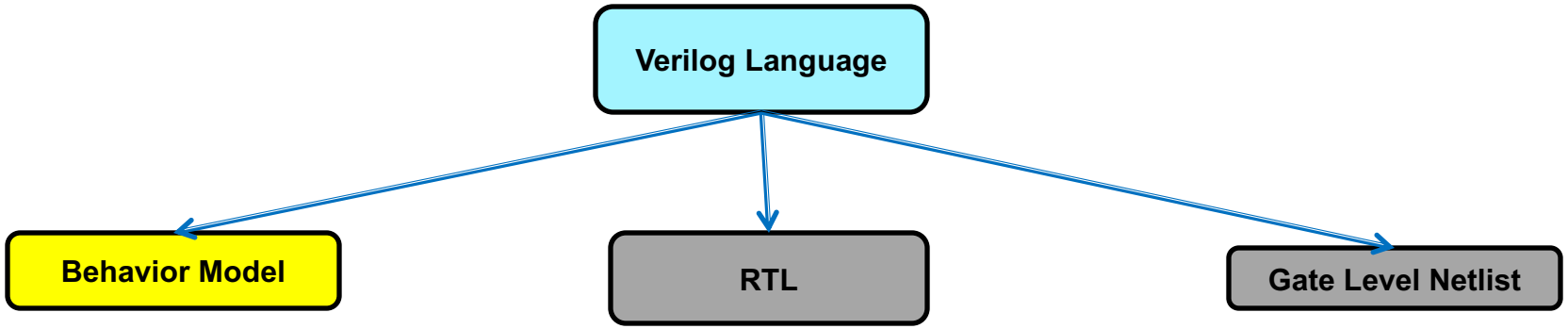

```
Initial begin
   while (1) begin
      Red = 1; Yellow = 0; Green = 0;
      # 30;
      Red = 0; Yellow = 0; Green = 1;
      # 27;
      Red = 0; Yellow = 1; Green = 0;
      #3;
   end
end
```

```
assign A = B&C;

always@(posedge CLK or negedge PRESET) begin
   if(!PRESET) begin
      Q <= 0;
   end
   else begin
      Q <= A;
   end
end
```

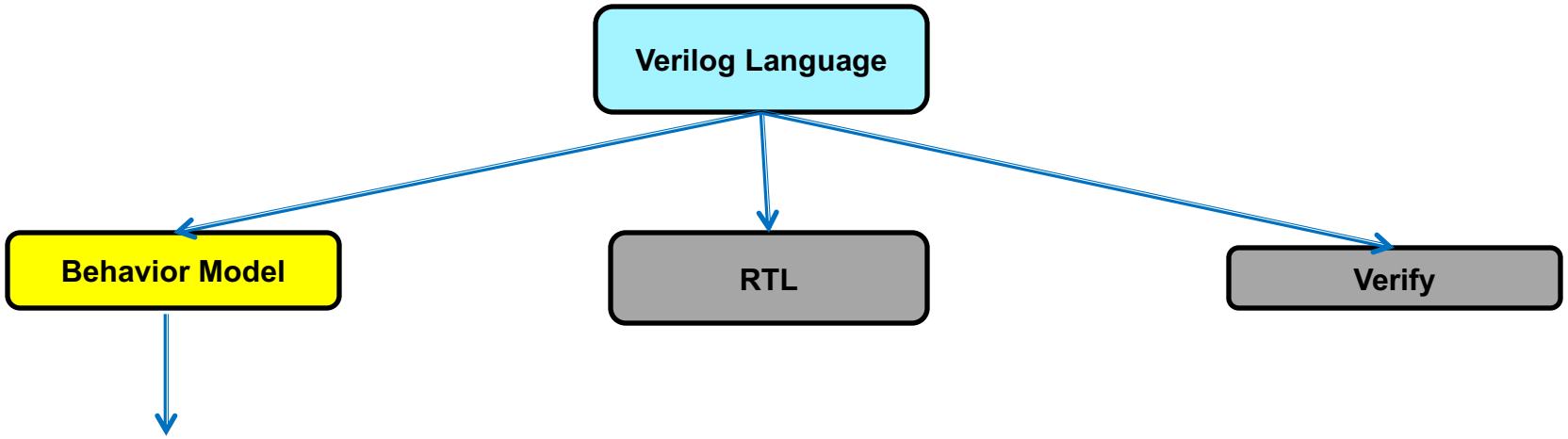
```
module sram_controller ( clk, rst_n,
ready, fs, addr_recog, sram_addr);

input [13:0] addr_recog;
output [13:0] sram_addr;
input clk, rst_n, ready, fs;

DFFSXL chip_ena_reg ( .D(vsr_ena_n)
 .CK(clk), .SN(rst_n), .Q(chip_ena) );

XAND2X1 U4 ( .A(fs), .B(ready),
 .Y(sram_oe_n) );

.....
```



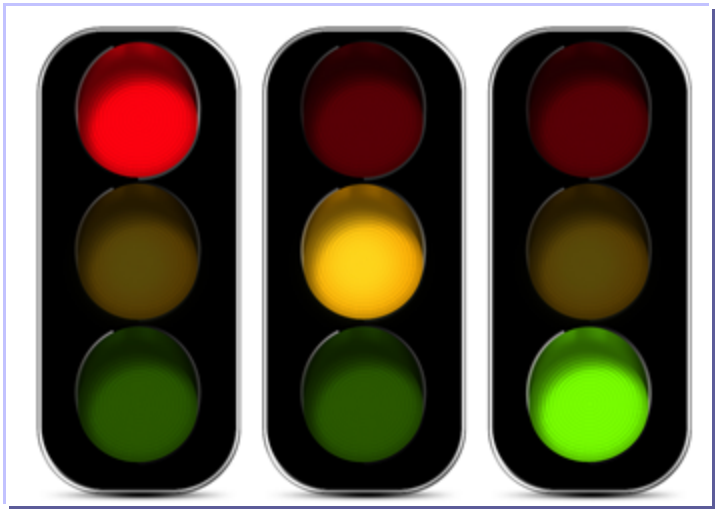
Behavior level



Behavior level

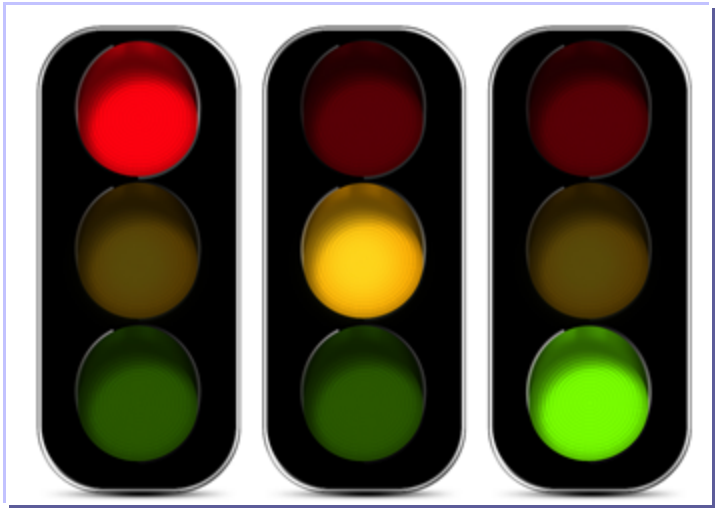


How to model a behavior of traffic light system?



- Red → Green : 30 s
- Green → Yellow : 27s
- Yellow → Red : 3 s

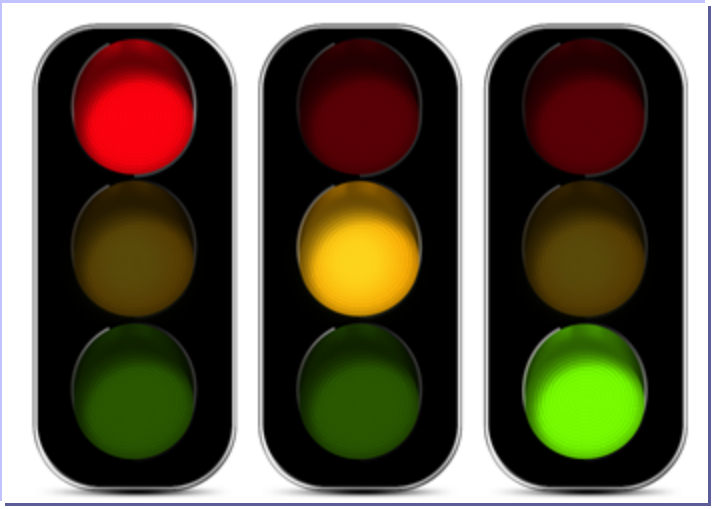
Behavior level



- Red → Green : 30 s
- Green → Yellow : 27s
- Yellow → Red : 3 s

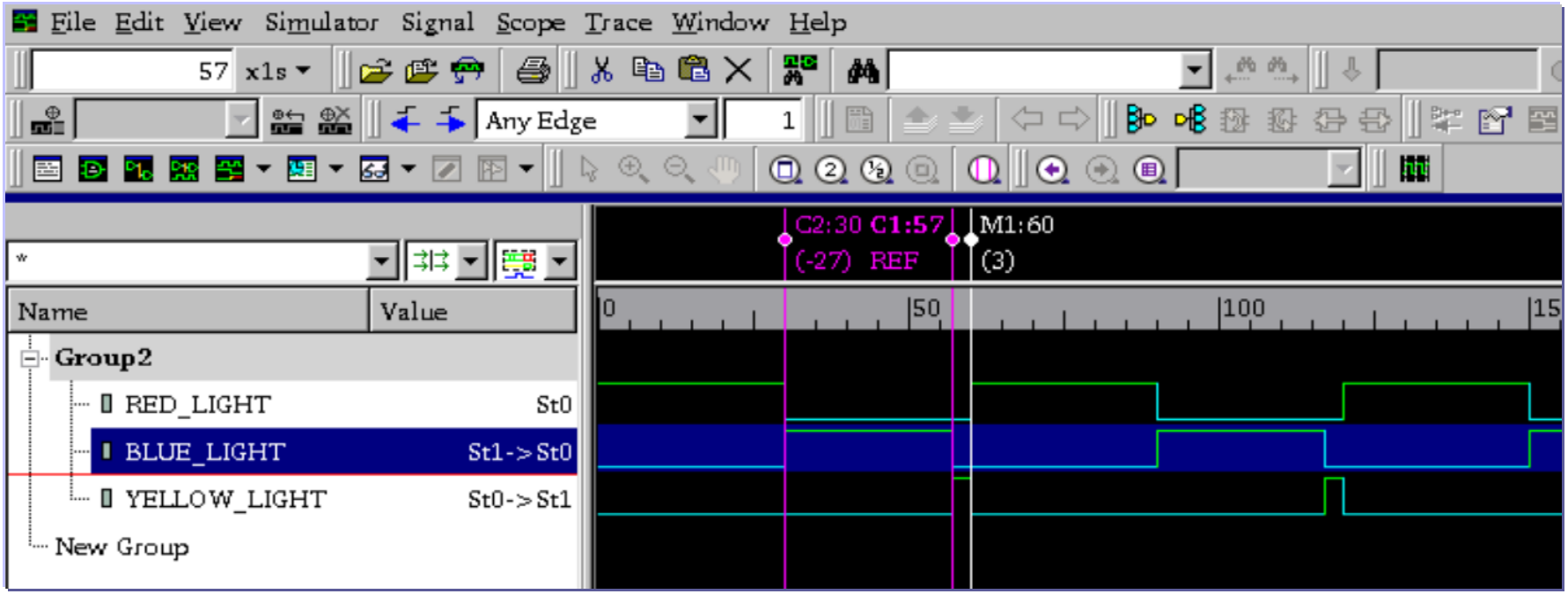
```
Initial begin
   while (1) begin
      Red = 1; Yellow = 0; Green = 0;
      # 30;
      Red = 0; Yellow = 0; Green = 1;
      # 27;
      Red = 0; Yellow = 1; Green = 0;
      #3;
   end
end
```

Behavior level

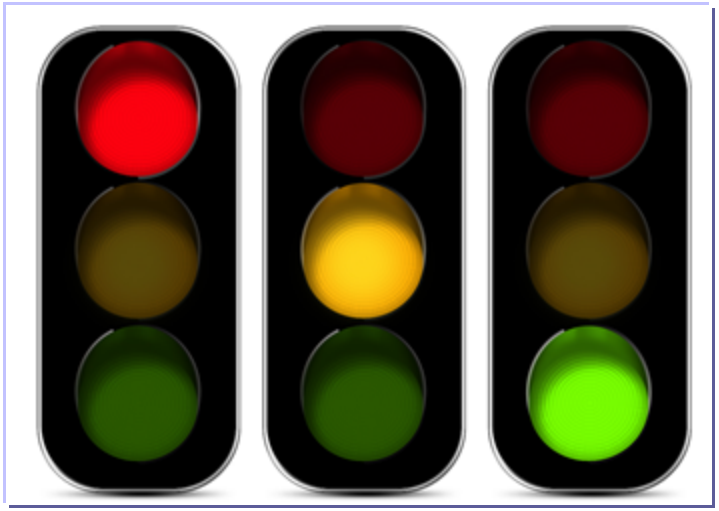


- Red → Green : 30 s
- Green → Yellow : 27s
- Yellow → Red : 3 s

Verificataion



Behavior level

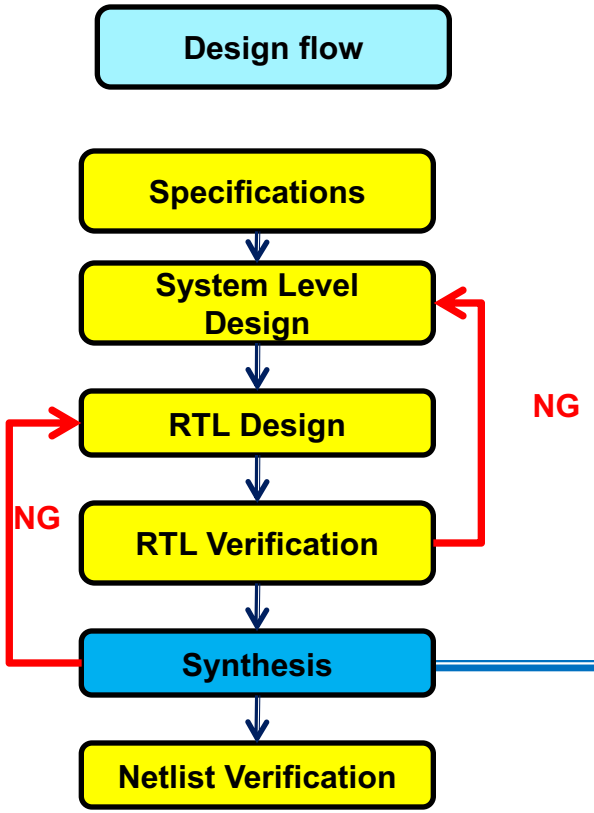


- Red → Green : 30 s
- Green → Yellow : 27s
- Yellow → Red : 3 s

```
Initial begin
   while (1) begin
      Red = 1; Yellow = 0; Green = 0;
      # 30;
      Red = 0; Yellow = 0; Green = 1;
      # 27;
      Red = 0; Yellow = 1; Green = 0;
      #3;
   end
end
```

Can simulate, but cannot be synthesized to Gate Level

Cell Based Design Flow Review



Can simulate, but cannot synthesize to Gate Level

Tools are not enough excellent to understand your writing

Cell Based Design Flow Review

```
Warning:  /home/quanghan/Work/01_Lab/01_vcs_demo/03_synthesize_mod
el_fail/01_rtl/model.v:21: The statements in initial blocks are ig
nored. (VER-281)
Warning:  /home/quanghan/Work/01_Lab/01_vcs_demo/03_synthesize_mod
el_fail/01_rtl/model.v:25: The statements in initial blocks are ig
nored. (VER-281)
Warning:  /home/quanghan/Work/01_Lab/01_vcs_demo/03_synthesize_mod
el_fail/01_rtl/model.v:30: The statements in initial blocks are ig
nored. (VER-281)
```

Synthesize Report

```
module model;

reg BLUE_LIGHT; //Blue
reg YELLOW_LIGHT; //Yello
reg RED_LIGHT; //Red

initial begin
    while(1) begin
    #0 BLUE_LIGHT = 0;
       YELLOW_LIGHT = 0;
       RED_LIGHT = 1;
    #30 BLUE_LIGHT = 1;
       YELLOW_LIGHT = 0;
       RED_LIGHT = 0;
    #27 BLUE_LIGHT = 0;
       YELLOW_LIGHT = 1;
       RED_LIGHT = 0;
    #3 ;
    end
end
initial begin
    # 10000 $finish;
end

initial begin
    $vcdplusfile("Traffic_life.vpd");
    $vcdpluson();
end

endmodule
~
```

Verilog

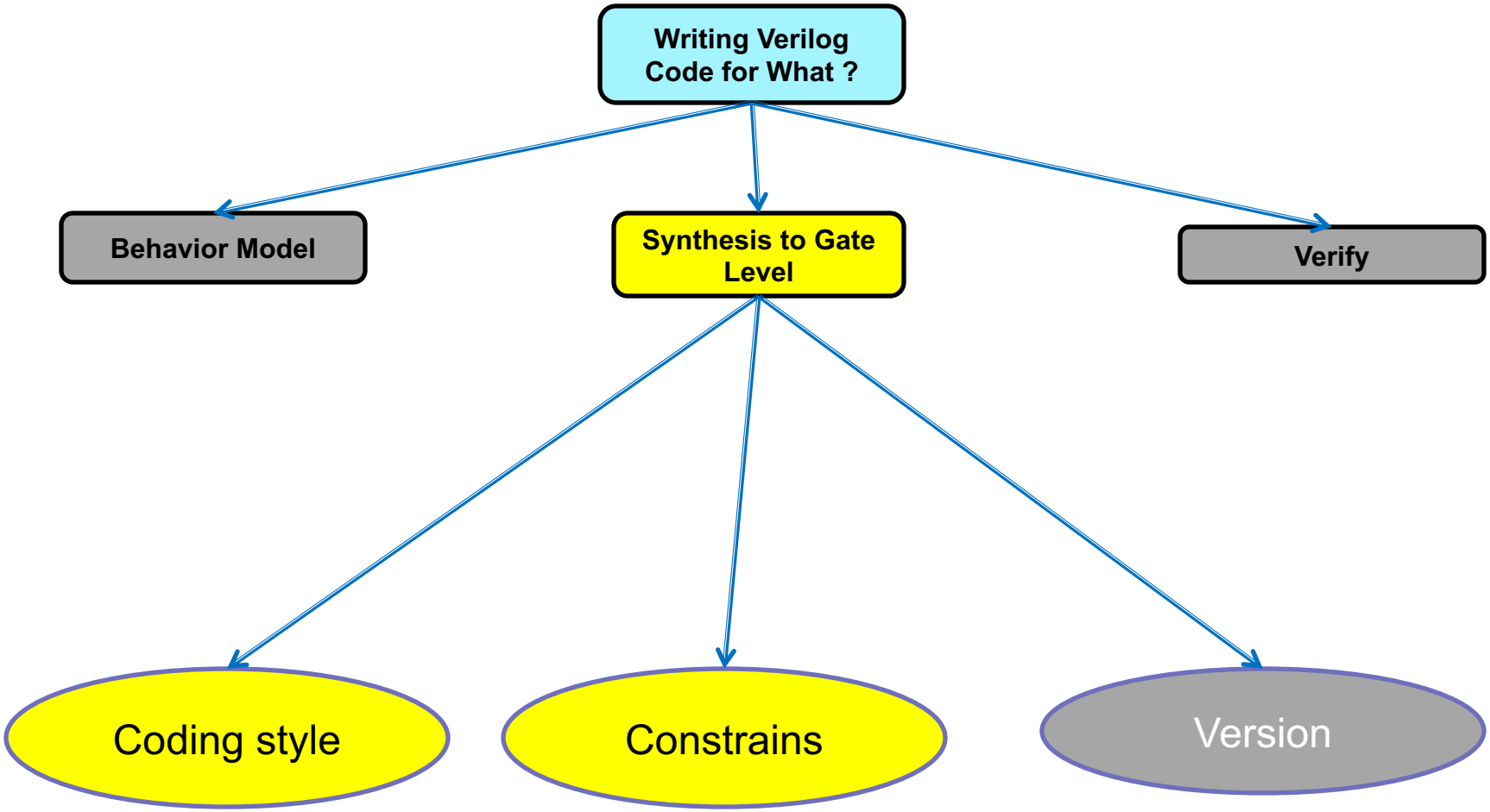
```
module model (   );


endmodule
~
"model.netlist.V" 6 lines --16%--
```

Netlist

Agenda

- Introduction
- Verilog language with different level?
- **How to write Verilog code well ?**

Writing Verilog code for what ?



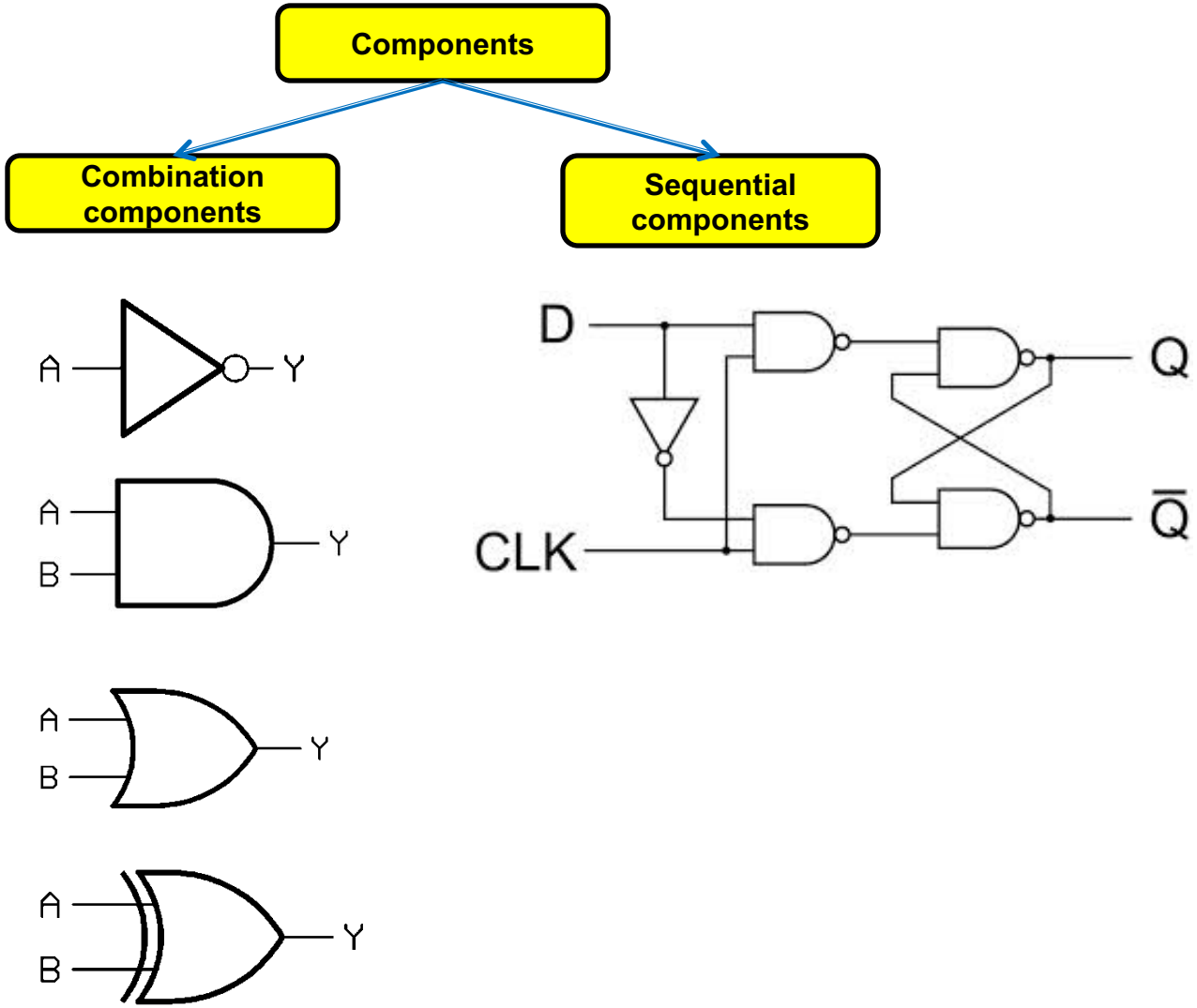
How to write Verilog code well ?

- Synthesized Verilog Introduce
- Verilog Basic

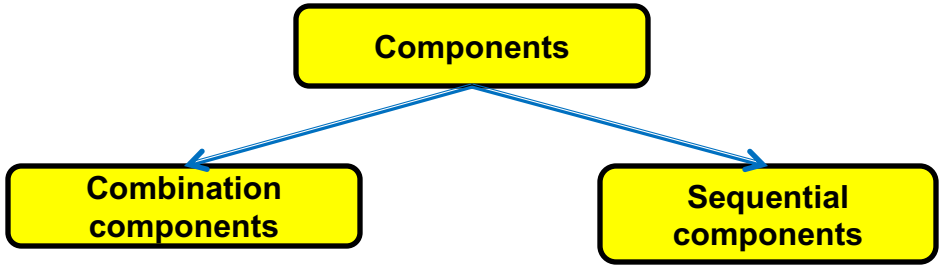
How to write Verilog code well ?

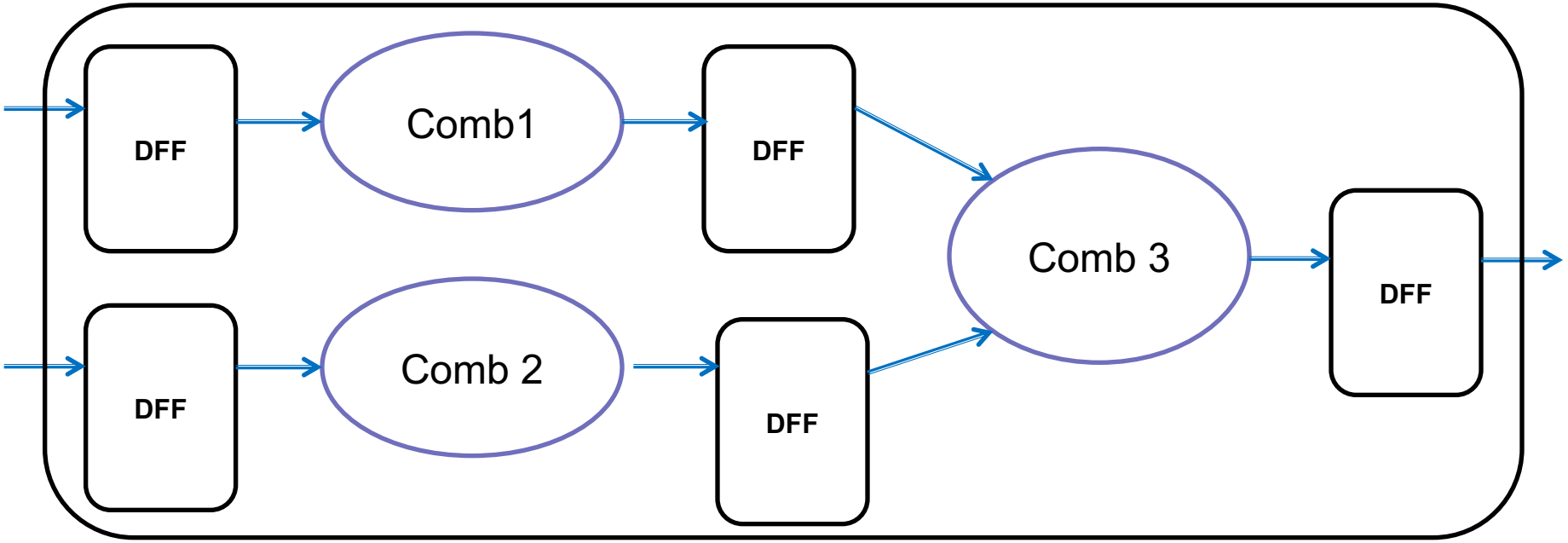
- Synthesized Verilog Introduce
- Verilog Basic

Synthesized Verilog Introduce



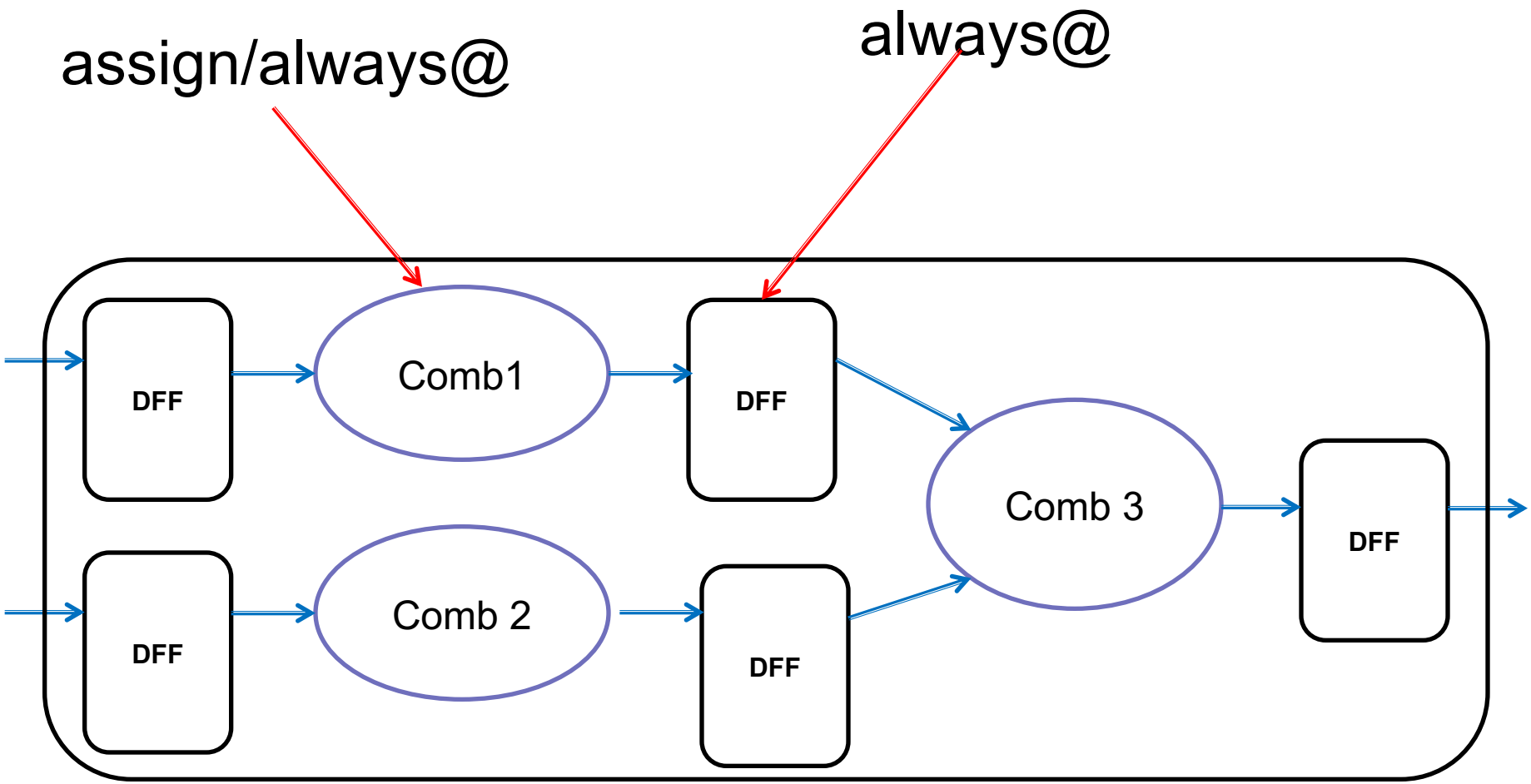
Synthesized Verilog Introduce





Synthesized Verilog Introduce

Prototype

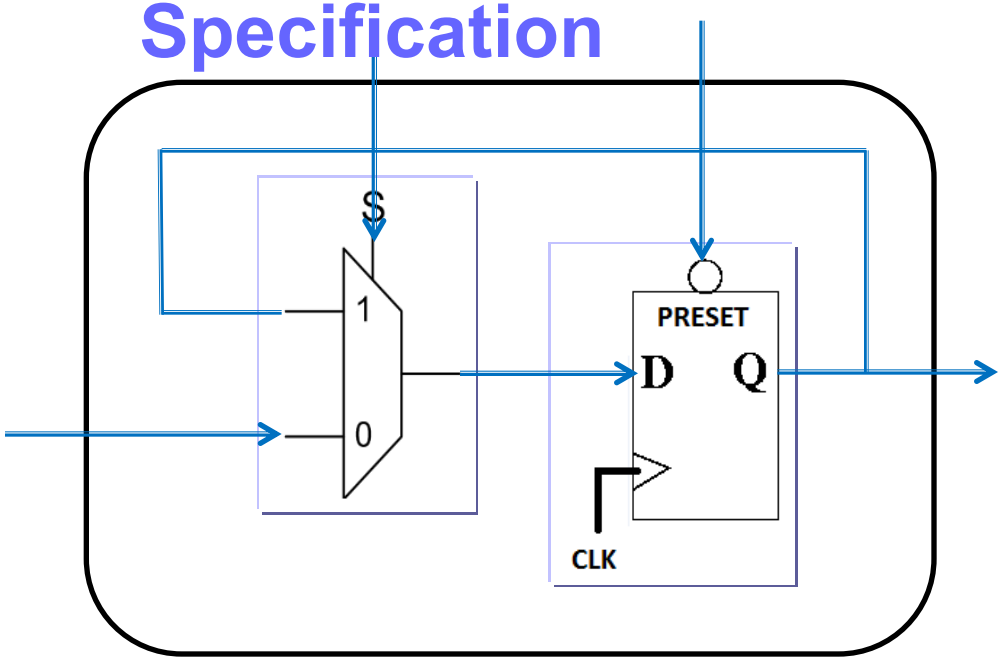


Synthesized Verilog Introduce

Thinking Flow

we have idea

Specification



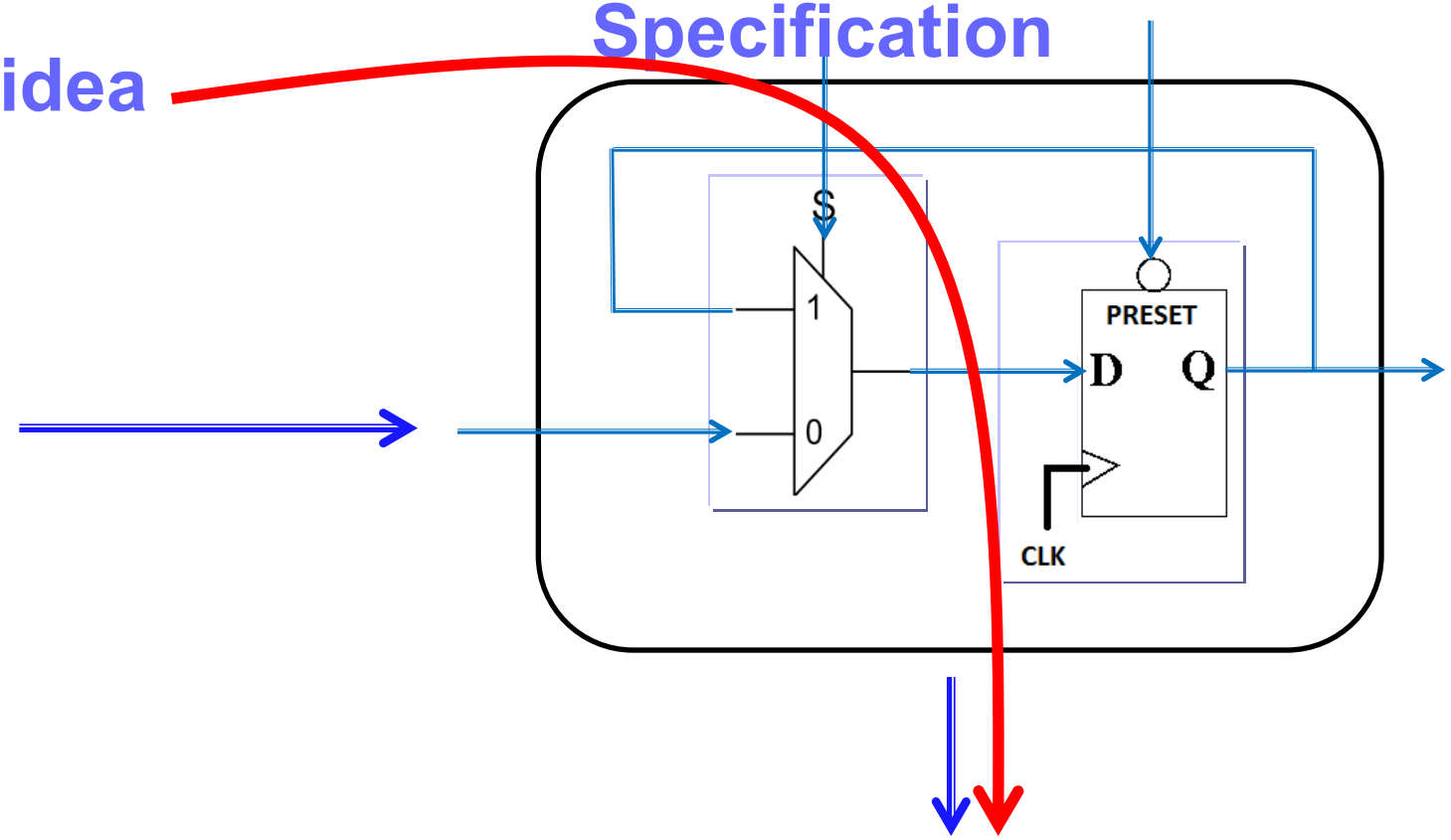
How to write source code Verilog ?

Synthesized Verilog Introduce

Thinking Flow

we have idea

Specification

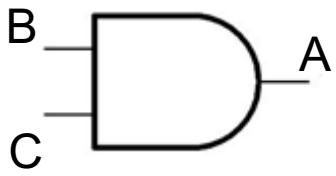


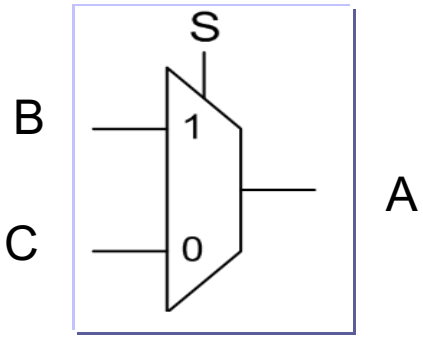
How to write source code Verilog ?

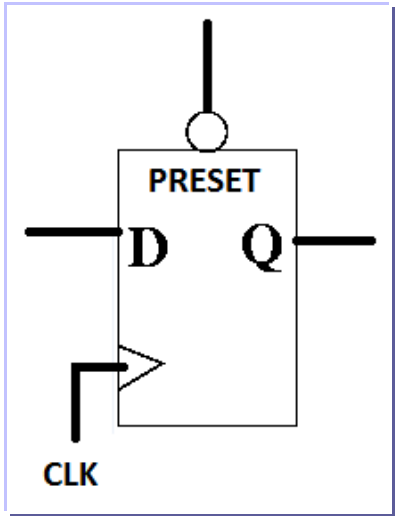
How to write Verilog code well ?

- Synthesized Verilog Introduce
- **Verilog Basic**

Verilog Basic

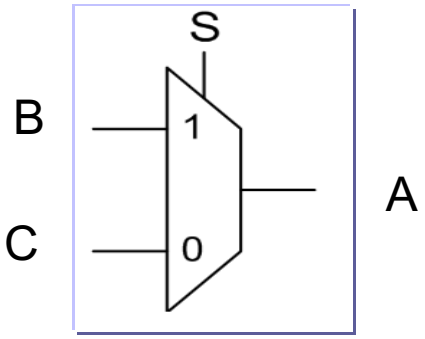




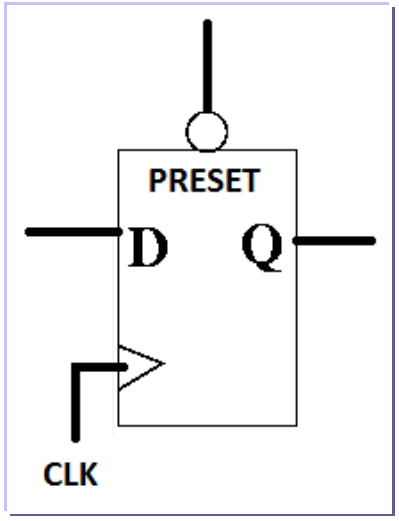


Verilog Basic

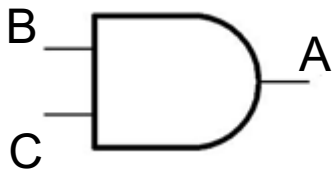

```
assign A = B&C;

always@(B  or C) begin
    A <= B&C;
end
```




```
assign A = (S == 1)?B:C;

always@(S or B or C) begin
    if(S == 1) begin
        A <= B;
    end
    else begin
        A <= C;
    end
end
```

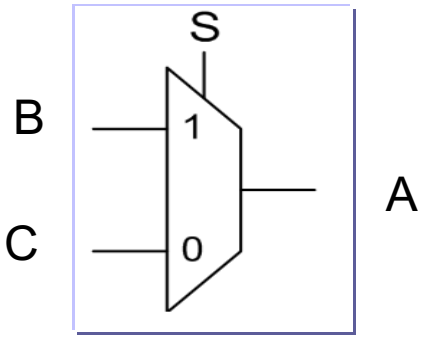



```
always@(posedge CLK or negedge PRESET) begin
    if(!PRESET) begin
        Q <= 0;
    end
    else begin
        Q <= D;
    end
end
```

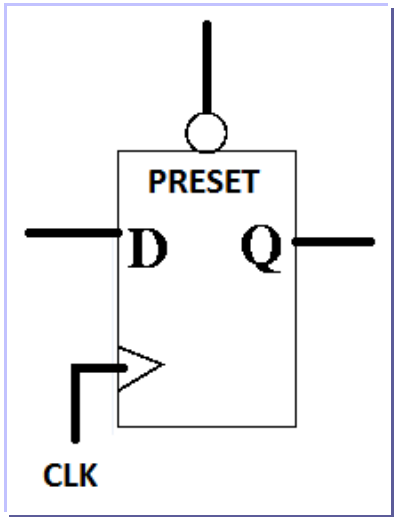



```
architecture rtl of AND_logic is
begin
   A <= B and C;
end architecture;
```

```
architecture rtl of AND_logic is
begin
   process (B, C) begin
      A <= B and C;
   end process
end architecture;
```




```
architecture rtl of mux is
begin
process (S, B, C) begin
 if (S= '0') then
     A <= C;
elsif (S = '1') then
     A <= B;
end if;
end process;
end architecture;
```




```
architecture rtl of dff_async_reset is
begin
process (CLK, PRESET) begin
 if (PRESET= '0') then
     q <= '0';
elsif (rising_edge(CLK)) then
     q <= D;
end if;
end process;
end architecture;
```

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

Verilog Basic

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

always@(X) **begin**
end

Verilog Basic

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    case (X)
    endcase
end
```

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    case (X)
        3'b000: begin
        end
        3'b001: begin
        end
        3'b010: begin
        end
        default : begin
        end
    endcase
end
```

Verilog Basic

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    case (X)
        3'b000: begin
            Y <= 2'b01;
        end
        3'b000: begin
            Y <= 2'b10;
        end
        3'b000: begin
            Y <= 2'b11;
        end
        default : begin
            Y <= 2'b00;
        end
    endcase
end
```

Verilog Basic

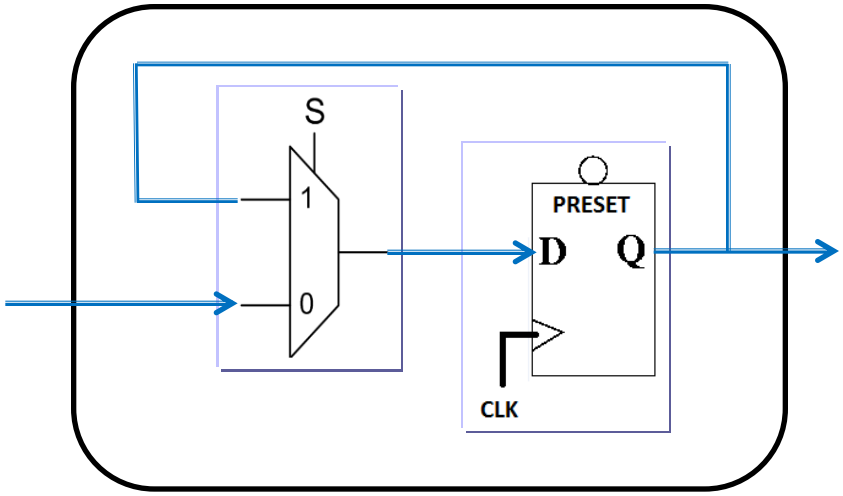
X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    case (X)
        3'b000: begin
            Y <= 2'b01;
        end
        3'b001: begin
            Y <= 2'b10;
        end
        3'b010: begin
            Y <= 2'b11;
        end
        default : begin
            Y <= 2'b00;
        end
    endcase
end
```

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

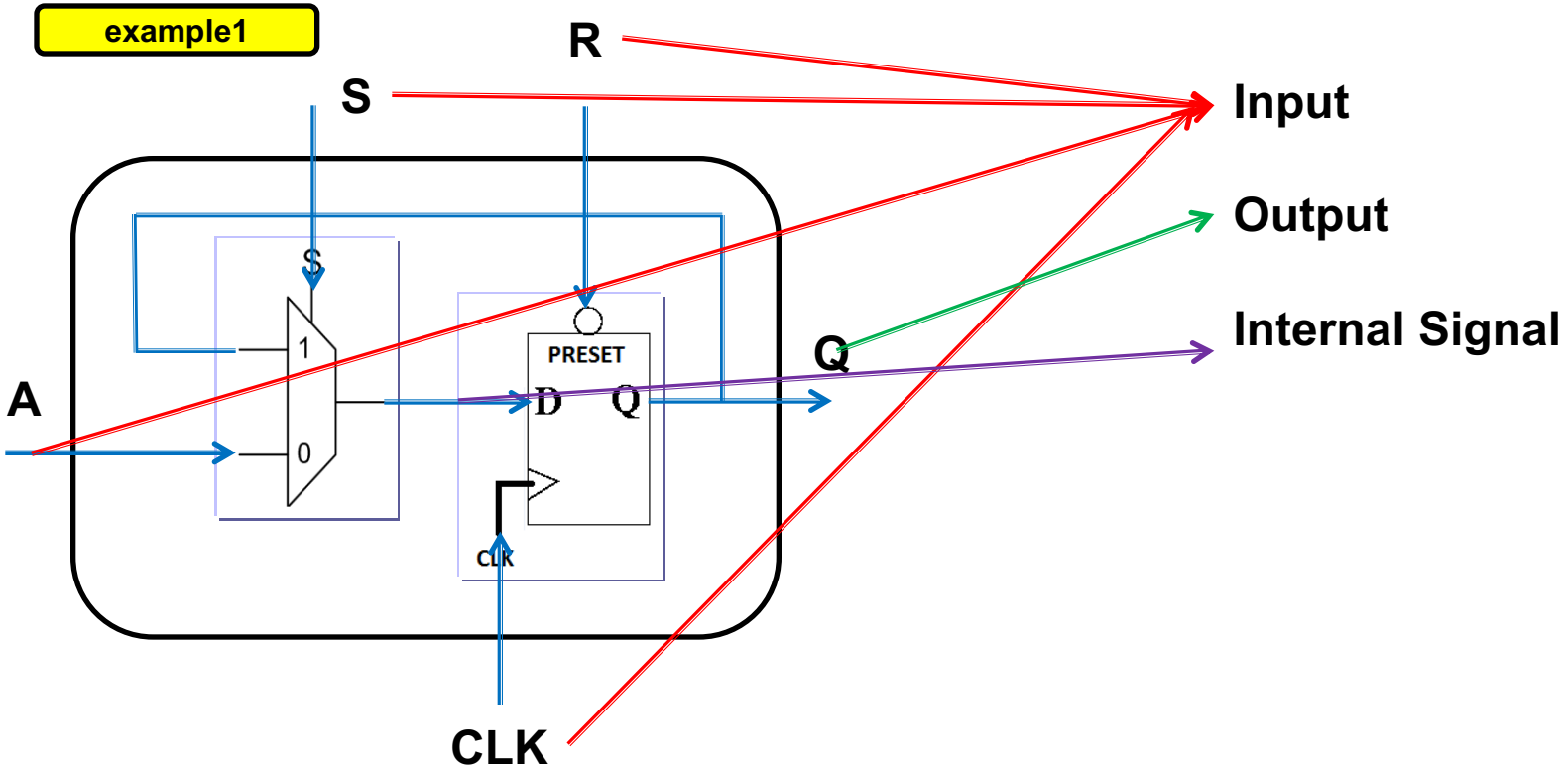
```
always@(X) begin
    if (X == 3'b000) begin
        Y <= 2'b01;
    end
    else if (X == 3'b001) begin
        Y <= 2'b10;
    end
    else if (X == 3'b010) begin
        Y <= 2'b11;
    end
    else begin
        Y <= 2'b00;
    end
end
```

example1



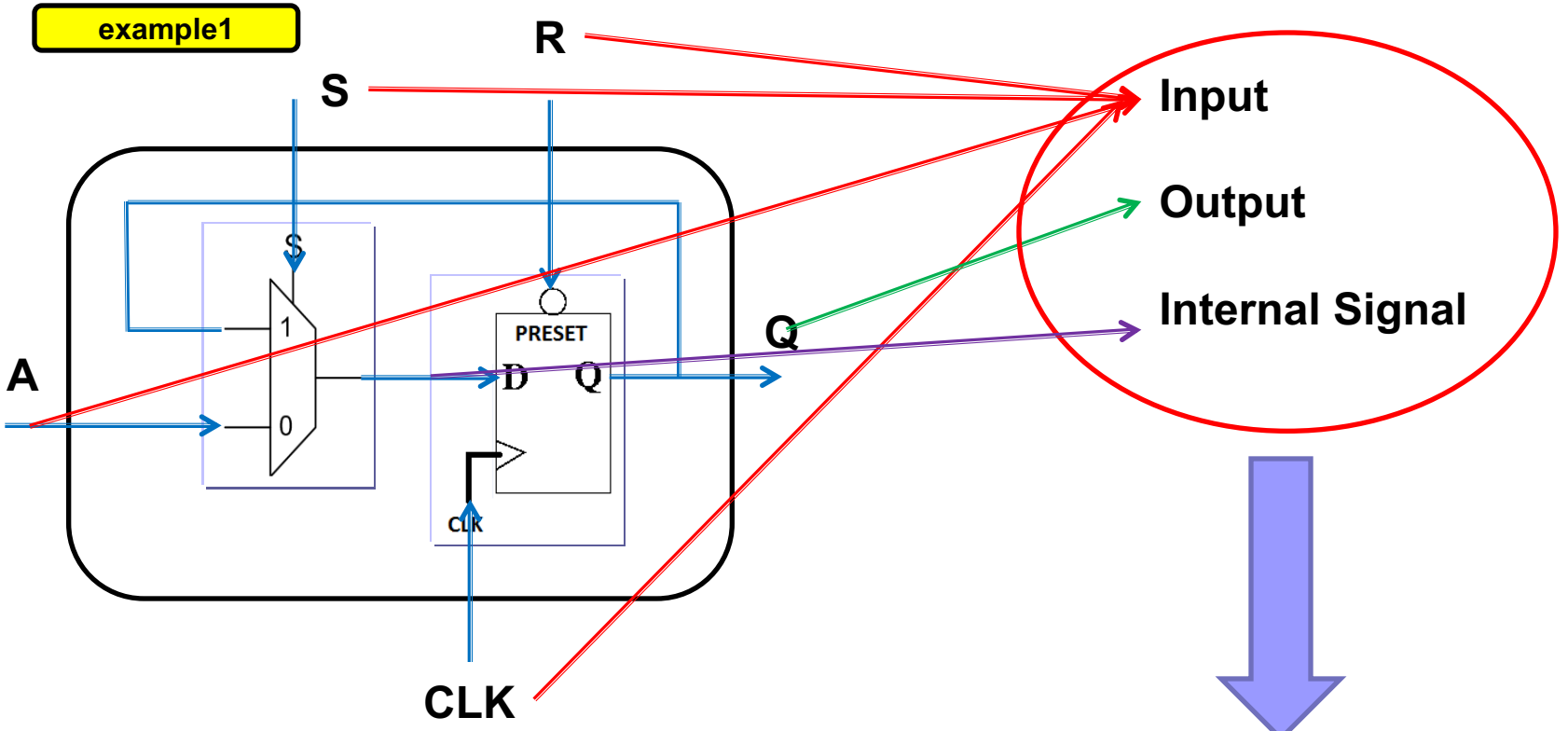
Verilog Basic

example1



Verilog Basic

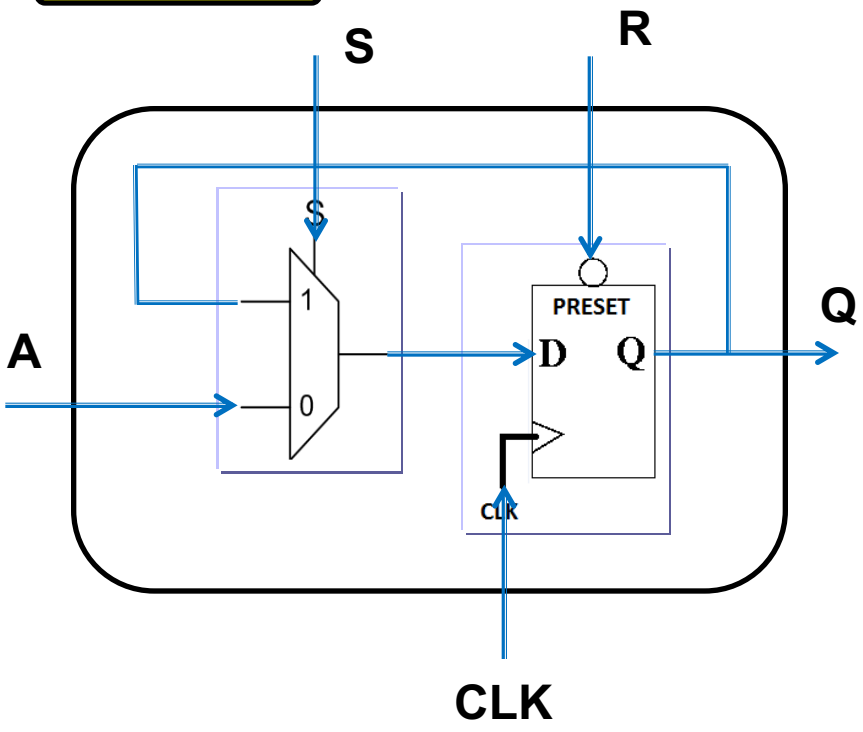
example1



Should be "reg/wire" data type

Verilog Basic

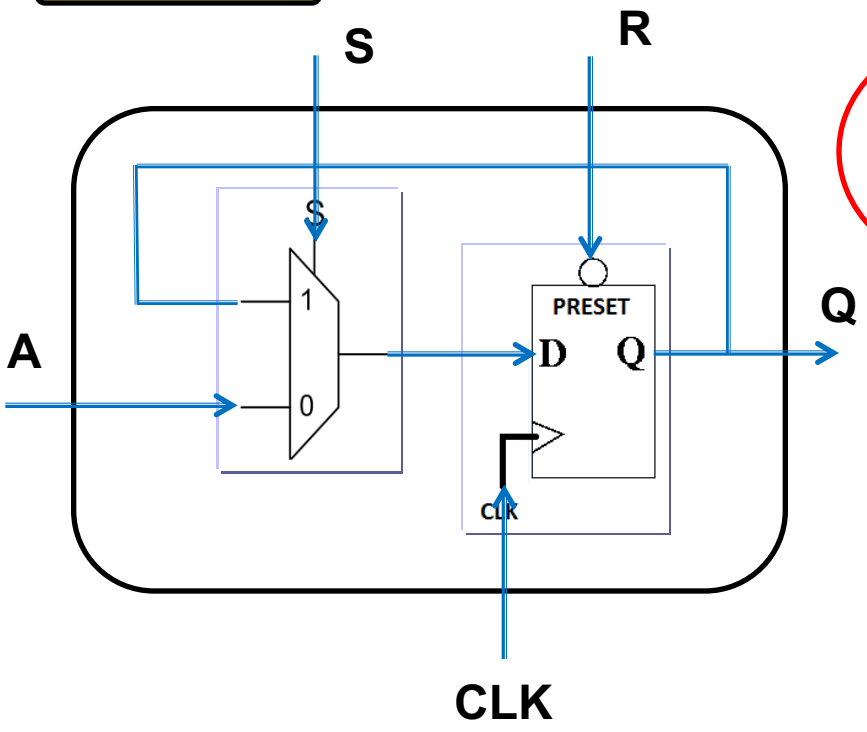
example1




```
module example1 (CLK, R, S, A, Q);

input CLK;
input R;
input S;
input A;

output Q
reg Q;

wire D;

assign D = (S==0)?A:Q;


always@(posedge CLK or negedge R) begin
    if(R == 0) begin
        Q <= 0;
    end
    else begin
        Q <= D;
    end
end
endmodule
```

Verilog Basic

example1




```
module example1 (CLK, R, S, A, Q);

input CLK;
input R;
input S;
input A;

output Q
reg Q;

wire D;

assign D = (S==0)?A:Q;

always@(posedge CLK or negedge R) begin
    if(R == 0) begin
        Q <= 0;
    end
    else begin
        Q <= D;
    end
end
endmodule
```

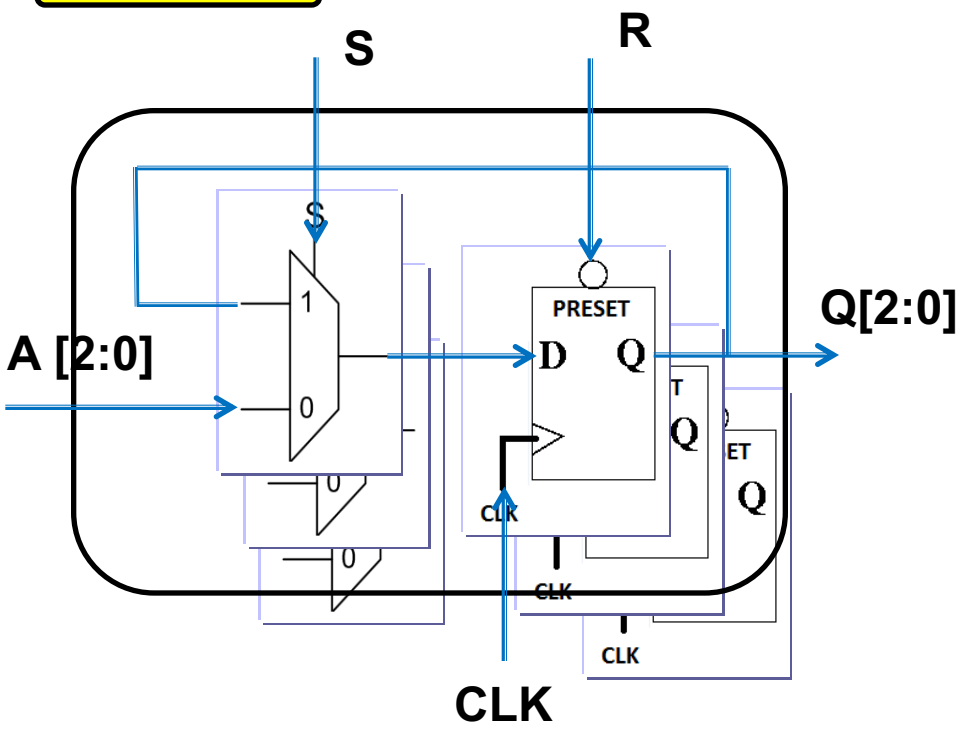
Why not declare the data type ?

Why declare “reg”data type ?

Why declare “type”data type ?

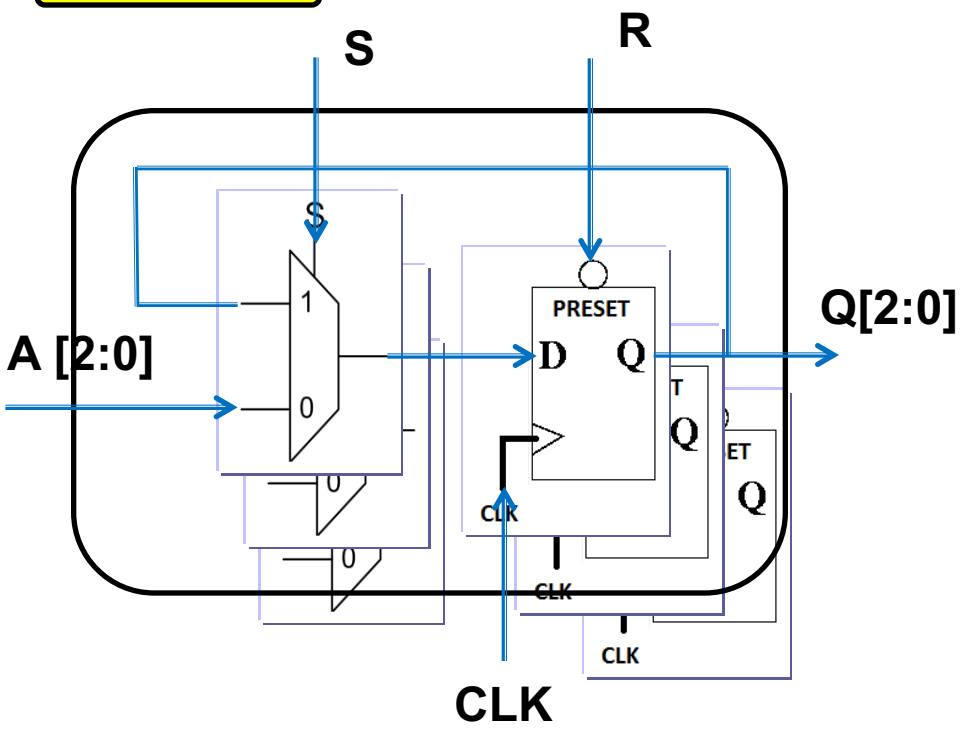
Verilog Basic

example1



Verilog Basic

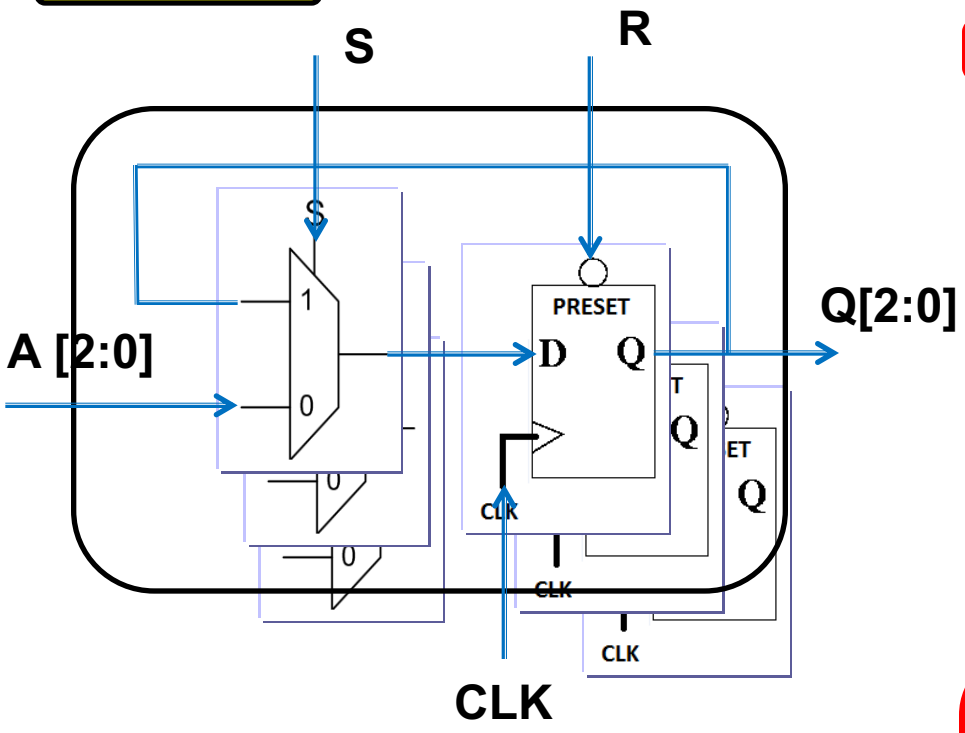
example1




```
module example1 (CLK, R, S, A, Q);

parameter DATA_WIDTH = 3;

input CLK;
input R;
input S;
input [DATA_WIDTH-1:0] A;

output [DATA_WIDTH-1:0] Q
reg    [DATA_WIDTH-1:0] Q;

wire [2:0] D;

assign D = (S==0)?A:Q;

always@(posedge CLK or negedge R) begin
   if(R == 0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end

endmodule
```

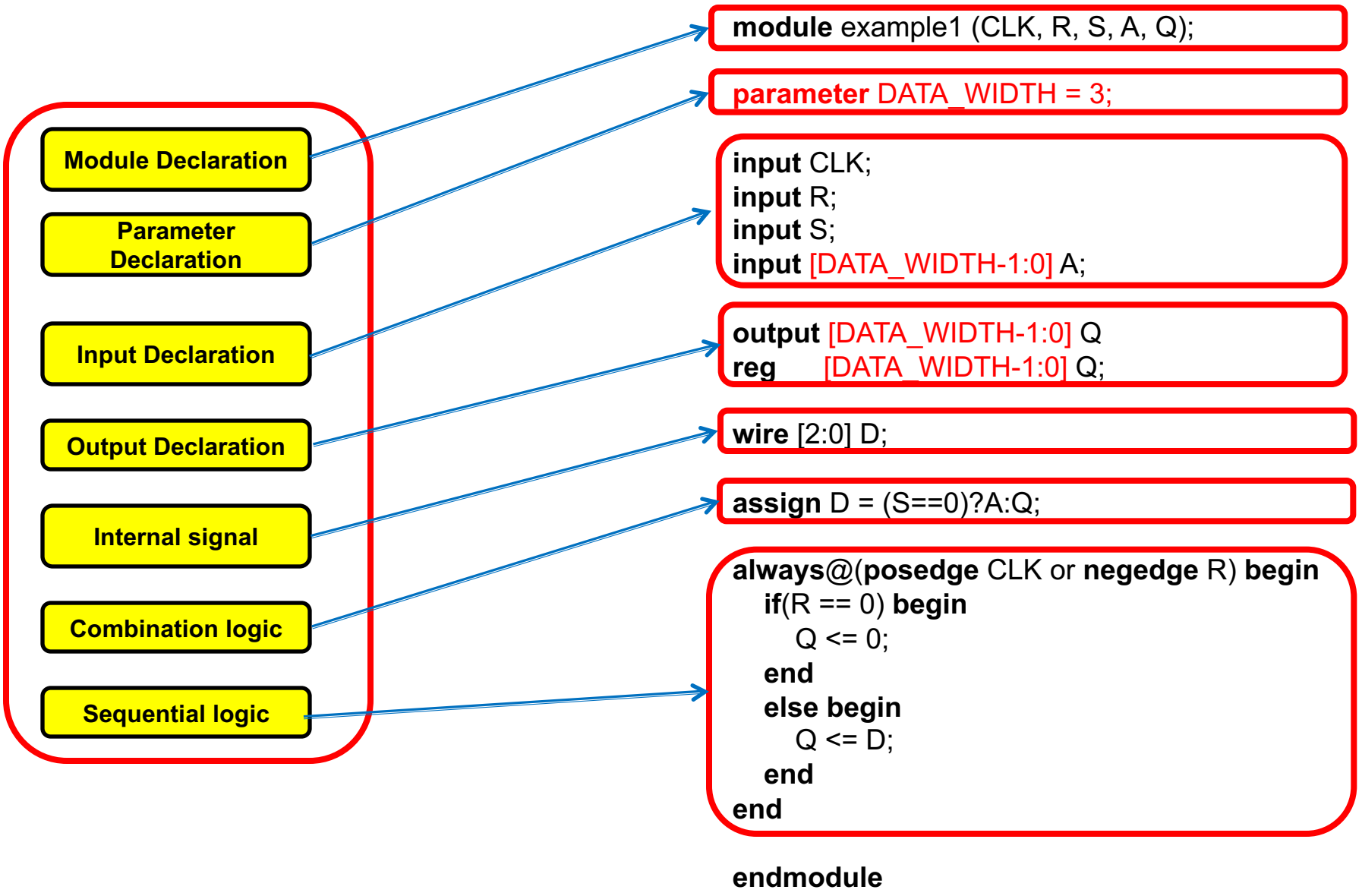
Verilog Basic

example1

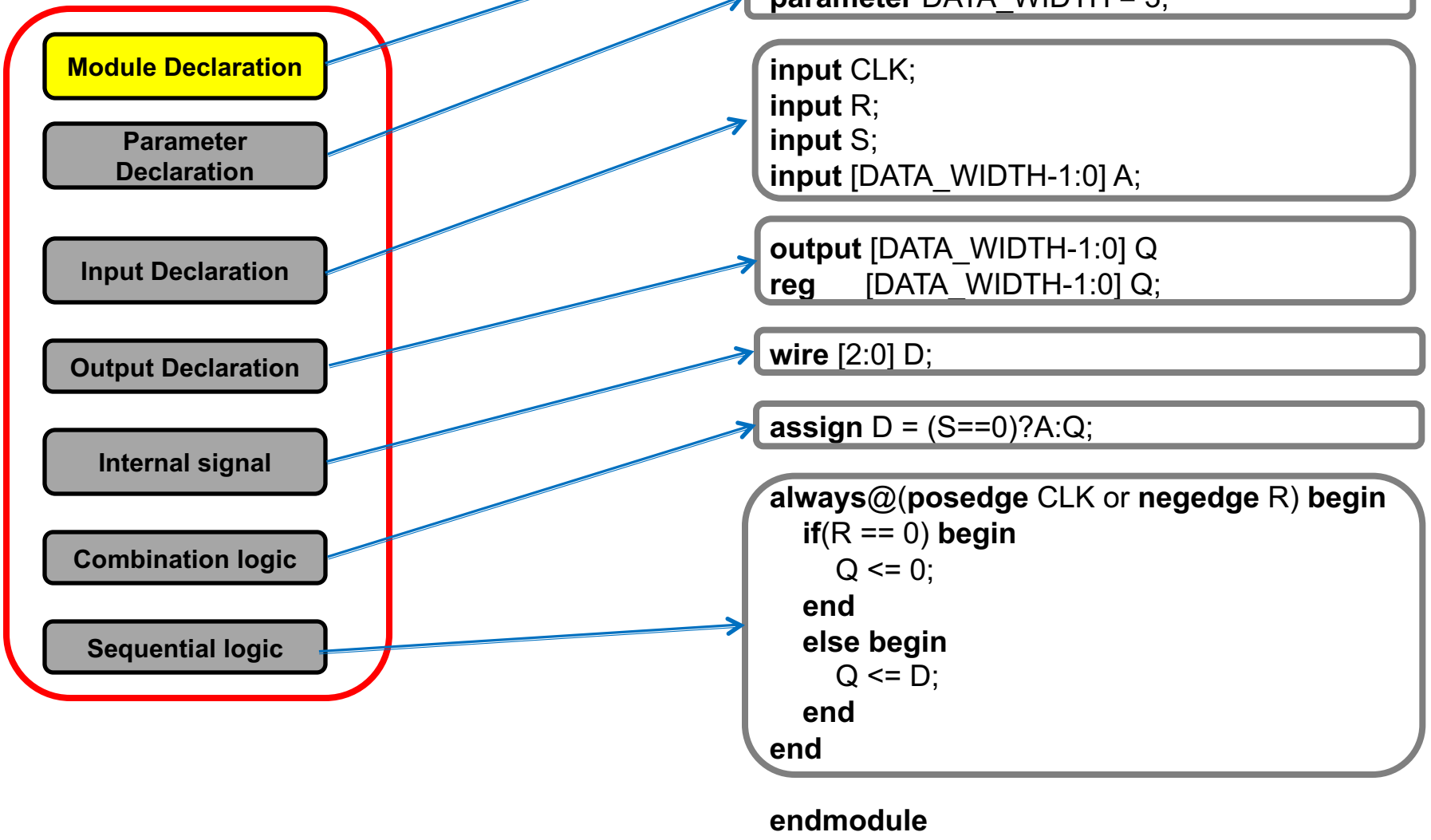



```
module example1 (CLK, R, S, A, Q);

parameter DATA_WIDTH = 3;

input CLK;
input R;
input S;
input [DATA_WIDTH-1:0] A;

output [DATA_WIDTH-1:0] Q
reg    [DATA_WIDTH-1:0] Q;

wire [2:0] D;

assign D = (S==0)?A:Q;

always@(posedge CLK or negedge R) begin
   if(R == 0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end

endmodule
```

Verilog Basic



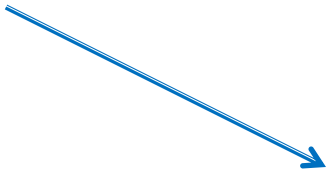
Verilog Basic



Verilog Basic

Module Declaration

module <module_name>(<input1>, <input_2>, <input_3>… <output_1>,.. <output_n>);



<module_name> should be same to file name
should be lower case characters

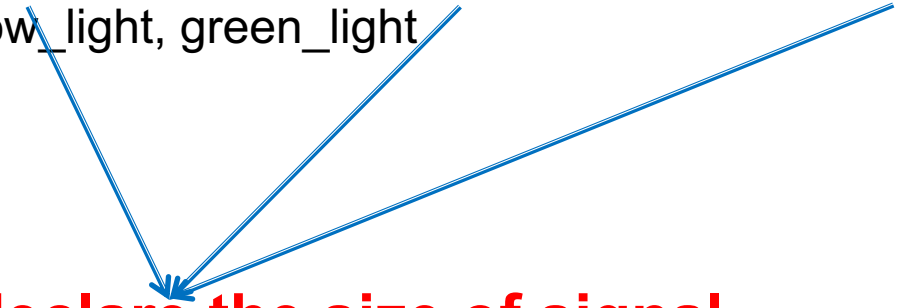
Verilog Basic

Module Declaration

module <module_name>(<input1>, <input_2>, <input_3>… <output_1>,.. <output_n>);

→ **Example:**

```
module traffic_light (  clk, rst_n, enable
                      , read_counter[2:0], yellow_counter[2:0], green_counter[2:0]
                      , red_light, yellow_light, green_light
                     );
```



Should not declare the size of signal in the module declaration

Verilog Basic

Module Declaration

module <module_name>(<input1>, <input_2>, <input_3>… <output_1>,.. <output_n>);

→ **Example:**

```
module traffic_light (  clk, rst_n, enable
                      , read_counter, yellow_counter, green_counter
                      , red_light, yellow_light, green_light
                     );
```

Only name of signal should be declared in the module declaration

Verilog Basic

Module Declaration

module <module_name>(<input1>, <input_2>, <input_3>… <output_1>,.. <output_n>);

→ **Example:**

```
module traffic_light (  clk, rst_n, enable
                      , read_counter, yellow_counter, green_counter
                      , red_light, yellow_light, green_light
                     );
```

- **<module name> and <port_name> are separated by lower & upper characters**
- **The <module name> and <port_name> should be used by reminiscent words**
- **The first input should be <clock> port**
- **The second input should be <reset> port**
- **The output port should be declared after input ports**

Verilog Basic

Module Declaration

module <module_name>(<input1>, <input_2>, <input_3>… <output_1>,.. <output_n>);

Q3?

```
module traffic_light (  clk, rst_n, enable
                      , read_counter, yellow_counter, green_counter
                      , red_light, yellow_light, green_light
                     );
```

New line & The comma is first

- <module name> and <port_name> are separated by lower & upper characters Q1?
- The <module name> and <port_name> should be used by reminiscent words Q2?
- The fist input should be <clock> port
- The second input should be <reset> port
- The output port should be declared after input ports

Verilog Basic

Character

- White space: Space, tab(\t), newline(\n) characters are ignored.

→ The space characters are members of string

- Comment:
  - + Line comment (//)
  - + Block comment (/* */)

→ Should not use the block comment

- Number: The constant number can be integer constant or real constant

→ Should be clear the type of number

- String : Is enclosed by double quotes (“”)

→ Example : “Hello World”

- Identifier: Use “$” character for identifiers.
  These can be system functions or system variables

→ Example: $time, $display

- Backslash (\): Use the define the escaped identifier

→ Example: “\$ ” is understand as “$ ” string
The identifier should be end with white space

Verilog Basic

Name Rule

- Use enough character for name (5 → 16)
- Use the underscore "_" to separate the name by meaning (not use at the end)
- Not use the same character with different lower & upper cases ("aaa" and "AAA")
- Do not mix the upper case and lower case (Dmac_Enable, Uart_Clock)
- Use reminiscent word for the name (dmac_enable, uart_tx)
- Use "_n" for the active low signal (dmac_rst_n, uart_rst_n)
- The clock should be start by "clk" character (clk_dmac, clk_uart)
- Do not use the name which is same with the keywords (vcc, vdd, clock, reset …)
- Use the upper cases for the parameter (parameter DATA_WIDTH = 32)
- Good example: clk_system, clk_sbi
  rst_n_timer, rst_n_system
  temp_01, temp_02
  rd_ena, wr_ena

Verilog Basic

Key Words

- <NETTYPE>
  wire, reg, wand, wor, supply0, supply1, tri, tri0, tri1, triand , trior, trireg
- <CAPACITOR_SIZE>
  small, medium, large
- <STRENGTH0>
  supply0, strong0, pull0, weak0, highz0
- <STRENGTH1>
  supply1, strong1, pull1, weak1, highz1
- <CAPACITOR_SIZE>
  small, medium, large
- <GATETYPE>
  and, nor, pullup, tran, buf, not, rcmos, tranif0, bufif0, notif0, rnmos, tranif1, bufif1, notif1 rpmos, xnor, cmos, or, rtran, xor, nand, pmos rtranif0, nmos, pulldown, rtranif1

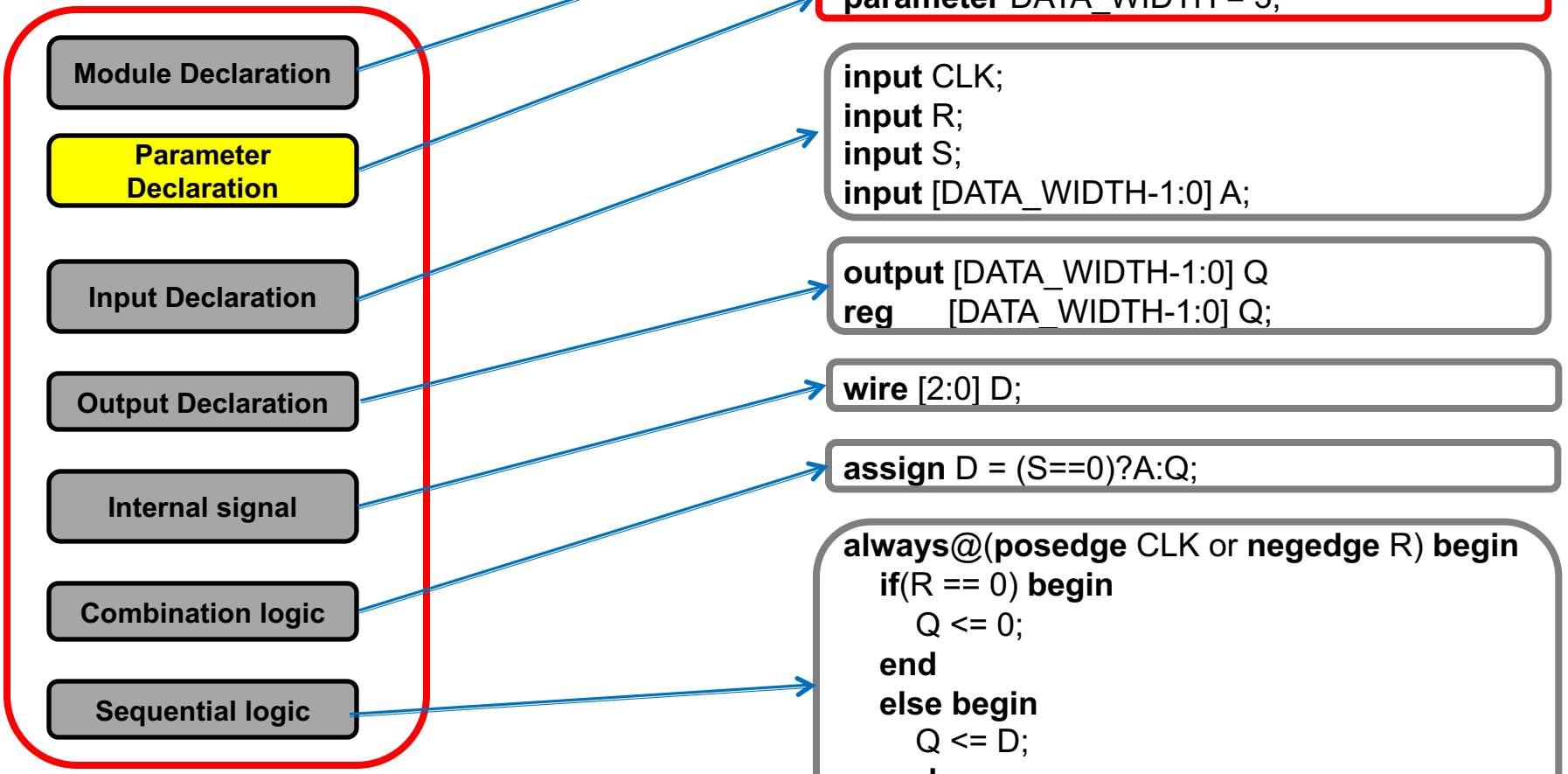
Verilog Basic

System Function/Variable

$display - Print to screen a line followed by an automatic newline.
$write - Write to screen a line without the newline.
$swrite - Print to variable a line without the newline.
$sscanf - Read from variable a format-specified string. (*Verilog-2001)
$fopen - Open a handle to a file (read or write)
$fdisplay - Write to file a line followed by an automatic newline.
$fwrite - Write to file a line without the newline.
$fscanf - Read from file a format-specified string. (*Verilog-2001)
$fclose - Close and release an open file handle.
$readmemh - Read hex file content into a memory array.
$readmemb - Read binary file content into a memory array.
$monitor - Print out all the listed variables when any change value.
$time - Value of current simulation time.
$dumpfile - Declare the VCD format output file name.
$dumpvars - Turn on and dump the variables.
$dumpports - Turn on and dump the variables in Extended-VCD format.
$random - Return a random value.

The system functions/variables are used to simulate
→ These are not synthesized

Verilog Basic



Verilog Basic

Parameter

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **Valid only in the module in which is declared**
- **Use for the bit/bus size, cycle, address, state name …**
- **How to define a parameter ?** **Q4?**
- **How to define a parameter outside of the module?** **Q5?**
- **How to separate between “define ” and “parameter”?** **Q6?**

Verilog Basic

Parameter

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter ?** **Q4?**

Example code

```
parameter DATA_WIDTH = 32;
parameter INITIAL = 2'b00;
parameter START  = 2'b01;
parameter BUS_ADDRESS = 8;
……….
reg [DATA_WIDTH-1:0] pci_data;
……….
assign pci_address = data[BUS_ADDRESS-1:0];
……….
INITIAL: begin
                if(pci_ena == 1) begin
                     state <= START;
                end
                else begin
                     state <= INITIAL;
                end
            end
```

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

```
module LEVER_A(….);
parameter WIDTH = 8;
…..
reg [WIDTH-1:0] level_a_temp;
```

8

```
module LEVER_B(….);
parameter WIDTH = 9;
…..
reg [WIDTH-1:0] level_b_temp;
```

9

The values of "WIDTH" are different in two independent modules

Verilog Basic

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

```
module LEVER_A(….);
parameter WIDTH = 8;
…..
reg [WIDTH-1:0] level_a_temp;
……
LEVEL_B inst_01 (…);
```

```
module LEVER_B(….);
parameter WIDTH = 9;
…..
reg [WIDTH-1:0] level_b_temp;
```

The values of "WIDTH" are different in the instance module

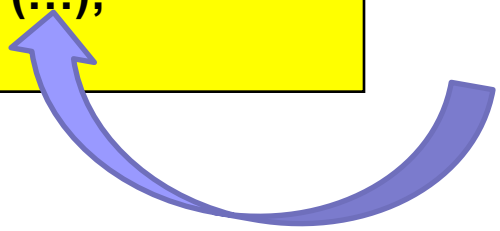
parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

```
module LEVER_A(….);
defparam inst_01.WIDTH = 8;
parameter WIDTH = 8;

…..
reg [WIDTH-1:0] level_a_temp;
……
LEVEL_B inst_01 (…);
```

The value of "WIDTH" in the LEVEL_B instance is redefined to 8



```
module LEVER_B(….);
parameter WIDTH = 9;
…..
reg [WIDTH-1:0] level_b_temp;
```

Verilog Basic

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

```
module LEVER_A(….);
defparam inst_01.WIDTH = 8;
parameter WIDTH = 8;

…..
reg [WIDTH-1:0] level_a_temp;
……
LEVEL_B inst_01 (…);
LEVEL_B inst_02 (…);
```

The value of "WIDTH" in the Inst_01 of LEVEL_B module is redefined to 8 but it is still 9 in The inst_02 of LEVEL_B module

```
module LEVER_B(….);
parameter WIDTH = 9;
…..
reg [WIDTH-1:0] level_b_temp;
```

Verilog Basic

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

```
module LEVER_A(….);
parameter WIDTH = 8;
…..
reg [WIDTH-1:0] level_a_temp;
```

8

```
module LEVER_B(….);
parameter WIDTH = 8;
…..
reg [WIDTH-1:0] level_b_temp;
```

8

If using same parameter, the same parameters can list in the parameter file

Verilog Basic

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to define a parameter outside of the module?** **Q5?**

Parameter_dev.v

```
parameter WIDTH = 8;
parameter ADRESS =  32;
…
```

```
module LEVER_A(….);
`include “parameter_def.v”
…..
reg [WIDTH-1:0] level_a_temp;
```

If using same parameters, these can be declared in the parameter file

→ The “parameter” is LOCAL

Verilog Basic

Parameter

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to differentiate “define ” and “parameter”?** **Q6?**

```
module A(…);
`define  ALW_PRT always@*begin
…
ALW_PRT
…..
```

```
module A(….);
…
….
always@*begin
…..
```

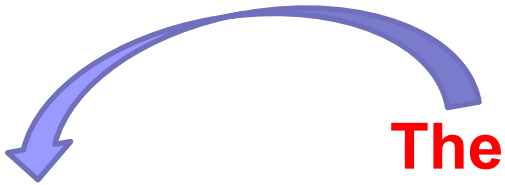
Use “define” to replace a string/character …

parameter <PARAMETER_NAME> = <value>;

parameter DATA_WIDTH = 3;

- **How to differentiate "define " and "parameter"?** **Q6?**

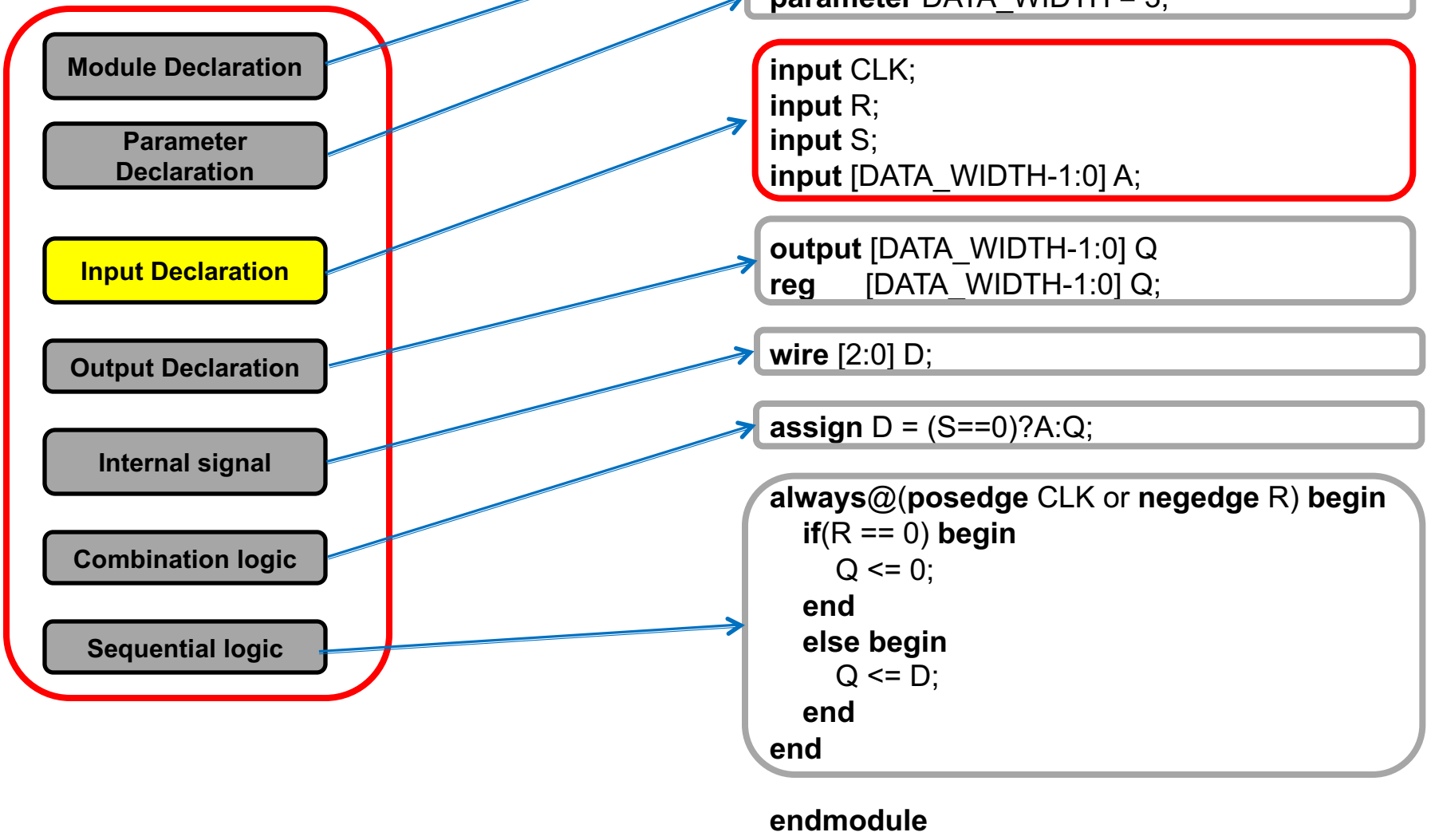
```
module A (…);
`define TEMP_ONE 3'b111
…..
endmodule

module B (…);
`define TEMP_ONE 3'b110
….
endmodule
```



The "TEMP_ONE" is updated after this code line
→ Same as "macro" concept in C++

→ The "define" is GLOBAL

Verilog Basic



Verilog Basic

Input Declaration

input <[side of signal]> <input name> ;

Input [7:0] data_01, data_02;
input clk, rst_n;
wire clk, rst_n;
wire [7:0] data_01, data_02;

- **Should declare the clk signal and reset signal firstly**
- **Should not declare many input signals in the same lines**
- **The default type of input is "wire"**

→ Should not declared "wire" data type

Verilog Basic

Input Declaration

input <[side of signal]> <input name> ;

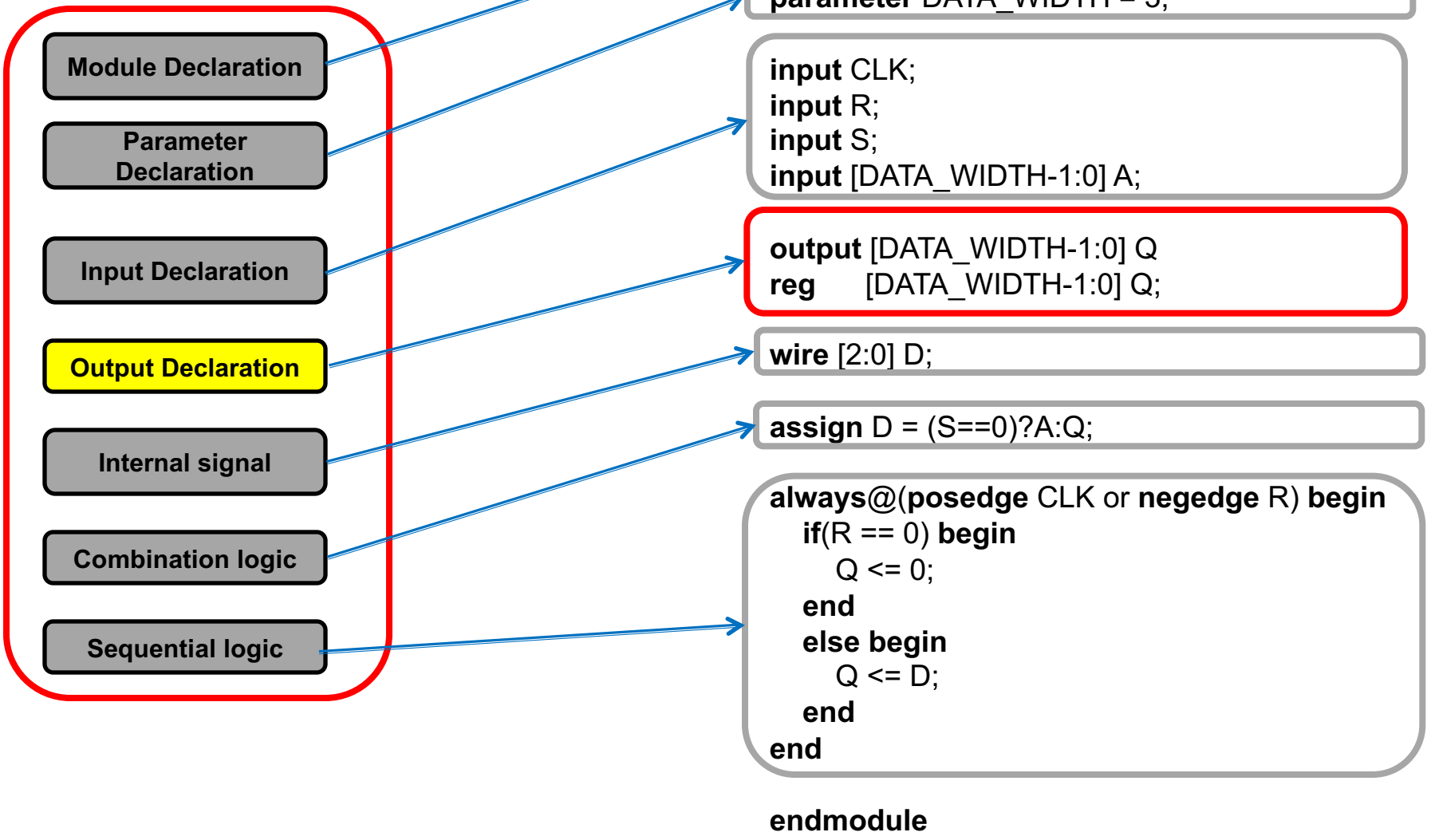
input [7:0] data_01, data_02;
input clk, rst_n;
wire clk, rst_n;
wire [7:0] data_01, data_02;

correct

→ **Should not declared "wire" data type for input**

input clk;
input rst_n
input [7:0] data_01;
input [7:0] data_02;

Verilog Basic



Verilog Basic

Input Declaration

output<[side of signal]> <input name> ;

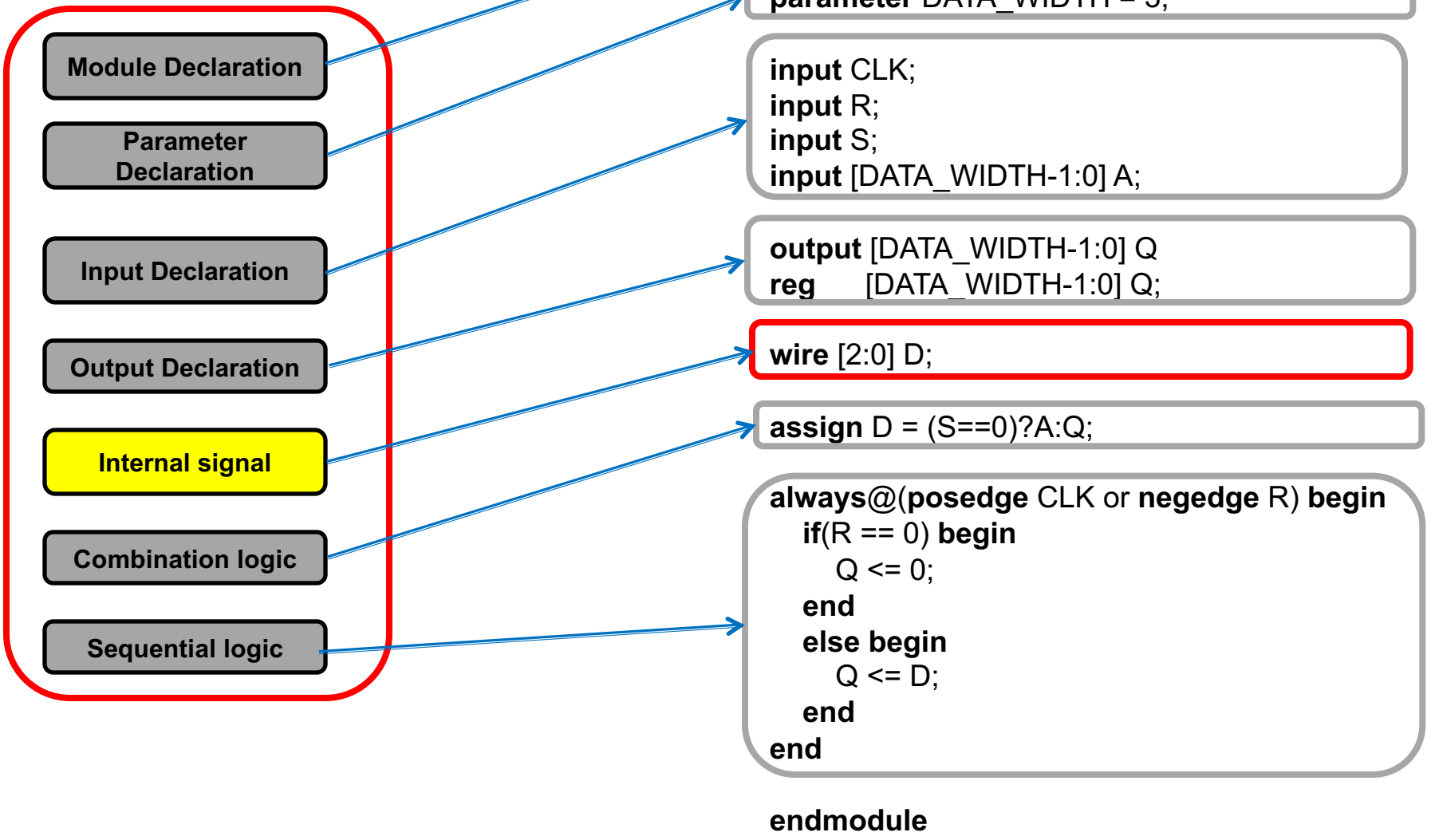
output [4:0] data_out;
wire[4:0] data_out

correct → Should not declared “wire” data type

output [4:0] data_out;

- **Question: When decalaring “reg” type to output ports ?**
  → **“reg” type is used whenever the signal is assigned in “always” prototype**

Verilog Basic



Verilog Basic

Internal Signal

wire<[side of signal]> <input name> ;
reg <[side of signal]> <input name> ;

reg [4:0] out_multiplier;
wire[4:0] out_adder;

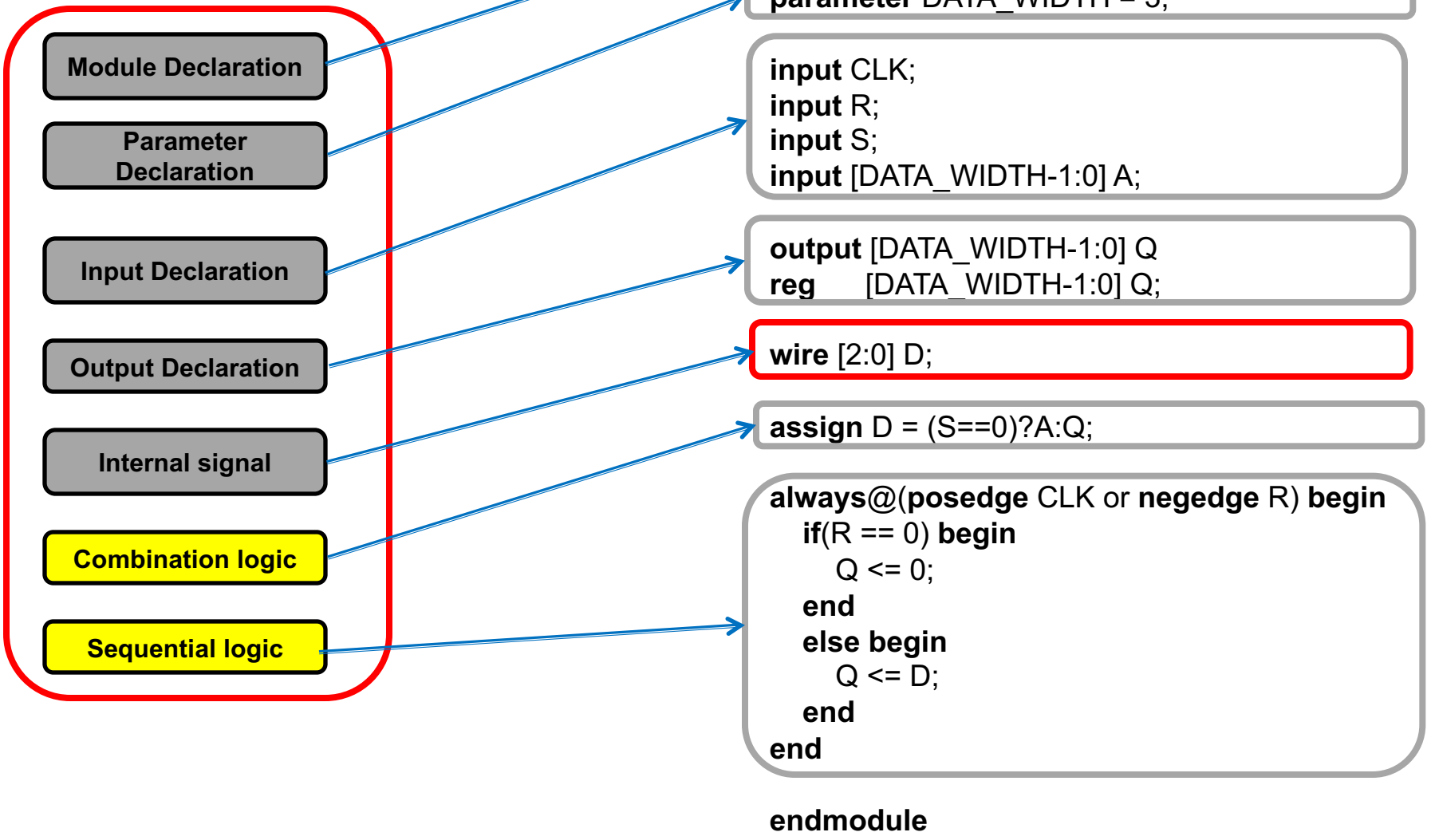
- **Question: When declaring "reg/type" type to internal nets ?**

→ "reg" type is used whenever the output port is assigned in "always" prototype.

→ "wire" type is used whenever the output port is assigned in "assign" prototype.

Verilog Basic

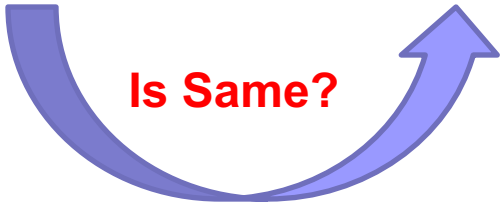
Main Structures



Verilog Basic

Rules

```
module abc (a, b);
input a;
output [3:0]b;
wire a;
reg [3:0]b;
always @(a) begin
   if (a==1'b1) b <= 4'b0011;
   else         b <= 4'b1010;
end
endmodule
```

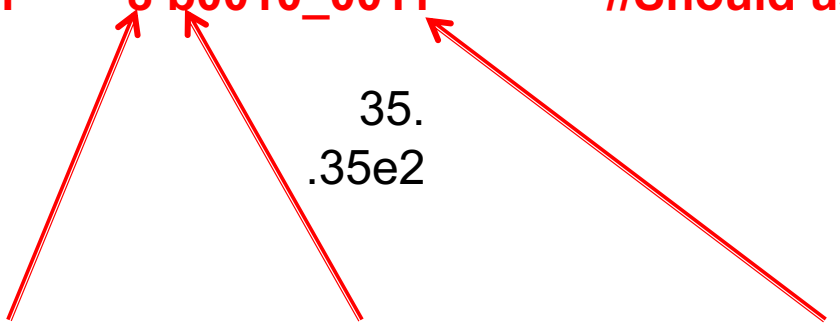



```
module abc (
          a,
          b
          );
input a; output [3:0]b ;
wire a;
reg [3:0]b;
always @(a)
begin
   if (a==1'b1)
      b <= 4'b0011;
   else
      b <= 4'b1010;
end
endmodule
```

Verilog Basic

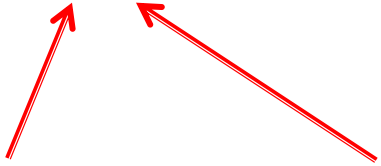
Number

Decimal number	35	
	8’**d**35	//10*3+5
Octal number	8’**o**043	//64*0+8*4+3
Hex number	8’h23	//16*2+3
Binary number	**8’b0010_0011**	**//Should use**
Real number	35.	
	.35e2	



Bit width Base number Value

4’sb1001 : Sized bit is 1 → negative number



Size format Signed bit

- Logic vs. Arithmetic
- Synchronous Reset vs. Asynchronous Reset
- None-Blocking vs. Blocking
- Use enough cases towards Case/If
- Limitation of Hierarchies
- How many levels when using Case/If
- How to connect many modules

Logic vs. Arithmetic

```
wire [2:0] a;
wire [2:0] b;
wire [2:0] c1; //Test logic
wire [2:0] c2; //test arithmetic

assign c1 = a && b;  // AND LOGIC
assign c2 = a & b; // AND BIT
```

→ **Is c1 and c2 similarly when a = 3'b101 and b = 3'b110**

Logic vs. Arithmetic

```
wire [2:0] a;
wire [2:0] b;
wire [2:0] c1; //Test logic
wire [2:0] c2; //test arithmetic

assign c1 = a && b;  // AND LOGIC
assign c2 = a & b; // AND BIT
```

a = 3'b101 and b = 3'b110

Result : **c1 = 3'b001;** // Ignore the c1[2] and c1[1]; c1[0] is 1 because a != 0 and b != 0

c2 = 3'b100; // c2[2] = a[2] && b[2]; …; a[0] && b[0];

Logic vs. Arithmetic

```
wire [2:0] a;
wire [1:0] b; // b is only two bits
wire [2:0] c1; //Test logic
wire [2:0] c2; //test arithmetic

assign c1 = a && b;  // AND LOGIC
assign c2 = a & b; // AND BIT
```

a = 3'b101 and b = 2'b10

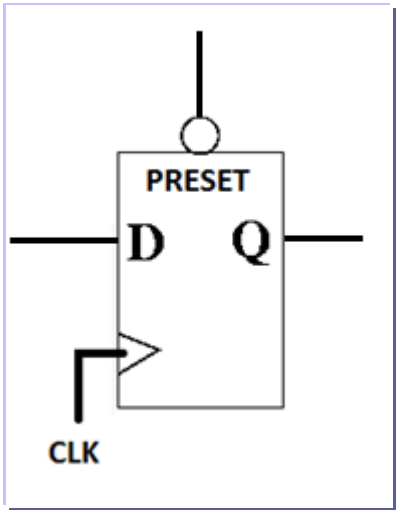
Result : **c1 = 3'b001;** // Ignore the c1[2] and c1[1]; c1[0] is 1 because a != 0 and b != 0

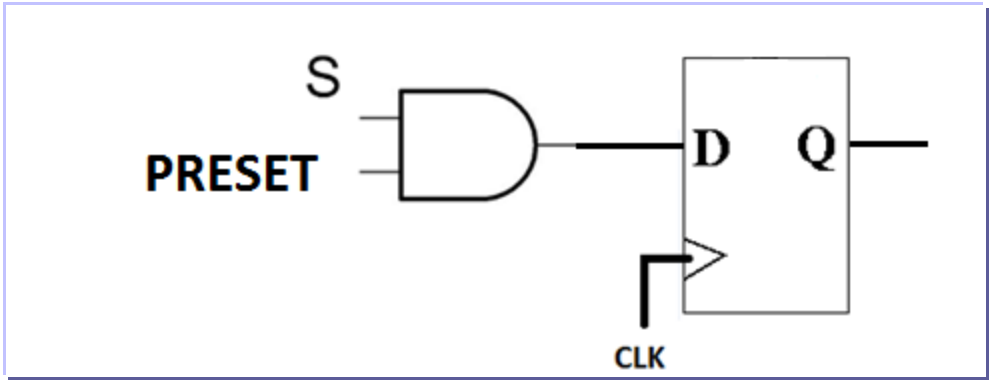
c2 = 3'bX00; // c2[2] is ignored; c2[1] = a[1] && b[1]; c2[0] = a[0] && b[0];

Logic vs. Arithmetic

Name	Test book style	Gate expression (MIL Logical notation)	Operation
AND	Y = AB		Y is 1 only when both A and B are 1
OR	Y = A + B		Y is 1 only when A or B is 1
NOT	$Y = \bar{A}$		Y is 1 only when A is 0
EOR	$Y = \bar{A}B + A\bar{B}$		Y is 1 only either one of A or B is 1
NAND	$Y = \overline{AB}$		Y is 1 only when A or B is 0
NOR	$Y = \overline{A + B}$		Y is 1 only when both A and B are 0

Synchronous Reset vs. Asynchronous Reset





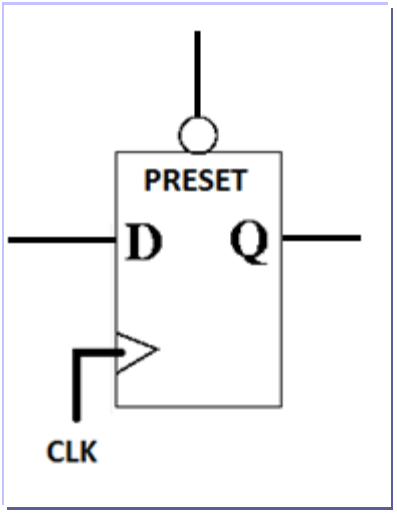
Asynchronous

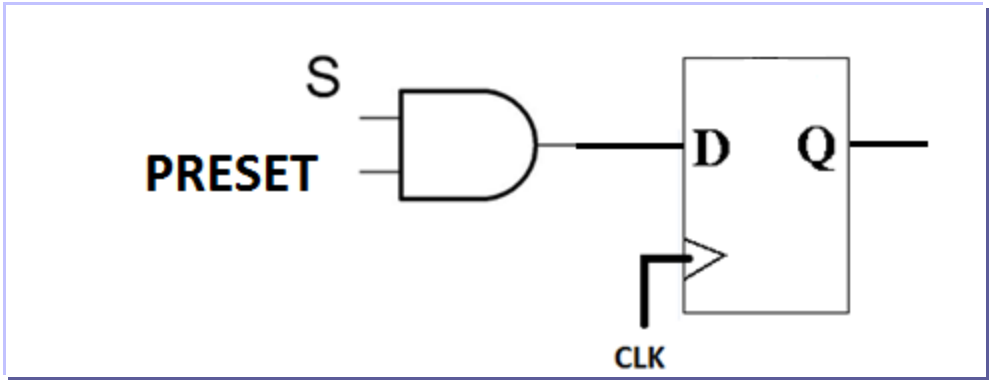
Synchronous

- Synchronous reset signal is similar to other signal as S, D or Q
- Asynchronous reset signal is special signal as CLK

Q7?

Synchronous Reset vs. Asynchronous Reset



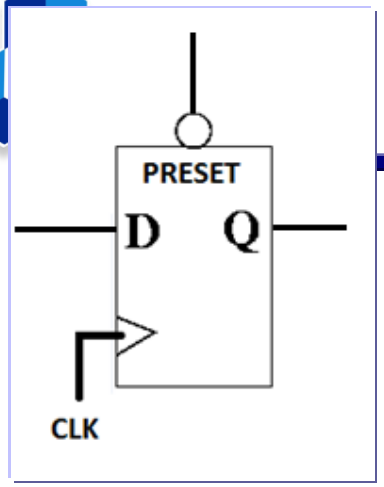


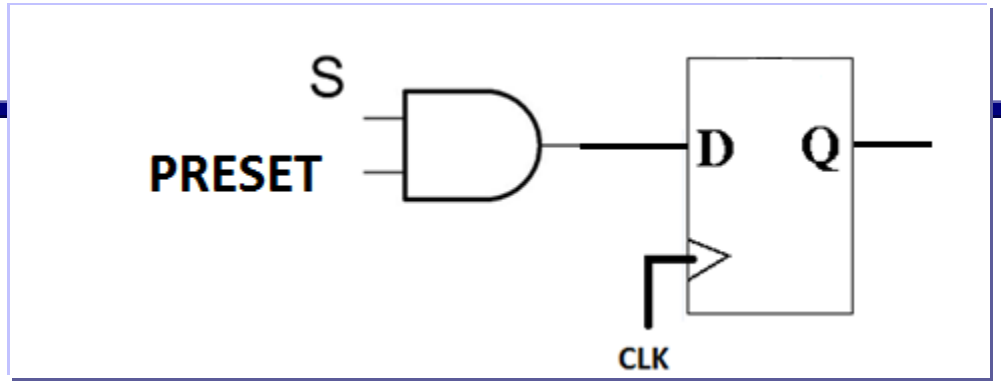
Asynchronous

```
always@(posedge CLK or negedge PRESET) begin
   if(PRESET == 0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end
```

Synchronous

```
assign D = S & PRESET;

always@(posedge CLK) begin
   Q <= D;
end
```





Asynchronous

```
always@(posedge CLK or negedge PRESET) begin
   if(PRESET == 0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end
```

```
process (CLK, PRESET) begin
 if (PRESET = '0') then
    Q <= '0';
elsif (rising_edge(CLK)) then
    Q <= D;
end if;
end process;
```

Synchronous

```
assign D = S & PRESET;

always@(posedge CLK) begin
   Q <= D;
end
```

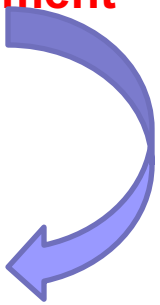
```
D <= S and PRESET;
process (CLK) begin
 if (PRESET = '0') then
    Q <= '0';
elsif (rising_edge(CLK)) then
    Q <= D;
end if;
end process;
```

Blocking vs. Non-Bloking

Non-Blocking : Only apply to ‘always’ prototype (<=)

Blocking : Apply for both “assign” and “always” prototype(=)

```
assign D = S & PRESET; // Blocking assignment

always@(posedge CLK) begin
    Q <= D;                          // Non- Blocking assignment
 end
```



No error,
Same function,
Same Hardware
→ ??? Difference

```
always@(posedge CLK) begin
    Q = D;                           // Blocking assignment
 end
```

Blocking vs. Non-Bloking

Non-Blocking : Only apply to ‘always’ prototype (<=)

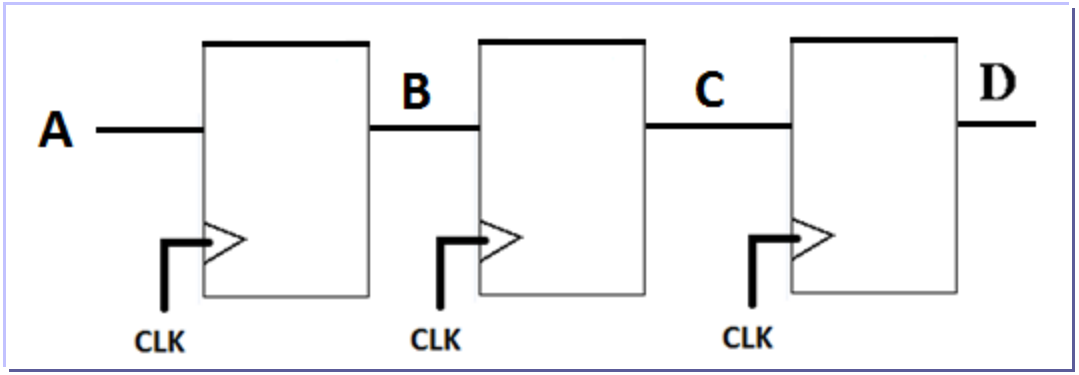
Blocking : Apply for both “assign” and “always” prototype(<=)

```
always@(posedge CLK) begin
    B <= A;                          // Non-Blocking assignment
    C <= B;
    D <= C;
 end

always@(posedge CLK) begin
    B = A;                           // Blocking assignment
    C = B;
    D = C;
end
```

Blocking vs. Non-Bloking

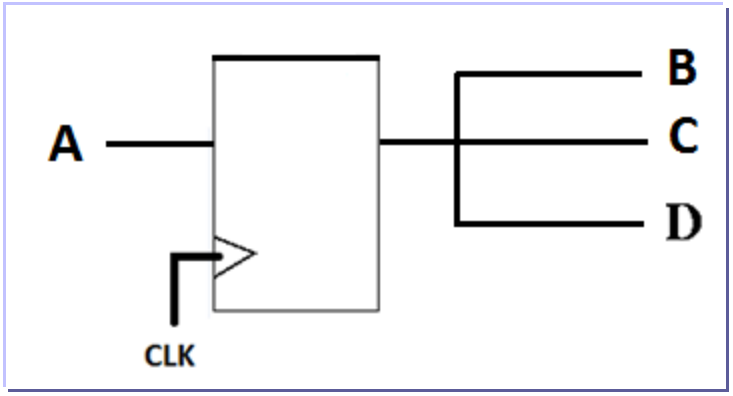
```
always@(posedge CLK) begin
   B <= A;
   C <= B;
   D <= C;
end
```

// Non-Blocking assignment




```
always@(posedge CLK) begin
   B = A;
   C = B;
   D = C;
end
```

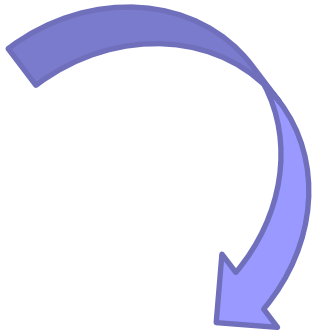
// Blocking assignment



Use K-Map to optimize function

Fundamentals

X[2]	X[1]	X[0]	Y[1]	Y[0]
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
--------	---------	--------	0	0



K-Map
→ Do not care how many cases

Y[1] = (!X[2] && X[0]) | (!X[2] && X[1])
Y[0] = !X[2] && (!X[0])

Use enough cases

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    if (X == 3'b000) begin
        Y <= 2'b01;
    end
    else if (X == 3'b001) begin
        Y <= 2'b10;
    end
    else if (X == 3'b010) begin
        Y <= 2'b11;
    end
    else begin
        Y <= 2'b00;
    end
end
```

'else' is for other cases to cover all cases of X[2:0]

Use enough cases

X[2]	X[1]	X[0]	Y[1:0]
0	0	0	01
0	0	1	10
0	1	0	11
0	1	1	00
--------	---------	--------	00

```
always@(X) begin
    case (X)
        3'b000: begin
            Y <= 2'b01;
        end
        3'b001: begin
            Y <= 2'b10;
        end
        3'b010: begin
            Y <= 2'b11;
        end
        default : begin
            Y <= 2'b00;
        end
    endcase
end
```

'default' is for other cases to cover all cases of X[2:0]

Use enough cases

Fundamentals

```
assign Y[1] = (!X[2] && X[0]) | (!X[2] && X[1])
assign Y[0] = !X[2] && (!X[0])
```

VS.

```
always@(X) begin
    case (X)
        3'b000: begin
            Y <= 2'b01;
        end
        3'b001: begin
            Y <= 2'b10;
        end
        3'b010: begin
            Y <= 2'b11;
        end
        default : begin
            Y <= 2'b00;
        end
    endcase
end
```

Limitation of hierarchies


```
always@(X) begin
    case (X)
        3'b000: begin
            case (Z)
                3'b000: begin
                    Y <= 2'b01;
                end
                ..............
            endcase
        end
        ..............
        default : begin
            Y <= 2'b00;
        end
    endcase
end
```

4 space bar after 'begin'

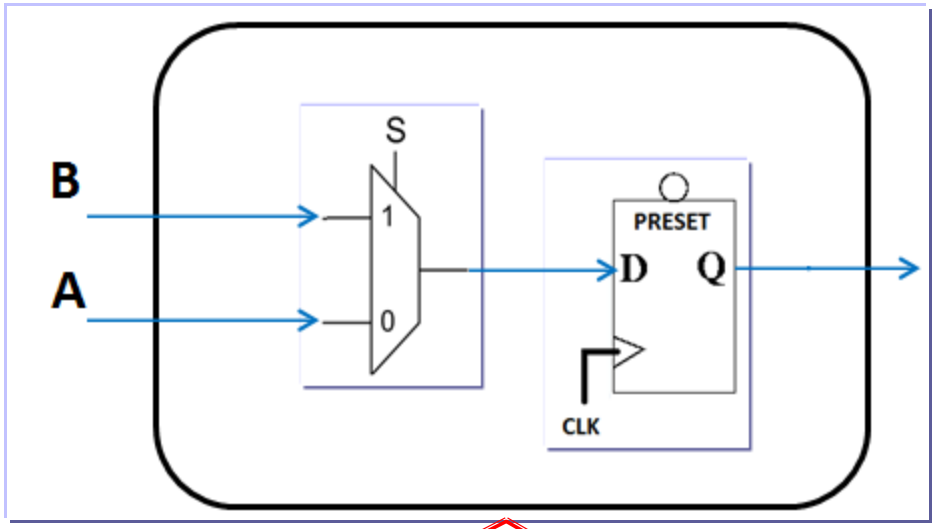
// Level 1 of case

// Level 2 of case

// How many levels ?

Maximum three levels are accepted
+ If
+ Case
+ If & Case together

Use “always”

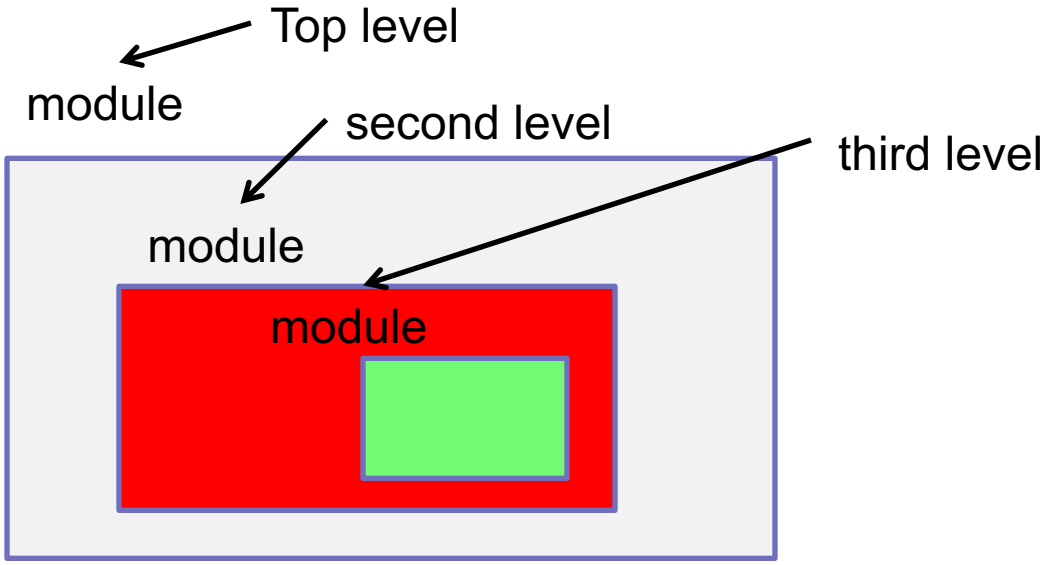


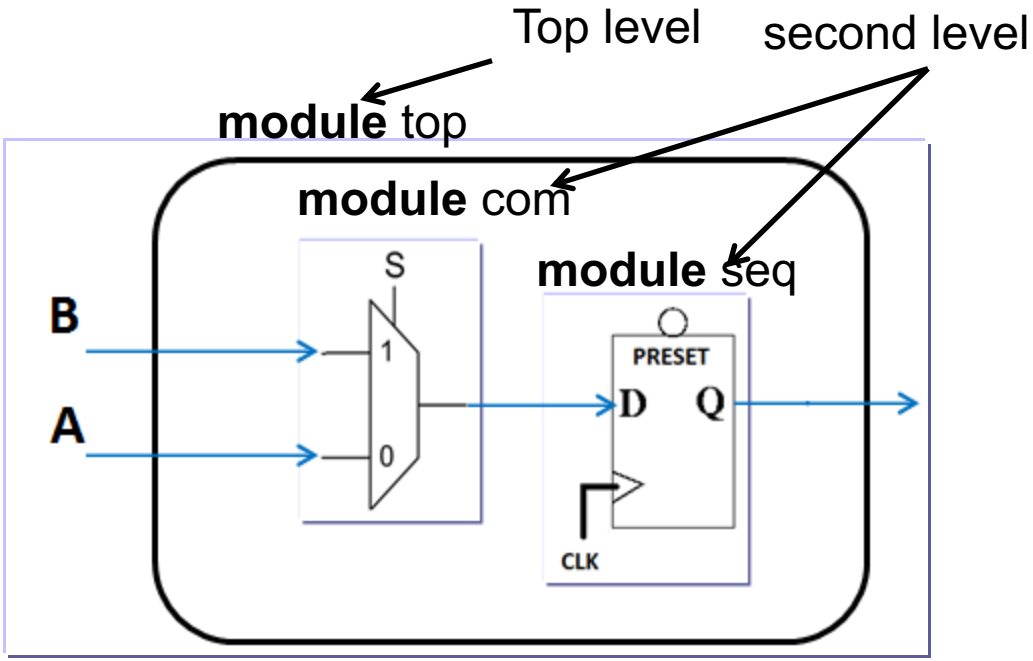

```
always( S or A or D) begin
   if(S == 1'b0) begin
      D = A;
   end
   else
       D = B;
   end
end

always@(posedge CLK or negedge R) begin
   if(R == 0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end
```

```
always@(posedge CLK or negedge R) begin
   if(R == 0) begin
      Q <= 0;
   end
   else if (S == 1'b0) begin
      Q <= A;
   end
   else
       Q <= B;
   end
end
```

Should not use ‘if’/’case’ in ‘always’ if it is too complicated

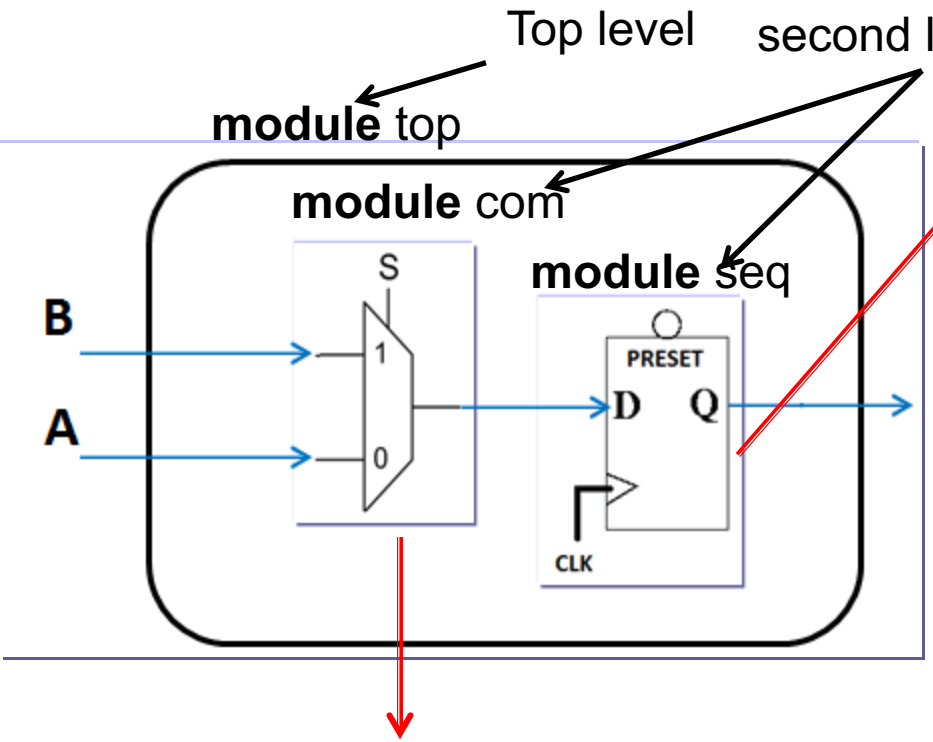
Connect modules





Connect modules

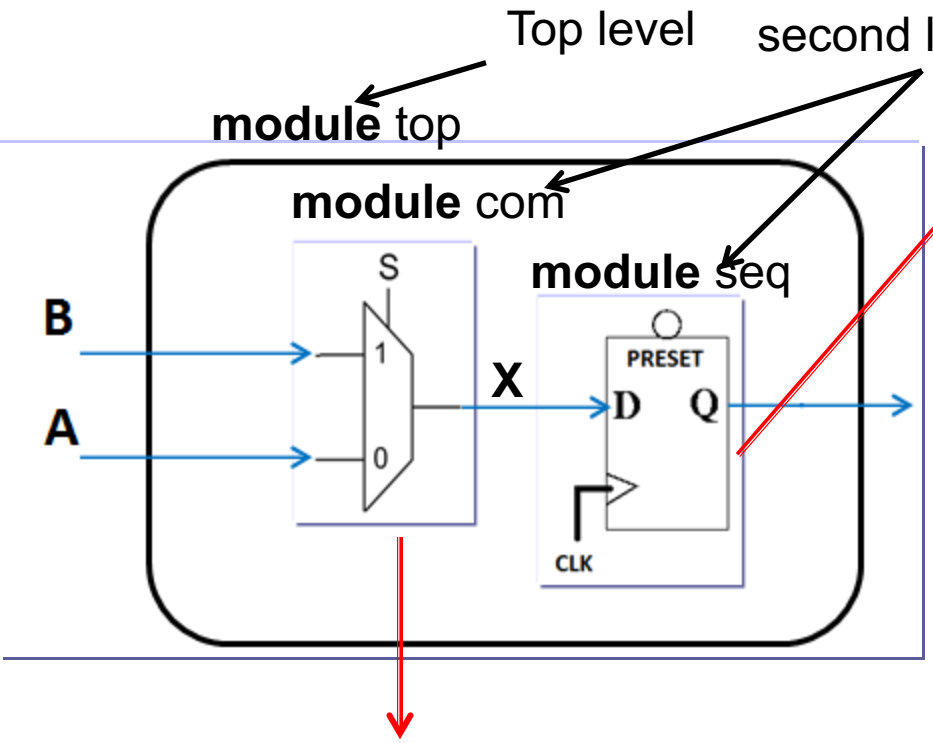
Fundamentals




```
module com (A, B, S, O);
input A;
input B;
input S;
output O;
wire O;
assign O = (S==1'b0)?A:B;
endmodule
```

```
module seq (CLK, R, D, Q);
input CLK;
input R;
input D;
output Q;
reg Q;
always@(posedge CLK or negedge R) begin
   if(R == 1'b0) begin
      Q <= 0;
   end
   else begin
      Q <= D;
   end
end

endmodule
```

Connect modules



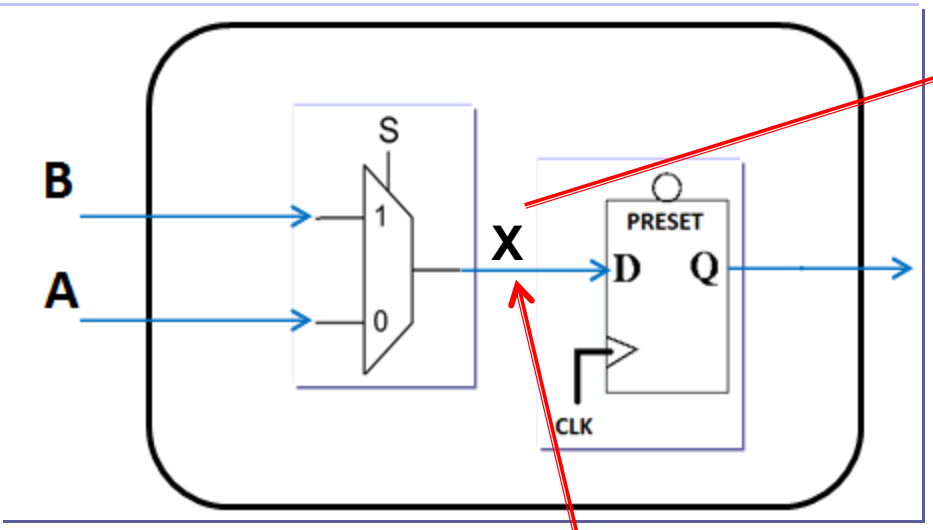

```
module com (A, B, S, O);
……..
endmodule
```

```
module seq (CLK, R, D, Q);
……..
endmodule
```

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named ‘com_01’
com com_01 ( .A(A), B(B), .S(S), .O(X) );

// Call module seq named : ‘seq_01’
seq seq_01 ( .CLK(CLK), .R(R), .D(X), .Q(Q) );

endmodule
```

Connect modules

Fundamentals

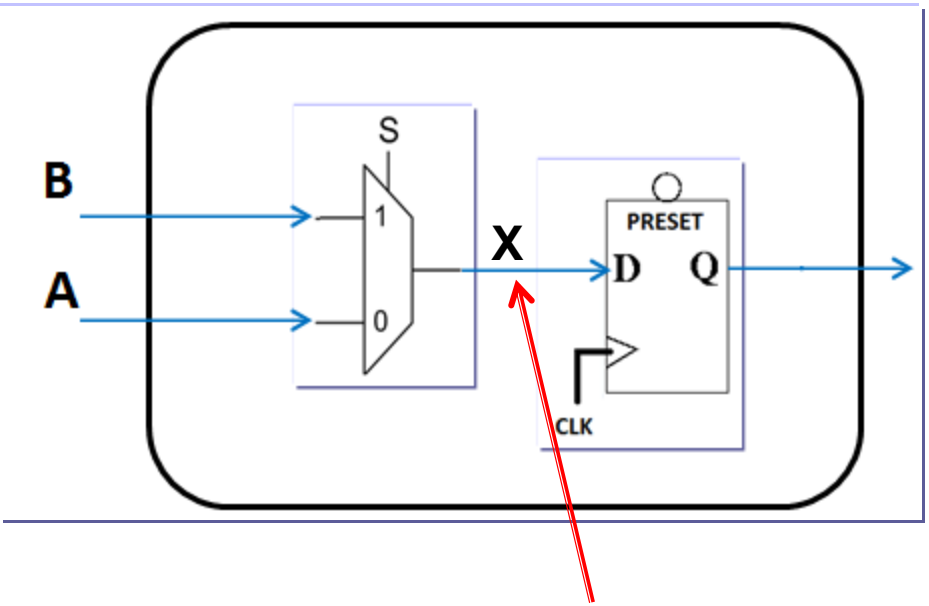



```
module com (A, B, S, O);
……..
endmodule
```

```
module seq (CLK, R, D, Q);
……..
endmodule
```

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named 'com_01'
com com_01 ( .A(A), B(B), .S(S), O(X) );

// Call module seq named : 'seq_01'
seq seq_01 ( .CLK(CLK), .R(R), D(X), .Q(Q) );

endmodule
```

Connect modules




```
module com (A, B, S, O);
……..
endmodule
```

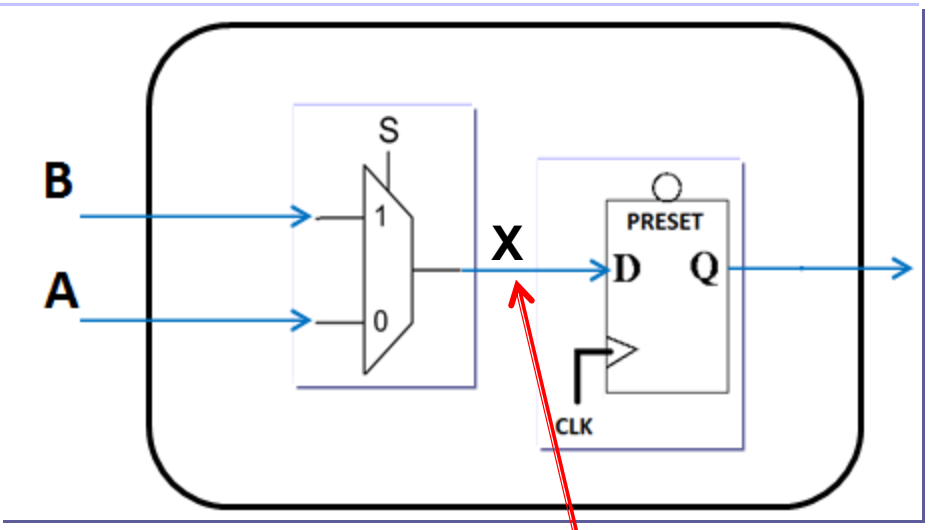
```
module seq (CLK, R, D, Q);
……..
endmodule
```

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named ‘com_01’
com com_01 ( .A(A), .B(B), .S(S), .O(X) );

// Call module seq named : ‘seq_01’
seq seq_01 ( .CLK(CLK), .R(R), .D(X), .Q(Q) );

endmodule
```

Second level

Top level

Connect modules




```
module com (A, B, S, O);
……..
endmodule
```

```
module seq (CLK, R, D, Q);
……..
endmodule
```

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named ‘com_01’
com com_01 ( .A(A), .B(B), .S(S), .O(X) );

// Call module seq named : ‘seq_01’
seq seq_01 ( .CLK(CLK), .R(R), .D(X), .Q(Q) );

endmodule
```

Declare wire due to connection instead of from “always”

Connect modules

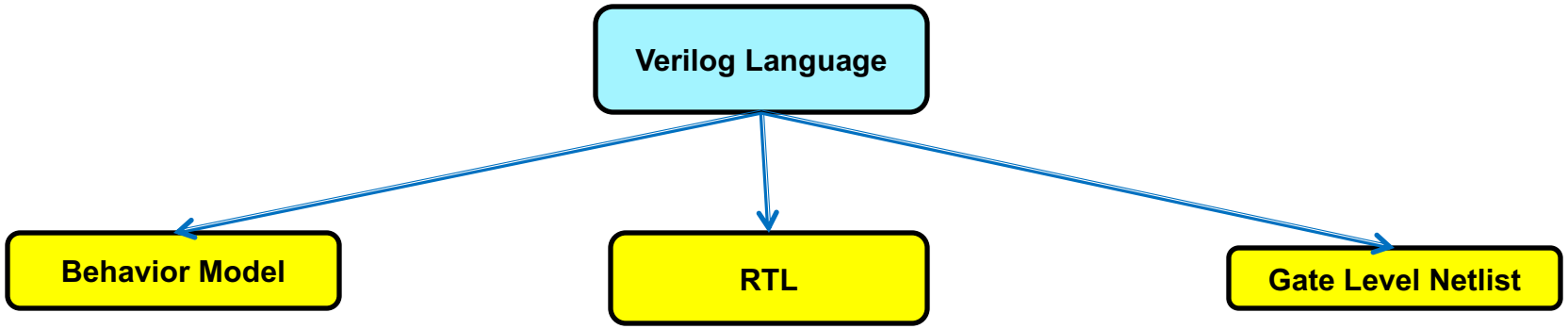

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named 'com_01'
com com_01 ( .A(A), .B(B), .S(S), .O(X) );

// Call module seq named : 'seq_01'
seq seq_01 ( .CLK(CLK), .R(R), .D(X), .Q(Q) );

endmodule
```

Name Assignment

```
module top (CLK, R, A, B, S, Q);

input CLK;
input R;
input A;
input B;
input S;

output Q;
wire Q;

wire X;
// Call module com named 'com_01'
com com_01 ( A, B, S, X);

// Call module seq named : 'seq_01'
seq seq_01 (CLK, R, X, Q);

endmodule
```

Order Assignment

Gate Level Netlist



+ Use any things, but not commit syntax error

+ Strict Rules
+ Only use simple structures
+ Do not use ‘for, while,…’ of complex structures

+ Declare gate level netlist in detail
+ Have other kind of writing Verilog called Verilog Primitives

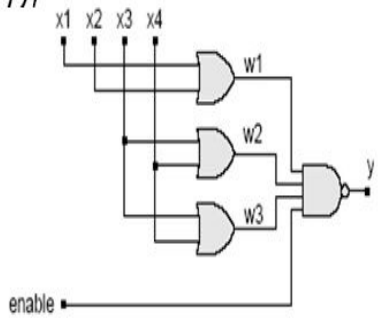
Gate Level Netlist

Verilog: Primitives Format

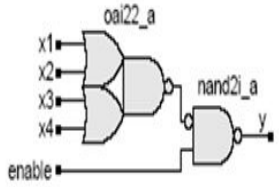
```
primitive multiplexer(mux, control, dataA, dataB )
;
output mux ;
input control, dataA, dataB ;
table
0 1 0 : 1 ;
0 1 1 : 1 ;
0 1 x : 1 ;
0 0 0 : 0 ;
0 0 1 : 0 ;
0 0 x : 0 ;
1 0 1 : 1 ;
1 1 1 : 1 ;
1 x 1 : 1 ;
1 0 0 : 0 ;
1 1 0 : 0 ;
1 x 0 : 0 ;
x 0 0 : 0 ;
x 1 1 : 1 ;
endtable
endprimitive
```

Verilog : Gate Level

```
module or_nand_1 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  wire w1, w2, w3;
  or (w1, x1, x2);
  or (w2, x3, x4);
  or (w3, x3, x4); // redundant
  nand (y, w1, w2, w3, enable);
endmodule
```



Pre-synthesis



Q & A