

Chương 2

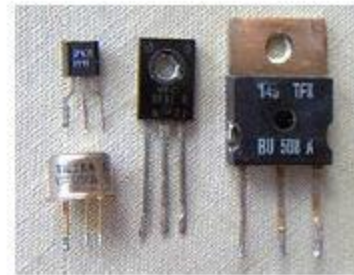
NGÔN NGỮ VERILOG

I. GIỚI THIỆU

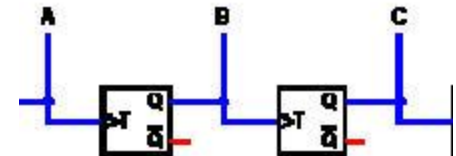
1.1. Ngôn ngữ mô tả phần cứng HDL (Hardware Description Language)

- Thiết kế mạch số (1950's, 1980's): vẽ mạch schematic -> lựa chọn linh kiện -> thi công.
- Mạch schematic gồm có:
 - Phần tử (component): Cổng (Gate), Điện trở, (LEDs, LCD) Chips,...
 - Dây kết nối các phần tử
 - Input, Output -> xem 1 mạch schematic như 1 phần tử -> kết nối phân cấp.

1950's
Idea



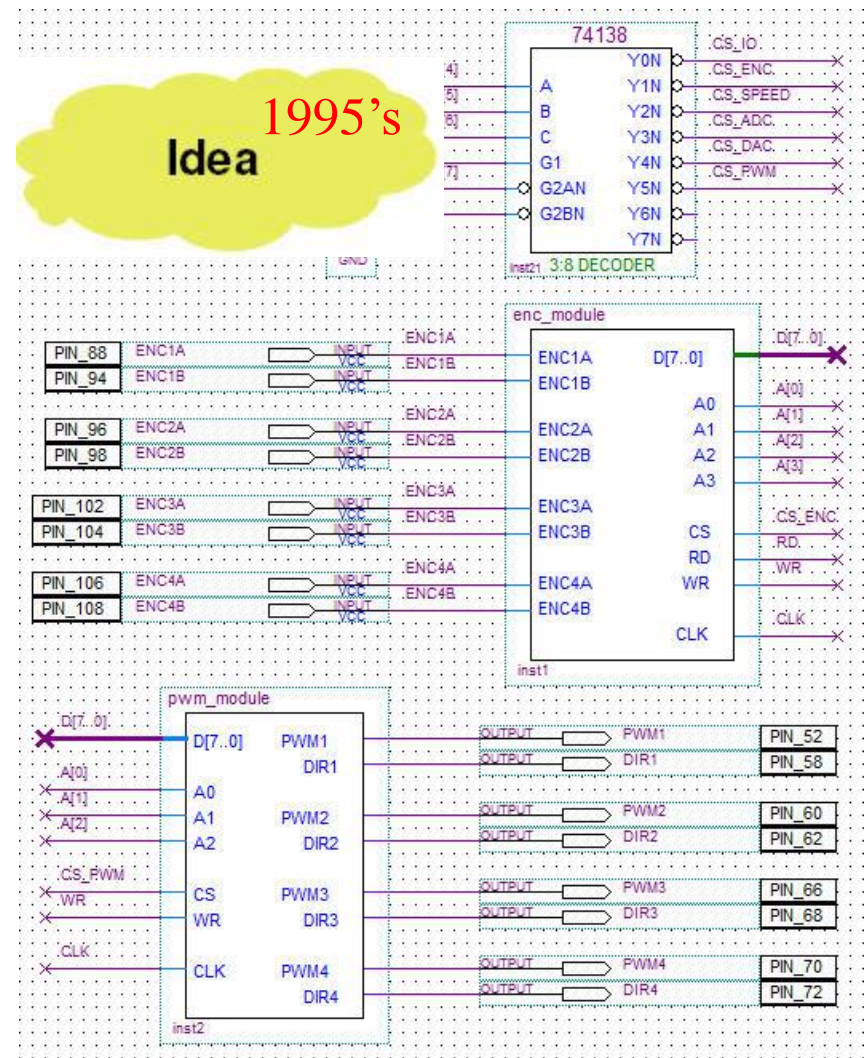
1980's
Idea



I. GIỚI THIỆU

1.1. Ngôn ngữ mô tả phần cứng HDL (Hardware Description Language)

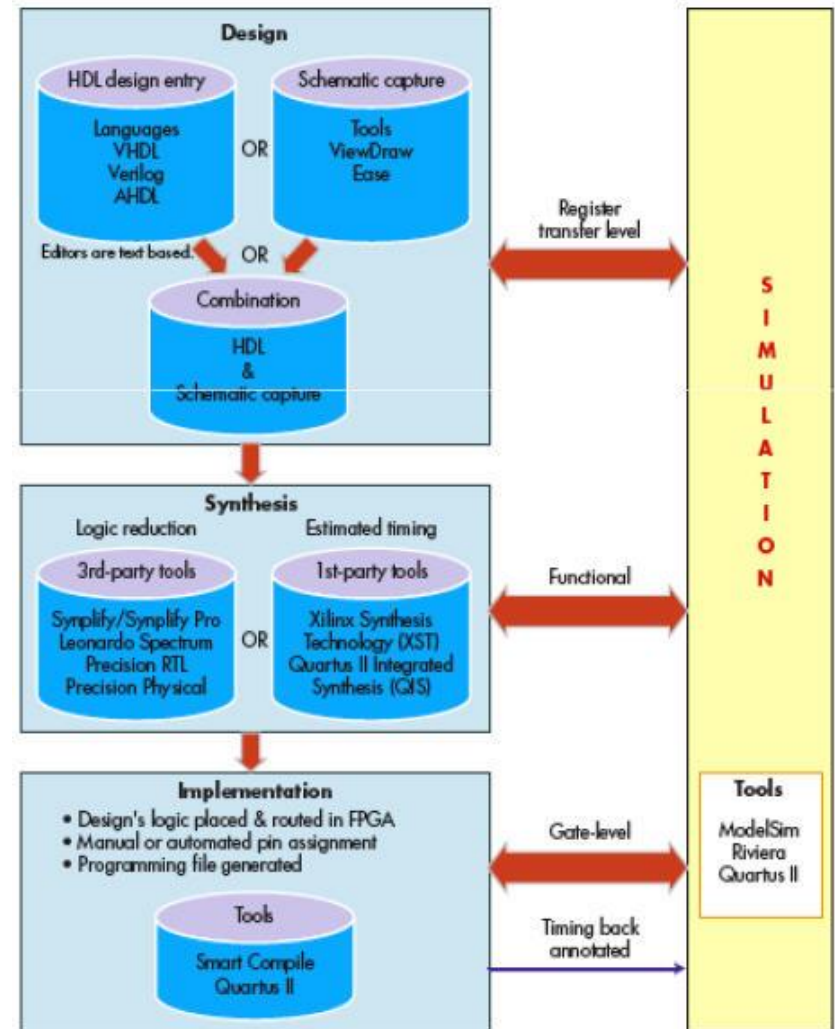
- Thiết kế mạch số (1995's ->): vẽ mạch schematic ->... hoàn tất
- Hai ngôn ngữ phổ biến: Verilog HDL (1984) và VHDL (1980).
- Được sử dụng rộng rãi trong thiết kế và mô phỏng mạch số ở mức độ thanh ghi (register-transfer level).
- Một thiết kế HDL bao gồm nhiều module, mỗi module chứa nhiều phân cấp và giao tiếp với các module khác thông qua tập input, output, và bidirectional port.



I. GIỚI THIỆU

1.2. Ưu điểm HDL so với Schematic:

- Xây dựng và lưu trữ HDL trong các file.
- Các file có thể đóng gói và xử lý bởi các công cụ:
 - Design: Viết HDL, vẽ sơ đồ
 - Synthesis: lựa chọn phần tử, tối ưu logic, ước lượng thời gian
 - Implementation: gán chân, lập trình vào FPGA
- Dễ dàng thay đổi, chỉnh sửa thiết kế mà không cần thay đổi phần cứng.
- Đáp ứng các yêu cầu thiết kế phức tạp

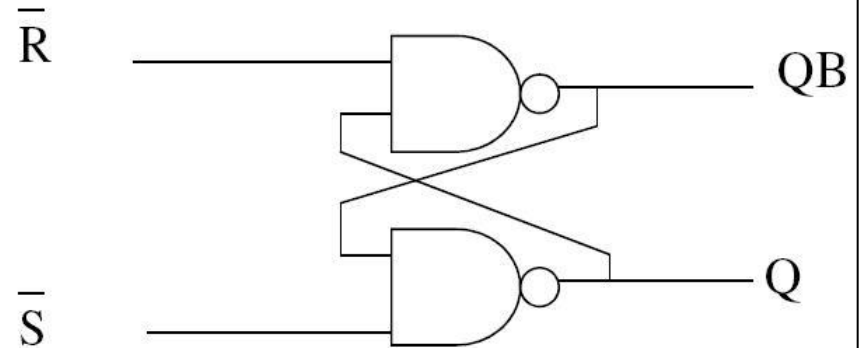


II. THIẾT KẾ PHÂN CẤP

2.1. Các mô tả (abstraction) trong thiết kế phần cứng:

- Mô tả cấu trúc (Structural modeling).
- Mô tả dòng dữ liệu (Dataflow modeling).
- Mô tả hành vi (Behavioral modeling).

Describe a module in terms of components



Netlist

- two modules: NAND_1 and NAND_2
- net 1: Rbar to NAND_1, input_1
- net 2: Sbar to NAND_2, input_2
- net 3: NAND_1, output_1 to NAND_2, input_1
- etc ..

II. THIẾT KẾ PHẦN CẤP

2.1. Các mô tả (abstraction) trong thiết kế phần cứng:

- Mô tả cấu trúc (Structural modeling).
- Mô tả dòng dữ liệu (Dataflow modeling).
- **Mô tả hành vi (Behavioral modeling)**

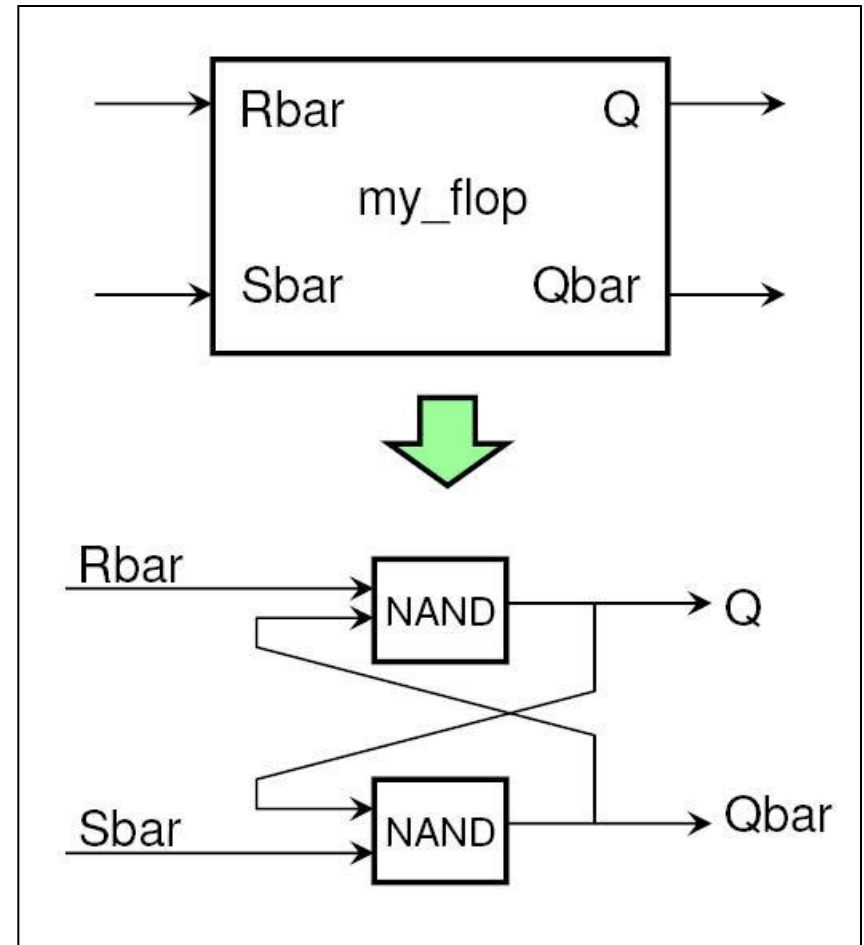
```
input S, R;  
output Q, QB;
```

```
if ( S == '1') and (R == '0') then  
    Set Q to '1', set QB to '0'  
else if ( S == '1') and (R == '1') then  
    Set Q to '1', set QB to '1'  
else if ( S == '0') and (R == '0') then  
    *Hold current state*  
else if ( S == '0') and (R == '1') then  
    Set Q to '0', set QB to '1'
```


II. THIẾT KẾ PHÂN CẤP

2.2. Khác nhau giữa Mô tả cấu trúc với Mô tả hành vi:

- Mô tả hành vi (Behavioral modeling): diễn tả chuyện gì xảy ra với Q, Qbar theo hàm của Rbar, Sbar.
- Mô tả cấu trúc (Structural modeling): diễn tả chuyện gì xảy ra với Q, Qbar theo hàm của 1 netlist gồm của các phần tử (các cổng) liên kết với nhau.
- Trong Verilog, một module có thể diễn đạt bằng cả Mô tả cấu trúc và Mô tả hành vi



II. THIẾT KẾ PHÂN CẤP

2.3. Ví dụ: Mô tả hành vi

```
module D_FF(q, d, clk, reset);
```

```
    output q;  one output port, three input ports
```

```
    input d, clk, reset;
```

```
    reg q;      'reg' means that q is a variable
```

```
    always @(posedge reset or negedge clk)
```

```
    if (reset)  always @(condition) means: whenever
                  (condition) is true, proceed.
```

```
        q <= 1'b0;
```

```
    else
```

```
        q <= d;
```

```
endmodule
```

The '<=' is called a *dataflow assignment*, in this case the effect is similar to assigning d to q

II. THIẾT KẾ PHÂN CẤP

2.3. Ví dụ: Mô tả cấu trúc

```
module T_FF(q, clk, reset);  
    output q;  
    input clk, reset;  
    wire d;  
  
    D_FF dff0(q, d, clk, reset);  
    not n1(d, q);  
  
endmodule
```

one output port, two input ports

a wire d has no storage: its value is always determined in terms of other variables

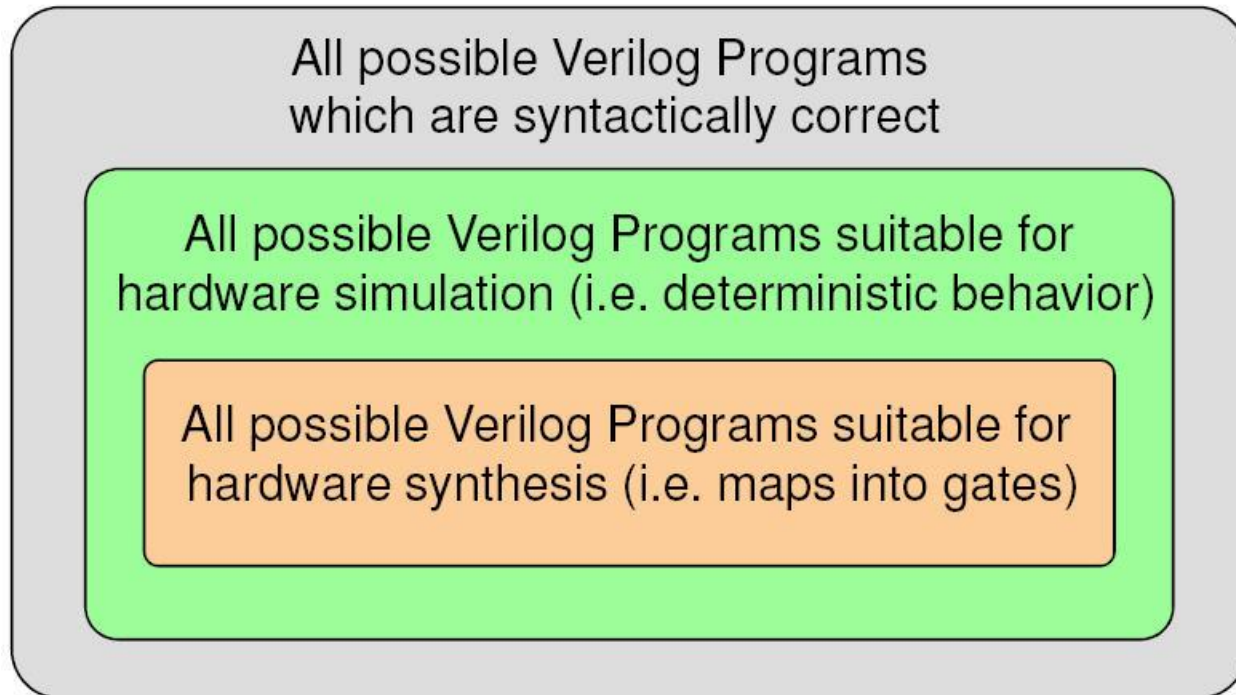
connect corresponding terminals of module

```
module D_FF(q, d, clk, reset);
```

'not' is a *primitive* in Verilog: the simulator understands what this module does.

III. VERILOG HDL

3.1. Tổng hợp (Synthesis) và Mô phỏng (Simulation)



- Tập trung vào mô phỏng và viết chính xác ngôn ngữ Verilog
- Ghi nhớ: Mô phỏng đúng không có nghĩa là thi công đúng

III. VERILOG HDL

3.1. Tổng hợp (Synthesis) và Mô phỏng (Simulation)

```
module syntst(clk);  
  init clk;  
  reg b;  
  
  initial  
  begin  
    b = 0;  
    #10 b = 1;  
  end  
  
endmodule;
```

```
module syntst(clk, q);  
  input clk;  
  output q;  
  reg b;  
  
  always @(posedge clk)  
  begin  
    b = ~b;  
  end  
  
  assign q = b;  
  
endmodule
```

III. VERILOG HDL

3.2. Module

module name(portlist);	-> tên module (danh sách port)
<i>port</i> declarations;	-> hướng của port (input, output, bidir)
<i>parameter</i> declarations;	-> tham số, khai báo module khác
<i>wire</i> declarations;	-> tín hiệu kết nối cục bộ
<i>reg</i> declarations;	-> lưu trữ cục bộ, biến cục bộ
<i>variable</i> declarations;	-> lưu trữ cục bộ trong module
<i>module</i> instantiations;	-> mô tả cấu trúc
<i>dataflow</i> statements;	-> mô tả hành vi
<i>always</i> blocks;	-> mô tả hành vi
<i>initial</i> blocks;	-> mô tả hành vi
tasks and functions;	
endmodule	

III. VERILOG HDL

3.2. Module – Ví dụ 1

```
module name(portlist);  
    port      declarations;  
    parameter declarations;  
    wire      declarations;  
    reg       declarations;  
    variable declarations;  
    module    instantiations;  
    dataflow statements;  
    always   blocks;  
    initial  blocks;  
    tasks and functions;  
endmodule
```

```
module or_nand_3 (enable, x1, x2, x3, x4, y);  
    input enable, x1, x2, x3, x4;  
    output y;  
    reg y;  
    always @ (enable or x1 or x2 or x3 or x4)  
        if (enable)  
            y = !((x1 | x2) & (x3 | x4));  
        else  
            y = 1; // operand is a constant.  
endmodule
```

III. VERILOG HDL

3.2. Module – Ví dụ 1

```
module name(portlist);  
    port      declarations;  
    parameter declarations;  
    wire      declarations;  
    reg       declarations;  
    variable  declarations;  
    module    instantiations;  
    dataflow  statements;  
    always    blocks;  
    initial   blocks;  
    tasks and functions;  
endmodule
```

```
module or_nand_4 (enable, x1, x2, x3, x4, y);  
    input enable, x1, x2, x3, x4;  
    output y;  
    assign y = or_nand(enable, x1, x2, x3, x4);  
    function or_nand;  
        input enable, x1, x2, x3, x4;  
        begin  
            or_nand = ~(enable & (x1 | x2) & (x3 |  
                x4));  
        end  
    endfunction  
endmodule
```


III. VERILOG HDL

3.2. Module – Ví dụ 2

```
module name(portlist);  
    port      declarations;  
    parameter declarations;  
    wire     declarations;  
    reg      declarations;  
    variable declarations;  
    module   instantiations;  
    dataflow statements;  
    always   blocks;  
    initial  blocks;  
    tasks and functions;  
endmodule
```

```
module or_nand_5 (enable, x1, x2, x3, x4, y);  
    input enable, x1, x2, x3, x4;  
    output y;  
    reg y;  
    always @ (enable or x1 or x2 or x3 or x4)  
        or_nand (enable, x1, x2, x3, x4, y);  
    task or_nand;  
        input enable, x1, x2, x3, x4;  
        output y1;  
        begin  
            y1 = !(enable & (x1 | x2) & (x3 | x4));  
        end  
    endtask  
endmodule
```

III. VERILOG HDL

3.3. Giá trị dữ liệu (Data value)

- 4 loại giá trị:
 - 0: mức thấp, 1: mức cao
 - Z: hi-Z, X: unknown
- Khi khởi tạo, các biến có giá trị X
- Gán giá trị các hằng số: **{bit width}'{base}{value}**
 - **parameter** RED = 6'b010_111 : 010111
 - **parameter** BLUE = 8'b0110 : 00000110
 - 4'bx01 : xx01
 - 16'H3AB : 0000001110101011
 - 24 : 0...0011000
 - 5'O36 : 11011
 - 8'hz : *ZZZZZZZZ*

III. VERILOG HDL

3.3. Loại dữ liệu (Data type)

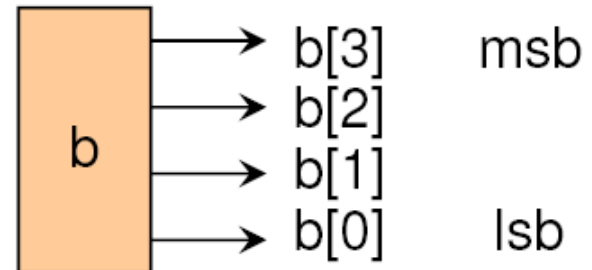
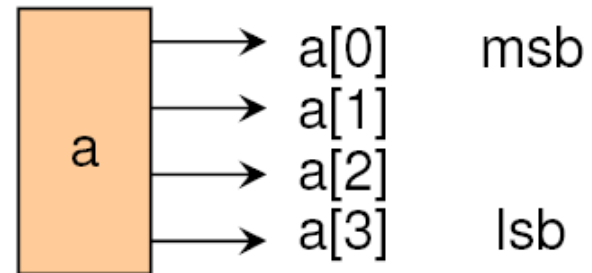
- 2 loại dữ liệu: **wire** và **reg**
 - **wire**: là giá trị được gán liên tục (continuously assigned), không có khả năng nhớ, không lưu trữ.
 - **reg (reg, integer)**: là giá trị gán thủ tục (procedurally assigned), có khả năng nhớ, lưu trữ
- Mô tả cấu trúc có thể sử dụng dữ liệu **wire**, không sử dụng dữ liệu **reg**
- Mô tả hành vi có thể sử dụng dữ liệu **reg** (trong cấu trúc **initial** và **always**), không sử dụng dữ liệu **wire**
- Ví dụ:

```
wire Reset;           // 1-bit wire      reg signed [3:0] counter;  // 4-bit register
wire [7:0] Addr;      // 8-bit wire      integer cla;           // maximum 32-bit
```

III. VERILOG HDL

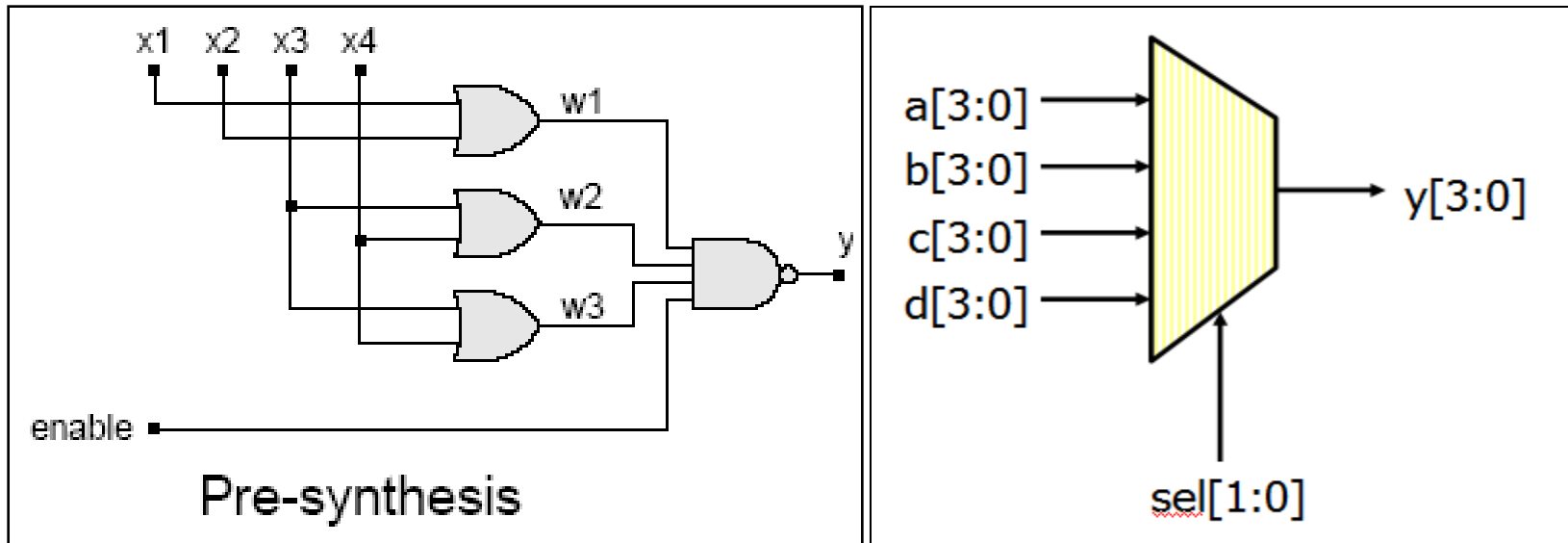
3.3. Loại dữ liệu (Data type)

```
module printit;  
  
    reg [0:3] a;  
    reg [3:0] b;  
  
    initial  
    begin  
  
        // some statements  
  
    end  
endmodule
```



III. VERILOG HDL

3.4. Ví dụ: Viết module cho các mạch sau bằng Verilog:



IV. PHÉP GÁN (ASSIGNMENT)

4.1. Khối initial và always

- Mô tả hành vi
- Khối **initial** chỉ thực hiện một lần khi khởi động (khi bật nguồn)
- Khối **always** thực hiện lặp lại liên tục
 - **always @(a or b)**
-> any changes in a or b
 - **always @(posedge a)**
-> a transitions from 0 to 1
 - **always @(negedge a)**
-> a transitions from 1 to 0
 - **always @***
-> any changes in “inputs”

```
module initalways (clk);  
  reg [7:0] a;  
  reg b;  
  initial  
    begin  
      a = 0;  
      b = 0;  
    end  
  always @ (posedge clk)  
    begin  
      a = a + 1;  
      b = 1;  
    end  
endmodule;
```


IV. PHÉP GÁN (ASSIGNMENT)

4.2. Phép gán liên tục (continuous assignment)

- Cú pháp:
assign LHS = RHS;
- LHS : kiểu dữ liệu bắt buộc là wire.
- RHS: kiểu dữ liệu có thể là wire, reg, hằng số, biểu thức
- Giá trị của RHS luôn luôn được gán cho LHS
- Tất cả các phép gán hoạt động đồng thời
- Thường sử dụng trong kết nối các cổng logic

// example 1

assign out = in1 ^ in2;

// example 2

wire product1, product2;

assign product1 = in1 & !in2;

assign product2 = !in1 & in2;

assign out = product1 | product2;

// example 3

assign out = (in1 != in2);

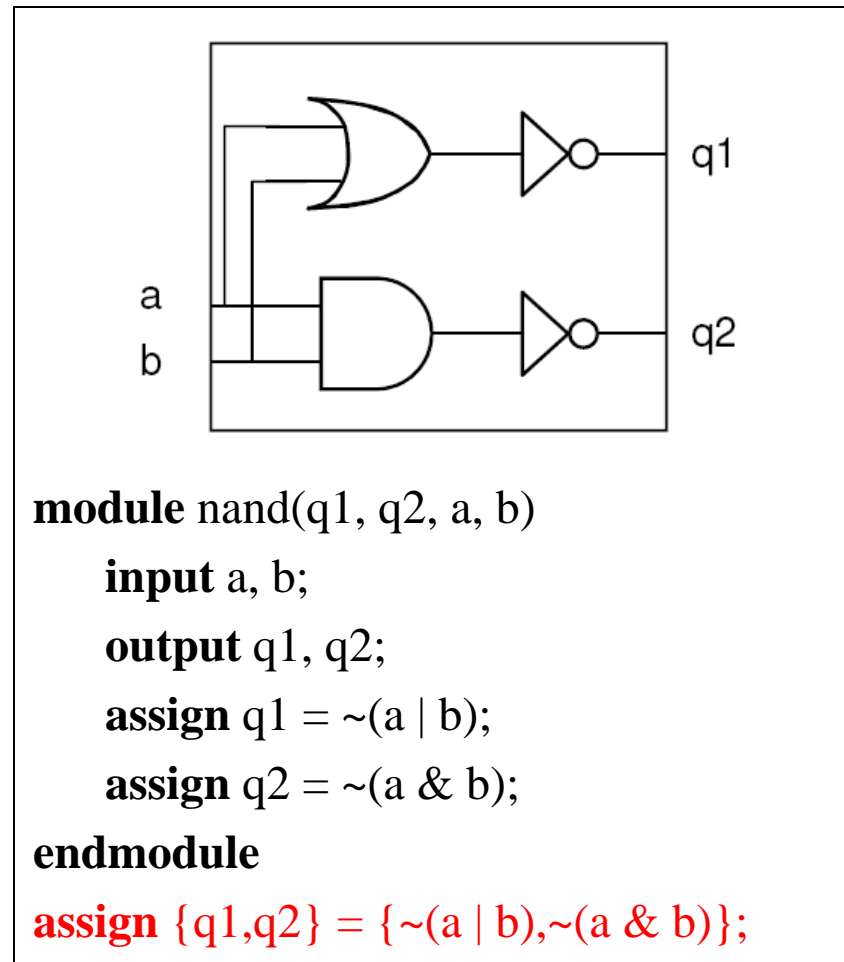
// example 4

assign out = in1 ? (!in2) : (in2);

IV. PHÉP GÁN (ASSIGNMENT)

4.2. Phép gán liên tục (continuous assignment)

- Cú pháp:
assign LHS = RHS;
- LHS : kiểu dữ liệu bắt buộc là wire.
- RHS: kiểu dữ liệu có thể là wire, reg, hằng số, biểu thức
- Giá trị của RHS luôn luôn được gán cho LHS
- Tất cả các phép gán hoạt động đồng thời, chỉ phụ thuộc netlist
- Mỗi ngõ ra chỉ được gán 1 lần
- Thường sử dụng trong kết nối các cổng logic



IV. PHÉP GÁN (ASSIGNMENT)

4.3. Phép gán thủ tục (procedural assignment)

- Cú pháp:
 LHS = RHS; // blocking
 LHS <= RHS; // non-blocking
- LHS : kiểu dữ liệu bắt buộc là reg.
- RHS: kiểu dữ liệu có thể là wire, reg, hằng số, biểu thức
- Thực thi ngay sau khi một sự kiện xảy ra
- Nếu có nhiều phép gán trong cùng 1 sự kiện thì sử dụng (**begin, end**)

```
module nand(q, a, b) //Continuous assignment
    output q;
    input a, b;
    assign q = ~(a | b);
endmodule
```

```
module nand(q, a, b) // Procedural assignment
    output q;
    reg q;
    input a, b;
    always @(a or b)
        q = ~(a | b);
endmodule
```

IV. PHÉP GÁN (ASSIGNMENT)

4.4. Phép gán thủ tục blocking và non-blocking

- Blocking: phép gán của hàng lệnh trên phải được hoàn thành trước khi thực hiện hàng lệnh bên dưới.
- Non-blocking: các biến dữ liệu chỉ được cập nhật sau khi ra khỏi 1 khối.
- Trong thực tế, không được gán blocking và non-blocking trong cùng 1 khối **always**.
- Ví dụ: Giá trị của a, b bằng bao nhiêu sau 3 chu kỳ xung clk?

// Blocking

```
reg [7:0] a, b;
```

```
initial b = 0;
```

```
initial a = 4;
```

```
always @(posedge clk) begin
```

```
    a = b + 2;
```

```
    b = a * 3;
```

```
end
```

// Non-blocking

```
reg [7:0] a, b;
```

```
always @(posedge clk) begin
```

```
    a <= b + 2;
```

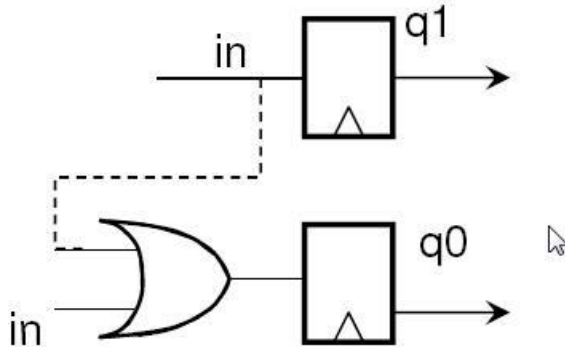
```
    b <= a * 3;
```

```
end
```

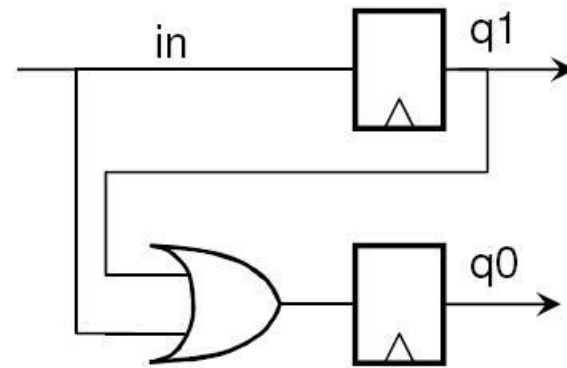
IV. PHÉP GÁN (ASSIGNMENT)

4.4. Phép gán thủ tục blocking và non-blocking

```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 = in;  
        q0 = in | q1;  
    end  
endmodule
```



```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 <= in;  
        q0 <= in | q1;  
    end  
endmodule
```

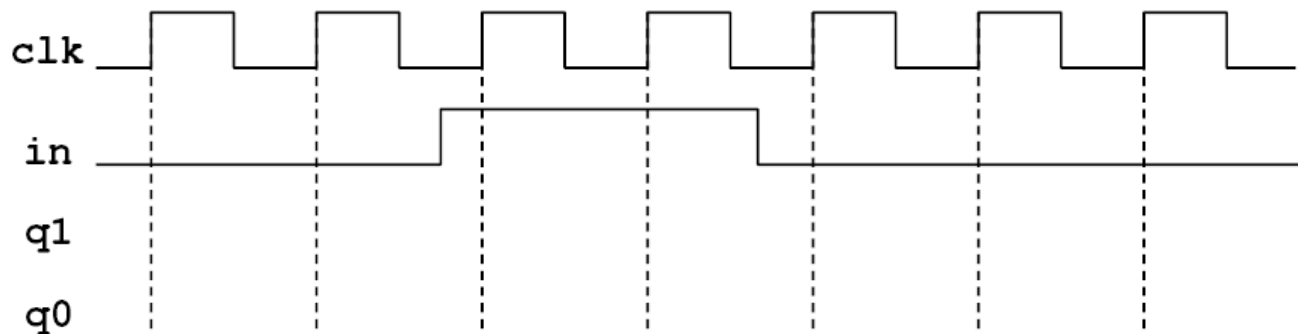


IV. PHÉP GÁN (ASSIGNMENT)

4.4. Phép gán thủ tục blocking và non-blocking

```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 = in;  
        q0 = in | q1;  
    end  
endmodule
```

```
module fsm_mod(q1, q0, in, clk);  
    output q1, q0;  
    input clk, in;  
    reg q1, q0;  
  
    always @(posedge clk) begin  
        q1 <= in;  
        q0 <= in | q1;  
    end  
endmodule
```



IV. PHÉP GÁN (ASSIGNMENT)

	Continuous Assignment	Proc. Blocking Assignment	Proc. Non-Blocking Assignment
Operation	=	=	<=
Where	using stand-alone <i>assign</i> statement	inside of <i>always</i> and <i>initial</i>	inside of <i>always</i> and <i>initial</i>
Example	<pre> wire q; reg a, b; assign q = a & b; </pre>	<pre> wire q; reg a, b; always @(b) a = b + 5; </pre>	<pre> wire q; reg a, b; always @(b) a <= 5; </pre>
Valid LHS	net (wire)	reg	reg
Valid RHS	expression of net or reg	expression of net or reg	expression of net or reg
Evaluated	when any part of RHS changes	procedural execution	at the end of current time step

© 4 Digital Design II

V. TOÁN TỬ (OPERATOR)

5.1. Các dạng toán tử:

- Arithmetic: + (cộng), - (trừ), * (nhân), / (chia), ** (mũ), % (số dư)
- Bitwise: & (and), | (or), ~ (negate), ^ (xor), ^~ (xnor)
- Reduction: **assign** q = &a, q = |(4'b0001) = 1, q = ^(4'b0111) = 1
- Logical: !, &&, ||, !=, (4'b1100) && (4'b0011) = 0
- Relational: a < b, a > b, a <= b, a >= b, a == b, a != b
- Logical shift: <<, >>
- Arithmetic shift: <<<, >>>
- Selection: **assign** q = c ? a : b
- Concatenation: {a, b[3:2], c} = {a, b[3], b[2], c} = a 4-bit vector
- Replication: 4{a} = {a, a, a, a}, {a, 3{b,c}} = {a, b, c, b, c, b, c}

V. TOÁN TỬ (OPERATOR)

5.2. Ví dụ:

reg [5:0] A = 6'b101111;

reg [5:0] B, C;

B = A >> 2;

B = A >>> 2; gia su **reg signed** [5:0] B;

C = A + B;

C = A & B;

C = {A[2:1], B[2:0]};

assign out = &A;

assign out = A[1] ? (A[0] ? B[1] : B[2]) : B[0];

C = out ? A : C;

V. TOÁN TỬ (OPERATOR)

5.2. Ví dụ:

```
reg [5:0] A = 6'b101111;
```

```
reg [5:0] B, C;
```

```
B = A >> 2;           // 6'b001011
```

```
B = A >>> 2;          // 6'b111011
```

```
C = A + B;            // 6'b101010
```

```
C = A & B;             // 6'b101011
```

```
C = {A[2:1], B[2:0]}; // 6'b011011
```

```
assign out = &A;       // A[0]&A[1]&A[2] &A[3]&A[4]&A[5] = 0
```

```
assign out = (A[1]) ? ((A[0]) ? B[1] : B[2]) : B[0]; // = 1
```

```
C = out ? A : C;      // 6'b101111
```

VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

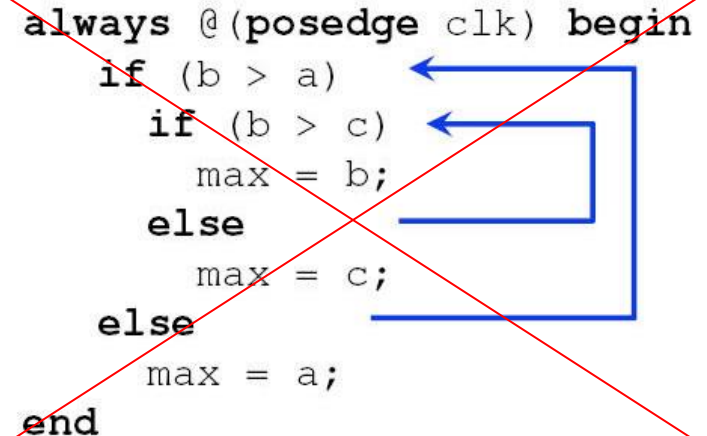
6.1. if-then-else:

```
reg [7:0] a, b, max;

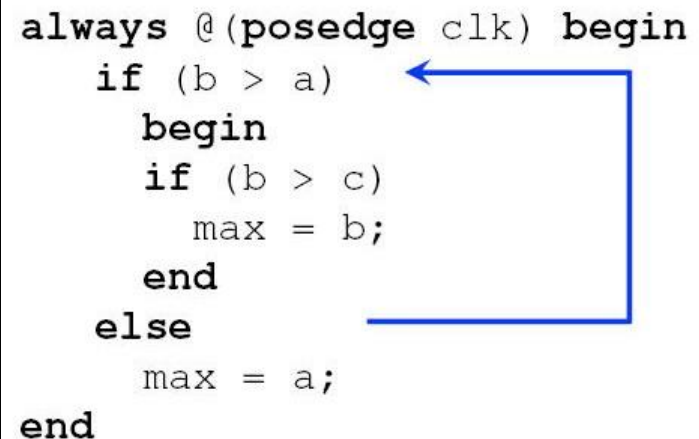
always @(posedge clk) begin
    max = b;
    if (a > b)
        max = a;
end

always @(posedge clk) begin
    if (a > b)
        max = a;
    else
        max = b;
end
```

```
always @(posedge clk) begin
    if (b > a)
        if (b > c)
            max = b;
        else
            max = c;
    else
        max = a;
end
```



```
always @(posedge clk) begin
    if (b > a)
        begin
            if (b > c)
                max = b;
            end
        else
            max = a;
end
```



VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

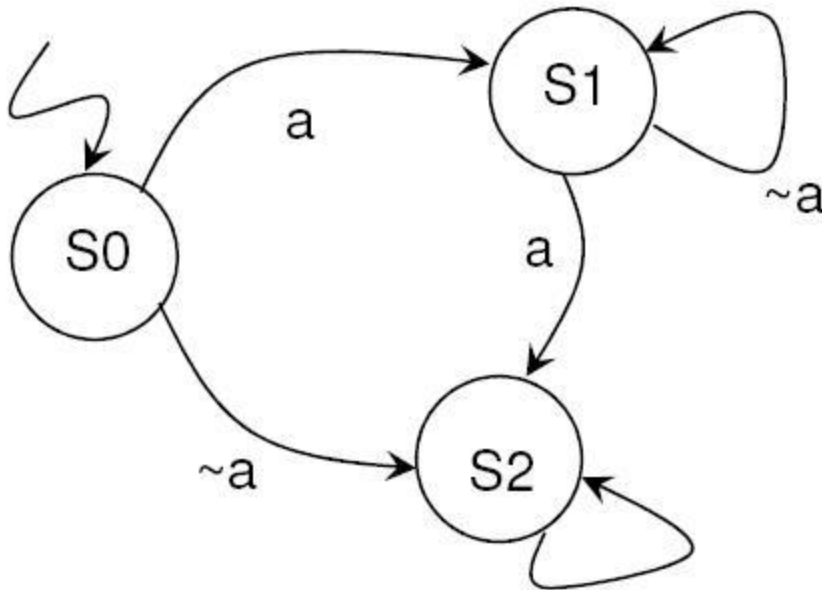
6.1. if-then-else:

```
reg [7:0] a, b, c, max;

always @(posedge clk) begin
    if (a < 10) begin
        // ... when a < 10
    end
    else if (a < 20) begin
        // ... when a >= 10 and a < 20
    end
    else if (a < 30) begin
        // .. when a >= 20 and a < 30
    end
    else begin
        // .. when a >= 30
    end
end
```


VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.2. Case:



```
reg a;  
reg [1:0] state;  
parameter s0 = 2'b00,  
           s1 = 2'b01,  
           s2 = 2'b10;  
  
always @(posedge clk)  
  case (state)  
    s0: if (a)  
         state <= s1;  
        else  
         state <= s2;  
    s1: if (a)  
         state <= s2;  
        else  
         state <= s1;  
    s2: state <= s2;  
    default state <= s0;  
  endcase
```

VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.2. Case:

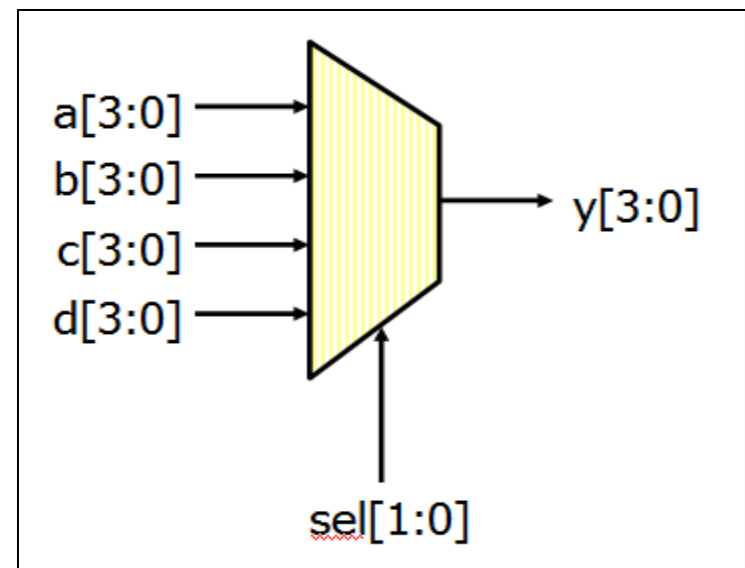
```
reg a, b;

always @(posedge clk)
    case ({a,b})
        2'b00: // will match a = 0, b = 0
        2'bx0: //           a = x, b = 0
        2'b10: //           a = 1, b = 1
        2'b01,
        2'b0x: //           a = 0, b = 1 or x
        2'b00: //           a = 0, b = 0
        default: // all other cases
    endcase
```

VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.3. Ví dụ: - Dùng toán hạng điều kiện:

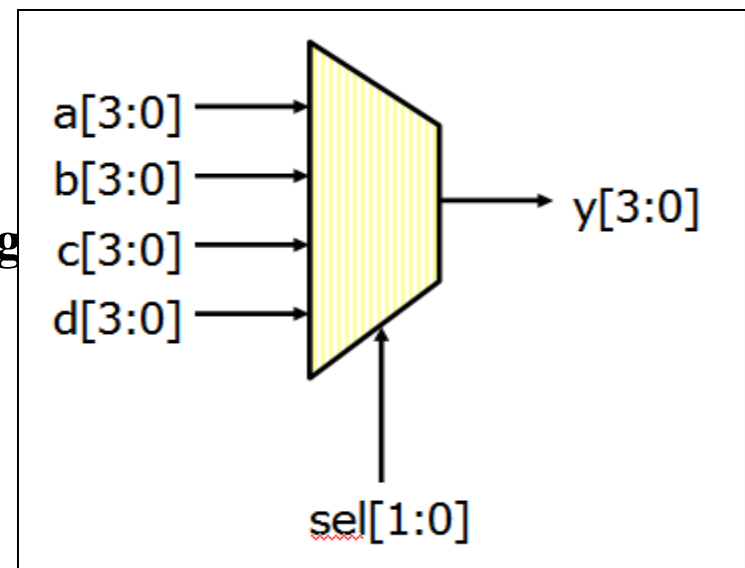
```
module mux_4bits(y, a, b, c, d, sel);  
  input [3:0] a, b, c, d;  
  input [1:0] sel;  
  output [3:0] y;  
  assign y =  
    (sel == 0) ? a :  
    (sel == 1) ? b :  
    (sel == 2) ? c :  
    (sel == 3) ? d : 4'bx;  
endmodule
```



VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.3. Ví dụ: - Dùng phát biểu if-then-else

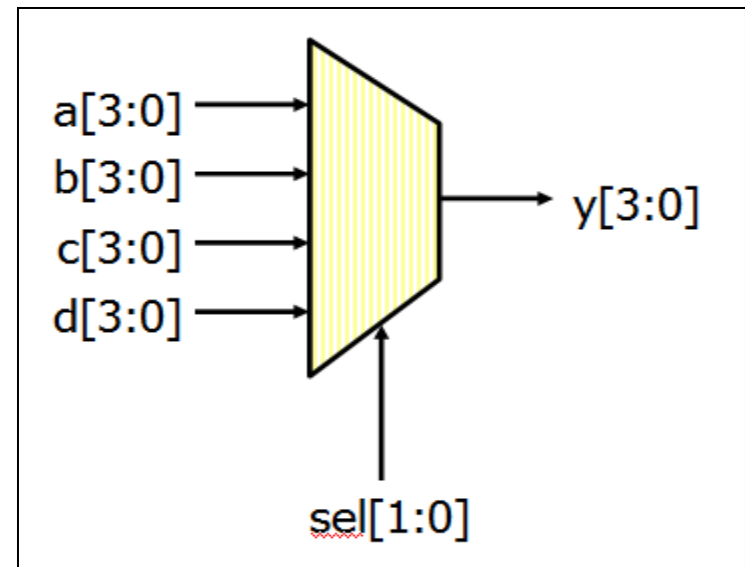
```
module mux_4bits(y, a, b, c, d, sel);  
  input [3:0] a, b, c, d;  
  input [1:0] sel;  
  output [3:0] y;  
  reg [3:0] y;  
  always @ (a or b or c or d or sel) begin  
    if (sel == 0) y = a;  
    else if (sel == 1) y = b;  
    else if (sel == 2) y = c;  
    else if (sel == 3) y = d;  
    else y = 4'bx;  
  end  
endmodule
```



VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.3. Ví dụ: - Dùng phát biểu Case

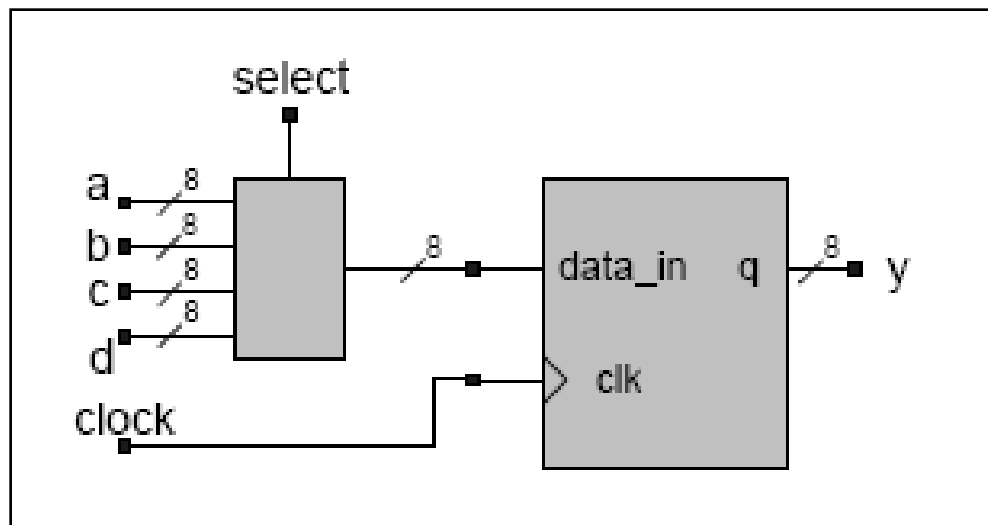
```
module mux_4bits(y, a, b, c, d, sel);  
  input [3:0] a, b, c, d;  
  input [1:0] sel;  
  output [3:0] y;  
  reg [3:0] y;  
  always @ (a or b or c or d or sel)  
    case (sel)  
      0: y = a;  
      1: y = b;  
      2: y = c;  
      3: y = d;  
      default: y = 4'bx;  
    endcase  
endmodule
```



VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.3. Ví dụ:

- Thực hiện sơ đồ sau dùng phát biểu if-then-else và Case

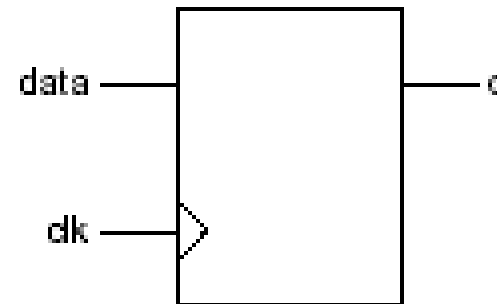


VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.4. Một số Flip-Flop tiêu biểu:

- D Flip-Flop:

```
module dff (data, clk, q);  
    input data, clk;  
    output q;  
    reg q;  
    always @(posedge clk)  
        q <= data;  
endmodule
```

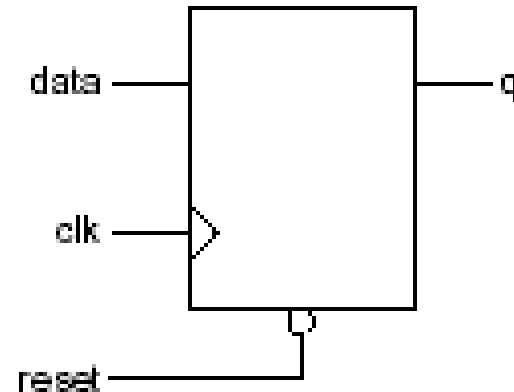


VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.4. Một số Flip-Flop tiêu biểu:

- D Flip-Flop với Reset không đồng bộ:

```
module dff (data, clk, reset, q);  
    input data, clk, reset;  
    output q;  
    reg q;  
    always @(posedge clk or negedge reset)  
    if (~reset)  
        q <= 1'b0;  
    else  
        q <= data;  
endmodule
```

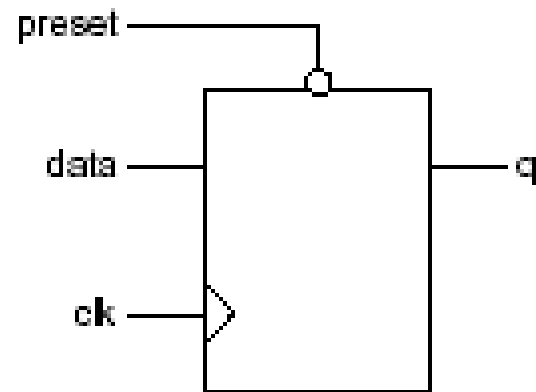


VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.4. Một số Flip-Flop tiêu biểu:

- D Flip-Flop với Preset không đồng bộ:

```
module dff (data, clk, preset, q);  
    input data, clk, preset;  
    output q;  
    reg q;  
    always @(posedge clk or negedge preset)  
    if (~preset)  
        q <= 1'b1;  
    else  
        q <= data;  
endmodule
```

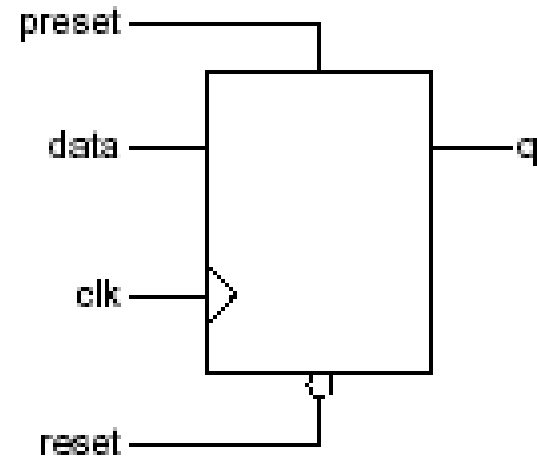


VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.4. Một số Flip-Flop tiêu biểu:

- D Flip-Flop với Preset, Reset không đồng bộ:

```
module dff (data, clk, preset, reset, q);  
    input data, clk, preset, reset;  
    output q;  
    reg q;  
    always @ (posedge clk or negedge reset or posedge preset)  
        if (~reset)  
            q <= 1'b0;  
        else if (preset)  
            q <= 1'b1;  
        else q <= data;  
endmodule
```



VI. PHÁT BIỂU CÓ ĐIỀU KIỆN

6.4. Một số Flip-Flop tiêu biểu:

