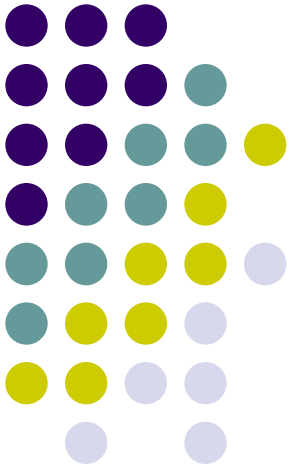
SEMICON Solutions

Cơ Bản Verilog HDL

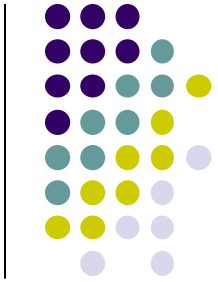


Trình bày: Đặng Tường Dương

2014

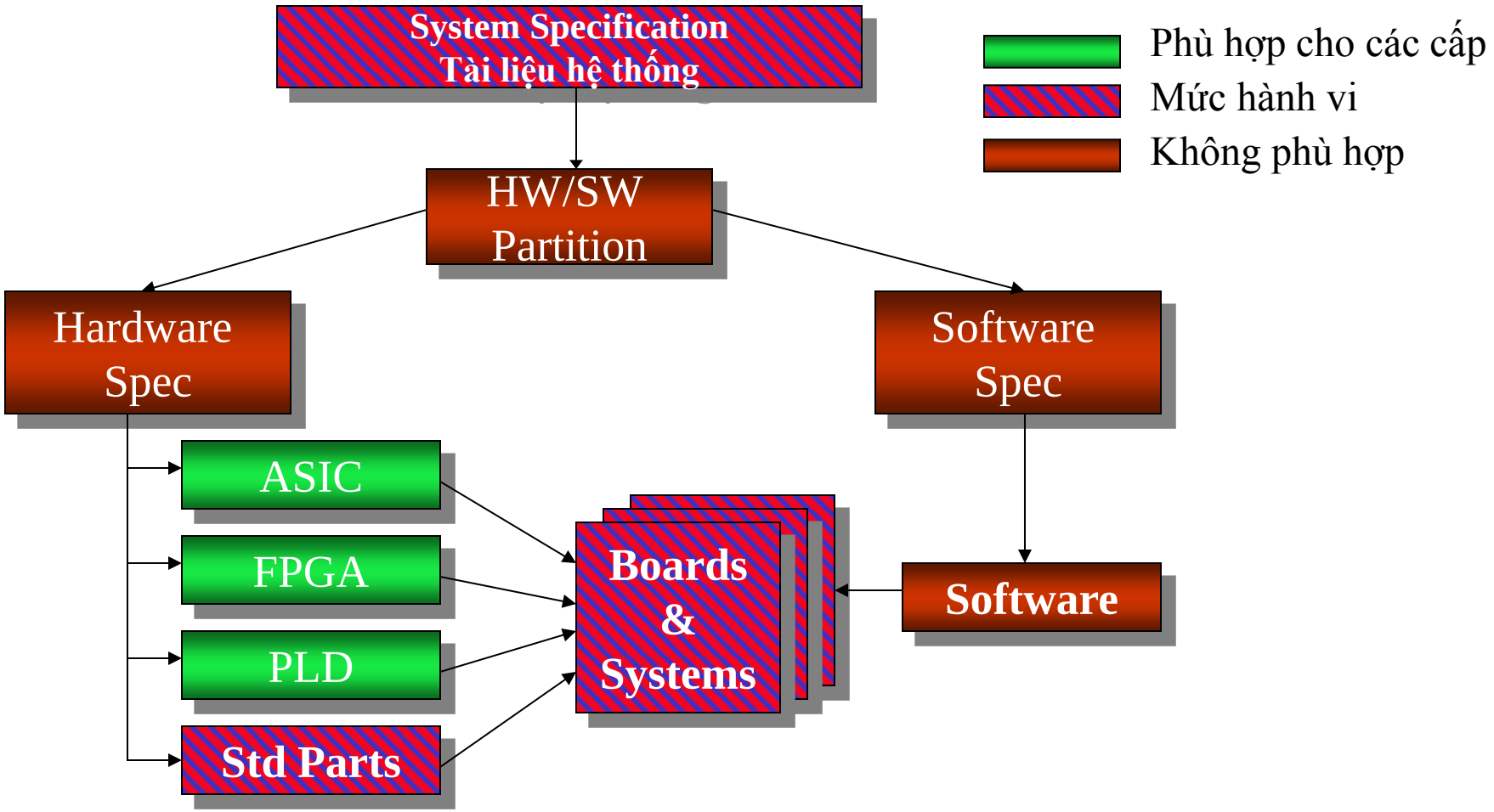
SEMICON
THINKING OUT OF THE BOX

Verilog là gì?



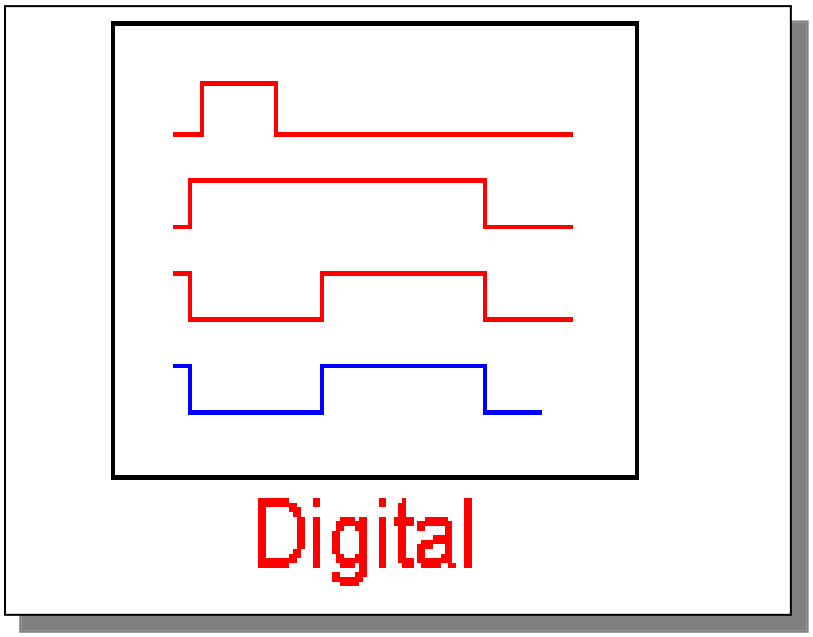
- Ngôn ngữ mô tả phần cứng - Hardware Description Language (HDL)
- Phát triển từ 1984
- Theo chuẩn 1364 IEEE: 12/2005

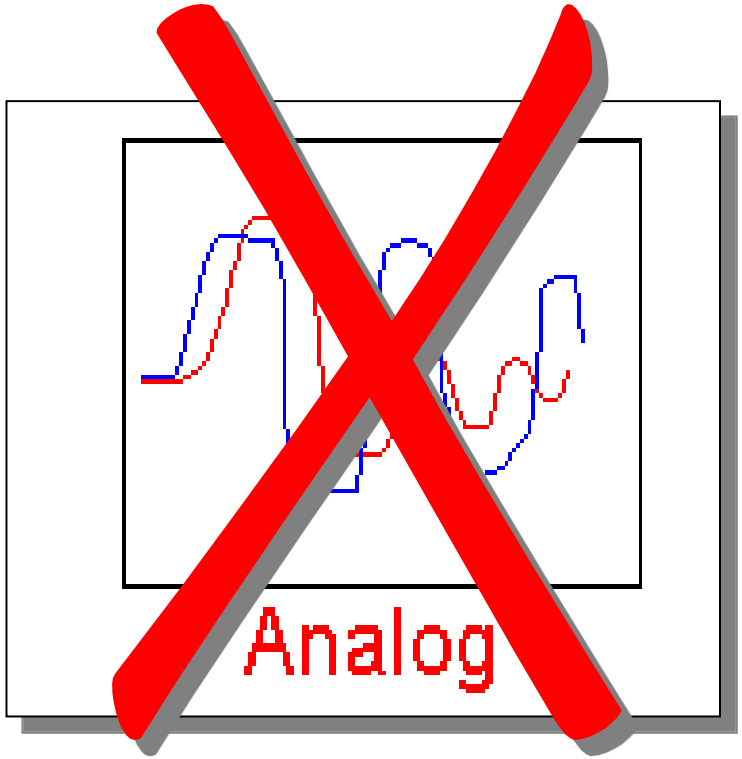
Ứng dụng của Verilog



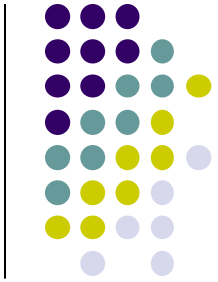
Giới hạn cơ bản của Verilog

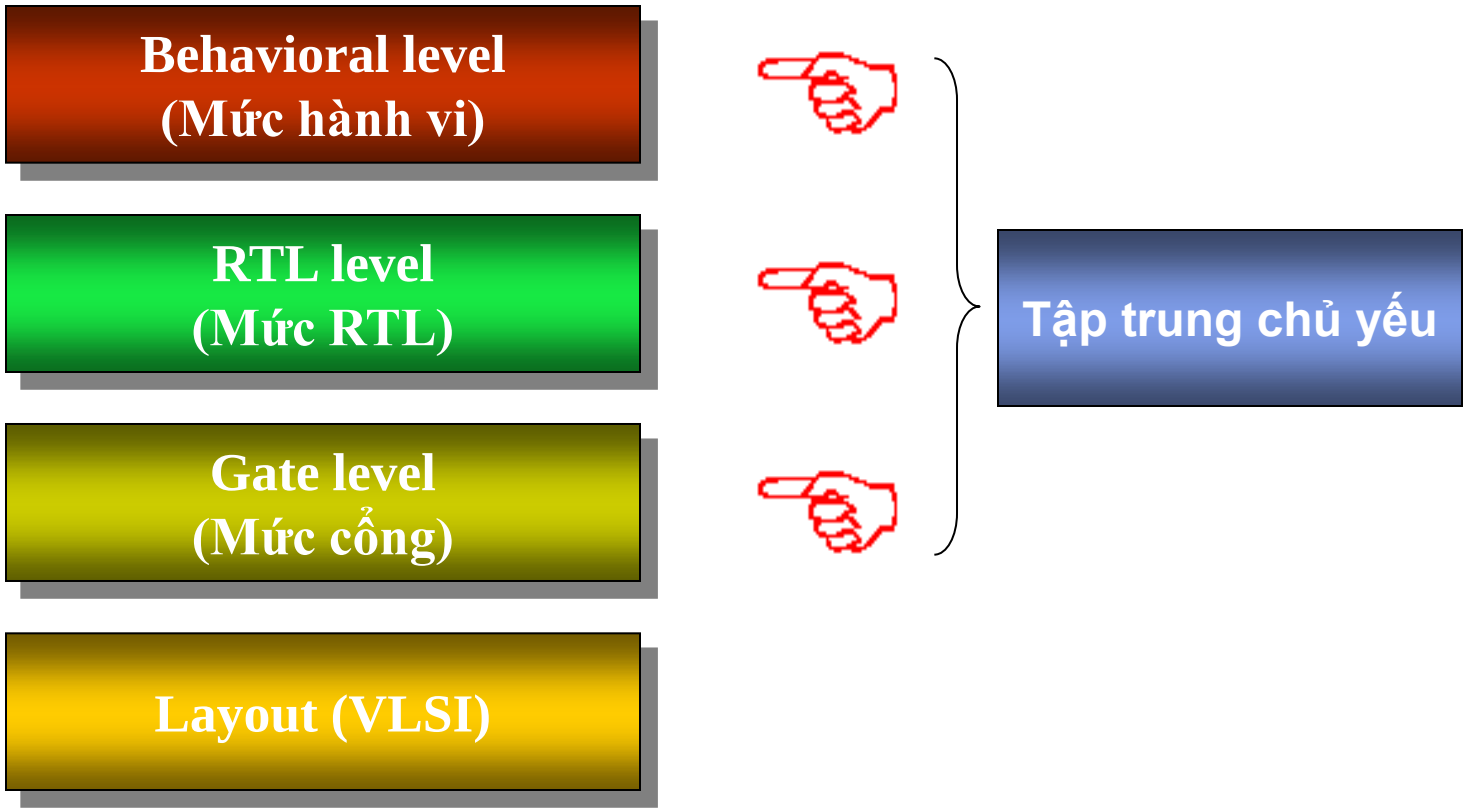
Chỉ đặc tả hệ thống số



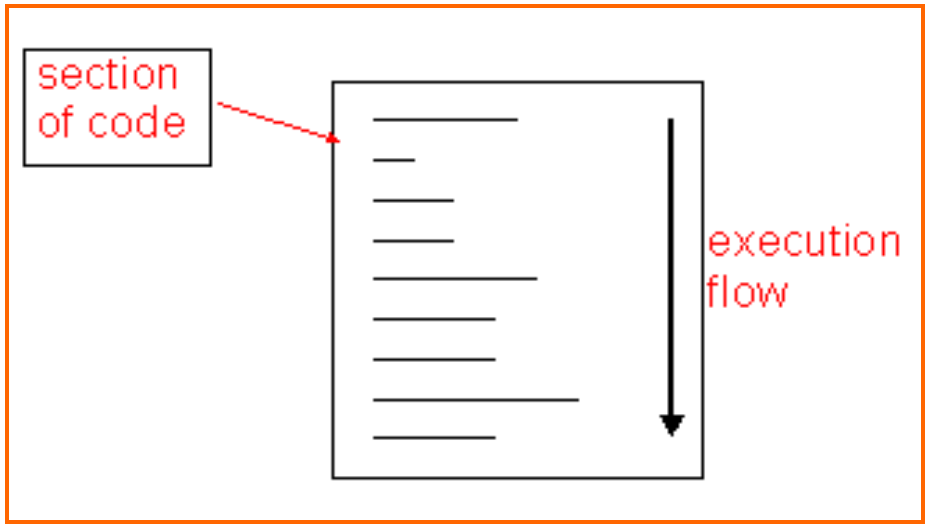


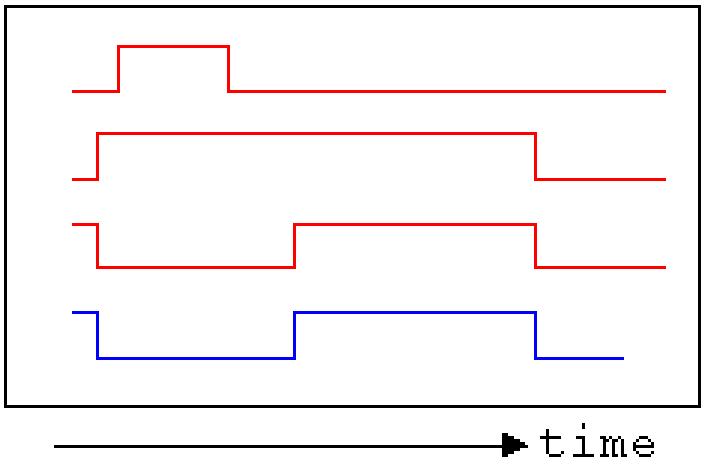
Các mức mô tả Verilog



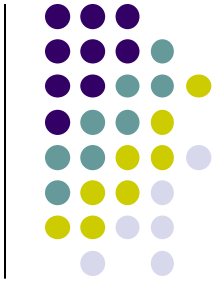


Điểm Chính về Verilog



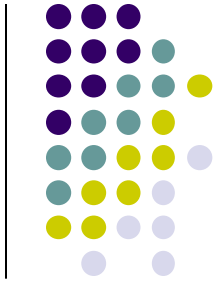


Quy định sử dụng



- Có thể dùng {[A-Z], [a-z], [0-9], _, $, …}
- Không bắt đầu $ hay [0-9]
  - `myidentifier` → ok
  - `m_y_identifier` → ok
  - **`3my_identifier`** → **not ok**
  - **`$my_identifier`** → **not ok**
  - `_myidentifier$` → ok
- Phân biệt chữ hoa và thường
  - `myid` ≠ `Myid`

Comments – Trích dẫn

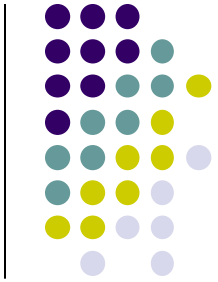


- // Dòng chú thích
- /*

Chú thích nhiều dòng

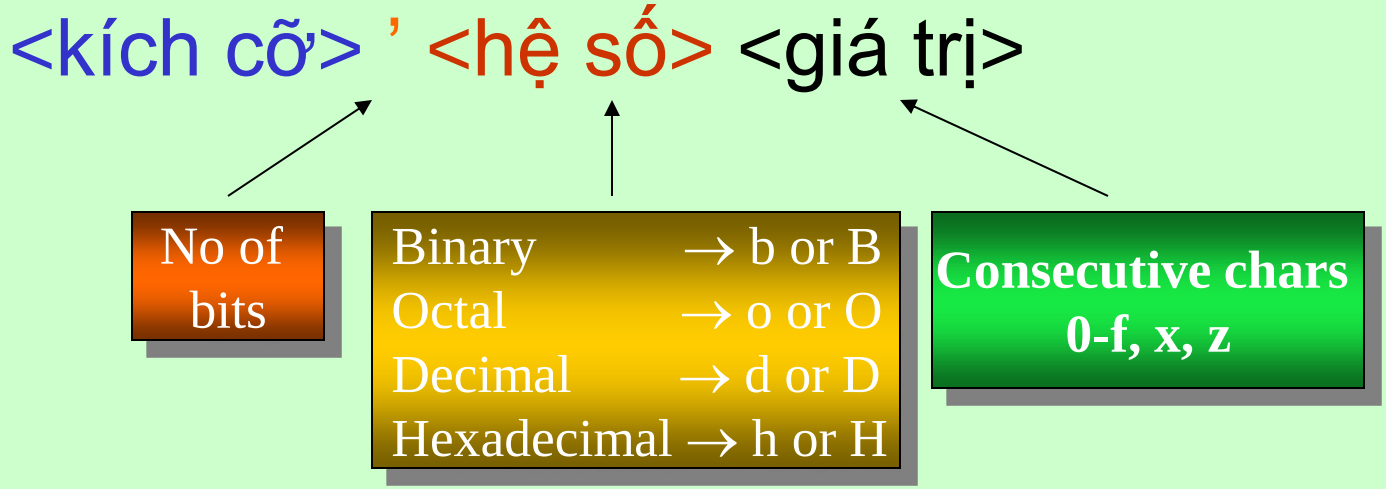
*/

Các giá trị Verilog



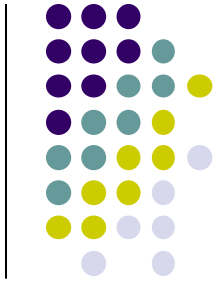
- *0* mức logic thấp hay điều kiện sai (false)
- *1* mức logic cao hay điều kiện đúng (true)
- *x* mức logic không biết (unknown level)
- *z* Mức logic tông trở cao (hi-Z)

Số trong Verilog (1)



- 8'h ax = 1010xxxx
- 12'o 3zx7 = 011zzzxxx111

Exercise



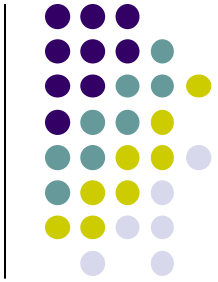
Y = 1010_1101(B) + 746 (O) + 126(D) + AF(H)

746(O) =

126(D) =

AF(H) =

Exercise



Y = 1010_1101(B) + 746 (O) + 126(D) + AF(H)

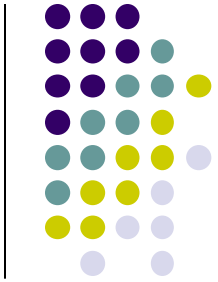
746(O) = 111_100_110 (B) = 1_1110_0110 (B)

126(D) = 7*16 + 14 (D) = 7E(H) = 0111_1110(B)

AF(H) = 1010_1111(B)

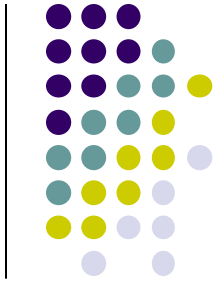
```
     0_1010_1101(B)
     1_1110_0110(B)
+    0_0111_1110(B)
     0_1010_1111(B)
  --------------------
    11_1100_0000(B)
```

Số trong Verilog (2)



- Có thể chèn "_" để dễ đọc
  - 12'b 000_111_010_100
  - 12'b 000111010100
  - 12'o 07_24
- Mở rộng BIT (MSB: Most Significant Bit)
  - MS bit = 0, x or z ⇒ mở rộng chính nó
    - 4'b x1 = 4'b xx_x1
  - MS bit = 1 ⇒ mở rộng 0
    - 4'b 1x = 4'b 00_1x

Số trong Verilog (3)

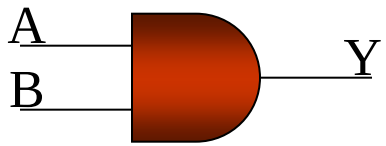


- Nếu hệ số không có, xem như hệ số 10
  - **15 = <size>'d 15**

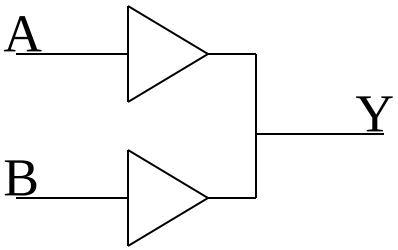
Đường nối (nets) (1)

- Xem như các dây phần cứng lái bởi logic
- Bằng Z khi không nối
- Các loại khác nhau của nets
  - `wire`
  - `wand` **(wired-AND)**
  - `wor` **(wired-OR)**
  - `tri` **(tri-state)**
- Ví dụ sau: Y sẽ xem xét tự động khi A hoặc B thay đổi

Đường nối (nets) (2)




```
wire Y;  // declaration
assign Y = A & B;
```

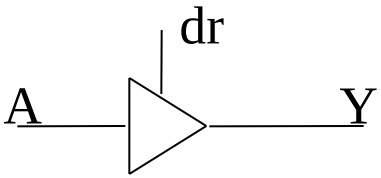



```
wand Y;  // declaration
assign Y = A;
assign Y = B;
```

Y	A=0	A=1
B=0	0	0
B=1	0	1

```
wor Y;  // declaration
assign Y = A;
assign Y = B;
```

Y	A=0	A=1
B=0	0	1
B=1	1	1




```
tri Y;  // declaration
assign Y = (dr) ? A : z;
```

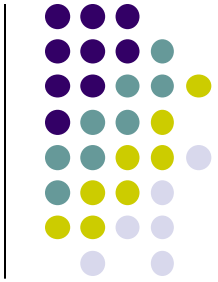
Các thanh ghi - Registers

- Là các biến lưu giá trị
- Không có thật trong phần cứng thật nhưng ..
- .. Phần cứng thật có thể bổ sung các thanh ghi
- Chỉ có 1 loại: **reg**

```
reg A, C; // Khai báo
// Phép gán luôn được thực thi bên trong một procedure
A = 1;
C = A; // C sẽ có giá trị logic 1
A = 0; // C vẫn là 1
C = 0; // Bây giờ C là 0
```

- Giá trị thanh ghi được ghi mới hiện hữu!!

Vectors



- Các Vector Bus

```
wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;
```

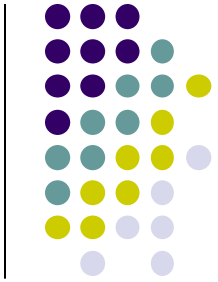
- Số bên trái có trọng số lớn nhất (MSB – Maximum Significant bit)
- Quản lý theo từng bit

```
busC[1] = busA[2];
busC[0] = busA[1];
```

- Gán vector (theo vị trí***!!***)

```
busB[1] = busA[3];
busB[2] = busA[2];
busB[3] = busA[1];
busB[4] = busA[0];
```

Loại dữ liệu thực và số nguyên



- Khai báo

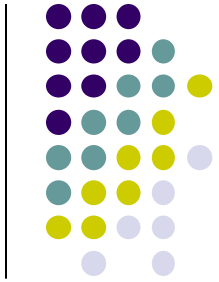
```
integer i, k;
real r;
```

- Dùng như thanh ghi (bên trong procedures)

```
i = 1; // gán bên trong procedures
r = 2.9;
k = r; // k được làm tròn 3
```

- Các số nguyên không có giá trị ban đầu
- Các số thực ban đầu là 0.0

Loại dữ liệu thời gian



- Loại dữ liệu đặc biệt để đo thời gian mô phỏng
- Khai báo

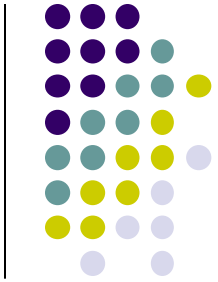
```
time my_time;
```

- Dùng bên trong procedure

```
my_time = $time; // lấy giá trị hiện hành
```

- Thời gian chạy mô phỏng không phải thời gian thực

Mảng – Arrays (1)



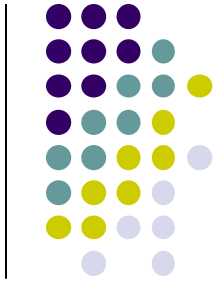
- Cú pháp

```
integer count[1:5]; // 5 số nguyên
reg var[-15:16]; // 32 thanh ghi 1-bit
reg [7:0] mem[0:1023]; // 1024 thanh ghi 8-bit
```

- Truy xuất các phần tử của mảng
  - `mem[10] = 8'b 10101010;`

```
reg [7:0] temp;
..
temp = mem[10];
var[6] = temp[2];
```

Mảng – Arrays (2)



- Giới hạn: Không thể truy xuất một phần mảng hoặc toàn mảng một lúc

```
var[2:9] = ???; // SAI!!
var = ???; // SAI!!
```

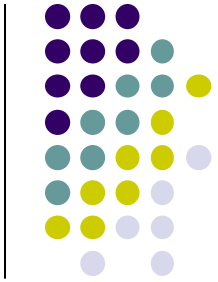
- Không có mảng đa chiều

```
reg var[1:10] [1:100]; // SAI!!
```

- Mảng không làm việc với loại dữ liệu thực

```
real r[1:10]; // SAI !!
```

Chuỗi - String

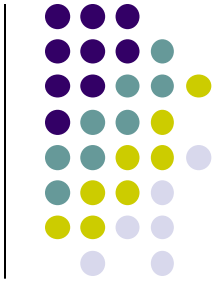


- Bổ sung các thanh ghi:

```
reg [8*13:1] string_val; // Có thể giữ tối đa 13 ký tự
..
string_val = “Hello Verilog”;
string_val = “hello”; // Các Byte trọng số cao được điền 0
string_val = “I am overflowed”; // “I ” không gán được
```

- Escaped chars:
  - \n dòng mới
  - \t khoảng TAB
  - %% %
  - \\ \
  - \““ Khoảng trắng

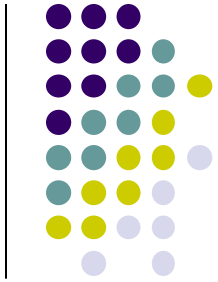
Toán tử logic



- `&&` → AND logic
- `||` → OR logic
- `!` → NOT logic
- Toán tử trả về 1 trong các giá trị: *0*, *1* or *x*
- Kết quả là một giá trị: *0*, *1* or *x*

```
A = 6;          A && B → 1 && 0 → 0
B = 0;          A || !B → 1 || 1 → 1
C = x;          C || B → x || 0 → x
```

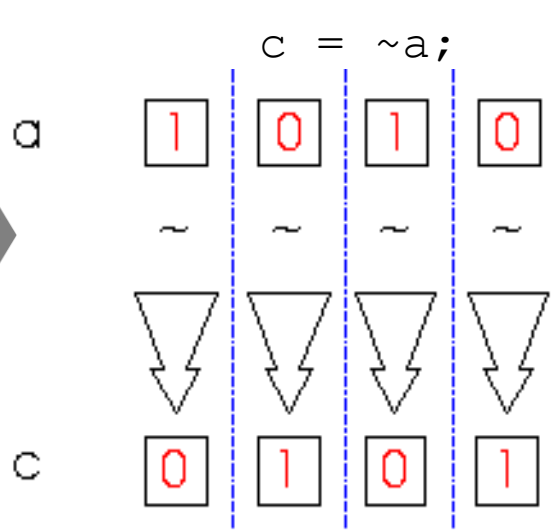
Toán tử trên bit - bitwise

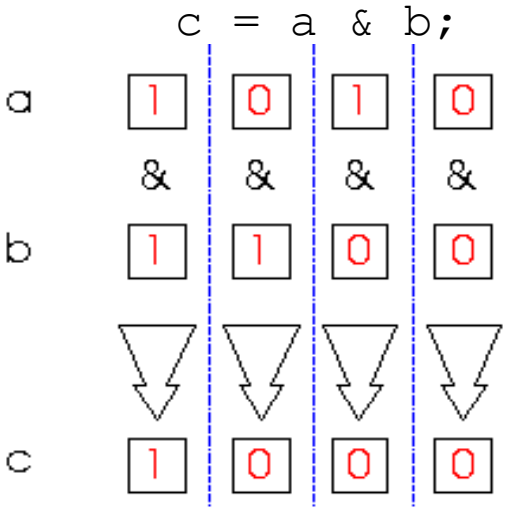


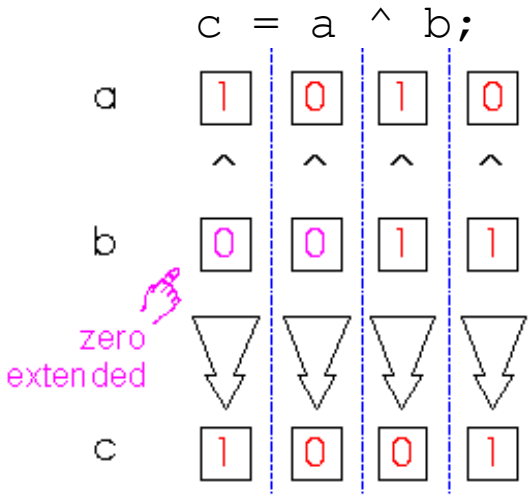
- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- ~^ hoặc ^~ → bitwise XNOR

Toán tử trên bit - bitwise

a = 4'b1010;
b = 4'b1100;



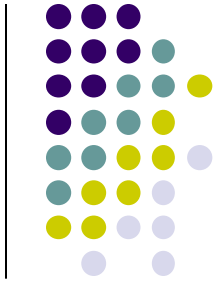




a = 4'b1010;
b = 2'b11;

SEMICON
THINKING OUT OF THE BOX

Các toán tử rút gọn

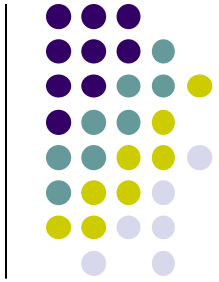


- `&` → AND
- `|` → OR
- `^` → XOR
- `~&` → NAND
- `~|` → NOR
- `~^` or `^~` → XNOR

- Một phép toán nhiều bit → Kết quả là 1 bit

```
a = 4'b1001;
..
c = |a; // c = 1|0|0|1 = 1
```

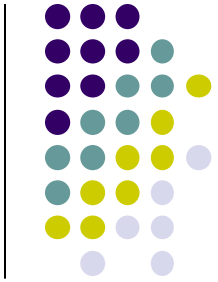
Toán tử dịch



- >> → dịch phải
- << → dịch trái
- Kết quả có cùng kích cỡ với phép toán ban đầu, 0 sẽ được chèn vào

```
a = 4'b1010;
...
d = a >> 2;    // d = 0010
c = a << 1;    // c = 0100
```

Toán tử dịch (Ví Dụ)



```
wire a = 8’b1;
always @(sel_reg)
  case (sel_reg[2:0])
      3’b000: b = 8’b 0000_0001;  // = a;
      3’b001: b = 8’b 0000_0010;  // = a << 1;
      3’b010: b = 8’b 0000_0100;  // = a << 2;
      3’b011: b = 8’b 0000_1000;  // = a << 3;
      3’b100: b = 8’b 0001_0000;  // = a << 4;
      3’b101: b = 8’b 0010_0000;  // = a << 5;
      3’b110: b = 8’b 0100_0000;  // = a << 6;
      3’b111: b = 8’b 1000_0000;  // = a << 7;
endcase
```

Toán tử gộp

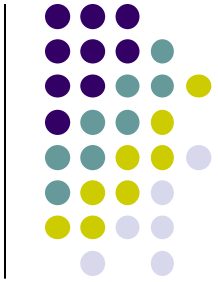
- {op1, op2, ..} → gộp op1, op2, .. Thành một số
- Phép toán phải định kích cỡ !!

```
reg a;
reg [2:0] b, c;
..
a = 1'b 1;
b = 3'b 010;
c = 3'b 101;
catx = {a, b, c};          // catx = 1_010_101
caty = {b, 2'b11, a};      // caty = 010_11_1
catz = {b, 1};             // SAI !!
```

- Gộp đa tầng

```
catr = {4{a}, b, 2{c}};
// catr = 1111_010_101101
```

Toán tử quan hệ

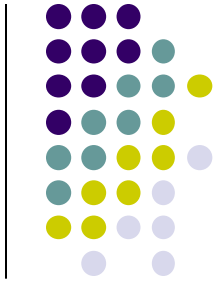


- `>` → **lớn hơn**
- `<` → **nhỏ hơn**
- `>=` → **lớn hơn hoặc bằng**
- `<=` → **nhỏ hơn hoặc bằng**

- Kết quả chỉ là một bit: *0*, *1* or *x*

```
1 > 0          → 1
4'b1x1 <= 0    → x
10 < z         → x
```

Toán tử bằng



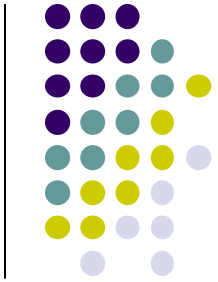
- == → bằng logic
- != → không bằng logic
- === → bằng trường hợp
- !== → không bằng trường hợp

Trả về *0, 1* or *x*

Trả về *0* or *1*

- `4'b 1z0x == 4'b 1z0x` → *x*
- `4'b 1z0x != 4'b 1z0x` → *x*
- `4'b 1z0x === 4'b 1z0x` → *1*
- `4'b 1z0x !== 4'b 1z0x` → *0*

Ví dụ so sánh với X & Z dùng === (!==)



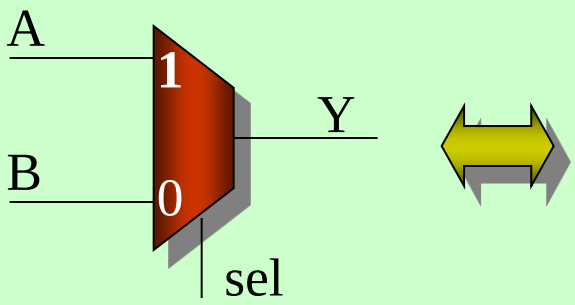
```
If ((a === 1’bx) | (a === 1’bz))
   $display(“Error: At time %t, Giá Trị a = %b”, $time, a);
```

Toán tử điều kiện

- Cú Pháp:

  assign output = (condition_expresion) ? true_expr : false_expr

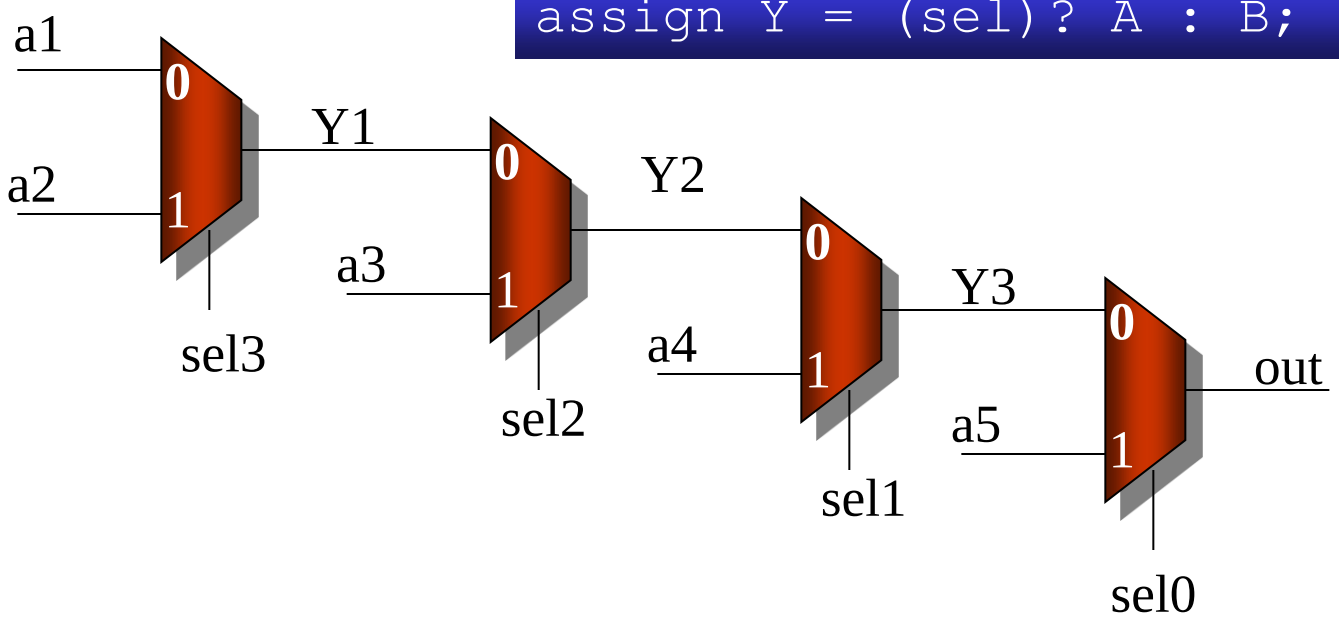
- Như MUX 2 -> 1




```
assign Y = (sel == 1)? A : B;
assign Y = (sel)? A : B;
```

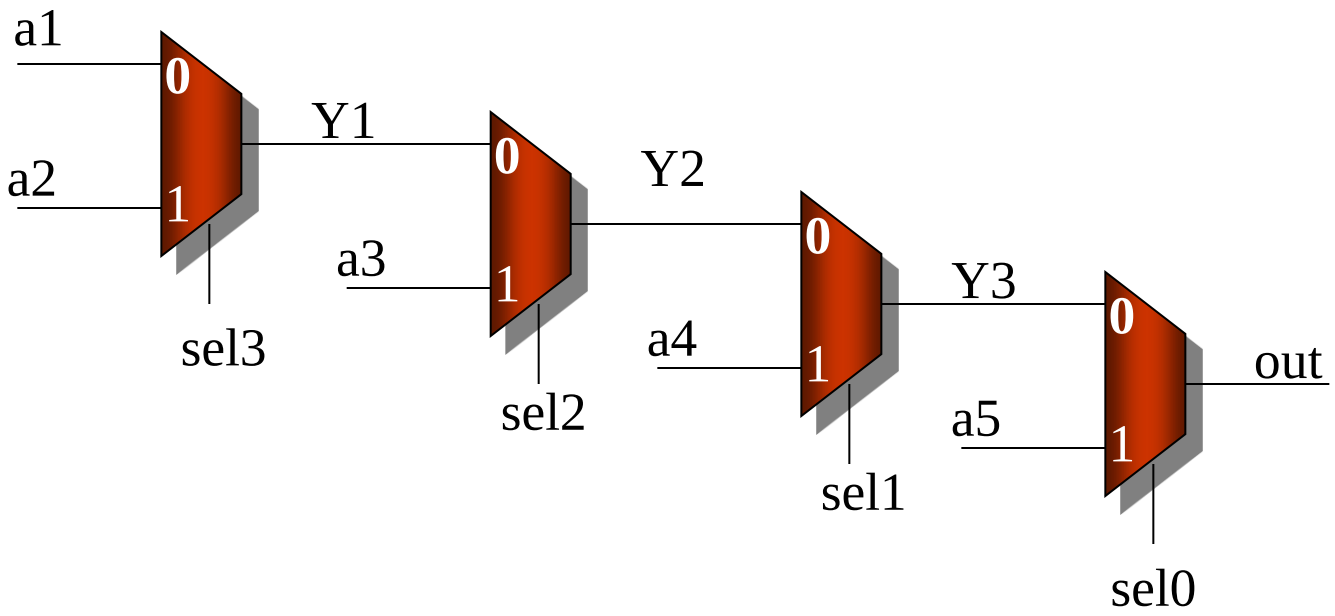
Ví dụ về dùng hàm điều kiện nhiều tầng

```
assign Y = (sel == 1)? A : B;
assign Y = (sel)? A : B;
```



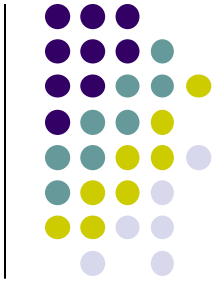
assign out = ??????;

Ví dụ về dùng hàm điều kiện nhiều tầng




```
assign out = sel0 ? a5 : sel1 ? a4 : sel2 ? a3 : sel3 ? a2 : a1;
```

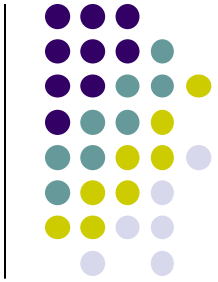
Toán tử toán học (1)



- **+(Cộng), -(Trừ) , *(Nhân), /(chia), %(Phần dư), **(lũy thừa)**
- Nếu bất kỳ toán hạn là x, kết quả sẽ là x
- Các thanh ghi âm:
  - Thanh ghi có thể gán với giá trị âm nhưng được xử lý như không dấu

```
reg [15:0] regA;
..
regA = -4'd12;    // được lưu như 2^16-12 = 65524
c = regA/3        // c có giá trị 21861
```

Toán tử toán học (2)



- Các thanh ghi âm:
  - Có thể gán các giá trị âm
  - Việc xử lý khác nhau dựa trên tài liệu cơ bản (base specification) hoặc không (no base specification)

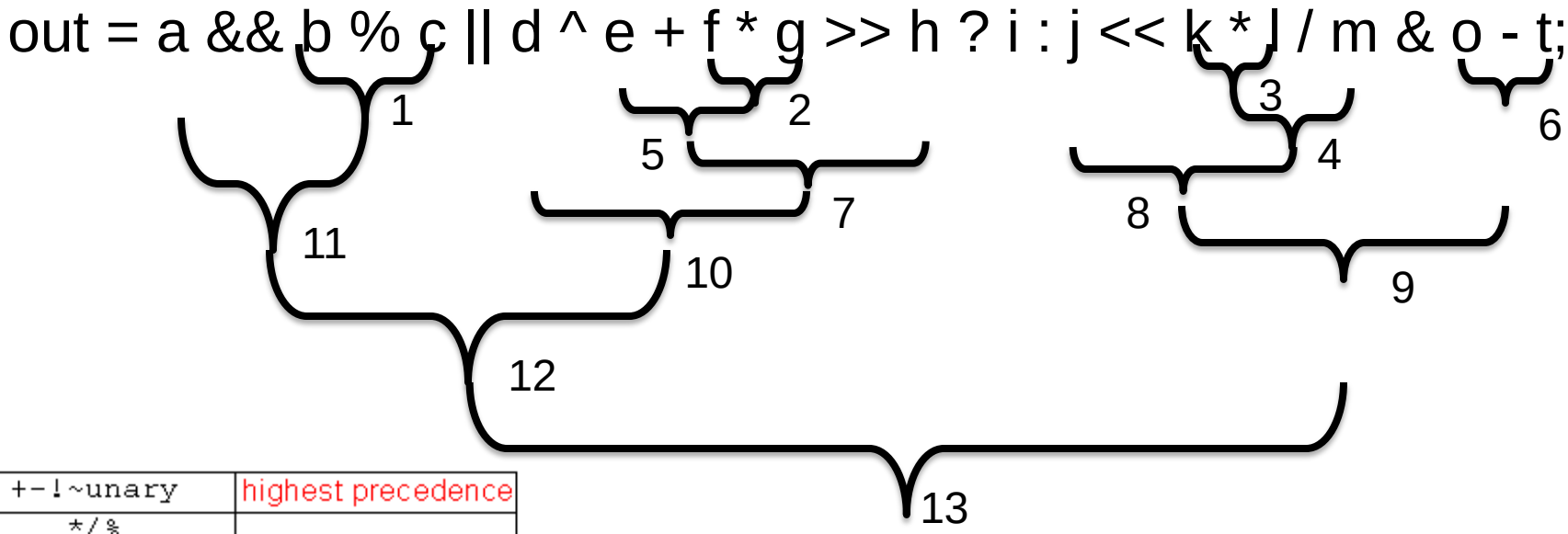
```
reg [15:0] regA;
integer intA;
..
intA = -12/3;          // evaluates to -4 (no base spec)
intA = -'d12/3;  // evaluates to 1431655761 (base spec)
```

Độ ưu tiên toán tử

+-!~unary	highest precedence
*/%	
+-(binary)	
<< >>	
< <= => >	
== != === !==	
& ~&	
^ ^~ ~^	
| ~|	
&&	
||	
?: conditional	lowest precedence

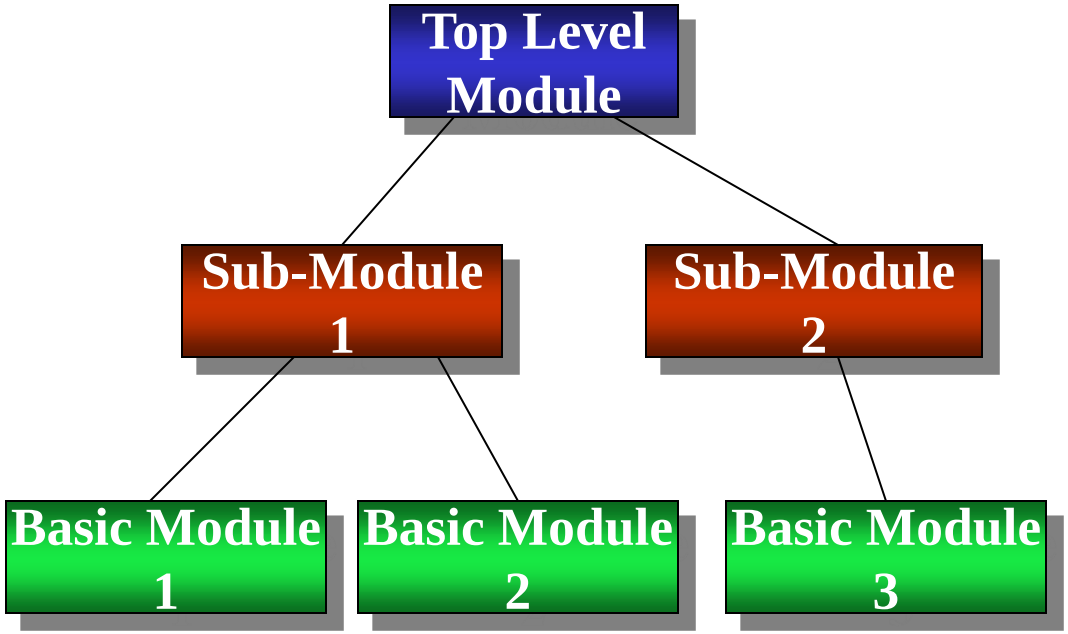
Dùng dấu trong ngoặc
Để yêu cầu độ ưu tiên

Ví dụ về ưu tiên toán tử

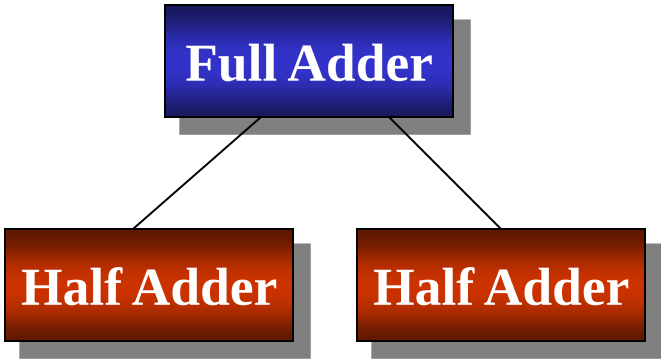


+-!~unary	highest precedence
*/%	
+-(binary)	
<< >>	
< <= => >	
== != === !==	
& ~&	
^ ^~ ~^	
| ~|	
&&	
||	
?: conditional	lowest precedence

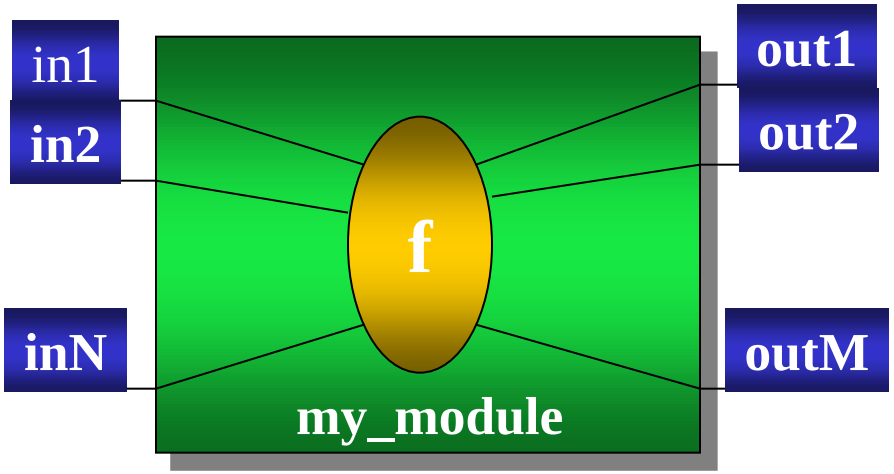
Thiết kế Cấp bậc module (module hierarchy)



E.g.



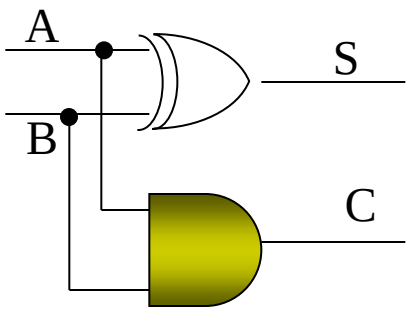
Module




```
module my_module(out1, .., inN);
output out1, .., outM;
input in1, .., inN;

.. // declarations
.. // description of f (maybe
.. // sequential)

endmodule
```

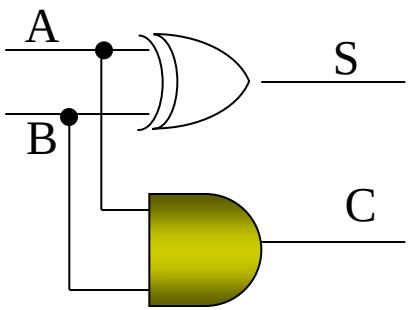
Mọi thứ phải được viết bên trong module
Ngoại trừ: compiler directives
Example: `timescale 1n/1p

Ví dụ: Half Adder



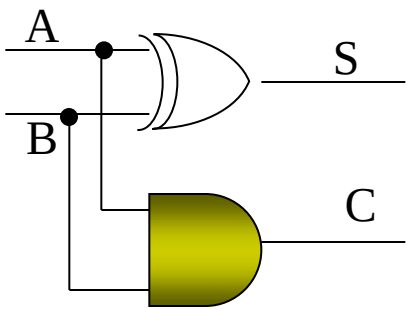

```
module half_adder (S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

Ví dụ: Half Adder



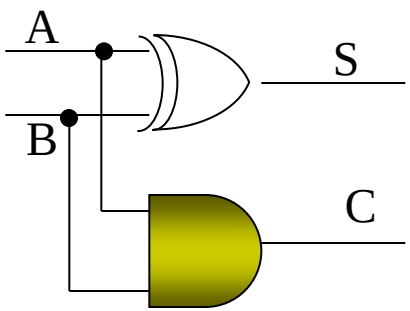

```
module half_adder (S, C, A, B);
output S, C;
input A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

Ví dụ: Half Adder



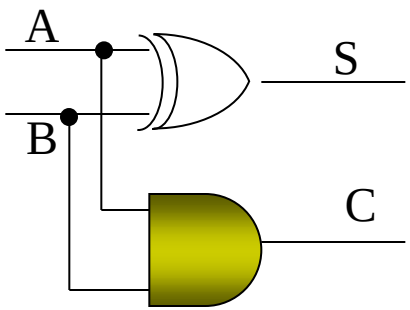

```
module half_adder (
output S, output C,
input A, input B
);

assign S = A ^ B;
assign C = A & B;

endmodule
```

Ví dụ: Half Adder



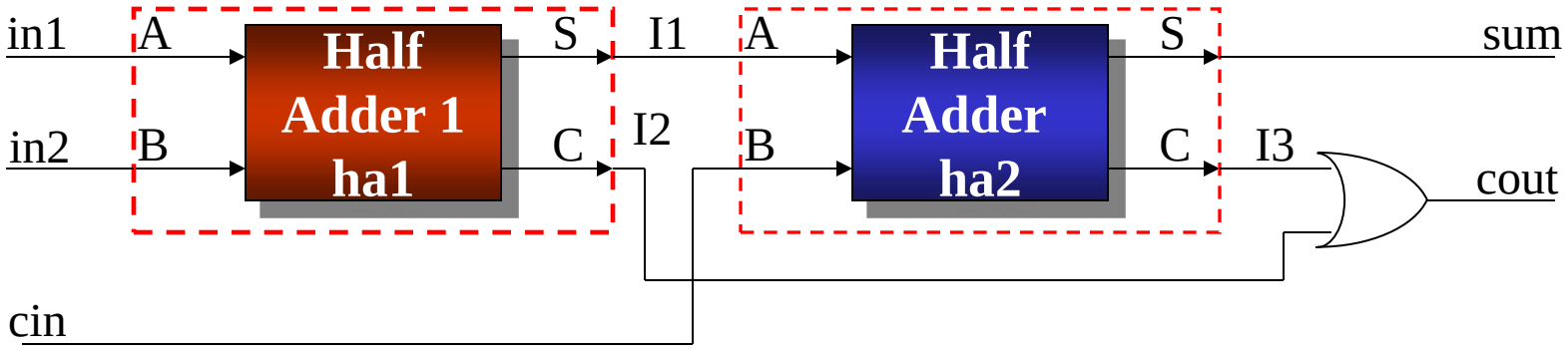

```
module half_adder (
output S, output C,
input A, input B
);

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```

Ví dụ: Half Adder




```
module half_adder (
output wire S, output wire C,
input wire A, input wire B );

assign S = A ^ B;
assign C = A & B;

endmodule
```

Ví dụ: Bộ cộng toàn phần

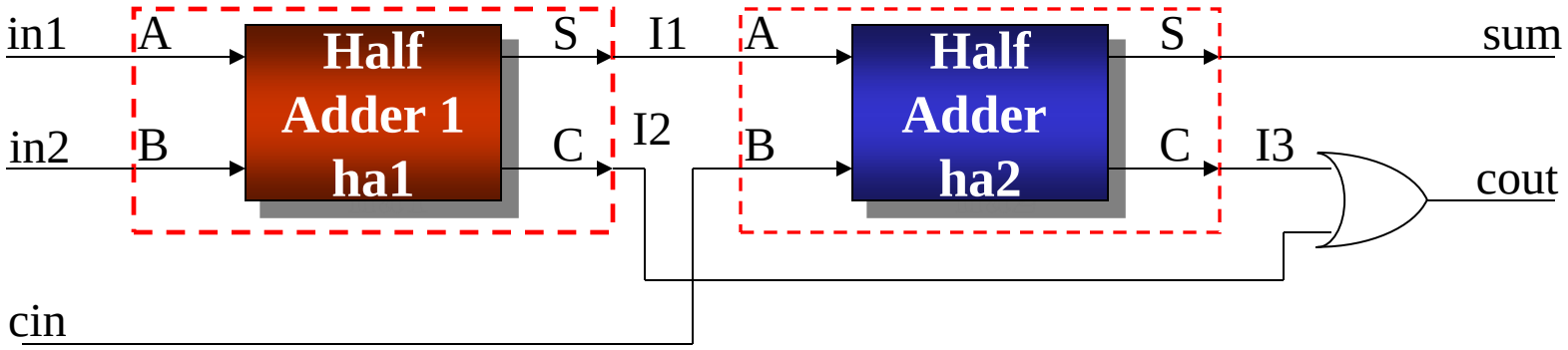



```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(.S(I1), .C(I2), .A(in1), .B(in2));
half_adder ha2(.S(sum), .C(I3), .A(I1), .B(cin));

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

Ví dụ: Bộ cộng toàn phần

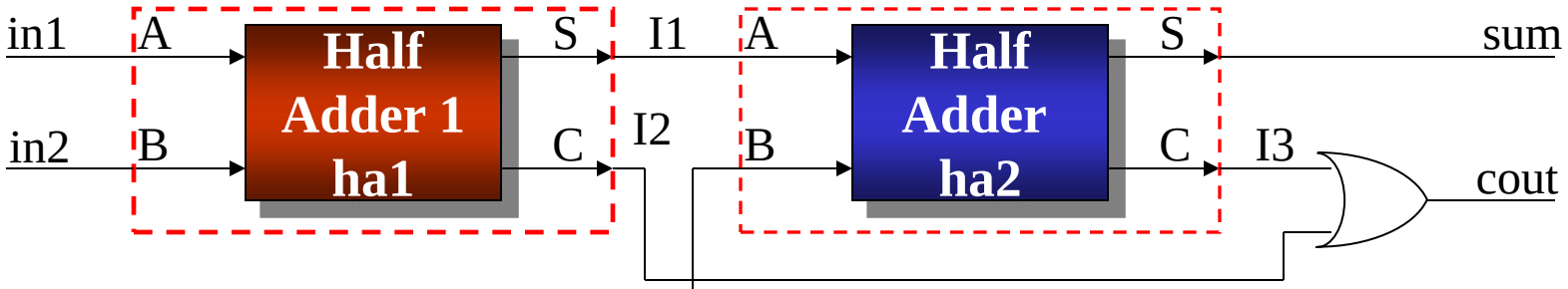



```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(.S(), .C(1'b0), .A(1'b1), .B(in2));
half_adder ha2(.S(sum), .C(I3), .A(I1), .B(cin));

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

Ví dụ: Bộ cộng toàn phần

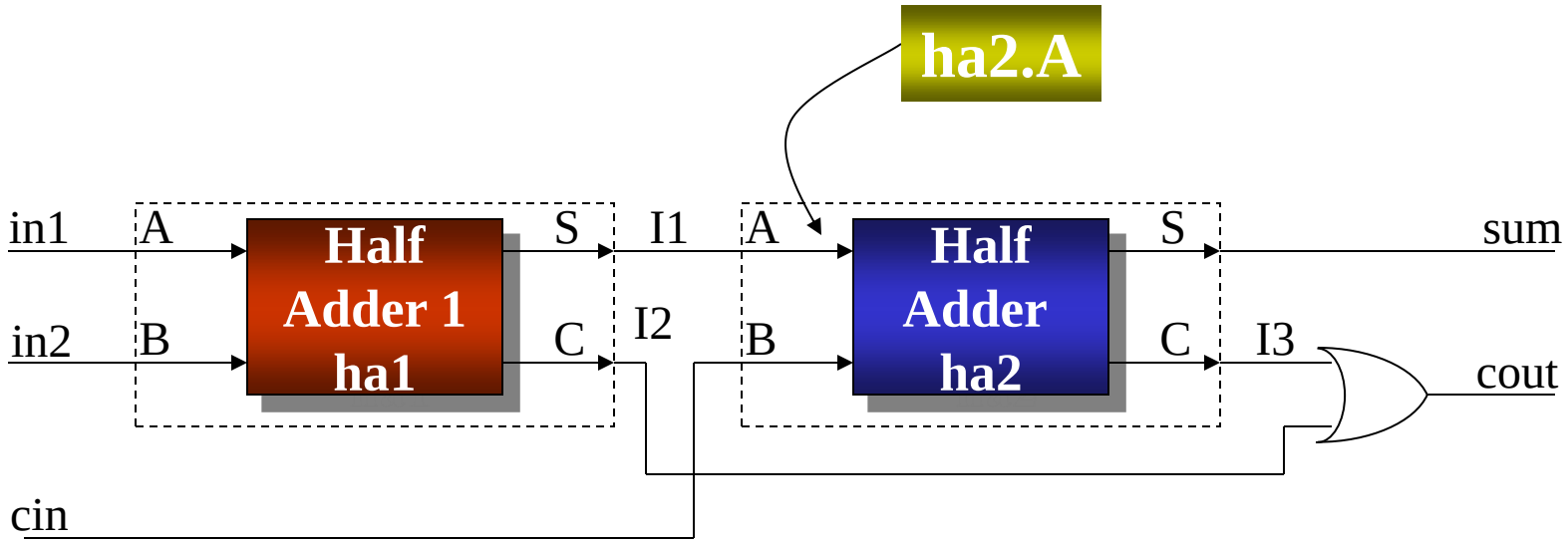



```
module full_adder(sum, cout, in1, in2, cin);
output sum, cout;
input in1, in2, cin;

wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(I1, I2, in1, in2);
half_adder ha2(sum, I3, I1, cin);

assign cout = I2 || I3;

// print signal A inside ha2
$display("%h", ha2.A);

endmodule
```

Module name

Instance name

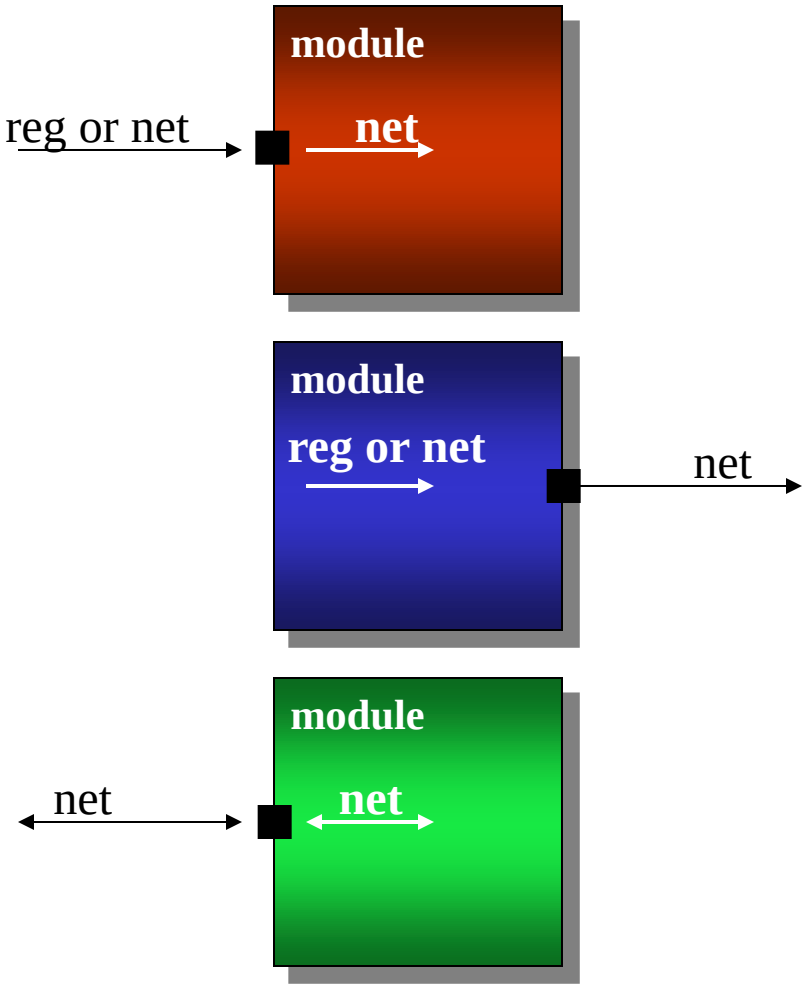
Các tên Hierarchy



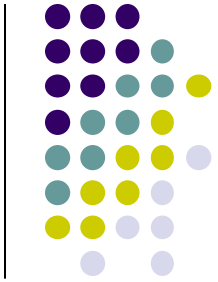
Nhớ dùng tên nhúng (instance names),
Không dùng tên module (module names)

Gán chân vào ra (ports)

- **Inputs**
- **Outputs**
- **Inouts**

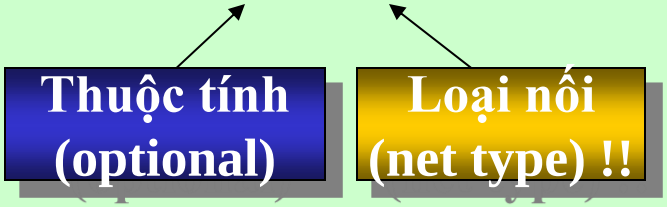


Phép gán liên tục (continuous assignment)



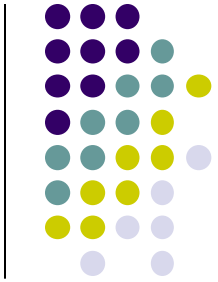
- Syntax:

```
assign #delay <output> =<expr>; Ex: assign #3 y = a & b;
```



- Nơi diễn tả:
  - Trong module
  - Bên ngoài procedures
- Đặc điểm:
  - Thực thi song song
  - Thứ tự độc lập
  - Kích hoạt liên tục

Mô hình cấu trúc (Mức cổng)



- Xây dựng ở mức cổng:

  `and, nand, nor, or, xor, xnor, buf, not, bufif0, bufif1, notif0, notif1`

- Sử dụng:

  `nand (out, in1, in2);` NAND 2 ngõ vào không trễ

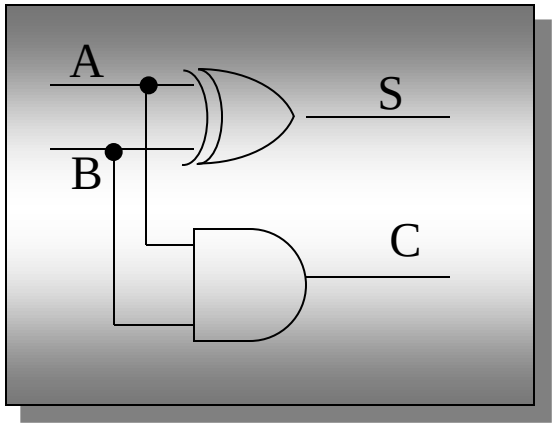
  `and #2 (out, in1, in2, in3);` AND 3 ngõ vào trễ 2 us

  `not #1 N1(out, in);` NOT trễ 1us và tên instace

  `xor X1(out, in1, in2);` XOR 2 ngõ vào và tên instance

- Viết bên trong module, bên ngoài procedures

Ví dụ: Half Adder theo cổng

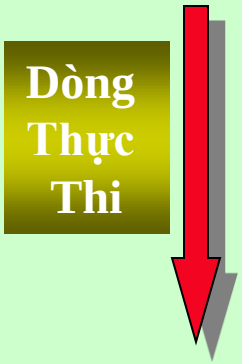


Đảm bảo:
- **XOR: trễ 2 us**
- **AND: trễ 1 us**

```
module half_adder(S, C, A, B);
output S, C;
input A, B;

wire S, C, A, B;

xor #2 (S, A, B);
and #1 (C, A, B);

endmodule
```

Mô hình hành vi – Procedure (1)

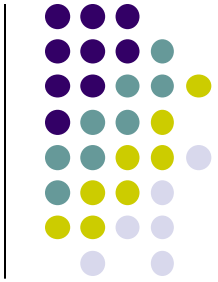
- Procedures = phần code được thực thi một cách tuần tự
- Diễn tả Procedural = diễn tả bên trong một procedure (chúng thực thi tuần tự)
- Kiểu diễn tả khác của MUX 2-1:




```
begin
  if (sel == 0)
      Y = B;
  else
      Y = A;
end
```

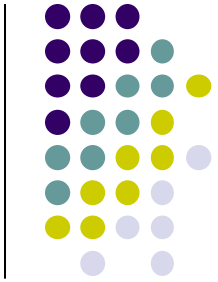
Phép gán Procedural
Y phải là thanh ghi !!

Mô hình hành vi – Procedure (2)



- Module có thể chứa bất cứ số lượng procedure
- Procedures thực thi song song (lên hệ với procedure khác) và ..
- .. Có thể diễn tả trong hai loại khối:
  - **initial → thực thi chỉ một lần**
  - **always → thực thi mãi (đến khi mô phỏng hoàn thành)**

Khối "initial"



- Bắt đầu thực thi ở thời gian 0 và hoàn thành khi diễn tả cuối cùng được thực thi

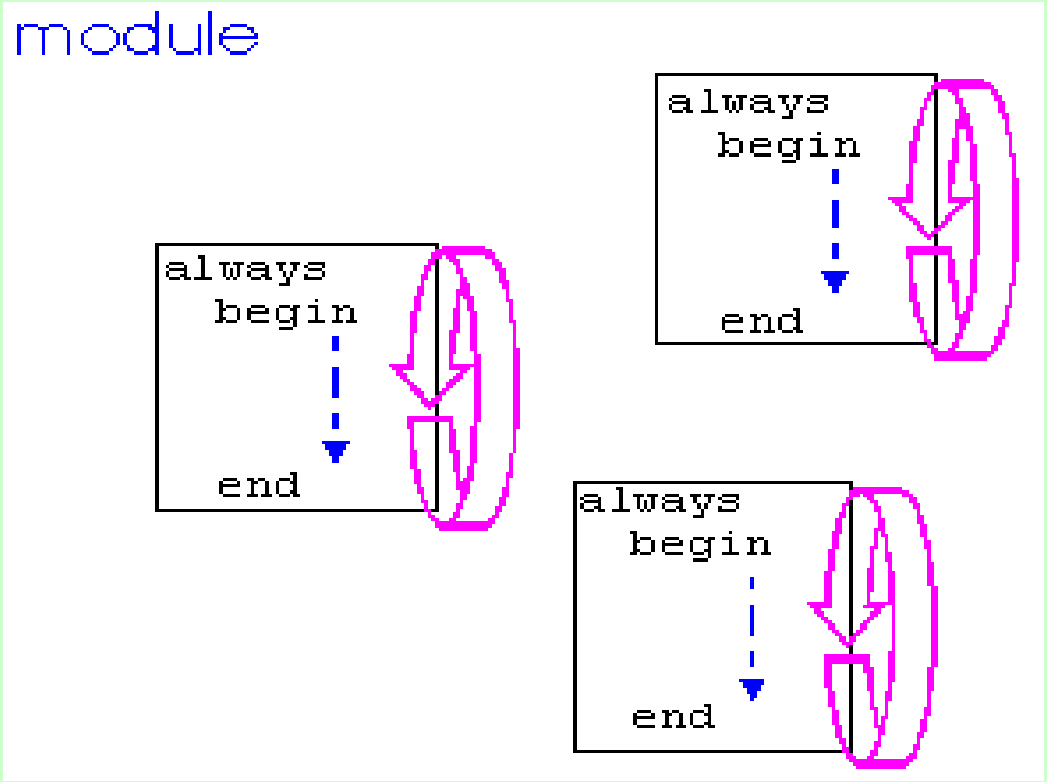
```
module nothing;

   initial
     begin
      #20;
      $display("I'm first");
     end

   initial begin
      #50;
      $display("Really?");
      end

   endmodule
```

Sẽ hiển thị ở thời Gian mô phỏng = 20

Sẽ hiển thị ở thời Gian mô phỏng = 50

Khối “always”

- Bắt đầu thực thi ở thời gian 0 và kết thúc khi thời gian mô phỏng hoàn thành



Sự kiện – Events (1)

Sensitivity list
(list biến nhạy)

- @

Bất cứ tín hiệu nào thay đổi làm begin..end

```
always @(signal1 or signal2 or ..) begin
      ..
      end
```

Positive edge (cạnh lên)

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

negative edge (cạnh xuống)

```
always @(negedge clk) begin
      ..
      end
```

Thực thi khi bất cứ tín Hiệu bất cứ thay đổi

Thực thi khi bất cứ tín Hiệu clk thay đổi từ 0 lên 1

Thực thi khi bất cứ tín Hiệu clk thay đổi từ 1 xuống 0

Ví dụ

- Half Adder – Cách 3

```
module half_adder(S, C, A,
   B);
output S, C;
input A, B;

reg S,C;
wire A, B;

always @(A or B) begin
   S = A ^ B;
   C = A && B;
   end
endmodule
```

- Hành Vi Latch

```
module latch(Q, D, enable);
output Q;
input D, enable;
reg Q;
wire D, enable;
always @(enable or D)
  begin
     if (enable == 1'b1)
       Q <= D;
     else
       Q <= Q;
  end
endmodule
```

Ví dụ

- Half Adder – Cách 3

```
module half_adder(S, C, A,
   B);
output S, C;
input A, B;

reg S,C;
wire A, B;

always @(A or B) begin
   S = A ^ B;
   C = A && B;
   end
endmodule
```

- Hành vi kích cạnh của DFF (asyn reset)

```
module dff(Q, D, Clk, rst_n);
output Q;
input D, Clk, rst_n;
reg Q;
wire D, Clk;
always @(posedge clk or
   negedge rst_n)
  begin
     if (rst_n == 1'b0)
       Q <= 1'b0;
     else
        Q <= D;
  end
endmodule
```

Ví dụ

- Half Adder – Cách 3

```
module half_adder(S, C, A,
   B);
output S, C;
input A, B;

reg S,C;
wire A, B;

always @(A or B) begin
   S = A ^ B;
   C = A && B;
   end

endmodule
```

- Hành vi kích cạnh của DFF (sync reset)

```
module dff(Q, D, Clk, rst_n);
output Q;
input D, Clk, rst_n;
reg Q;
wire D, Clk;
always @(posedge clk)
  begin
     if (rst_n == 1'b0)
       Q <= 1'b0;
     else
        Q <= D;
  end
endmodule
```

Ví dụ

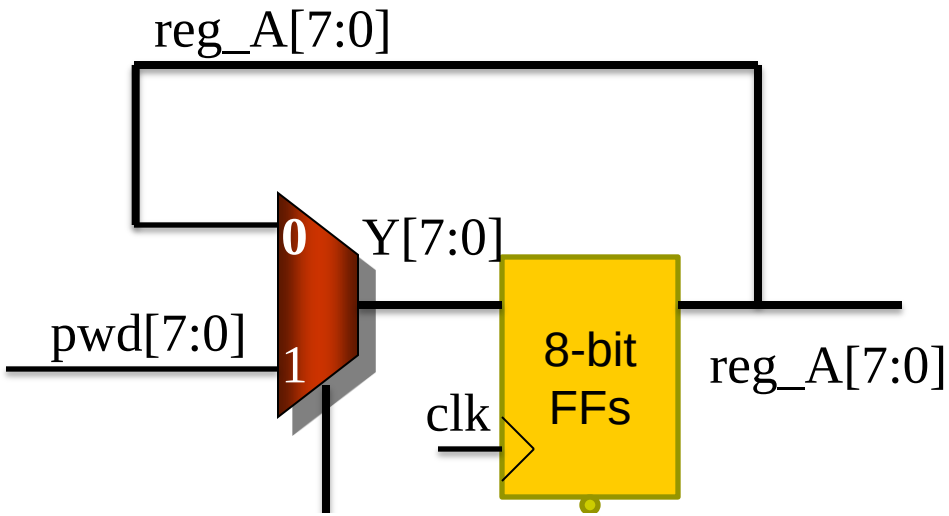
- Half Adder – Cách 3

```
module half_adder(S, C, A,
   B);
output S, C;
input A, B;

reg S,C;
wire A, B;

always @(*) begin
   S = A ^ B;
   C = A && B;
   end

endmodule
```

- Hành vi kích cạnh của DFF

```
module dff(Q, D, Clk,
   rst_n);
output Q;
input D, Clk, rst_n;
reg Q;
wire D, Clk;
always @(posedge clk or
   negedge rst_n)
  begin
      if (rst_n == 1'b0)
        Q <= 1'b0;
      else
         Q <= D;
  end

endmodule
```

Ví dụ Coding cho mạch Register

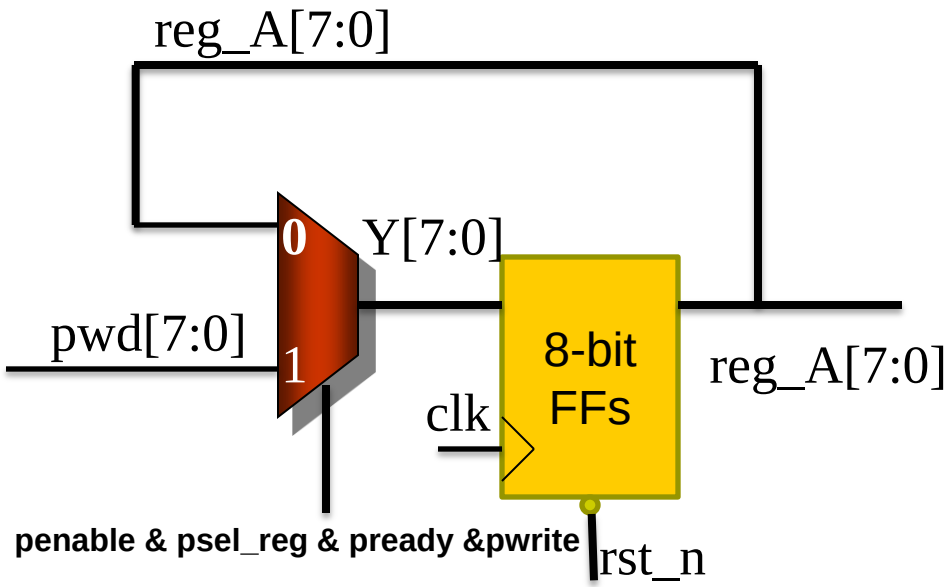


penable & psel_reg & pready &pwrite

- Hành vi kích cạnh của DFF

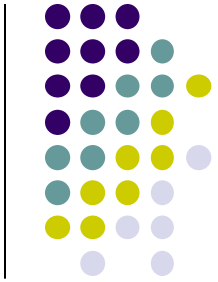
```
module dff(Q, D,
   Clk, rst_n);
output Q;
input D, Clk,
   rst_n;
reg Q;
wire D, Clk;
always @(posedge
   clk or negedge
   rst_n)
  begin
     if (rst_n ==
   1'b0)
        Q <= 1'b0;
     else
        Q <= D;
  end
endmodule
```

Ví dụ Coding Register




```
always @(posedge clk or negedge rst_n)
  begin // can delete
    if (!rst_n)
      reg_A <= 8'h0;
    else if (penable & psel_reg & pready &pwrite)
      reg_A <= pwd;
    else
      reg_A <= reg_A;
  end // can delete
```

Sự kiện – Events (2)

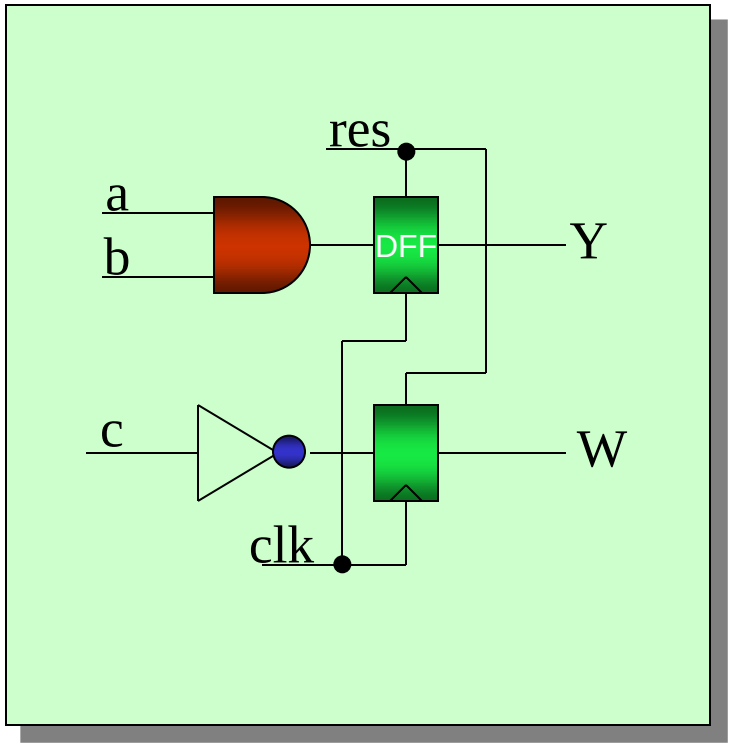


- wait (expr)

```
always @(...) begin
   wait (ctrl)
   #10 cnt = cnt + 1;
   #10 cnt2 = cnt2 + 2;

   end
```

- Kích mức của DFF thì sao?

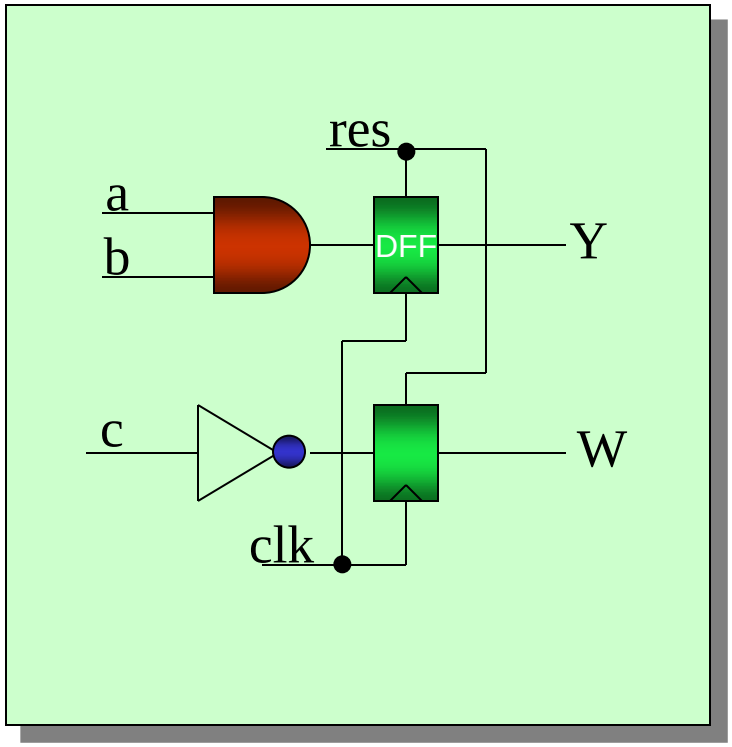
Ví dụ



Viết Verilog cho mạch bên trái

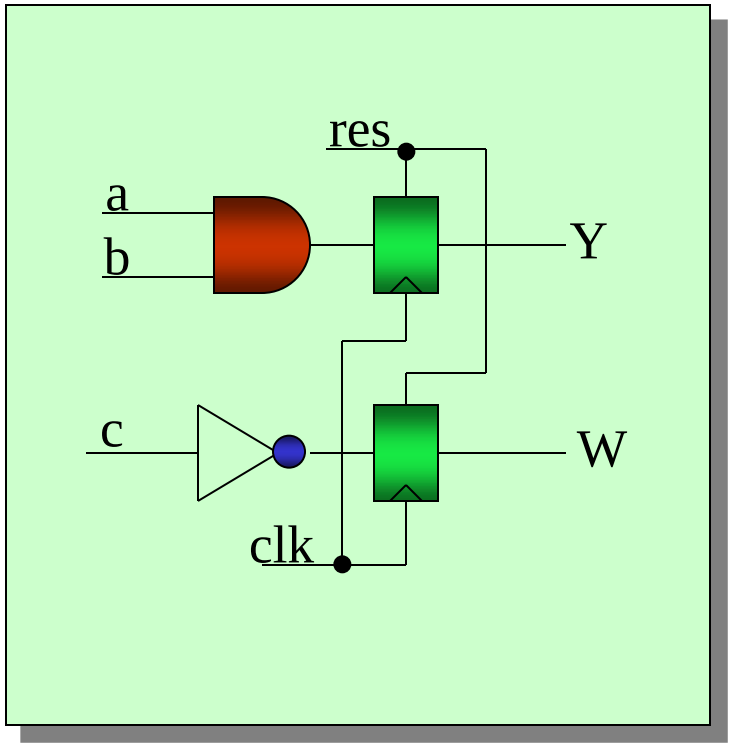
Ví dụ

Viết Verilog cho mạch bên trái

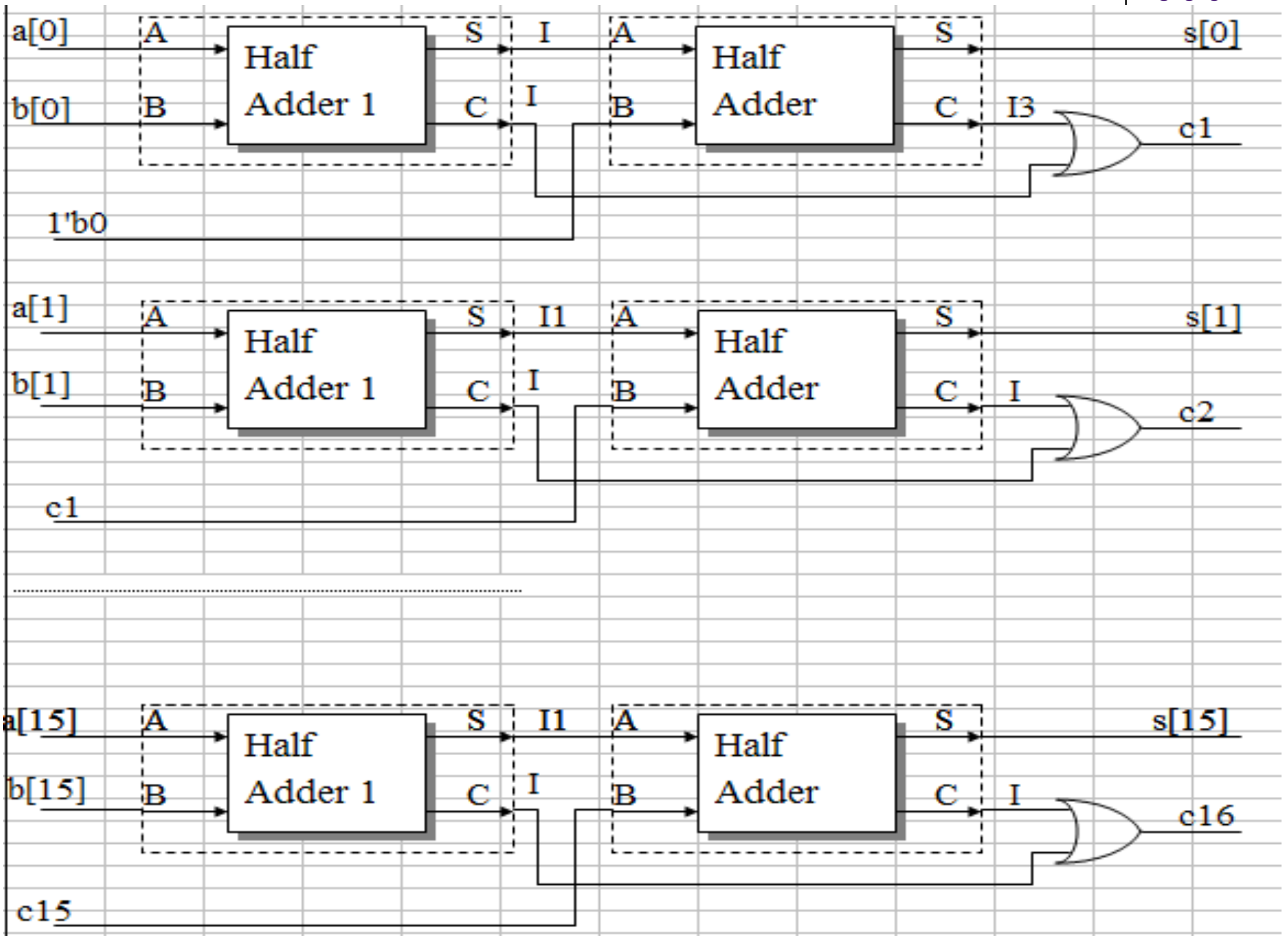



```
always @(res or posedge clk) begin
        if (res) begin
                Y = 0;
                W = 0;
                end
        else begin
                Y = a & b;
                W = ~c;
                end
        end
```

Ví dụ

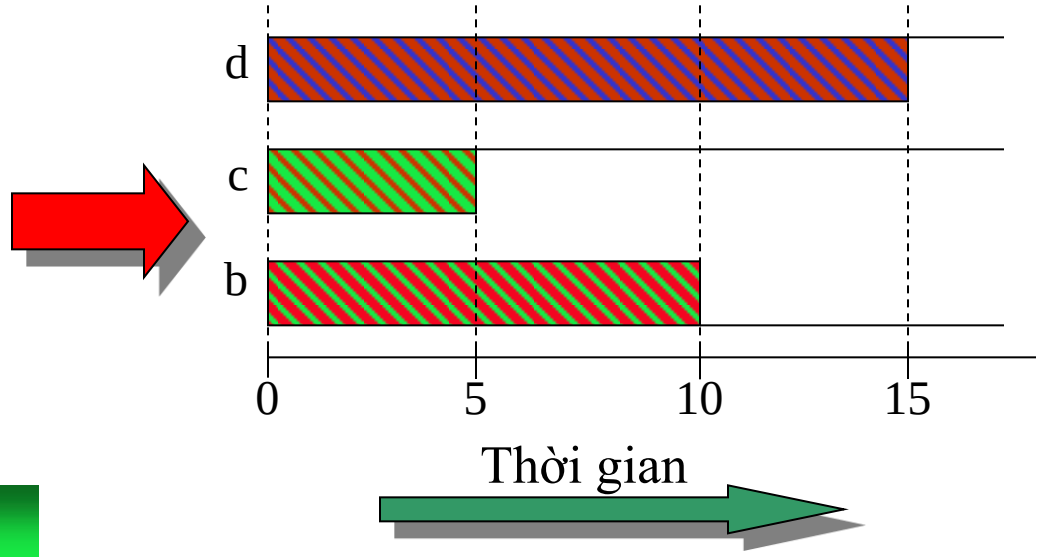



```
module sample (a, b, c, Y, W, clk,
rst);

input a, b, c, clk, res;
output Y,W;

wire a,b,c, clk, res;
reg Y,W;

always @(posedge res or posedge clk)
  begin
       if (res) begin
                Y <= 0;
                W <= 0;
                end
       else begin
                Y <= a & b;
                W <= ~c;
                end
  end
```


a[0]
b[0]
A
B
Half
Adder 1
S
C
I
I
A
B
Half
Adder
S
C
I3
s[0]
c1
1'b0
a[1]
b[1]
A
B
Half
Adder 1
S
C
I1
I
A
B
Half
Adder
S
C
I
s[1]
c2
c1
a[15]
b[15]
A
B
Half
Adder 1
S
C
I1
I
A
B
Half
Adder
S
C
I
s[15]
c16
c15

Thời gian – timing (1)

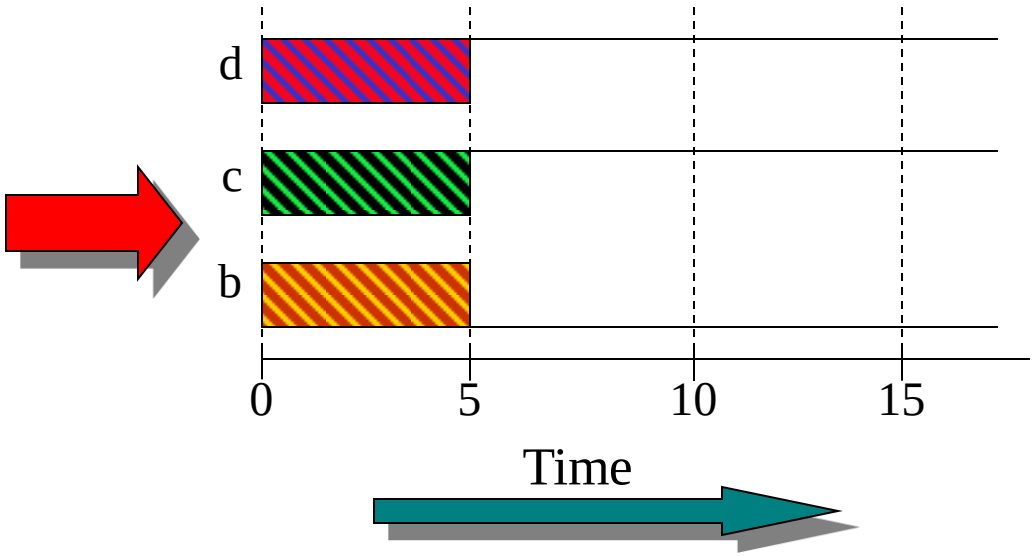
```
initial begin
        #5 c = 1;
        #5 b = 0;
        #5 d = c;
        end
```



Mỗi phép gán là một khối đằng trước nó

Thời gian – timing (2)

```
initial begin
        fork
        #5 c = 1;
        #5 b = 0;
        #5 d = c;
        join
        end
```



Phép gán không hình thành khối ở đây

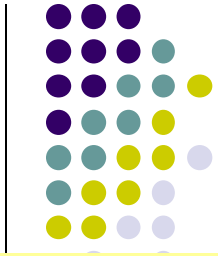
Diễn tả Procedure "if"

```
if (expr1)
        true_stmt1;

else if (expr2)
        true_stmt2;
..
else
        def_stmt;
```

Mux: 4-1:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        if (sel == 0)
                out = in[0];
        else if (sel == 1)
                out = in[1];
        else if (sel == 2)
                out = in[2];
        else
                out = in[3];
endmodule
```

Diễn tả Procedure "case"

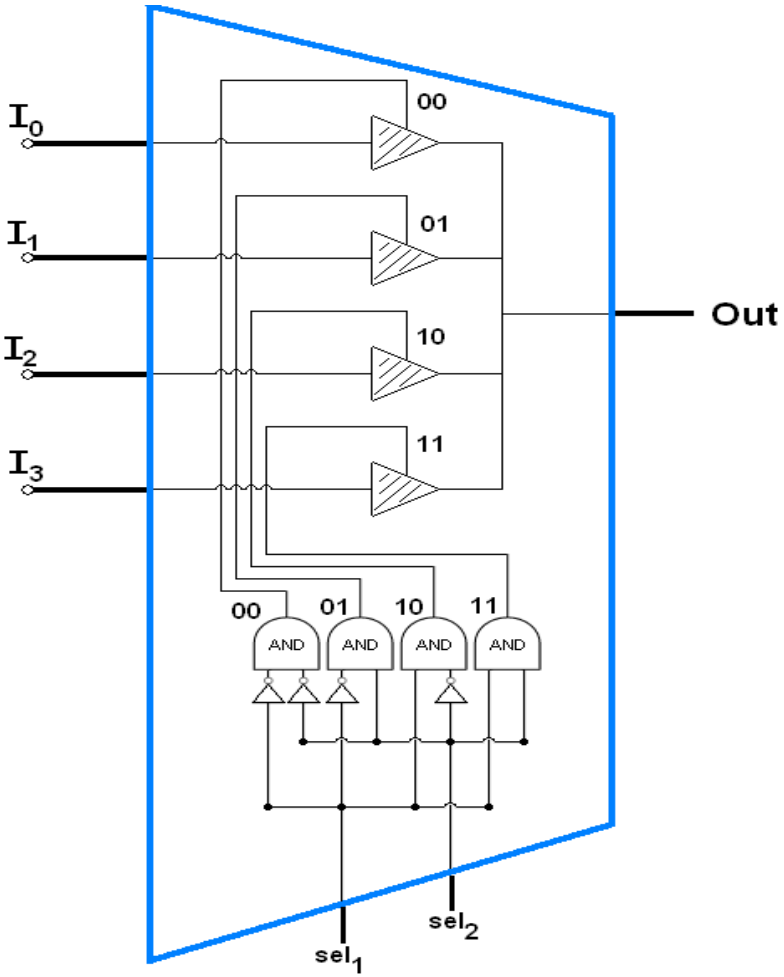


```
case (expr)

item_1, .., item_n:        stmt1;
item_n+1, .., item_m:  stmt2;
..
default:                        def_stmt;

endcase
```

Mux 4-1:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        case (sel)
          0: out = in[0];
          1: out = in[1];
          2: out = in[2];
          3: out = in[3];
         default: out = 1'bx;
        endcase
endmodule
```

Diễn tả Procedure "case"



Mux 4-1:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        case (sel)
          2'b00: out = in[0];
          2'b01: out = in[1];
          2'b10: out = in[2];
          2'b11: out = in[3];
         default: out = 1'bx;
        endcase
endmodule
```

Diễn tả Procedure “casex”

```
casex (expr)

item_1, .., item_n:       stmt1;
item_n+1, .., item_m:  stmt2;
..
default:                        def_stmt;

endcase
```

Mux 4-1:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        casex (sel)
          2'b00: out = in[0];
          2'b01: out = in[1];
          2'b10: out = in[2];
          2'b11: out = in[3];
         default: out = 1'bx;
        endcase
endmodule
```

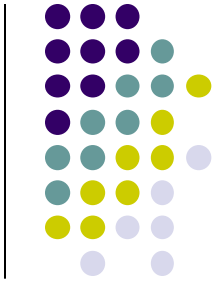
Diễn tả Procedure "casez"

```
casez (expr)

item_1, .., item_n:        stmt1;
item_n+1, .., item_m:  stmt2;
..
default:                        def_stmt;

endcase
```

Mux 4-1:

```
module mux4_1(out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;

reg out;
wire [3:0] in;
wire [1:0] sel;

always @(in or sel)
        casez (sel)
          //2'b?0: out = in[0];
           2'b00: out = in[0];
           2'b10: out = in[0];
          2'b01: out = in[2];
          2'b11: out = in[3];
         default: out = 1'bx;
        endcase
endmodule
```

Diễn tả Procedure “for”



for (gán giá trị ban đầu; điều kiện; gán nhảy)
diễn tả;

E.g.

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start)
        for (i = 0; i < 3; i = i + 1)
                #10 Y = Y + 1;
endmodule
```

Diễn tả Procedure “while”

while (expr) stmt;

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;
integer i;

initial
        Y = 0;

always @(posedge start) begin
        i = 0;
        while (i < 3) begin
                #10 Y = Y + 1;
                i = i + 1;
                end
        end
endmodule
```

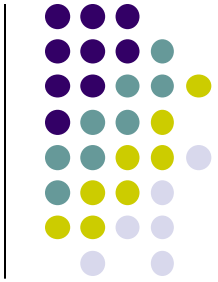
Diễn tả Procedure “repeat”

repeat (thời gian) diễn tả;

Có thể là một số nguyên hay là một biến

```
module count(Y, start);
output [3:0] Y;
input start;

reg [3:0] Y;
wire start;

initial
        Y = 0;

always @(posedge start)
        repeat (4) #10 Y = Y + 1;
endmodule
```

Diễn tả Procedure "forever"



forever diễn_tả;

Thực thi cho đến khi thời gian mô phỏng hoàn tất

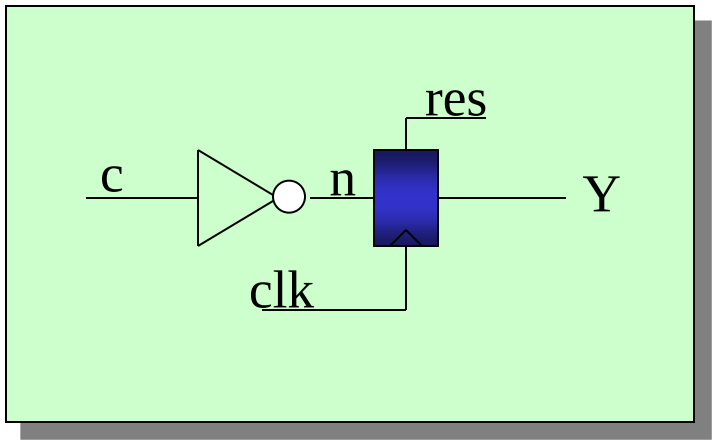
Ví dụ điển hình:

Tạo clock trong module kiểm tra

```
module test;

reg clk;

initial begin
        clk = 0;
        forever #10 clk = ~clk;
        end

other_module1 o1(clk, ..);
other_module2 o2(.., clk, ..);

endmodule
```

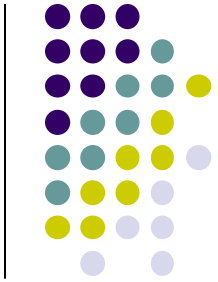
T_{clk} = 20 đơn vị thời gian

Mô hình hỗn hợp – Mixed model



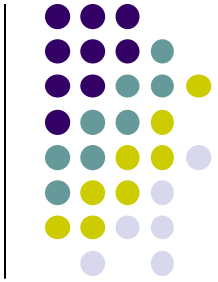

```
module simple(Y, c, clk, res);
output Y;
input c, clk, res;

reg Y;
wire c, clk, res;
wire n;

not(n, c); // gate-level

always @(res or posedge clk)
       if (res)
              Y = 0;
       else
              Y = n;
endmodule
```

Các Task hệ thống – system tasks



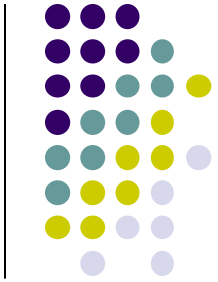
- $display(“..”, arg2, arg3, ..); → như printf(), hiển thị chuỗi được định dạng trong ngọ ra chuẩn khi gặp
- $monitor(“..”, arg2, arg3, ..); → giống $display(), nhưng hiển thị chuỗi khi arg2, arg3, .. thay đổi
- $stop; → treo mô phỏng khi gặp
- $finish; → hoàn tất mô phỏng khi gặp
- $fopen(“filename”); → trả về file descriptor (integer); rồi có thể dùng $fdisplay(fd, “..”, arg2, arg3, ..); hoặc $fmonitor(fd, “..”, arg2, arg3, ..); để viết đến file
- $fclose(fd); → đóng file
- $random(seed); → trả về giá trị ngẫu nhiên; cho seed mot so integer ngẫu nhiên

Các Task hệ thống – system tasks



```
always @(add or sub)
  if (a == 32'h cafecafe) begin
    $display("I want to print the value of signal a = %h", a);
  end

$monitor("I want to print the value of signal a = %h", a);
reg [7:0] a;
a = $random();

reg [7:0] b;
b = $random(0);
c =  b * 32;
```

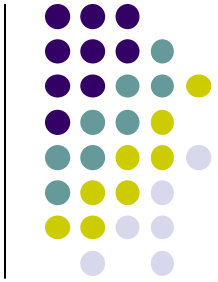
Định dạng chuỗi $display & $monitor



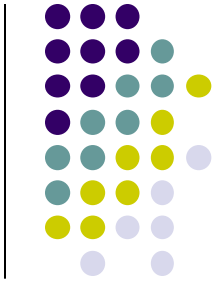
Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Compiler Directives

- `include “filename” → chèn nội dung file vào file hiện tại; viết nó bất kỳ trong code ..
- `define <text1> <text2> → đặt text1 là text2;
  - VD. `define BUS reg [31:0]` trong phần khai báo: `BUS data;`
- `timescale <time unit>/<precision>
  - VD. `timescale 10ns/1ns` sau: `#5 a = b;`

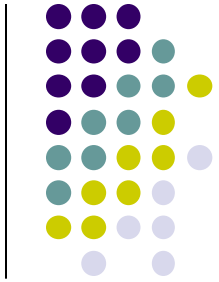


```
`define 2_BIT

`ifdef 2_BIT
……. // verilog code
`else
……..// verilog code
`endif


`ifndef 2_BIT
……. // verilog code
`else
……..// verilog code
`endif
```



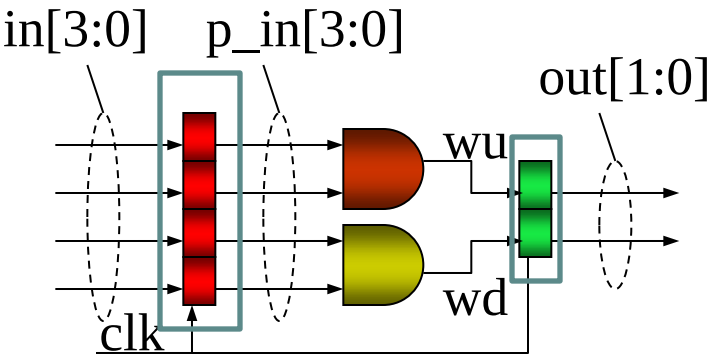
```
`define TEST_1

`ifdef TEST_1
//……. // verilog code
`include “test1.vt”
`endif
`ifdef TEST_2
……. // verilog code
`endif
`ifdef TEST_3
……. // verilog code
`endif
`ifdef TEST_4
……. // verilog code
`endif
```

Using generate



```
generate if (USE_A) begin : use_a
   // Add code here
end else begin : use_b
  // Add code here
end
endgenerate
```

Thông số - Parameters (1)



Không dùng thông số

```
module dff4bit(Q, D, clk);
output [3:0] Q;
input [3:0] D;
input clk;

reg [3:0] Q;
wire [3:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

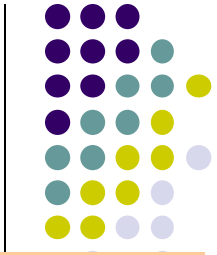
```
module dff2bit(Q, D, clk);
output [1:0] Q;
input [1:0] D;
input clk;

reg [1:0] Q;
wire [1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

Thông số - Parameters (2)

Không dùng thông số (tiếp)

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;    // internal nets
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff4bit instA(p_in, in, clk);
dff2bit instB(out, {wu, wd}, clk);
// notice the concatenation!!

endmodule
```

Thông số - Parameters (3)



Dùng thông số

```
module dff(Q, D, clk);
parameter WIDTH = 8;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff instA(p_in, in, clk);
       defparam instA.WIDTH = 4;

dff instB(out, {wu, wd}, clk);
       defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```

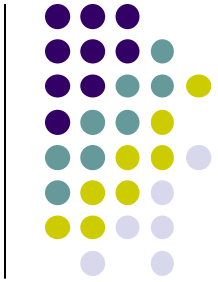
Thông số - Parameters (3)

Dùng thông số

```
module dff(Q, D, clk);
parameter WIDTH = 8;
output [WIDTH-1:0] Q;
input [WIDTH-1:0] D;
input clk;

reg [WIDTH-1:0] Q;
wire [WIDTH-1:0] D;
wire clk;

always @(posedge clk)
       Q = D;

endmodule
```

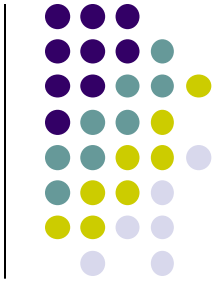
```
module top(out, in, clk);
output [1:0] out;
input [3:0] in;
input clk;

wire [1:0] out;
wire [3:0] in;
wire clk;

wire [3:0] p_in;
wire wu, wd;

assign wu = p_in[3] & p_in[2];
assign wd = p_in[1] & p_in[0];

dff #(.WIDTH(4)) instA (p_in, in,
clk);
      // defparam instA.WIDTH = 4;

dff #(.WIDTH(2)) instB (out, {wu,
wd}, clk);
       //defparam instB.WIDTH = 2;
// We changed WIDTH for instB only

endmodule
```

Kiểm tra module



```
module top_test;

wire [1:0] t_out;          // Top's signals
reg [3:0] t_in;
reg clk;

top inst(t_out, t_in, clk); // Top's instance

initial begin                          // Generate clock
      clk = 0;
      forever #10 clk = ~clk;
end

initial begin                          // Generate remaining inputs
      $monitor($time, " %b -> %b", t_in, t_out);
      #5 t_in = 4'b0101;
      #20 t_in = 4'b1110;
      #20 t_in[0] = 1;
      #300 $finish;
end

endmodule
```

Khai báo “task”



A *task* can be seen as a named piece of code with defined *inputs*, *outputs* and *inouts*. It allows a more abstract view of the specified system and assures that the same piece of code can be reused in other design sections.

To specify a *task*, follow this procedure:

1. Specify the code that is to be encapsulated as a task. If there is more than one statement inside, enclose it in the **begin** and **end** brackets.

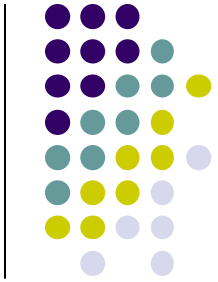
2. Determine the role of each variable inside the code: *inputs*, *inouts*, *outputs* and *local* or *temporal* variables. Specify them before the code listing.

3. Define a name for the task and specify it after the keyword **task** but preceding the declarations. Add the **endtask** keyword after the code.

```
task taskname;
  declarations_of_arguments;
  task_statements
endtask
```

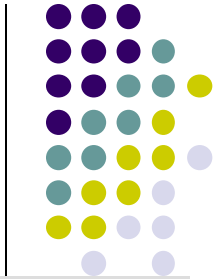
```
task factorial;
  output [31:0] OutFact;
  input [3:0] n;

integer Count;

  begin
    OutFact = 1;
    for (Count=n; Count>0; Count=Count-1)
      OutFact = OutFact * Count;
   end

endtask
```

Task – Gọi “task”



```
task SomeTask;
   input r, s;
   output w, x;

   begin
      w = r + s;
      x = r - s;
   end
endtask

SomeTask ( r , s , w , x )
```

Khai báo “function”



Function Declaration

The differences between *tasks* and *functions* ar reflected in the way they are defined and invoked. First of all a *function* returns a single result that is available through the function's invocation. This requires an assignment to a local variable that has the same name as the function itself. Any other assignment target in local variables cannot return more than one output.

Second, the result of a function is by default a 1-bit register. If a function of any other registered type is needed (either integer, real, realtime or a vector), it must be specified in the function header

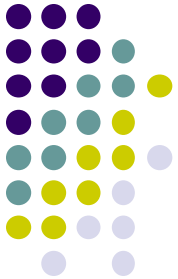
Third, the declarative part of a function may contain only declarations of inputs and local variables – no outputs or inouts are permitted.

Finally, a function starts with the **function** keyword and is terminated with the **endfunction** keyword.

```
function [function_type] function_name
  declarations_of_inputs;
  [declarations_of_local_variables];
  begin
    behavioral_statements;
    function_name = expression;
  end
endfunction
```

```
function [31:0] Factorial;
  input [3:0] Operand;
  reg [3:0] i;
  begin
   Factorial = 1;
    for (i = 2; i <= Operand; i = i + 1)
      Factorial = i * Factorial;
  end
endfunction

function ParityCheck;
  input [3:0] Data;
  begin
   ParityCheck = ^Data;
  end
endfunction
```

Funtion – gọi “function”



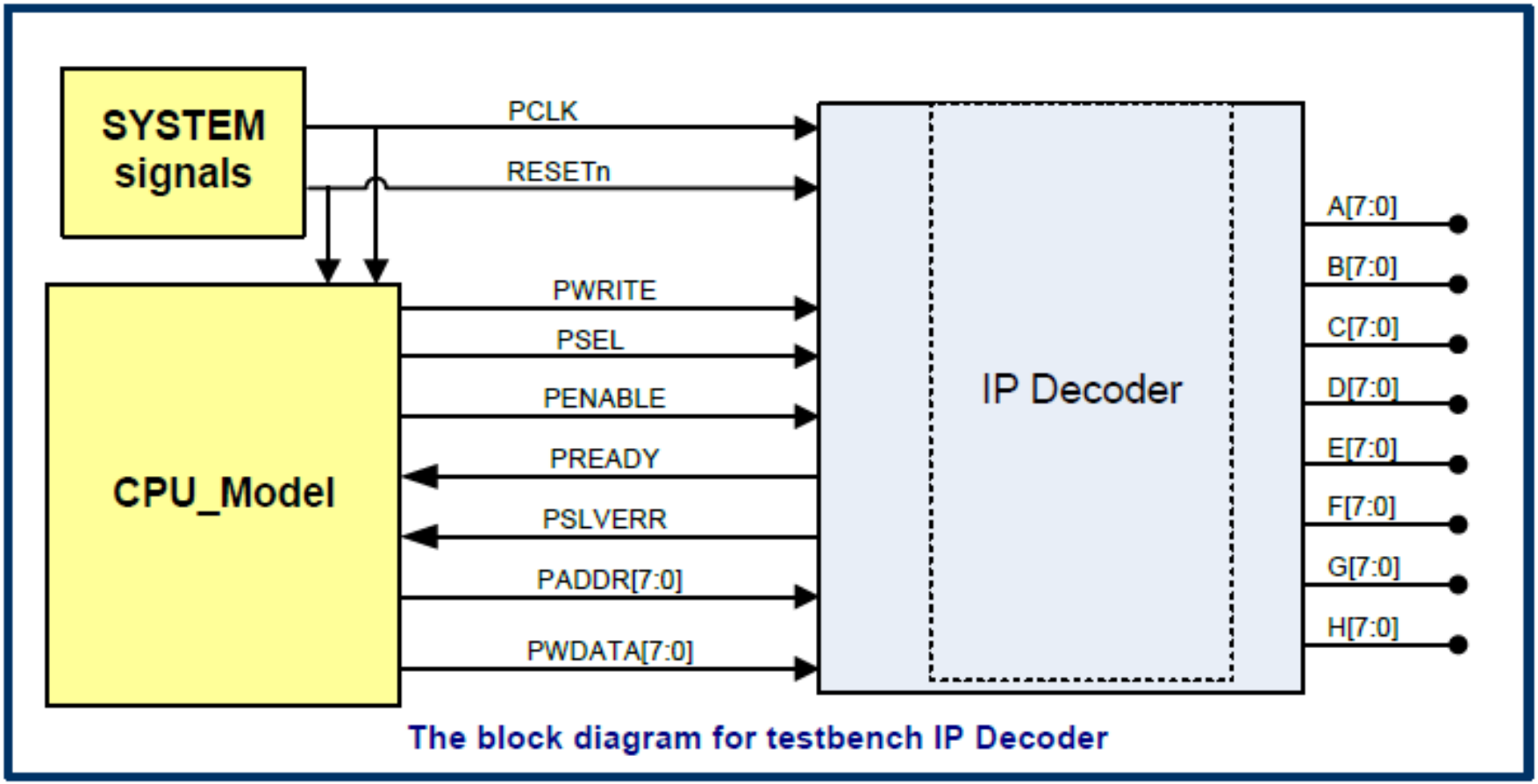
```
function ParityOdd;
  input [15:0] DataVectorLong;
  begin
    ParityOdd = ^DataVectorLong;
  end
endfunction
```

```
Flags[3] = ParityOdd(RegAX);
```

```
Select = ParityOdd(RegAX) & RegAX[0];
```

```
if (ParityOdd(AddressBus))
  . . .
```

Testbench of IP Decoder



The block diagram for testbench IP Decoder

APB Write Protocol Transfer

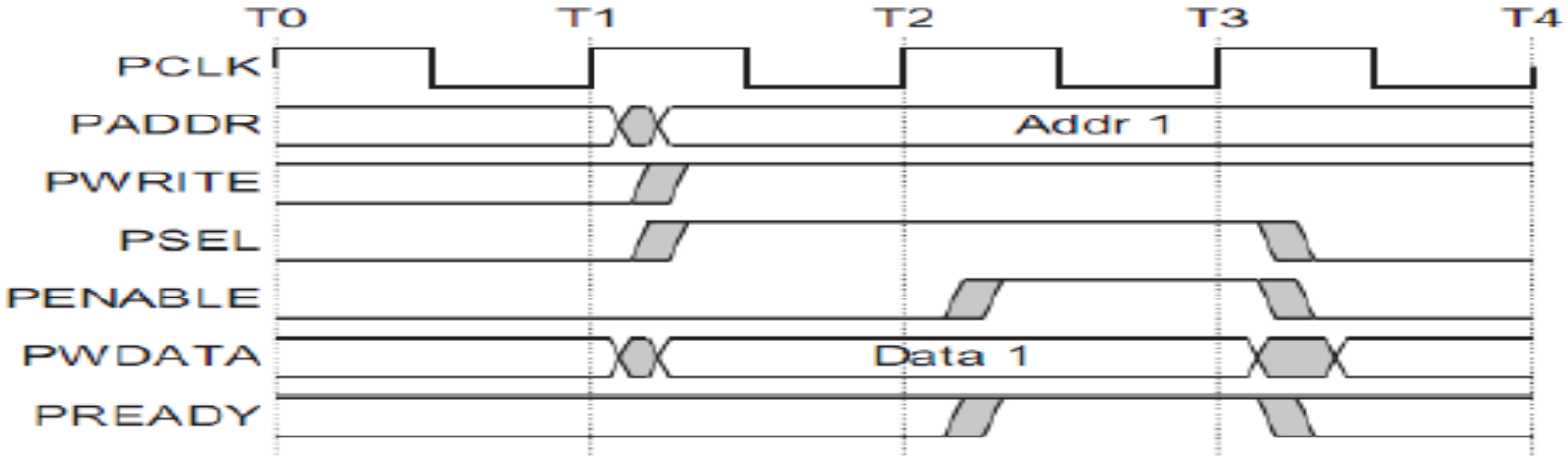


Figure 4-1: Write transfer with no wait states

MOV

T2: Generate penable to 1

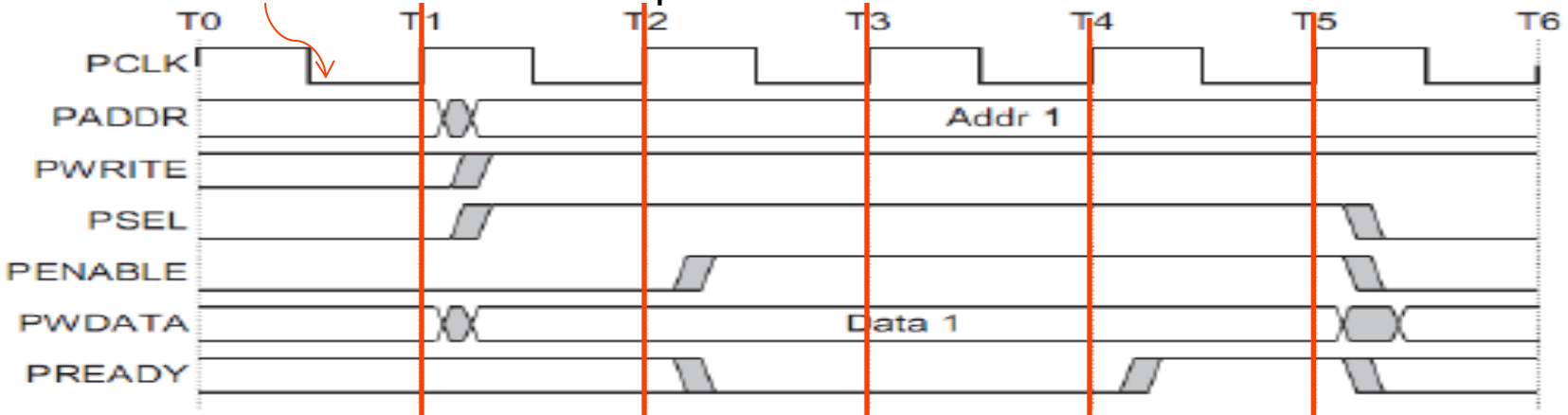
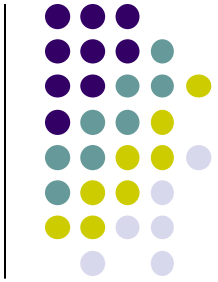


Figure 4-2: Write transfer with wait states

T1: Input address, data, psel = 1, pwrite

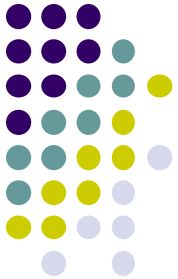
T3: Check pready = 1, if not, wait until it is 1

```
module cpu_model (clk, rst_n, psel, pwrite, penable,  paddr, pdata, pready, pslverr);
input clk,…..;
output psel,……;

task MOVT;
input [7:0] address;
input [7:0] data;
  begin
    @(posedge clk);
    ……................
end
Endtask

Task ADD;
Endtask

Task SUB;
Endtask

Task MUL;
Endtask

Task DIV;
endtask
endmodule
```



```
module cpu_model (clk, rst_n, psel, pwrite, penable,
paddr, pdata, pready, pslverr);
input clk,…..;
output psel,……;

task MOVT;
input [7:0] address;
input [7:0] data;
  begin
    @(posedge clk);
    #1;
    psel = 1'b1; pwrite = 1'b1; penable =1'b0; pdata = data; paddr= address;

 @(posedge clk);
    #1;
    penable = 1'b1;

@(posedge clk);
   while (pready !=1) @(posedge clk); // check pready = 1? Neu it is 0, wait until it is 1

    #1;
    psel =1'b0; pwrite = 1'b0; penable = 1'b0; pdata = 8'h00; paddr = 8'h00;

end
endtask
endmodule
```

```
module cpu_model (clk, rst_n, psel, pwrite, penable,
paddr, pdata, pready, pslverr);
input clk,…..;
output psel,……;

task MOVT;
input [7:0] address;
input [7:0] data;
  begin
    @(posedge clk);
    #1;    psel = 1'b1; pwrite = 1'b1; penable =1'b0; pdata = data; paddr= address;

 @(posedge clk);
    #1;    penable = 1'b1;

//@(posedge clk);
    if (pready == 0)
       @(posedge pready) @(posedge clk); // check pready = 1? Neu it is 0, wait until it is 1
    else
       @(posedge clk);
    #1;
    psel =1'b0; pwrite = 1'b0; penable = 1'b0; pdata = 8'h00; paddr = 8'h00;

end
Endtask
Initial begin
  repeat (500) begin
      data = $random();
      address = $random();
      MOVT({5'h0, address[3:0]} , data) ;
  end
```



Câu Hỏi & Trả Lời