

Nội dung môn học

1. Nhắc lại kiến thức điện tử số (1 LT + 0 BT)
2. Giới thiệu công nghệ IC lập trình được (1 LT + 0 BT)
3. Ngôn ngữ lập trình Verilog HDL (3 LT + 2 BT)
4. Thiết kế mạch số thông dụng (2 LT + 2 BT)
5. Công cụ là quy trình thiết kế với IC. lập trình được của Xilinx (2 LT + 2 BT)
6. Thực hành thiết kế trên FPGA (2 LT + 3 BT)
7. Kiểm tra 1 buổi + dự phòng 1 buổi

Ngôn ngữ lập trình Verilog HDL

- Giới thiệu chung
 - Phương pháp thiết kế dùng Verilog HDL
 - Thiết kế top-down
 - Phần mềm hỗ trợ thiết kế
- Các mô hình thiết kế Verilog
 - Mô hình mức cổng (gate-level)
 - Mô hình mức luồng dữ liệu
 - Mô hình hành vi
- Một số ví dụ cụ thể

Giới thiệu chung

- Phương pháp thiết kế dùng HDL (Hardware Description Languages)

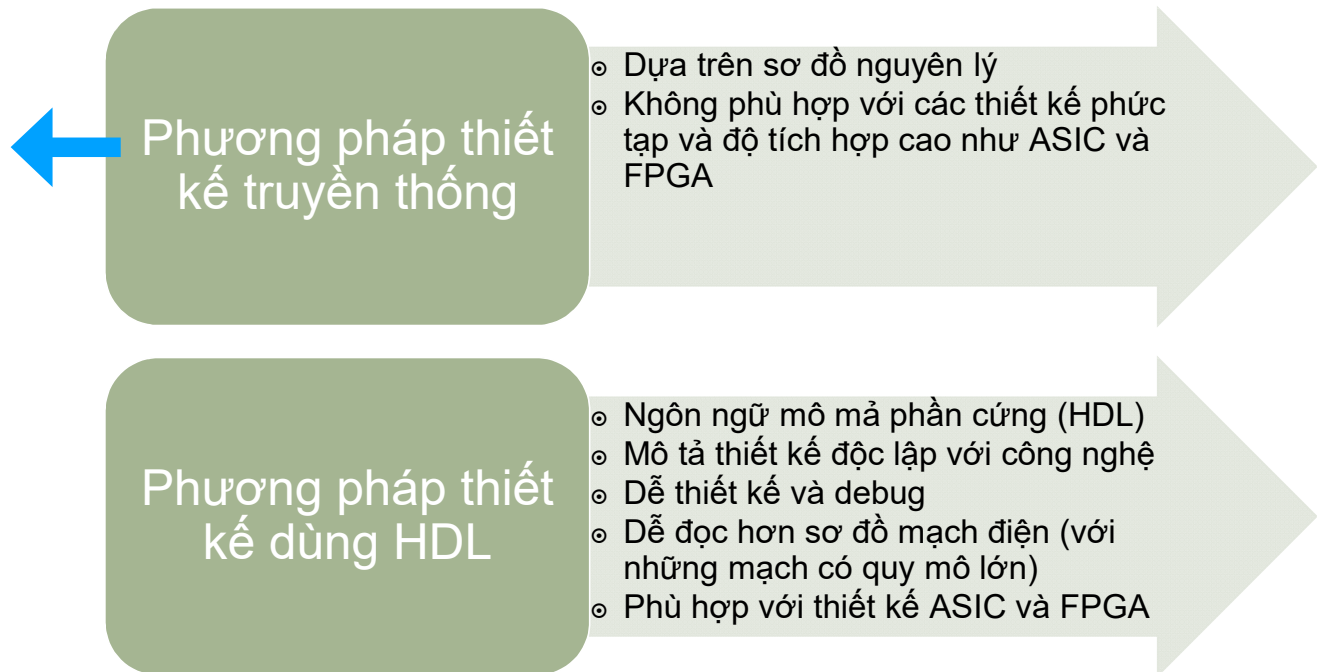
✓ Dễ sử dụng

✓ Dịch thủ công mô tả thiết kế vào

- tập hợp các biểu thức logic
- hoặc sơ đồ nguyên lý



- Cần hiểu rõ về FF và Gate
- Dễ thay đổi thành phần thiết kế
- Thân thiện với người dùng
- Nhưng không phù hợp với mạch tích hợp cao



Verilog và VHDL

- Verilog

- Đơn giản, dễ sử dụng
- Thiếu các cấu trúc cần thiết cho các tham số ở mức hệ thống
- Mô tả đơn giản, hiệu quả, trực quan cho các mạch số
- Cung cấp thiết kế mô hình chuyển mạch

- VHDL

- Phức tạp hơn
- Thích hợp với các thiết kế rất phức tạp
- Phù hợp cho mô hình mức hành vi
- Cung cấp nhiều hàm, thủ tục, thư viện
- Câu lệnh phức tạp

Verilog HDL

- Verilog là ngôn ngữ mô tả phần cứng (Hardware Description Language)
- Phát triển từ năm 1984-1985, công bố trên thế giới 1990, thành chuẩn công nghiệp IEEE năm 1995
- Mô tả mạch số đơn giản, hiệu quả và trực quan
- Dùng để mô tả một số hệ thống số như
 - Network switch
 - Microprocessor
 - Memory
 - Flip-flop
- Có thể dùng HDL để mô tả bất cứ thành phần cứng số nào tại bất kì mức nào

Verilog HDL

- Verilog cho phép thiết kế hệ thống ở 4 mức
 - Mức hành vi: dùng các thuật toán cấu trúc `if`, `case`, `for`, `while`,...
 - Mức thanh ghi - RTL: kết nối bằng biểu thức logic
 - Mức cổng Gate: kết nối bằng các cổng logic AND, OR, NOT,...
 - Mức chuyển mạch: kết nối bằng BJT, FET,...
- Mô tả thiết kế bằng cấu trúc hành vi, chưa cần đưa ra chi tiết về thực hiện thiết kế

Ví dụ các mức độ trừu tượng: mạch chia 2

- Hành vi:

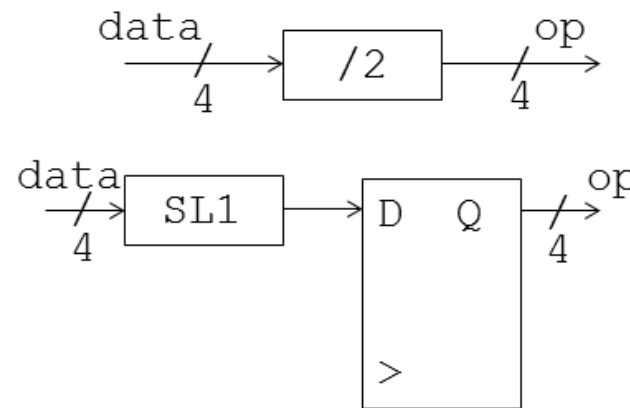
```
always @ (data)
    op <= data / 2;
```

- RTL:

```
always @ (posedge clk)
    op <= data >> 1;
```

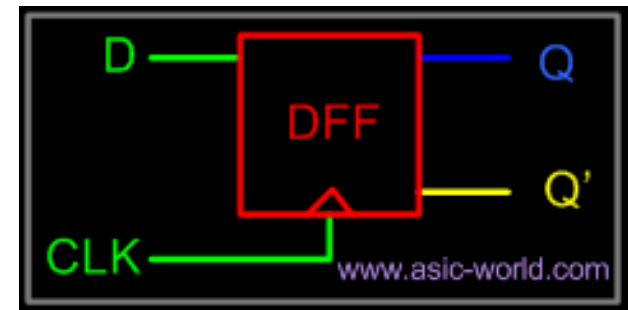
- Nối dây mức cổng

```
FD1 opreg2 (.D(data[3]), .CP(clk), .Q(log[2]));
FD1 opreg1 (.D(data[2]), .CP(clk), .Q(log[1]));
FD1 opreg0 (.D(data[1]), .CP(clk), .Q(log[0]));
FD1 opreg3 (.D(1'b0), .CP(clk), .Q(log[3]));
```



Ví dụ Verilog Code

```
// D flip-flop Code
module d_ff ( d, clk, q, q_bar);
input d ,clk;
output q, q_bar;
  wire d ,clk;
  reg q, q_bar;
  always @ (posedge clk)
  begin
    q <= d;
    q_bar <= ! d;
  end
endmodule
```



Mức độ trừu tượng

Nhập vào thiết kế /
mô phỏng nhanh hơn

Hành vi – Thuật
toán

Ít chi tiết hơn

Mức truyền thanh
ghi

Mức cổng – Cấu
trúc – Sơ đồ dây

Nhập vào thiết kế /
mô phỏng chậm hơn

Mức vật lý –
Silicon – Đa giác

Chi tiết hơn

Các kiểu thiết kế

- Bottom-up Design

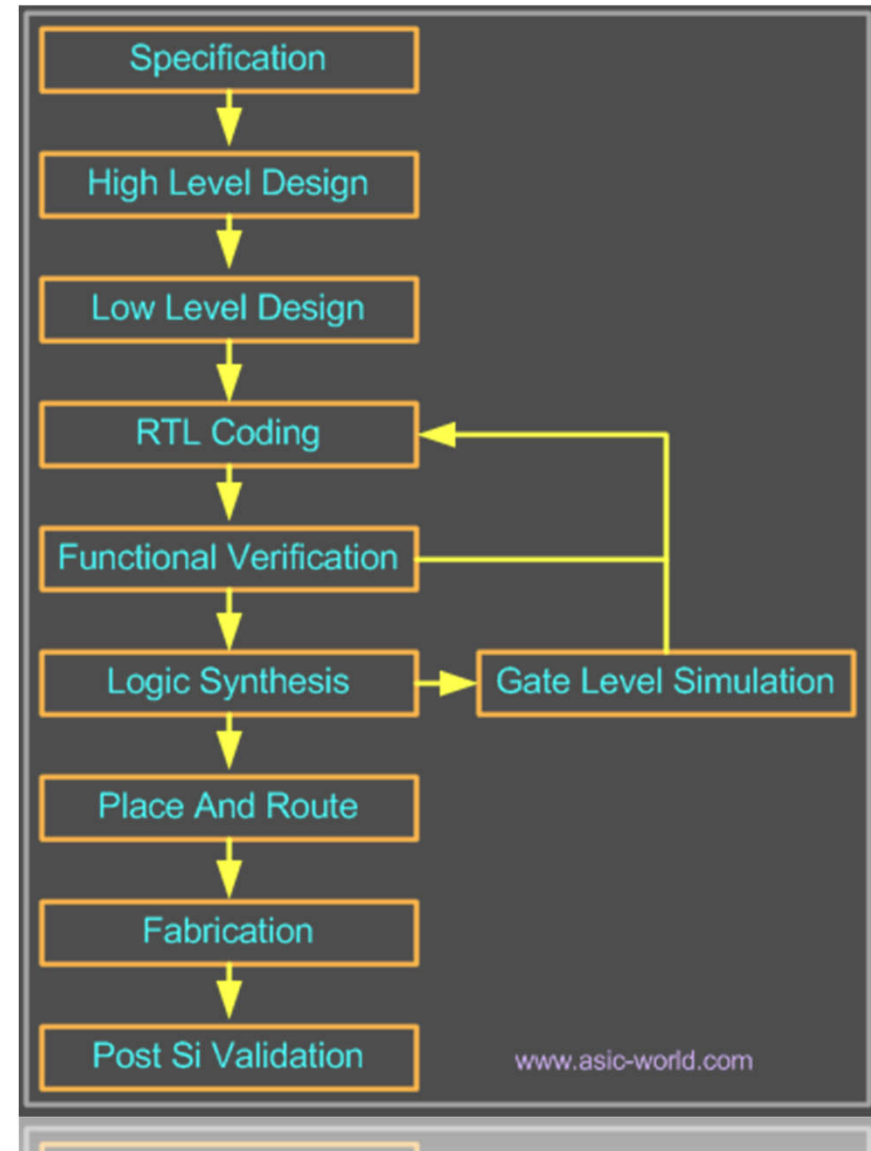
- Phương pháp thiết kế truyền thống
- Thực hiện tại mức Gate
- Không phù hợp với các thiết kế phức tạp cho các mạch ASIC hoặc Vi xử lý
- Cần thêm các cấu trúc mới và phương pháp thiết kế phân cấp

- Top-down Design

- Là phương pháp mong muốn của các nhà thiết kế
- Cho phép kiểm tra sớm, dễ dàng thay đổi công nghệ, thiết kế hệ thống có cấu trúc
- Khó thực hiện nếu không kết hợp với phương pháp Bottom-Up

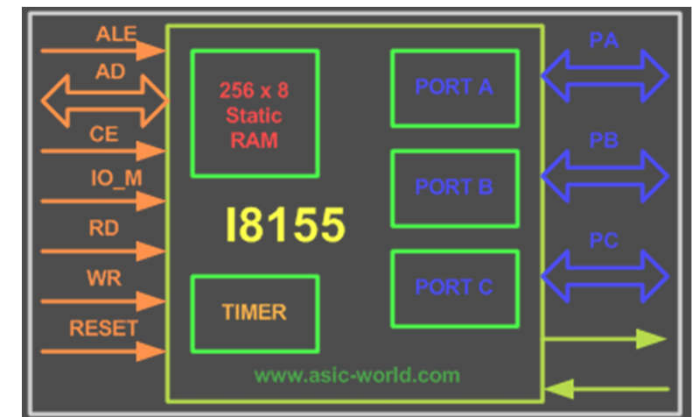
Top-down Design

- Xuất phát từ những thông số quan trọng nhất của hệ thống, triển khai các thiết kế ở mức thấp hơn
- Thực hiện chi tiết hoá đến mức cổng Gate
- Thực hiện mô phỏng, điều chỉnh các thiết kế các mức cho đến khi đạt yêu cầu
- Sau đó mới triển khai các bước chế tạo



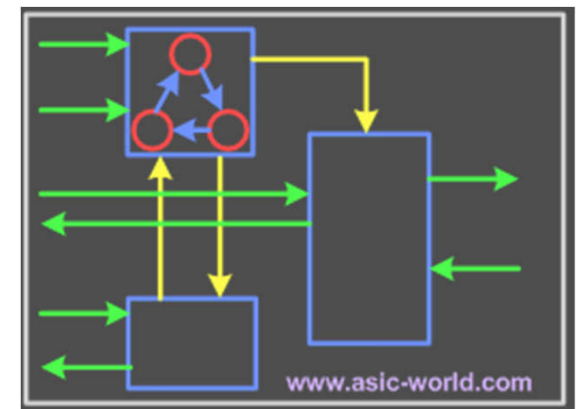
Top-down Design - High level Design

- Xác định khối cần thiết và liên kết giữa các khối đó
- Ví dụ cần thiết kế bộ VXL



Top-down Design - Low Level Design

- Mô tả cách thực hiện mỗi khối chức năng
- Gồm chi tiết các máy trạng thái, bộ đếm, giải mã, các thanh ghi bên trong
- Vẽ dạng sóng tại mỗi giao diện
- Bước này quan trọng và mất nhiều thời gian



Top-down Design - RTL Coding

- Chuyển thiết kế mức thấp
- Dùng các cấu trúc của ngôn ngữ mô tả phần cứng HDL
- Cần kết nối các đoạn mã trước khi kiểm tra và tổng hợp

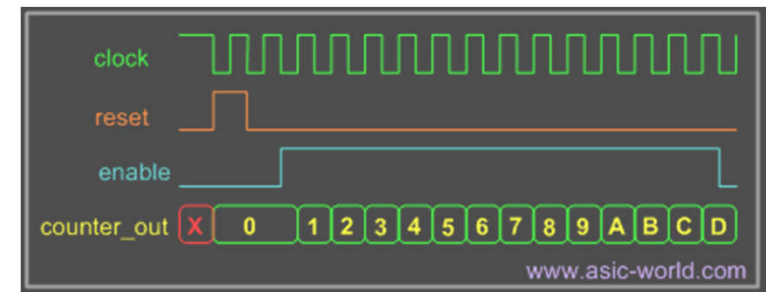
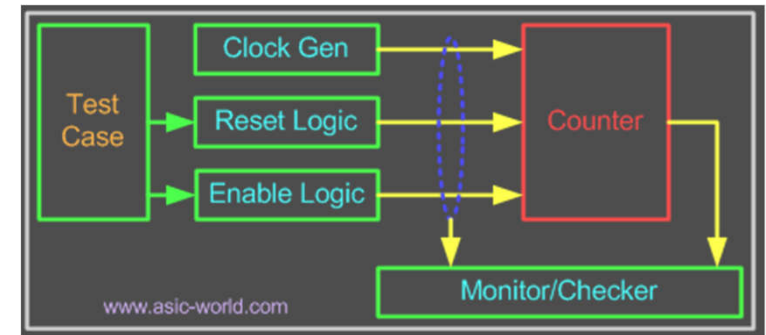
```
1 module addbit (  
2     a        , // first input  
3     b        , // Second input  
4     ci       , // Carry input  
5     sum      , // sum output  
6     co       , // carry output  
7 );  
8 //Input declaration  
9 input a;  
10 input b;  
11 input ci;  
12 //Output declaration  
13 output sum;  
14 output co;  
15 //Port Data types  
16 wire a;  
17 wire b;  
18 wire ci;  
19 wire sum;  
20 wire co;  
21 //Code starts here  
22 assign {co, sum} = a + b + ci;  
23  
24 endmodule // End of Module addbit
```

Mô phỏng

- Dùng các chương trình mô phỏng để kiểm tra chức năng của mô hình
- Kiểm tra toàn bộ chức năng của các khối RTL
- Cần viết testbench - tạo ra các tín hiệu clk, reset, và các vector kiểm tra
- Chiếm 60-70% thời gian của việc thiết kế
- Dùng dạng sóng đầu ra để kiểm tra hoạt động của các khối chức năng

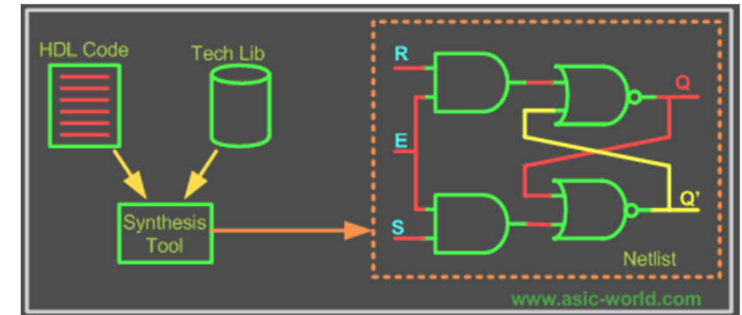
Timing Simulation

- Thực hiện sau khi tổng hợp hoặc Place and Route
- Kiểm tra mạch gồm trễ cổng, trễ dây dẫn tại tốc độ CLK (SDF simulation) hoặc Gate level simulation



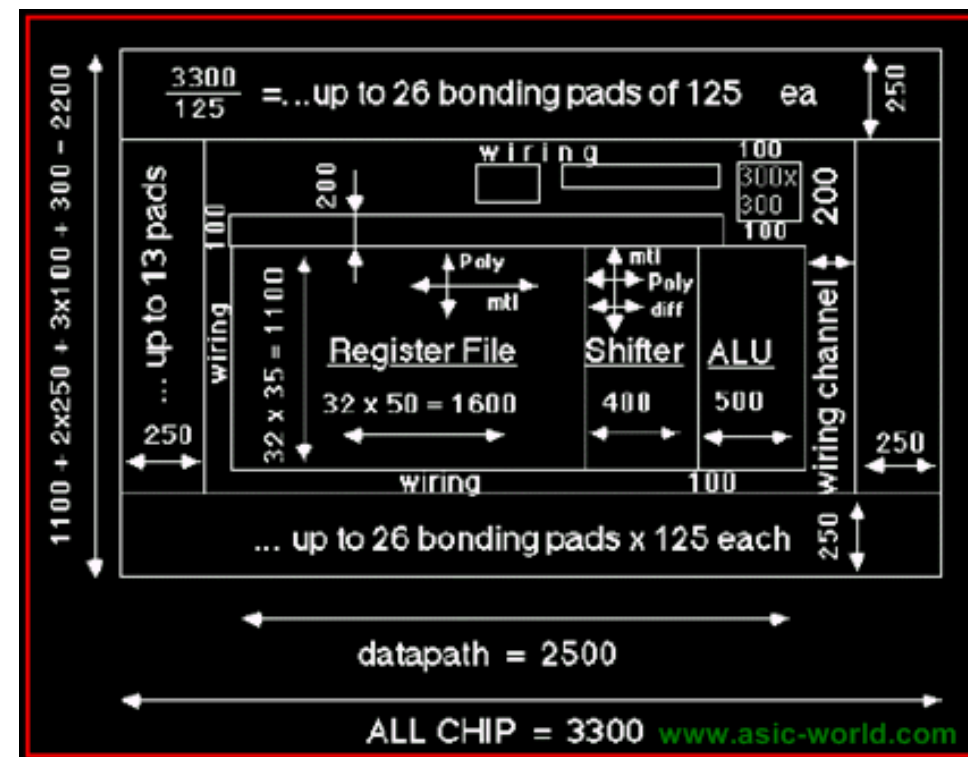
Tổng hợp - Synthesis

- Dùng phần mềm tổng hợp thiết kế
- Biên dịch mã HDL, các ràng buộc của đầu vào công nghệ thực hiện
- Ánh xạ mã RTL vào các cổng
- Phân tích thời gian (phần mềm chỉ xét tới trễ cổng mà chưa xét tới trễ dây dẫn)
- Kiểm tra
 - Formal Verification: kiểm tra tính chính xác của việc ánh xạ mã RTL sang sơ đồ cổng
 - Scan insertion: Kiểm tra chuỗi thực hiện trong trường hợp ASIC
- Kết quả sơ đồ mạch nguyên lý - kết nối các cổng → file netlist



Đặt khối và định tuyến - Place and Route

- Đặt khối và định tuyến theo định dạng Netlist Verilog
- Đặt vị trí các cổng và FF
- Định tuyến: xung Clock, reset, các khối
- Kết quả: file GDN, được sử dụng để chế tạo ASIC



Kiểm tra sau chế tạo - Post Silicon Validation

- Chế tạo xong chip bán dẫn
- Cần kiểm tra trong môi trường thực tế trước khi bán ra thị trường
- Do tốc độ mô phỏng trong RTL rất thấp nên thường xuất hiện vấn đề khi kiểm tra chip bán dẫn

Phần mềm thiết kế

- ModelSim: viết Xcode và testbench mô phỏng
- Phần mềm: https://fpgasoftware.intel.com/?product=modelsim_ae#tabs-2
- Chọn phiên bản Quartus Prime Lite Edition

Cài đặt cả 2

Quartus Prime Lite Edition (Free)

Quartus Prime (includes Nios II EDS) Size: 1.6 GB MD5: 5F6CFEBDB7B3CB35E033D2A9F5D59AC4 <small>** Nios II EDS on Windows requires Ubuntu 18.04 LTS on Windows Subsystem for Linux (WSL), which requires a manual installation. ** Nios II EDS requires you to install an Eclipse IDE manually.</small>	↓
ModelSim-Intel FPGA Edition (includes Starter Edition) Size: 1.2 GB MD5: 65024AF2CE888426125246A4BC7720CD	↓

Cài đặt Cyclone V device support

Arria II device support Size: 499.1 MB MD5: 4561D23010DD1FD359FE12348B102AC6	↓
Cyclone IV device support Size: 466.0 MB MD5: E6527CBC876426C4ECD8737D8B68369C	↓
Cyclone 10 LP device support Size: 265.7 MB MD5: D47100035A5A97F44048DF19218B09E4	↓
Cyclone V device support Size: 1.3 GB MD5: 78D59D548756F81E67B9D7CD2149E2B8	↓
MAX II, MAX V device support Size: 11.4 MB MD5: 9E8B802C6B4768933362A0E6398B7E2E	↓
MAX 10 FPGA device support Size: 287.4 MB MD5: FEA82DF785421CD0C0BF75CA94790804	↓

Ngôn ngữ lập trình Verilog HDL

- Giới thiệu chung
 - Phương pháp thiết kế dùng Verilog HDL
 - Thiết kế top-down
 - Phần mềm hỗ trợ thiết kế
- Các mô hình thiết kế Verilog
 - Mô hình mức cổng (gate-level)
 - Mô hình mức luồng dữ liệu
 - Mô hình hành vi
- Một số ví dụ cụ thể

Các mô hình thiết kế Verilog

- Mô hình hành vi (Behavioral level)
- Mức thanh ghi (Register-Transfer Level)
- Mức cổng (Gate Level)
- Mức chuyển mạch (switch): GJT, FET,...

Mô hình mức hành vi

- Mô tả hệ thống bằng các thuật toán đồng thời (hành vi)
- Mỗi thuật toán gồm một tập các câu lệnh được thực hiện tuần tự
- Các khối Function, Tasks, Always là các thành phần chính của mô hình
- Không liên quan đến việc thực hiện cấu trúc của thiết kế

Mô hình mức thanh ghi RTL

- Mô tả hoạt động của mạch qua luồng dữ liệu chuyển giữa các thanh ghi
- Sử dụng xung Clock chính xác
- Thiết kế mức RTL chứa giới hạn timing chính xác: các hoạt động được xảy ra tại thời gian nhất định
- Tất cả các mã có thể tổng hợp được là mã RTL

Mô hình mức cổng

- Mô tả hệ thống bằng các kết nối logic và các thuộc tính timing của chúng
- Tất cả các tín hiệu đều rời rạc (nhận mức logic 0,1)
- Dùng các cổng logic cơ bản để mô tả mạch
- Mã mức cổng được tạo bằng các công cụ tổng hợp và netlist được dùng để mô phỏng mức cổng

Cấu trúc chương trình Verilog

- Ngôn ngữ Verilog mô tả hệ thống như một tập hợp các module liên kết với nhau
- Trong Verilog, **module** khác với các **hàm** và **thủ tục**:
 - Một module không bao giờ được gọi tới
 - Một module được khởi tạo tại thời điểm bắt đầu chương trình và tồn tại trong suốt thời gian tồn tại của chương trình

Cấu trúc chương trình Verilog

- Khai báo các câu tiền xử lý của trình biên dịch
 - Include
 - Define
- Bắt đầu module bằng từ khoá **module** <tên_module> (dòng 8)
- Khởi Initial bắt đầu bởi Begin, kết thúc bởi end
 - Có 2 lệnh nằm giữa dòng 10 và 13
 - Thực hiện 1 lần duy nhất khi bắt đầu mô phỏng tại t=0
- Kết thúc bởi từ khoá **endmodule**

```
1 //-----
2 // This is my first Verilog Program
3 // Design Name : hello_world
4 // File Name : hello_world.v
5 // Function : This program will print 'hello world'
6 // Coder : Deepak
7 //-----
8 module hello_world ;
9
10 initial begin
11     $display ("Hello World by Deepak");
12     #10 $finish;
13 end
14
15 endmodule // End of Module hello_world
```

Cấu trúc module

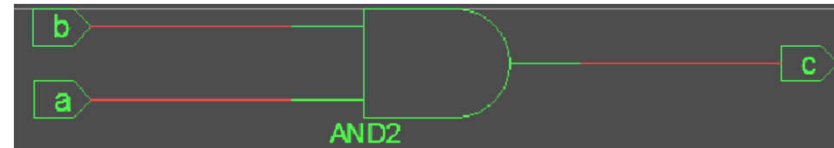
- Interface: khai báo các Ports là danh sách các tín hiệu vào, tín hiệu ra hoặc tín hiệu vào/ra để kết nối giữa các module
- Body: thông số các thành phần bên trong module
- Add-on: tùy chọn

module ten_module (danh_sach_Port)	
Khai báo cổng Khai báo các tham số	Giao diện (Interface)
Các dẫn hướng biên dịch	Tùy chọn (add-on)
Khai báo biến Khởi tạo các module mức thấp Các khối initial và always Các hàm và tác vụ	Thân (body)
endmodule	Kết thúc

Giao diện module

- Khai báo một giao diện module gồm 2 thành phần
 - Danh sách các cổng -port-list: chứa các tên cổng trong ngoặc đơn
 - Khai báo cổng: mô tả chi tiết hơn các cổng đã liệt kê trong danh sách

□ Ví dụ module AND2_1 có cổng vào ra như sau



□ Ngôn ngữ Verilog sẽ mô tả như sau

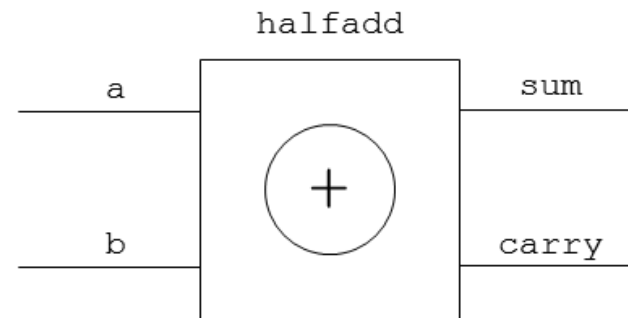
```
module AND2_1(a, b, c);  
  input a, b;  
  output c;  
  assign c = a & b;  
endmodule
```

Module

- Mô tả giao diện và hành vi
- Các module giao tiếp với nhau thông qua các cổng
 - Tên các cổng được liệt kê trong dấu ngoặc kép đằng sau tên module
- Các cổng có thể được xác định là đầu vào (input), đầu ra (output) hoặc hai chiều (inout)
- ^ được dành riêng cho toán tử
- & vừa là điều kiện, vừa là toán tử

(Δ) Verilog phân biệt chữ hoa và chữ thường. Các từ khóa phải viết bằng chữ thường.

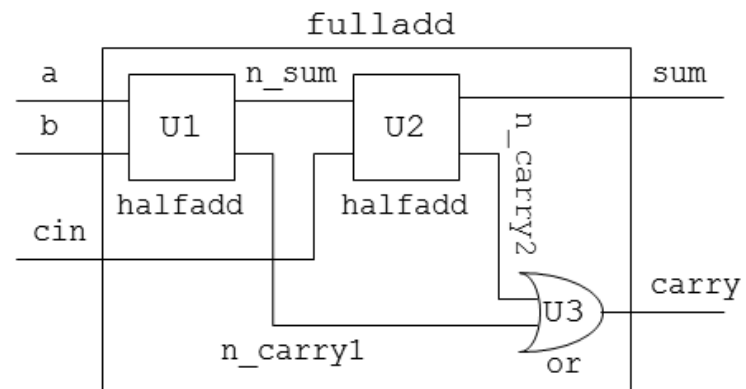
```
module halfadd (a, b, sum, carry);  
    output sum, carry;  
    input a, b;  
  
    assign sum = a ^ b;  
    assign carry = a & b;  
  
endmodule
```



Mô hình phân cấp

Tạo ra một mô hình phân cấp bằng cách

- Tạo ra module
- Nối các cổng của module tới các cổng cục bộ hoặc dây
 - Các dây nối cục bộ cần phải được định nghĩa
- Cấu trúc or là một phần tử cổng "dùng sẵn"

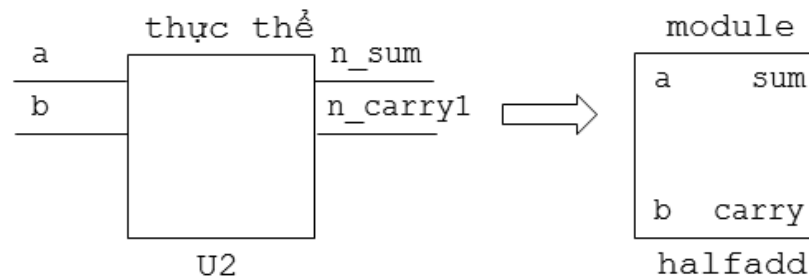


```
module fulladd (a, b, cin, sum, carry);
input a, b, cin;
output sum, carry;
wire n_sum, n_carry1, n_carry2;

halfadd U1 (.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
halfadd U2 (.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));
or U3 (carry, n_carry2, n_carry1);
endmodule
```

Kết nối phân cấp - Kết nối cổng có tên

- Phân biệt rõ ràng cổng nào của module được nối với cổng/dây cục bộ nào



```
module fulladd (a, b, cin, sum, carry);
input a, b, cin;
output sum, carry;
wire n_sum, n_carry1, n_carry2;
...
halfadd U1 (.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
...
```

... được nối với dây
n_carry1 của module
fulladd

```
module halfadd (a, b, sum, carry);
output sum, carry;
input a, b;
...
endmodule
```

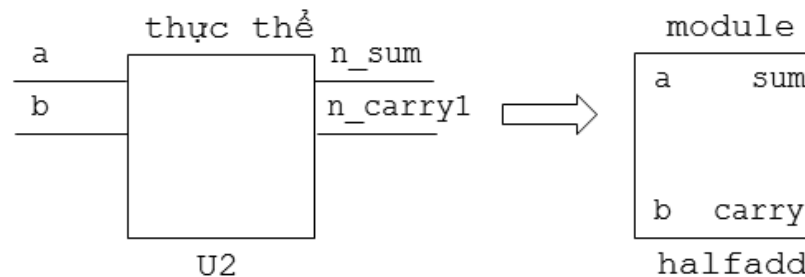
Đầu ra carry của
module halfadd

(T) Mẹo: Hãy sử dụng kết nối các cổng có tên để gắn các khối nhỏ trong mô hình phân cấp

Kết nối phân cấp - Kết nối cổng theo thứ tự

- Kết nối các cổng theo thứ tự:

- Cổng thứ nhất của thực thể sẽ ứng với cổng thứ nhất của module
- Cổng thứ hai của thực thể sẽ ứng với cổng thứ của module
- v.v và v.v...



```
module fulladd (a, b, cin, sum, carry);
input a, b, cin;
output sum, carry;
wire n_sum, n_carry1, n_carry2;
...
halfadd U1 (a, b, n_sum, n_carry1));
...
```

```
module halfadd (a, b, sum, carry);
output sum, carry;
input a, b;
...
endmodule
```

Đầu ra carry của module halfadd

Đầu vào a của fulladd ứng với đầu vào a của halfadd

Đầu vào b của fulladd ứng với đầu vào b của halfadd

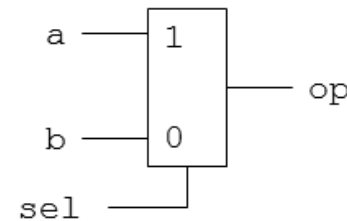
(!) **Chú ý:** Kiểu kết nối này khó đọc và dễ mắc lỗi hơn kiểu kết nối cổng có tên

Các khối thủ tục

- Chứa các biểu thức thực thi
 - Nhiều thủ tục hoạt động song song
 - thủ tục always
 - Thực thi khi bất kỳ biến nào trong danh sách sự kiện thay đổi giá trị
 - Chạy suốt trong quá trình mô phỏng
 - thủ tục initial
 - Thực thi đúng một lần vào lúc bắt đầu mô phỏng
 - Được sử dụng cho thủ tục khởi tạo, testbench...
- (&) – Tổng hợp:** Các khối initial không tổng hợp được

```
always @ (a or b or sel)
  if (sel == 1);
    op = a;
  else
    op = b;
```

Danh sách sự kiện



```
initial
begin
  a = 1;
  b = 0
end
```

Danh sách sự kiện

- thủ tục always thực thi khi một trong số các biến của danh sách sự kiện thay đổi giá trị
- Sự kiện là một sự thay đổi trong giá trị của biến logic

or là một từ khóa được dùng trong danh sách sự kiện

```
always @ (a or b or sel)
    if (sel == 1);
    op = a;
else
    op = b;
```

Danh sách sự kiện

Ngôn ngữ Verilog HDL

- Các quy ước cơ bản
 - Verilog HDL dùng tương tự như ngôn ngữ C
 - Có phân biệt CHỮ HOA và chữ thường
 - Tất cả các khoảng trắng là chữ thường

Ngôn ngữ Verilog HDL

- Khoảng trắng

- Gồm các ký tự: khoảng trống, Tab, xuống dòng, sang trang mới
- Thông thường các khoảng trắng đều được bỏ qua
- Nhưng trong các chuỗi, khoảng trắng và Tab có ý nghĩa riêng

Bad Code : Never write code like this.

```
1 module addbit(a,b,ci,sum,co);
2 input a,b,ci;output sum co;
3 wire a,b,ci,sum,co;endmodule
```

Good Code : Nice way to write code.

```
1 module addbit (
2     a,
3     b,
4     ci,
5     sum,
6     co);
7     input          a;
8     input          b;
9     input          ci;
10    output         sum;
11    output         co;
12    wire           a;
13    wire           b;
14    wire           ci;
15    wire           sum;
16    wire           co;
17
18 endmodule
```

Ngôn ngữ Verilog HDL

- Chú thích

- Quy tắc giống ngôn ngữ C
- Chú thích nằm giữa hai dấu //
- hoặc giữa /* */

```
1  /* This is a
2     Multi line comment
3     example */
4  module addbit (
5     a,
6     b,
7     ci,
8     sum,
9     co);
10
11  // Input Ports Single line comment
12  input      a;
13  input      b;
14  input      ci;
15  // Output ports
16  output     sum;
17  output     co;
18  // Data Types
19  wire       a;
20  wire       b;
21  wire       ci;
22  wire       sum;
23  wire       co;
24
25  endmodule
```

Ngôn ngữ Verilog HDL

- **Chữ hoa và chữ thường**

- Có sự phân biệt chữ hoa và chữ thường
- Tất cả các từ khoá đều là chữ thường
- Không dùng từ khoá của Verilog làm tên riêng, kể cả khi khác loại chữ

1	input
2	wire
3	WIRE
4	Wire

// a Verilog Keyword
// a Verilog Keyword
// a unique name (not a keyword)
// a unique name (not a keyword)

Ngôn ngữ Verilog HDL

- **Từ định danh - tên riêng**

- dùng cho tên biến, hàm, khối, module, tên trường hợp
- Bắt đầu bằng ký tự hoặc đường gạch dưới “_”
- Phân biệt dạng chữ
- Độ dài tối đa 1024 kí tự
- Ví dụ: data_input, muclk_input , my\$clk, i386, A

Ngôn ngữ Verilog HDL

- Từ khoá

- dùng các ký tự dành riêng để định nghĩa cấu trúc của Verilog
- Ví dụ: **assign**, **case**, **while**, **wire**, **reg**, **and**, **or**, **nand**, và **module**
- Chỉ toàn các ký tự thường
- Không được sử dụng để làm tên riêng
- Từ khóa Verilog cũng bao gồm cả chỉ dẫn chương trình biên dịch và System Task và các hàm

Ngôn ngữ Verilog HDL

- **Chữ số**

- Dùng lưu giá trị các con số nhị phân, thập phân, hecta,...
- Cú pháp: `<size>'<base format> <number>` Ví dụ: 8'hAA
 - Độ dài dữ liệu
 - Kiểu dữ liệu
 - Giá trị ban đầu
- mặc định là 32 bit
 - D hoặc d: hệ thập phân
 - B hoặc b: hệ nhị phân
 - H hoặc h: hệ thập lục phân
 - O hoặc o: hệ bát phân
- Số âm được biểu diễn bằng số bù 2
- Dấu ? được dùng thay thế cho z (trở kháng cao)
- Có các loại số: integer, real, số có dấu và không dấu

Ngôn ngữ Verilog HDL

- **Số nguyên**

- Là kiểu biến thanh ghi (reg) dùng chung cho tính toán, thao tác.
- Từ khoá khai báo: **integer**
- Được xác định cỡ hoặc không
- Chiều dài mặc định 32 bit
- Cho phép các khoảng trống giữa kích cỡ (size), cơ số (radix) và giá trị (value)
- Cú pháp: `<size>'<radix><value>`
- Ví dụ: `integer a; // số nguyên 32 bit, mặc định hệ cơ số 10`

Ngôn ngữ Verilog HDL

- Ví dụ về số nguyên

- Verilog mở rộng giá trị cho phù hợp với kích cỡ bằng cách xét từ phải qua trái
- Khi kích cỡ nhỏ hơn giá trị, những bit phía trái sẽ bị cắt bỏ
- Khi kích cỡ lớn hơn giá trị
 - 0 hoặc 1 sẽ được thêm vào phía trái
 - Bit Z sẽ được thêm Z
 - Bit X sẽ được thêm X

Integer	Stored as
1	00000000000000000000000000000001
8'hAA	10101010
6'b10_0011	100011
'hF	00000000000000000000000000001111

Integer	Stored as
6'hCA	001010
6'hA	001010
16'bZ	ZZZZZZZZZZZZZZZZ
8'bx	xxxxxxx

X - Bit chưa biết

Z - Trở kháng cao

Ngôn ngữ Verilog HDL

- **Số thực - Real**

- Từ khoá: **real**
- Có thể ở dạng thông thường (3.14) hoặc dạng khoa học (312 e-2)
- Giá trị mặc định của biến là 0
- Khi biến kiểu Real bị gán cho biến kiểu integer, giá trị sẽ được làm tròn đến giá trị integer gần nhất

```
real a;    // biến thực a
initial
begin
a=3e10;    // a gán giá trị dưới dạng khoa học
a=3.14;    // a gán giá trị = 3.14
end
integer i; // định nghĩa biến integer i
initial
i=a;       //i nhận giá trị =3 (làm tròn)
```

Ngôn ngữ Verilog HDL

- Số có dấu và không dấu
 - Bất kì số nào không có dấu (-) đứng trước đều là số dương hoặc số không dấu
 - các số có dấu (-) đứng trước size của số
 - Nội bộ verilog sẽ biểu diễn số âm dưới dạng số bù 2

Number	Description
32'hDEAD_BEEF	Unsigned or signed positive number
-14'h1234	Signed negative number

```
1 module signed_number;
2
3 reg [31:0] a;
4
5 initial begin
6     a = 14'h1234;
7     $display ("Current Value of a = %h", a);
8     a = -14'h1234;
9     $display ("Current Value of a = %h", a);
10    a = 32'hDEAD_BEEF;
11    $display ("Current Value of a = %h", a);
12    a = -32'hDEAD_BEEF;
13    $display ("Current Value of a = %h", a);
14    #10 $finish;
15 end
16
17 endmodule
```

```
Current Value of a = 00001234
Current Value of a = ffffedcc
Current Value of a = deadbeef
Current Value of a = 21524111
```

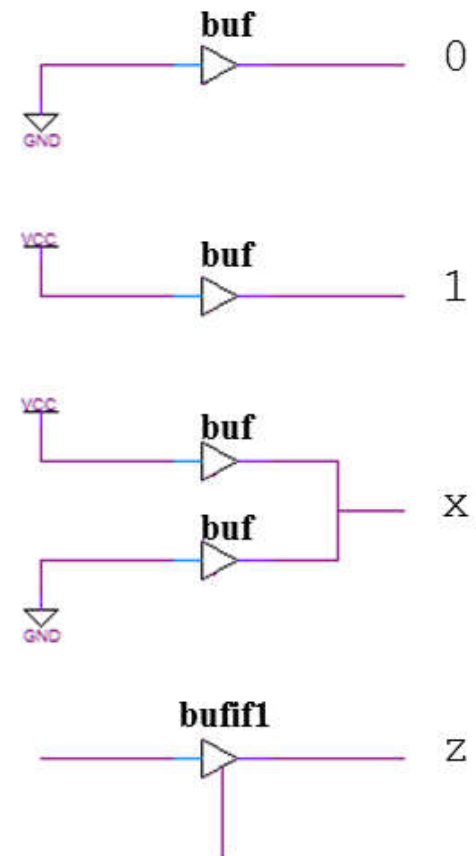
Kiểu dữ liệu

- Có 2 kiểu dữ liệu cơ bản
 - **Nets** - biểu diễn liên kết cấu tạo giữa các thành phần, thể hiện liên kết vật lý giữa các phần tử thuộc về phần cứng
 - **Registers** - biểu diễn các biến dùng để lưu trữ dữ liệu
- Tất cả các tín hiệu đều có kiểu dữ liệu liên kết với nó
 - **Khai báo rõ ràng** bằng khai báo trong mã Verilog HDL
 - **Khai báo ngầm định**: luôn là kiểu net “wire” và có độ rộng 1 bit

Giá trị logic trong Verilog - 4 giá trị logic

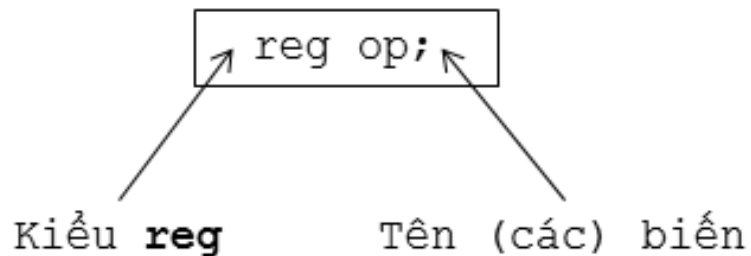
- Mức 0: mức thấp, sai, số không, mức logic âm, đất, giá trị âm
- Mức 1: mức cao, đúng, số 1, mức logic dương, nguồn, giá trị dương
- Mức “x”: chưa xác định (xung đột bus), chưa khởi tạo
- Mức “z”: trở kháng cao, 3 trạng thái, chưa điều khiển, chưa nối, chưa biết bộ điều khiển

Giá trị “x” khác với don't care



Khái niệm kiểu dữ liệu

- Cần xác định kiểu dữ liệu khi định nghĩa một biến
- Các cổng của module được mặc định kiểu `win`
- Mặc định các kiểu biến `win`, vô hướng
- Việc sử dụng các kiểu dữ liệu được quy định chặt chẽ



```
module mux (a, b, sel, op);  
input a, b;  
input sel;  
output op;  
reg    op;  
  
always @ (a or b or sel)  
    if (sel == 1)  
        op = a;  
    else  
        op = b;  
  
endmodule
```


Vector

- là một biến có độ dài từ 2 bit trở lên
- Kích cỡ của biến được định nghĩa khi khai báo

```
module mux (a, b, sel, op);  
  input  [3:0] a, b; ← Dây vector 4 bit  
  input  sel;  
  output [3:0] op;  
  reg    [3:0] op; ← Thanh ghi vector 4 bit  
  
  always @ (a or b or sel)  
    if (sel == 1)  
      op = a;  
    else  
      op = b;  
  
endmodule
```

Phép gán vector và thứ tự bit

- Gán theo vị trí
- có thể lấy giá trị các bit riêng lẻ từ vector
- Thứ tự bit có thể được định nghĩa

```
module mux (a, b, sel, op);  
  input  [3:0] a, b; ← Dây vector 4 bit  
  input  sel;  
  output [3:0] op;  
  reg    [3:0] op; ← Thanh ghi vector 4 bit  
  
  always @ (a or b or sel)  
    if (sel == 1)  
      op = a;  
    else  
      op = b;  
  
endmodule
```

Phép gán vector và độ dài bit

- Trong phép gán, các vector không cần có cùng độ dài
 - Nguồn lớn hơn đích: nguồn sẽ được cắt bớt từ MSB
 - Nguồn ngắn hơn đích: nguồn sẽ được thêm các số 0 từ MSB
- Sử dụng [] hoặc { } để phối hợp độ dài vector

```
reg [3:0] zbus;    // vector 4 bit  
reg [5:0] widebus; // vector 6 bit
```

zbus = widebus

widebus[5]
widebus[4]
zbus[3] ← widebus[3]
zbus[2] ← widebus[2]
zbus[1] ← widebus[1]
zbus[0] ← widebus[0]

Ngoặc vuông

```
zbus = widebus[3:0];
```

widebus = zbus

widebus[5] ← 0
widebus[4] ← 0
widebus[3] ← zbus[3]
widebus[2] ← zbus[3]
widebus[1] ← zbus[3]
widebus[0] ← zbus[3]

Ngoặc nhọn

```
widebus = {2'b00, zbus};
```

Kiểu dữ liệu register và Net

	Register (Thanh ghi)	Net (Dây)
Từ khoá	reg, integer, time, real	wire, wand, wor, tri, triand, trior, supply0, supply1
Điều khiển	Theo sự kiện	
Lưu trữ	Dữ liệu	Không lưu dữ liệu, chỉ là một kết nối
Sử dụng trong	khối “always”	Không được xuất hiện trong “always”
Lưu ý		<i>Input, output, inout</i> được mặc định kiểu wire

Kiểu dữ liệu Net

- Được dùng để mô tả các kiểu phần cứng khác nhau như PMOS, NMOS, CMOS...
- Là kết nối điện đi từ 1 khối mạch điện đến 1 khối mạch điện khác
- Không được gán giá trị cho các biến kiểu net
- **wire** được dùng phổ biến nhất

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand, triand	Wired outputs AND together (models open-collector)
tri0, tri1	Net pulls-down or pulls-up when not driven
supply0, supply1	Net has a constant logic 0 or logic 1 (supply strength)
triereg	Retains last value, when driven by z (tristate).

Ví dụ - wor

```
1 module test_wor();
2
3 wor a;
4 reg b, c;
5
6 assign a = b;
7 assign a = c; // giá trị a là mức logic của phép OR a và b
8
9 initial begin
10     $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
11     #1 b = 0;
12     #1 c = 0;
13     #1 b = 1;
14     #1 b = 0;
15     #1 c = 1;
16     #1 b = 1;
17     #1 b = 0;
18     #1 $finish;
19 end
20
21 endmodule
```

- %g: hiển thị số thực theo số lũy thừa hoặc hệ 10
- %b: hệ nhị phân

Simulator Output

```
0 a=x b=x c=x
1 a=x b=0 c=x
2 a=0 b=0 c=0
3 a=1 b=1 c=0
4 a=0 b=0 c=0
5 a=1 b=0 c=1
6 a=1 b=1 c=1
7 a=1 b=0 c=1
```

Ví dụ - wand

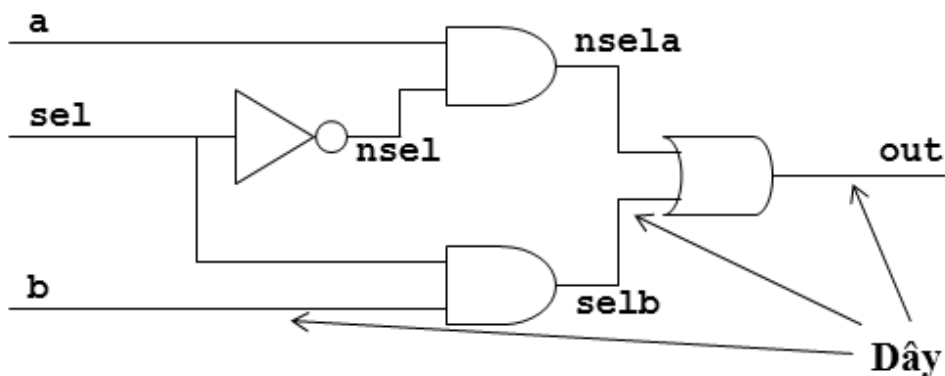
```
1 module test_wand();
2
3 wand a;
4 reg b, c;
5
6 assign a = b;
7 assign a = c;
8
9 initial begin
10     $monitor("%g a = %b b = %b c = %b", $time, a, b, c);
11     #1 b = 0;
12     #1 c = 0;
13     #1 b = 1;
14     #1 b = 0;
15     #1 c = 1;
16     #1 b = 1;
17     #1 b = 0;
18     #1 $finish;
19 end
20
21 endmodule
```

Simulator Output

```
0 a=x b=x c=x
1 a=0 b=0 c=x
2 a=0 b=0 c=0
3 a=0 b=1 c=0
4 a=0 b=0 c=0
5 a=0 b=0 c=1
6 a=1 b=1 c=1
7 a=0 b=0 c=1
```

wire và assign

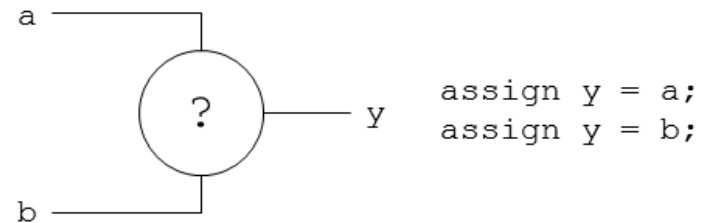
- Các giá trị wire thay đổi với câu lệnh assign trong phép gán liên tục
- việc khai báo và gán chân cho wire có thể kết hợp làm một



```
module mux (a, b, sel, out);  
  input a, b;  
  output out;  
  wire nsela, selb, nsel;  
  
  assign nsel = ~sel;  
  assign selb = sel & b;  
  assign nsela = nsel & a;  
  assign out = nsela | selb;  
  
endmodule
```


wire: cách giải quyết xung đột logic

- Xuất hiện khi 1 wire được điều khiển bằng nhiều nguồn khác nhau
- Giải quyết bằng cách quyết định giá trị cuối cùng của đích



Khai báo y:
wire y;
tri y;

	b				
a \	0	1	x	z	
0	0	x	x	0	
1	x	1	x	1	
x	x	x	x	x	
z	0	1	x	z	

Khai báo y:
wand y;
triand y;

	b				
a \	0	1	x	z	
0	0	0	0	0	
1	0	1	x	1	
x	0	x	x	x	
z	0	1	x	z	

Dữ liệu kiểu Register

- Register lưu trữ giá trị cuối cùng được gán cho đến khi có câu lệnh làm thay đổi giá trị của nó
- Register biểu diễn các cấu trúc lưu trữ dữ liệu (reg, integer, real, time)
- Có thể tạo mảng regs → memories
- Được dùng như các biến trong các khối thủ tục
- Các khối thủ tục bắt đầu với các từ khoá **initial** và **always**

- Trong các kiểu biến register, kiểu **reg** được sử dụng phổ biến nhất
- Từ khoá reg, giá trị mặc định là x - không xác định
- cú pháp

reg [MSB:LSB] reg_name

```
reg a; // biến thanh ghi đơn giản 1 bit
reg [7:0] A; // biến thanh ghi(vector) 8 bit
reg [5:0] a,b; // 2 biến thanh ghi 6 bit
```

Gán dữ liệu cho thanh ghi

- Giá trị thanh ghi thay đổi theo lệnh gán thủ tục, xuất hiện trong các khối initial, always,...

```
reg [3:0] vect; // vector không dấu 4 bit
reg [2:0] p, q; // 2 vector không dấu 3 bit
integer aint;   // số nguyên có dấu 32 bit
reg s;          // thanh ghi mặc định là 1 bit
time value;     // giá trị thời gian
```

```
module mux (a, b, sel, op);
input a, b;
input sel;
output op;
reg op;

always @ (a or b or sel)
begin
    if (sel == 1)
        op = a;
    else
        op = b;
end

endmodule
```

Gán dữ liệu cho thanh ghi

- Thanh ghi chỉ cập nhật giá trị trong các thủ tục
- Các thủ tục chỉ cập nhật giá trị của thanh ghi
- Vế phải của phép toán có thể là reg hoặc wire

(X) **Lỗi:** Dây được gán trong thủ tục

(X) **Lỗi:** Thanh ghi được gán ngoài thủ tục

```
module mux (a, b, c, sel, mux);
input  a, b, c;
input  sel;
output mux;
wire aandb, nmux;
reg  mux, nota;

always @ (a or b or sel)
    if (sel == 1)
        begin
            mux = a;
            nmux = b;
        end
    else
        begin
            mux = b;
            nmux = a;
        end
    end

assign nota = ~a;
assign aandb = a & b;
...
```

Time

- Là kiểu dữ liệu đặc biệt time, được dùng để lưu trữ thời gian mô phỏng được tính theo đơn vị giây (s)
- Chiều dài tối thiểu của thanh ghi này là 64 bit
- Có thể dùng hàm hệ thống \$time để có được thời gian hiện tại
- Ví dụ

time c;

c = **\$time**; // c = thời gian mô phỏng mạch điện

Data Types	Functionality
reg	Unsigned variable
integer	Signed variable - 32 bits
time	Unsigned integer - 64 bits
real	Double precision floating point variable

Note : Of all register types, reg is the one which is most widely used

Chuỗi - String

- là dãy ký tự đặt giữa 2 dấu nháy “”
- Chuỗi chỉ được viết trên 1 dòng
- Kiểu **string** có thể được lưu trữ trong reg
- Độ dài của biến thanh ghi phải đủ lớn để lưu trữ string
- Mỗi ký tự của string cần có 8 bit
- Ví dụ

```
reg [8*18:0] string_value; // khai báo biến 18 bytes  
initial
```

```
String_value= “ Hello Verilog word”; //chuỗi lưu trong biến
```

Character	Description
\n	New line character
\t	Tab character
\\	Backslash (\) character
\"	Double quote (") character
\ddd	A character specified in 1-3 octal digits (0 <= d <= 7)
%%	Percent (%) character

Chuỗi - String

- Ví dụ

```
1 //-----  
2 // Design Name : strings  
3 // File Name : strings.v  
4 // Function : This program shows how string  
5 // can be stored in reg  
6 // Coder : Deepak Kumar Tala  
7 //-----  
8 module strings();  
9 // Declare a register variable that is 21 bytes  
10 reg [8*21:0] string ;  
11  
12 initial begin  
13     string = "This is sample string";  
14     $display ("%s \n", string);  
15 end  
16  
17 endmodule
```

Thông số - parameter

- Thông số được dùng ở bất kỳ đâu để khai báo các hằng số thời gian
- Giúp code đọc dễ hơn
- Được đặt trong module mà chúng được định nghĩa
- có thể dùng để định cỡ các khai báo cục bộ (các cổng module, cần khai báo trước khi dùng)

```
// Danh sách thông số
parameter P1 = 8,
          REAL_P = 2.039,
          X_WORD = 16'bx;
```

```
module mux (a, b, sel, out);

parameter WIDTH = 2;

input  [WIDTH-1:0] a;
input  [WIDTH-1:0] b;
input  sel;
output [WIDTH-1:0] out;
reg    [WIDTH-1:0] out;

always @ (a or b or sel)
    if (sel == 0);
        out = a;
    if (sel == 1);
        out = b;

endmodule
```


Ghi đè giá trị của thông số

- Giá trị của thông số có thể thay đổi với mỗi thực thể của module

```
module muxs (abus, bbus, anib, bnib, opbus, opnib, opnib1, sel);
```

```
parameter NIB = 4;
```

```
input  [NIB-1:0] anib, bnib;
```

```
input  [7:0] abus, bbus;
```

```
input  sel;
```

```
output [NIB-1:0] opnib, opnib1;
```

```
output [7:0] opbus;
```

```
// Các thực thể của cùng một module với kích cỡ ghép kênh khác nhau
```

```
mux #(8) mux8 (.a(abus), .b(bbus), .sel(sel), .out(opbus));
```

```
mux #(NIB) mux4 (.a(anib), .b(bnib), .sel(sel), .out(opnib));
```

```
mux mux4a (.a(anib), .b(bnib), .sel(sel), .out(opnib));
```

```
defparam mux4a.WIDTH = 4;
```

```
endmodule
```

```
module mux (a, b, sel, out);
```

```
parameter WIDTH = 2;
```

```
...
```

(&) Tổng hợp: Cả # và defparam đều được tổng hợp bình thường

Mảng 2 chiều

- Verilog hỗ trợ mảng thanh ghi 2 chiều

```
reg [15:0] mem [0:1023]; // Mảng hai chiều 1K x 16 bit
integer int_array [99:0]; // Mảng số nguyên 100 phần tử
```

- Mỗi phần tử của mảng được đánh địa chỉ bởi một chỉ số trong mảng 2 chiều
 - Chỉ được tham chiếu 1 phần tử mảng tại một thời điểm
 - Truy cập vào nhiều phần tử cần nhiều câu lệnh
 - Truy cập đến 1 bit đơn lẻ cần một biến trung gian

```
reg [7:0] mem_array [0:225]; // Mảng bộ nhớ
reg [7:0] mem_word;
reg membit;

mem_word = mem_array[5]; // Truy cập địa chỉ 5
mem_word = mem_array[10]; // Truy cập địa chỉ 10
mem_bit = mem_word[7]; // Truy cập bit 7 của địa chỉ 10
```

Module

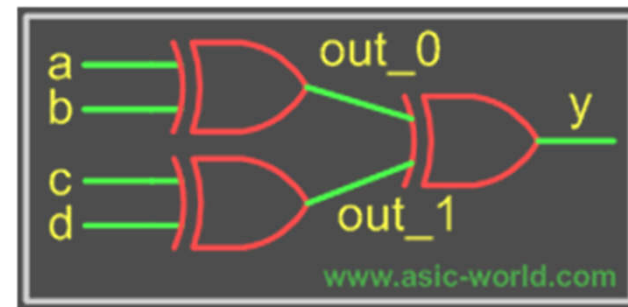
- là các khối xây dựng nên thiết kế verilog
- Tạo phân cấp thiết kế bằng cách khởi tạo các module trong module khác (module mức cao hơn)
- Các module giao tiếp với nhau thông qua các cổng



Khởi tạo một module

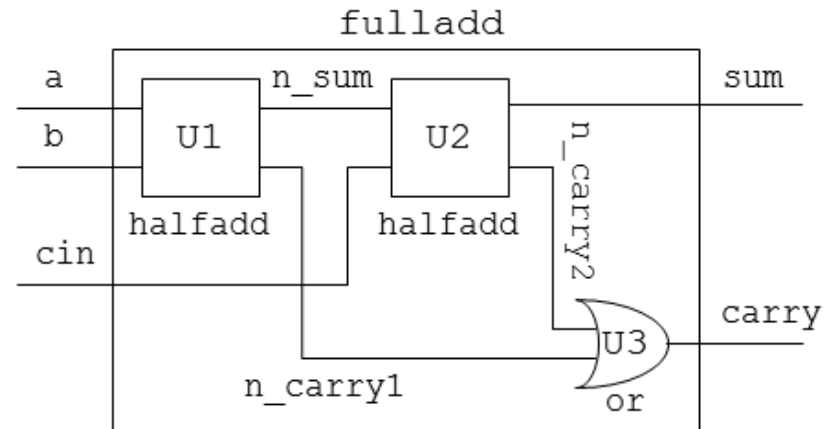
```
9 module parity (  
10 a      , // First input  
11 b      , // Second input  
12 c      , // Third Input  
13 d      , // Fourth Input  
14 y      // Parity output  
15 );  
16  
17 // Input Declaration  
18 input    a      ;  
19 input    b      ;  
20 input    c      ;  
21 input    d      ;  
22 // Output Declaration  
23 output    y      ;  
24 // port data types  
25 wire      a      ;  
26 wire      b      ;  
27 wire      c      ;  
28 wire      d      ;  
29 wire      y      ;
```

```
30 // Internal variables  
31 wire      out_0 ;  
32 wire      out_1 ;  
33  
34 // Code starts Here  
35 xor u0 (out_0, a, b) ;  
36  
37 xor u1 (out_1, c, d) ;  
38  
39 xor u2 (y, out_0, out_1) ;  
40  
41 endmodule // End Of Module parity
```



Mô hình phân cấp

- Tạo mô hình phân cấp
 - Tạo module
 - Nối các cổng của module tới các cổng cục bộ hoặc dây
 - Cần định nghĩa các dây nối cục bộ



```
module fulladd (a, b, cin, sum, carry);  
  input a, b, cin;  
  output sum, carry;  
  wire n_sum, n_carry1, n_carry2;  
  
  halfadd U1 (.a(a), .b(b), .sum(n_sum), .carry(n_carry1));  
  halfadd U2 (.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));  
  or      U3 (carry, n_carry2, n_carry1);  
endmodule
```

← Dây nối cục bộ (wire)

Port

- Port kết nối các module với nhau và với môi trường
- Tất cả các module (trừ top-level module)
- Các port liên kết theo **thứ tự** hoặc **theo tên**
- Port được khai báo **input**, **output** hoặc **inout**
- Chú ý: để dễ theo dõi khi viết chương trình, nên khai báo mỗi Port một dòng

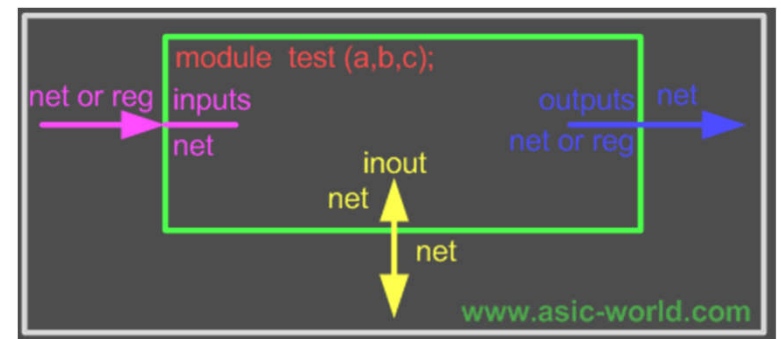
- Khai báo Port

```
input [range_val: range_var] list_of_identifiers;  
output [range_val:range_var] list_of_identifiers;  
inout [range_val:range_var] list_of_identifiers;
```

1	input		clk	; // clock input
2	input	[15:0]	data_in	; // 16 bit data input bus
3	output	[7:0]	count	; // 8 bit counter output
4	inout		data_bi	; // Bi-Directional data bus

Kết nối Port

- **input**: các input nội bộ luôn là kiểu net, các input bên ngoài có thể nối đến các biến kiểu reg hoặc net
- **output**: các output nội bộ là kiểu net hoặc reg, các output bên ngoài phải được kết nối đến một biến kiểu net.
- **inout**: luôn là kiểu net, chỉ có thể được nối đến một biến kiểu net
- Độ rộng: các Port khác size có thể được kết nối hợp lệ. Tuy nhiên có thể phần mềm tổng hợp sẽ báo lỗi
- Các port không kết nối, được chấp nhận bằng dấu phẩy
- Kiểu dữ liệu net được dùng để kết nối cấu trúc



Kết nối các module theo thứ tự Port

- Thứ tự Port phải chính xác
- Xảy ra vấn đề khi debug (VD xác định vị trí Port có lỗi biên dịch), khi xóa hay thêm Port → không nên dùng

```
module adder_implicit (result,carry,r1, r2, ci)

// Input Port Declarations
input  [3:0]  r1;
input  [3:0]  r2;
input                ci;

// Output Port Declarations
output [3:0]  result;
output  carry;

// Port Wires
wire [3:0] r1;
wire [3:0] r2;
```

```
wire ci;
wire [3:0] result;
wire  carry ;
// Internal variables
wire  c1 ;
wire  c2 ;
wire  c3;
//Code Starts Here
addbit u0 ( r1[0] , r2[0] , ci , result[0] , c1 );
addbit u1 ( r1[1] , r2[1] , c1 , result[1] , c2 );
addbit u2 ( r1[2] , r2[2] , c2 , result[2] , c3 );
addbit u3 ( r1[3] , r2[3] , c3 , result[3] , carry
); endmodule
// End Of Module adder
```


Kết nối các module theo tên Port

- Chỉ cần đúng tên port trong các module con
- Không quan tâm đến thứ tự port

```
9 module adder_explicit (  
10 result      , // Output of the adder  
11 carry       , // Carry output of adder  
12 r1          , // first input  
13 r2          , // second input  
14 ci          , // carry input  
15 );  
16  
17 // Input Port Declarations  
18 input  [3:0]  r1      ;  
19 input  [3:0]  r2      ;  
20 input                ci      ;  
21  
22 // Output Port Declarations  
23 output [3:0]  result    ;  
24 output                carry ;  
25  
26 // Port Wires  
27 wire  [3:0]  r1      ;  
28 wire  [3:0]  r2      ;  
29 wire                ci      ;  
30 wire  [3:0]  result    ;  
31 wire                carry ;  
32
```

```
38 // Code Starts Here  
39 addbit u0 (  
40 .a      (r1[0])      ,  
41 .b      (r2[0])      ,  
42 .ci     (ci)         ,  
43 .sum     (result[0]) ,  
44 .co     (c1)         ,  
45 );  
46  
47 addbit u1 (  
48 .a      (r1[1])      ,  
49 .b      (r2[1])      ,  
50 .ci     (c1)         ,  
51 .sum     (result[1]) ,  
52 .co     (c2)         ,  
53 );  
54  
55 addbit u2 (  
56 .a      (r1[2])      ,  
57 .b      (r2[2])      ,  
58 .ci     (c2)         ,  
59 .sum     (result[2]) ,  
60 .co     (c3)         ,  
61 );  
62  
63 addbit u3 (  
64 .a      (r1[3])      ,  
65 .b      (r2[3])      ,  
66 .ci     (c3)         ,  
67 .sum     (result[3]) ,  
68 .co     (carry)      ,  
69 );  
70  
71 endmodule // End Of Module adder
```

Ví dụ port không kết nối ngầm định

```
1 module implicit();
2 reg clk,d,rst,pre;
3 wire q;
4
5 // Here second port is not connected
6 dff u0 ( q,clk,d,rst,pre);
7
8 endmodule
9
10 // D fli-flop
11 module dff (q, q_bar, clk, d, rst, pre);
12 input clk, d, rst, pre;
13 output q, q_bar;
14 reg q;
15
16 assign q_bar = ~q;
17
18 always @ (posedge clk)
19 if (rst == 1'b1) begin
20     q <= 0;
21 end else if (pre == 1'b1) begin
22     q <= 1;
23 end else begin
24     q <= d;
25 end
26
27 endmodule
```

Hàm hệ thống - System Tasks

- Dùng để giám sát và điều khiển quá trình mô phỏng
- Ký tự \$ chỉ định hàm hệ thống
- **\$monitor**
 - \$monitor(\$time, “%d %d %d”, address, sinout, cosout);
 - Hiển thị giá trị biến khi biến thay đổi giá trị
 - Mỗi lần thay đổi hiển thị giá trị trên 1 dòng
- **\$display**
 - \$display (“%d %d %d”,address,sinout,cosout);
 - hiển thị giá trị hiện tại của biến theo định dạng xác định ở một dòng mới
- **\$finish**: Thoát khỏi mô phỏng
- **\$stop**: Dừng mô phỏng, có thể tiếp tục bởi user

Ngôn ngữ lập trình Verilog HDL

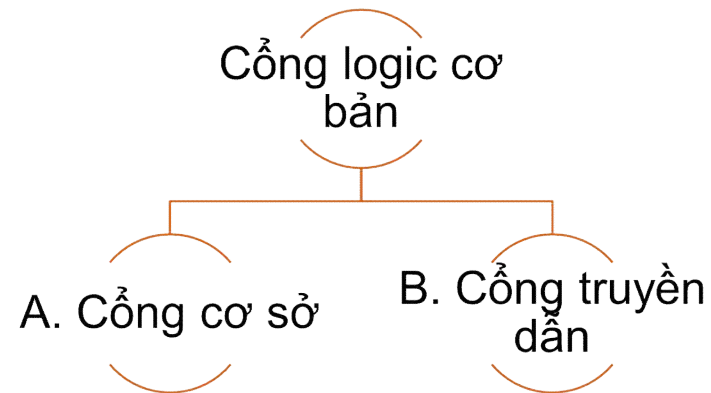
- Giới thiệu chung
 - Phương pháp thiết kế dùng Verilog HDL
 - Thiết kế top-down
 - Phần mềm hỗ trợ thiết kế
- Các mô hình thiết kế Verilog
 - Mô hình mức cổng (gate-level)
 - Mô hình mức luồng dữ liệu
 - Mô hình hành vi
- Một số ví dụ cụ thể

Mô hình thiết kế Verilog mức cổng

- Là mức thấp nhất trong 4 mô hình của Verilog
- Mạch được mô tả trên cơ sở các cổng AND, OR, NAND,...
- Tương ứng 1-1 giữa sơ đồ mạch logic và mô tả Verilog
- Phù hợp với người có kiến thức cơ bản về mạch số
- Ít được dùng trong thiết kế
- Dùng trong giai đoạn tổng hợp (synthesis) để mô tả các cell ASIC/FPGA

Mô hình thiết kế Verilog mức cổng

- Các kiểu cổng



- Giá trị logic các cổng

Logic Value	Description
0	zero, low, false
1	one, high, true
z or Z	high impedance, floating
x or X	unknown, uninitialized, contention

Các cổng cơ sở

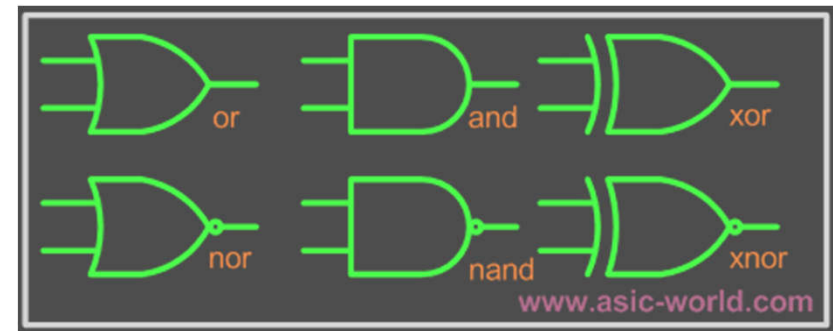
- Nhóm cổng thực hiện phép toán - 6 cổng:
and, nand, or, nor, xor, xnor

- Khai báo

```
wire OUT, IN1, IN2;
```

- Sử dụng

```
and a1 (OUT, IN1, IN2);  
nand na1 (OUT, IN1, IN2);  
nand na1_3in (OUT, IN1, IN2, IN3);  
// cổng 3 đầu vào  
and (OUT, IN1, IN2);  
// cổng không cần tên khởi tạo
```



Gate	Description
and	N-input AND gate
nand	N-input NAND gate
or	N-input OR gate
nor	N-input NOR gate
xor	N-input XOR gate
xnor	N-input XNOR gate

Các cổng buf/not

- Nhóm cổng **đệm/đảo** - 6 cổng: **buf**, **not**, **bufif1**, **notif1**, **bufif0**, **notif0**
- Có thể có 1 hoặc nhiều đầu ra
- Khai báo

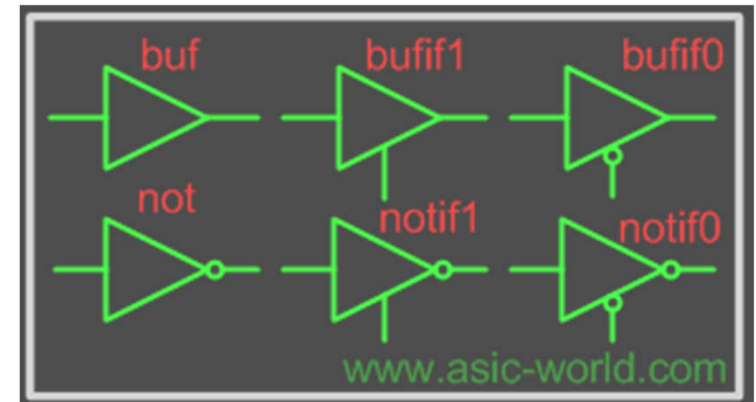
```
wire OUT, OUT1, IN;
```

- Sử dụng

```
buf b1 (OUT, IN);
not n1 (OUT1, IN);
buf b1_2output (OUT1, OUT2, IN);
```

- Chuỗi cổng

```
wire [7:0] OUT, IN1, IN2;
nand n_gate [7:0] (OUT, IN1, IN280);
```



Gate	Description
not	N-output inverter
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Các cổng buf/not

- Nhóm cổng **đệm/đảo** - 6 cổng: **buf**, **not**, **bufif1**, **notif1**, **bufif0**, **notif0**
- Có thể có 1 hoặc nhiều đầu ra
- Khai báo

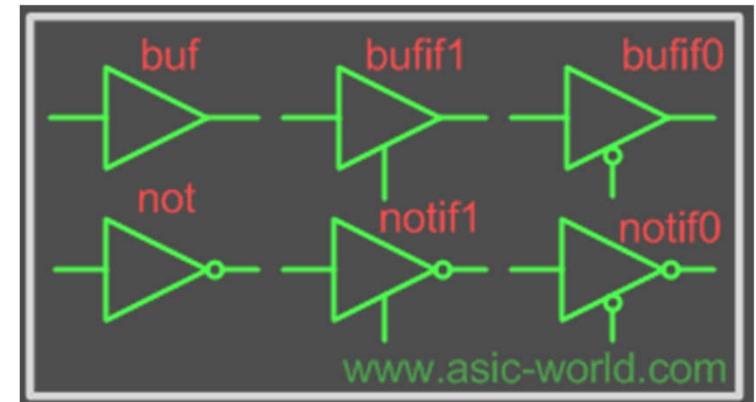
```
wire OUT, OUT1, IN;
```

- Sử dụng

```
buf b1 (OUT, IN);
not n1 (OUT1, IN);
buf b1_2output (OUT1, OUT2, IN);
```

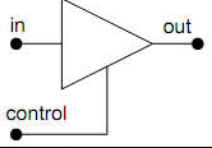
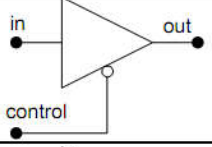
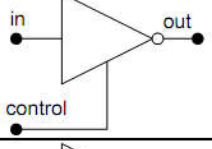
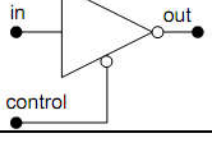
- Chuỗi cổng

```
wire [7:0] OUT, IN1, IN2;
nand n_gate [7:0] (OUT, IN1, IN281);
```



Gate	Description
not	N-output inverter
buf	N-output buffer.
bufif0	Tri-state buffer, Active low en.
bufif1	Tri-state buffer, Active high en.
notif0	Tristate inverter, Low en.
notif1	Tristate inverter, High en.

Thông tin thêm

Typical instantiation	Functional representation	Functional description
bufif1 (out, in, control);		Out = in if control = 1; else out = z
bufif0 (out, in, control);		Out = in if control = 0; else out = z
notif1 (out, in, control);		Out = complement of in if control = 1; else out = z
notif0 (out, in, control);		Out = complement of in if control = 0; else out = z

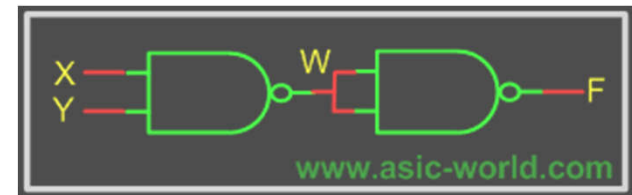
Type of gate	0 output state	1 output state	x output state
AND	Any one of the inputs is zero	All the inputs are at one	All other cases
NAND	All the inputs are at one	Any one of the inputs is zero	
OR	All the inputs are at zero	Any one of the inputs is one	
NOR	Any one of the inputs is one	All the inputs are at zero	
XOR	If every one of the inputs is definite at zero or one, the output is zero or one as decided by the XOR or XNOR function		If any one of the inputs is at x or z state, the output is at x state
XNOR			
BUF	If the only input is at 0 state	If the only input is at 1 state	All other cases of inputs
NOT	If the only input is at 1 state	If the only input is at 0 state	

Thiết kế dùng cổng

- Chỉ dùng khi xây dựng thư viện
 - Nhà cung cấp ASIC sẽ cung cấp thư viện ASIC Verilog dùng các cổng cơ bản
 - Và người dùng có thể tự định nghĩa các cổng (mạch) - UDP
- Yêu cầu sinh viên
 - Bài 1: Xây dựng cổng AND từ cổng NAND
 - Bài 2: Xây dựng D-FF từ cổng NAND
 - Bài 3: Xây dựng bộ cộng 1 bit và bộ cộng 4 bit
 - Bài 4: Xây dựng bộ ghép kênh 4:1
 - Bài 5: Xây dựng bộ so sánh 8 bit
 - Bài 6: Xây dựng bộ chuyển mã BCD

Bài 1: Xây dựng cổng AND từ cổng NAND

```
1 // Structural model of AND gate from two NANDS
2 module and_from_nand();
3
4 reg X, Y;
5 wire F, W;
6 // Two instantiations of the module NAND
7 nand U1(W,X, Y);
8 nand U2(F, W, W);
9
10 // Testbench Code
11 initial begin
12     $monitor ("X = %b Y = %b F = %b", X, Y, F);
13     X = 0;
14     Y = 0;
15     #1 X = 1;
16     #1 Y = 1;
17     #1 X = 0;
18     #1 $finish;
19 end
20
21 endmodule
```



X = 0	Y = 0	F = 0
X = 1	Y = 0	F = 0
X = 1	Y = 1	F = 1
X = 0	Y = 1	F = 0

Ngôn ngữ lập trình Verilog HDL

- Giới thiệu chung
 - Phương pháp thiết kế dùng Verilog HDL
 - Thiết kế top-down
 - Phần mềm hỗ trợ thiết kế
- Các mô hình thiết kế Verilog
 - Mô hình mức cổng (gate-level)
 - **Mô hình mức luồng dữ liệu**
 - Mô hình hành vi
- Một số ví dụ cụ thể

Mô hình mức luồng dữ liệu

- Thiết kế dựa trên mô tả luồng dữ liệu giữa các thanh ghi
- Module được thiết kế dựa trên việc mô tả luồng dữ liệu vào ra và cách xử lý chúng
- Hiệu quả hơn mô hình mức cổng trong các thiết kế phức tạp, số lượng cổng lớn

Mức cổng VS mức luồng dữ liệu

Gate level

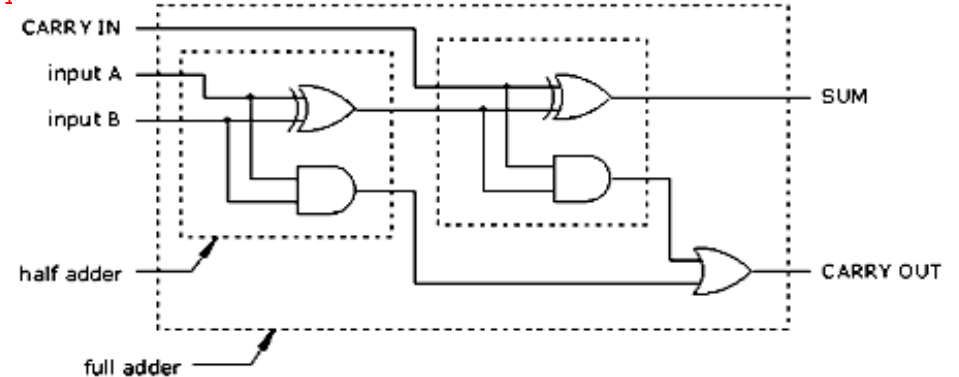
```
module fulladder
(sum,c_out,a, b, c_in);
output sum,c_out;
input a, b, c_in;
wire s1, c1, c2;

//khai báo cổng logic
xor (s1,a,b);
and (c1,a,b);
xor (sum,s1,c_in);
and (s2,s1,c_in);
xor (c_out, s2,c1);

endmodule
```

Dataflow level

```
module full_adder
(sum, c_out, a, b, c_in);
output sum, c_out;
input in0, in1, c_in;
assign
{c_out,sum} = a + b + c_in;
endmodule
```



Phép gán liên tục - assign

- là phát biểu cơ bản nhất trong mô hình luồng dữ liệu
- được dùng để đưa 1 giá trị vào biến nét
- Phép gán assign sẽ thay thế các cổng trong việc mô tả mạch điện

```
assign out = in1&in2;
```

net (wire) net or reg or function

- Phía bên trái của phép gán chỉ là **net**, không được phép là reg
- Toán hạng bên phải có thể là **net**, **reg** hoặc **function**

Phép gán liên tục - assign

- Câu lệnh nằm ngoài các khối thủ tục (always và initial block)
- Phép gán liên tục sẽ ghi đè mọi phép gán thủ tục
- Phân chia thành 2 loại

Gán liên tục thông thường

```
//gán liên tục thông thường  
wire out;  
assign out = in1&in2;
```

Gán liên tục ngầm định

```
// gán ngầm định  
wire out = in1&in2;
```

Thay thế việc khai báo net và viết phép gán liên tục trên net

Phương trình, toán tử và toán hạng

- Mô hình luồng dữ liệu là thiết kế trên cơ sở các phương trình, toán tử và toán hạng
- Toán hạng: có thể là bất cứ dữ liệu nào như hằng số, số nguyên, số thực, dữ liệu kiểu net hoặc reg
- Toán tử: có thể là các phép toán số học (+ - * /) phép toán logic (AND OR) hoặc phép so sánh (= < >)
- Phương trình: là sự kết hợp của toán tử, toán hạng để đưa ra kết quả

```
real a, b, c;  
c= a-b;
```

Toán tử số học

- Thực hiện phép toán số học: cộng (+), trừ (-), nhân (*), chia (/), lũy thừa (**), chia lấy phần dư modulus (%)
- Nếu có bất kì 1 bit nào của 1 trong 2 toán hạng là “x” thì kết quả là “x”
- Toán tử modulus không được phép có biến kiểu dữ liệu thực, kết quả lấy dấu của toán hạng đầu tiên
- Toán hạng + và - còn được sử dụng để chỉ kiểu dấu (kiểu unary). Khi làm dấu, nó có độ ưu tiên cao hơn so với phép toán

```
-4          // Negative 4
+5          // Positive 5
-10 / 5     // Evaluates to -2
```

Toán tử số học

```
A = 4'b0011; B = 4'b0100;
// A and B are register vectors
D = 6; E = 4; F=2
// D and E are integers

A * B // Multiply A and B. Evaluates to
4'b1100

D / E // Divide D by E. Evaluates to 1.
Truncates any fractional part.

A + B // Add A and B. Evaluates to
4'b0111

B - A // Subtract A from B. Evaluates to
4'b0001

F = E ** F; //E to the power F, yields 16
```

```
//Nếu có bất kỳ một toán hạng nào là
'x' → kết quả là 'x'
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be
evaluated to the value 4'bx

//modulus: lấy phần dư
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign
of the first operand
7 % -2 // Evaluates to +1, takes
sign of the first operand
```

Toán tử logic

- Phép toán logic luôn có giá trị 1 bit, nhận 1 trong các giá trị FALSE (0), TRUE (1) và KHÔNG XÁC ĐỊNH (x)
- Phép toán nhận các biến và các biểu thức như các toán hạng
- Coi tất cả các giá trị khác 0 đều là 1
- Toán tử logic được dùng nhiều trong các câu điều kiện (if...else), khi chúng làm việc trên biểu thức

Toán tử logic

```
1 module logical_operators();
2
3 initial begin
4     // Logical AND
5     $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
6     $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
7     $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
8     // Logical OR
9     $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
10    $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
11    $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
12    // Logical Negation
13    $display ("! 1'b1      = %b", (! 1'b1));
14    $display ("! 1'b0      = %b", (! 1'b0));
15    #10 $finish;
16 end
17
18 endmodule
```

Operator	Description
!A	not A
A && B	A and B
A B	A or B

```
1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1
1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
! 1'b1      = 0
! 1'b0      = 1
```

Toán tử quan hệ

- Toán tử quan hệ so sánh 2 toán hạng với nhau, trả về một bit đơn là 0 hoặc 1
- Biểu thức nhận giá trị 1 nếu đúng và 0 nếu sai
- Nếu có bất kỳ giá trị x hoặc z nào, biểu thức sẽ nhận giá trị là x

Operator Description	
$a < b$	a nhỏ hơn b
$a > b$	a lớn hơn b
$a \leq b$	a nhỏ hơn hoặc bằng b
$a \geq b$	a lớn hơn hoặc bằng b

```
1 module relational_operators();
2
3 initial begin
4     $display (" 5  <= 10 = %b", (5      <= 10));
5     $display (" 5  >= 10 = %b", (5      >= 10));
6     $display (" 1'bx <= 10 = %b", (1'bx  <= 10));
7     $display (" 1'bz <= 10 = %b", (1'bz  <= 10));
8     #10 $finish;
9 end
10
11 endmodule
```

```
5  <= 10 = 1
5  >= 10 = 0
1'bx <= 10 = x
1'bz <= 10 = x
```

Toán tử so sánh

- So sánh từng bit các toán hạng
- Kết quả là 0 (false) hoặc 1 (true) hoặc x (không xác định)

Operator	Description	Result
a === b	a bằng b (gồm cả x và z)	1 or 0
a !== b	a không bằng b (gồm cả bit x và z)	1 or 0
a == b	a bằng b, kết quả có thể không xác định	1,0,x
a != b	a không bằng b, kết quả có thể không xác định	1,0,x

Toán tử so sánh

```
1 module equality_operators();
2
3 initial begin
4     // Case Equality
5     $display (" 4'bx001 === 4'bx001 = %b", (4'bx001 === 4'bx001));
6     $display (" 4'bx0x1 === 4'bx001 = %b", (4'bx0x1 === 4'bx001));
7     $display (" 4'bz0x1 === 4'bz0x1 = %b", (4'bz0x1 === 4'bz0x1));
8     $display (" 4'bz0x1 === 4'bz001 = %b", (4'bz0x1 === 4'bz001));
9     // Case Inequality
10    $display (" 4'bx0x1 !== 4'bx001 = %b", (4'bx0x1 !== 4'bx001));
11    $display (" 4'bz0x1 !== 4'bz001 = %b", (4'bz0x1 !== 4'bz001));
12    // Logical Equality
13    $display (" 5 == 10 = %b", (5 == 10));
14    $display (" 5 == 5 = %b", (5 == 5));
15    // Logical Inequality
16    $display (" 5 != 5 = %b", (5 != 5));
17    $display (" 5 != 6 = %b", (5 != 6));
18    #10 $finish;
19 end
20
21 endmodule
```

4'bx001	===	4'bx001	=	1
4'bx0x1	===	4'bx001	=	0
4'bz0x1	===	4'bz0x1	=	1
4'bz0x1	===	4'bz001	=	0
4'bx0x1	!==	4'bx001	=	1
4'bz0x1	!==	4'bz001	=	1
5	==	10	=	0
5	==	5	=	1
5	!=	5	=	0
5	!=	6	=	1

Toán tử bitwise

- Thực hiện **phép toán logic theo từng bit** của 2 toán hạng theo đúng thứ tự
- Nếu 1 toán hạng có chiều dài nhỏ hơn, nó sẽ thêm vào các bit 0 để 2 toán hạng có độ dài bằng nhau
- Sự khác biệt đối với phép toán logic: Kết quả phép toán logic là 1 bit, kết quả của phép bitwise là nhiều bit phụ thuộc độ dài của dữ liệu
 - $\sim x = x$
 - $0 \& x = 0$
 - $1 \& x = x \& x = x$
 - $1 | x = 1$
 - $0 | x = x | x = x$
 - $0 \wedge x = 1 \wedge x = x \wedge x = x$
 - $0 \wedge \sim x = 1 \wedge \sim x = x \wedge \sim x = x$

Operator	Description
\sim	negation
$\&$	and
$ $	inclusive or
\wedge	exclusive or
$\wedge \sim$ or $\sim \wedge$	exclusive nor (equivalence)

Toán tử bitwise

```

1 module bitwise_operators();
2
3 initial begin
4     // Bit Wise Negation
5     $display (" ~4'b0001      = %b", (~4'b0001));
6     $display (" ~4'bx001      = %b", (~4'bx001));
7     $display (" ~4'bz001      = %b", (~4'bz001));
8     // Bit Wise AND
9     $display (" 4'b0001 & 4'b1001 = %b", (4'b0001 & 4'b1001));
10    $display (" 4'b1001 & 4'bx001 = %b", (4'b1001 & 4'bx001));
11    $display (" 4'b1001 & 4'bz001 = %b", (4'b1001 & 4'bz001));
12    // Bit Wise OR
13    $display (" 4'b0001 | 4'b1001 = %b", (4'b0001 | 4'b1001));
14    $display (" 4'b0001 | 4'bx001 = %b", (4'b0001 | 4'bx001));
15    $display (" 4'b0001 | 4'bz001 = %b", (4'b0001 | 4'bz001));
16    // Bit Wise XOR
17    $display (" 4'b0001 ^ 4'b1001 = %b", (4'b0001 ^ 4'b1001));
18    $display (" 4'b0001 ^ 4'bx001 = %b", (4'b0001 ^ 4'bx001));
19    $display (" 4'b0001 ^ 4'bz001 = %b", (4'b0001 ^ 4'bz001));
20    // Bit Wise XNOR
21    $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001));
22    $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001));
23    $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001));
24    #10 $finish;
25 end
26
27 endmodule

```

```

~4'b0001      = 1110
~4'bx001      = x110
~4'bz001      = x110
4'b0001 & 4'b1001 = 0001
4'b1001 & 4'bx001 = x001
4'b1001 & 4'bz001 = x001
4'b0001 | 4'b1001 = 1001
4'b0001 | 4'bx001 = x001
4'b0001 | 4'bz001 = x001
4'b0001 ^ 4'b1001 = 1000
4'b0001 ^ 4'bx001 = x000
4'b0001 ^ 4'bz001 = z000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111

```

Thuật toán Reduction

- Thực hiện phép toán bitwise giữa các bit của 1 toán hạng, và lấy kết quả 1 bit
- Các bit có giá trị x được xử lý như trong thuật toán bitwise

Operator	Description
&	and
~&	nand
	or
~	nor
^	xor
^~ or ~^	xnor

```

& 4'b1001 = 0
& 4'bx111 = x
& 4'bz111 = x
~& 4'b1001 = 1
~& 4'bx001 = 1
~& 4'bz001 = 1
| 4'b1001 = 1
| 4'bx000 = x
| 4'bz000 = x
~| 4'b1001 = 0
~| 4'bx001 = 0
~| 4'bz001 = 0
^ 4'b1001 = 0
^ 4'bx001 = x
^ 4'bz001 = x
~^ 4'b1001 = 1
~^ 4'bx001 = x
~^ 4'bz001 = x

```

100

```

1 module reduction_operators();
2
3 initial begin
4     // Bit Wise AND reduction
5     $display (" & 4'b1001 = %b", (& 4'b1001));
6     $display (" & 4'bx111 = %b", (& 4'bx111));
7     $display (" & 4'bz111 = %b", (& 4'bz111));
8     // Bit Wise NAND reduction
9     $display (" ~& 4'b1001 = %b", (~& 4'b1001));
10    $display (" ~& 4'bx001 = %b", (~& 4'bx001));
11    $display (" ~& 4'bz001 = %b", (~& 4'bz001));
12    // Bit Wise OR reduction
13    $display (" | 4'b1001 = %b", (| 4'b1001));
14    $display (" | 4'bx000 = %b", (| 4'bx000));
15    $display (" | 4'bz000 = %b", (| 4'bz000));
16    // Bit Wise NOR reduction
17    $display (" ~| 4'b1001 = %b", (~| 4'b1001));
18    $display (" ~| 4'bx001 = %b", (~| 4'bx001));
19    $display (" ~| 4'bz001 = %b", (~| 4'bz001));
20    // Bit Wise XOR reduction
21    $display (" ^ 4'b1001 = %b", (^ 4'b1001));
22    $display (" ^ 4'bx001 = %b", (^ 4'bx001));
23    $display (" ^ 4'bz001 = %b", (^ 4'bz001));
24    // Bit Wise XNOR
25    $display (" ^~ 4'b1001 = %b", (^~ 4'b1001));
26    $display (" ^~ 4'bx001 = %b", (^~ 4'bx001));
27    $display (" ^~ 4'bz001 = %b", (^~ 4'bz001));
28    #10 $finish;
29 end
30
31 endmodule

```

Toán tử điều kiện

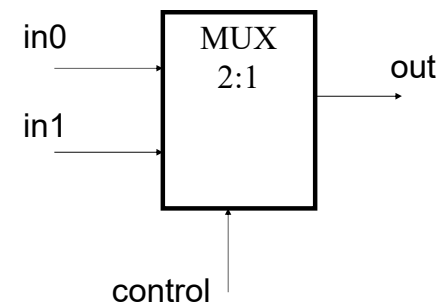
- Có 3 toán hạng
- Cú pháp

`Condition_expr>true_expr:false_expr`

- Kiểm tra `Condition_expr`
 - Đúng: nhận giá trị `true_expr`
 - Sai: nhận giá trị `false_expr`

- Bộ ghép kênh MUX 2:1
 - `control=0` thì `out=in0`
 - `control=1` thì `out=in1`

```
assign out = control ? in1 : in0;
```



Một số toán tử khác

- Phép dịch
 - Khi các bit được dịch, vị trí các bit bỏ trống sẽ bằng 0
 - Phép dịch sẽ không quay vòng
 - Toán hạng bên trái sẽ dịch số bit tương ứng với toán hạng bên phải

Ký hiệu	Mô tả
<<	Dịch trái
>>	Dịch phải

```
1 module shift_operators();
2
3 initial begin
4     // Left Shift
5     $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
6     $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
7     $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
8     // Right Shift
9     $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
10    $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
11    $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
12    #10 $finish;
13 end
14
15 endmodule
```

```
4'b1001 <<1 = 0010
4'b10x1 <<1 = 0x10
4'b10z1 <<1 = 0z10
4'b1001 >> 1 = 0100
4'b10x1 >> 1 = 010x
4'b10z1 >> 1 = 010z
```

Một số toán tử khác

- Ghép nối các toán hạng
 - Ghép nhiều toán hạng thành 1 toán hạng
 - Các toán hạng phải được định cỡ trước

```
1 module concatenation_operator();  
2  
3 initial begin  
4     // concatenation  
5     $display (" {4'b1001,4'b10x1} = %b", {4'b1001,4'b10x1});  
6     #10 $finish;  
7 end  
8  
9 endmodule
```

{4'b1001,4'b10x1} = 100110x1

Một số toán tử khác

- Lặp toán hạng
 - dùng để lặp n lần một nhóm bit

Ký hiệu	Mô tả
$\{n\{m\}\}$	Lặp lại n lần giá trị m

- Có thể kết hợp ghép và lặp toán hạng

```
{3{a}} // Tương ứng với {a, a, a}
{b, {3{c, d}}}
```

// tương ứng với {b, c, d, c, d, c, d}

```
1 module replication_operator();
2
3 initial begin
4     // replication
5     $display (" {4{4'b1001}} = %b", {4{4'b1001}});
6     // replication and concatenation
7     $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
8     #10 $finish;
9 end
10
11 endmodule
```

```
{4{4'b1001}} = 1001100110011001
{4{4'b1001,1'bz}} = 1001z1001z1001z1001z
```

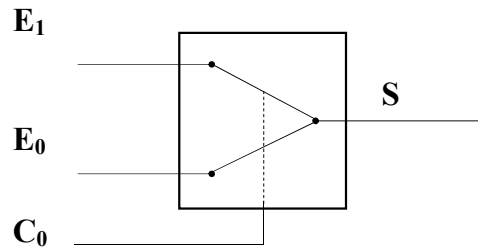

Thứ tự ưu tiên

Operator	Symbols
Unary, Multiply, Divide, Modulus	!, ~, *, /, %
Add, Subtract, Shift	+, - , <<, >>
Relation, Equality	<, >, <=, >=, ==, !=, ===, !==
Reduction	&, !&, ^, ^~, , ~
Logic	&&,
Conditional	? :

Bài tập

- Bài 1: Thiết kế bộ ghép kênh
 - MUX 2:1
 - MUX 4:1
- Bài 2: Thiết kế bộ so sánh
- Bài 3: Mạch mã hoá/giải mã
- Bài 4: Tạo và kiểm tra bit chẵn lẻ

Bài 1: Bộ MUX 2:1



C_0	S
0	E_0
1	E_1

$$S = \overline{C_0}E_0 + C_0E_1$$

Logic equation

```

module mux_2_1(c0,e0,e1,s);
input c0,e0,e1;
output s;
assign
    s=!c0 && e0||c0 && e1;
endmodule
    
```

Conditional operator

```

module mux_2_1(c0,e0,e1,s);
input c0,e0,e1;
output s;
assign s = (c0)?e1 :e0;
endmodule
    
```

	Msgs								
/mux_2_1_tb/c0	1'h0								
/mux_2_1_tb/e0	1'h0								
/mux_2_1_tb/e1	1'h0								
/mux_2_1_tb/s	1'h0								

Ngôn ngữ lập trình Verilog HDL

- Giới thiệu chung
 - Phương pháp thiết kế dùng Verilog HDL
 - Thiết kế top-down
 - Phần mềm hỗ trợ thiết kế
- Các mô hình thiết kế Verilog
 - Mô hình mức cổng (gate-level)
 - Mô hình mức luồng dữ liệu
 - **Mô hình hành vi**
- Một số ví dụ cụ thể

Mô hình hành vi

- là mô hình mức cao nhất, mô tả hành vi của mạch logic
- Sử dụng các ngôn ngữ mức cao
 - for loop
 - if else
 - while
- Các lệnh nằm trong khối thủ tục
- Có 2 kiểu khối thủ tục
 - always - được thực hiện lặp đi lặp lại
 - initial - được thực hiện tại thời điểm 0

Các khối thủ tục

- Thành phần khối thủ tục
 - Các lệnh gán thủ tục
 - Các cấu trúc mức cao (vòng lặp, các lệnh điều kiện,...)
 - Điều khiển thời gian

```
module initial_example();  
    reg clk,reset,enable,data;  
    initial begin  
        clk = 0;  
        reset = 0;  
        enable = 0;  
        data = 0;  
    end  
endmodule
```

Khối **initial**

- Được thực hiện tại thời điểm 0
- Chỉ thực hiện các câu lệnh trong khoảng **begin** và **end** mà không cần đợi event
- Một module có thể có nhiều initial
 - đều được thực hiện tại $t=0$
 - Trong các khối khác nhau được thực hiện đồng thời
 - Trong một khối thực hiện tuần tự

```
module initial_example();  
    reg clk,reset,enable,data;  
    initial begin  
        clk = 0;  
        reset = 0;  
        enable = 0;  
        data = 0;  
    end  
endmodule
```

Khối **always**

- Khi xuất hiện sự kiện, đoạn mã giữa **begin** và **end** sẽ được thực hiện
- Luôn đợi 1 sự kiện (event) để thực hiện
- Quá trình đợi và thực hiện lệnh sẽ lặp đi lặp lại đến khi kết thúc mô phỏng
- Được dùng để mô hình hoá 1 khối hành vi được lặp đi lặp lại trong mạch số
- Các câu lệnh giữa **begin** và **end** được thực hiện tuần tự

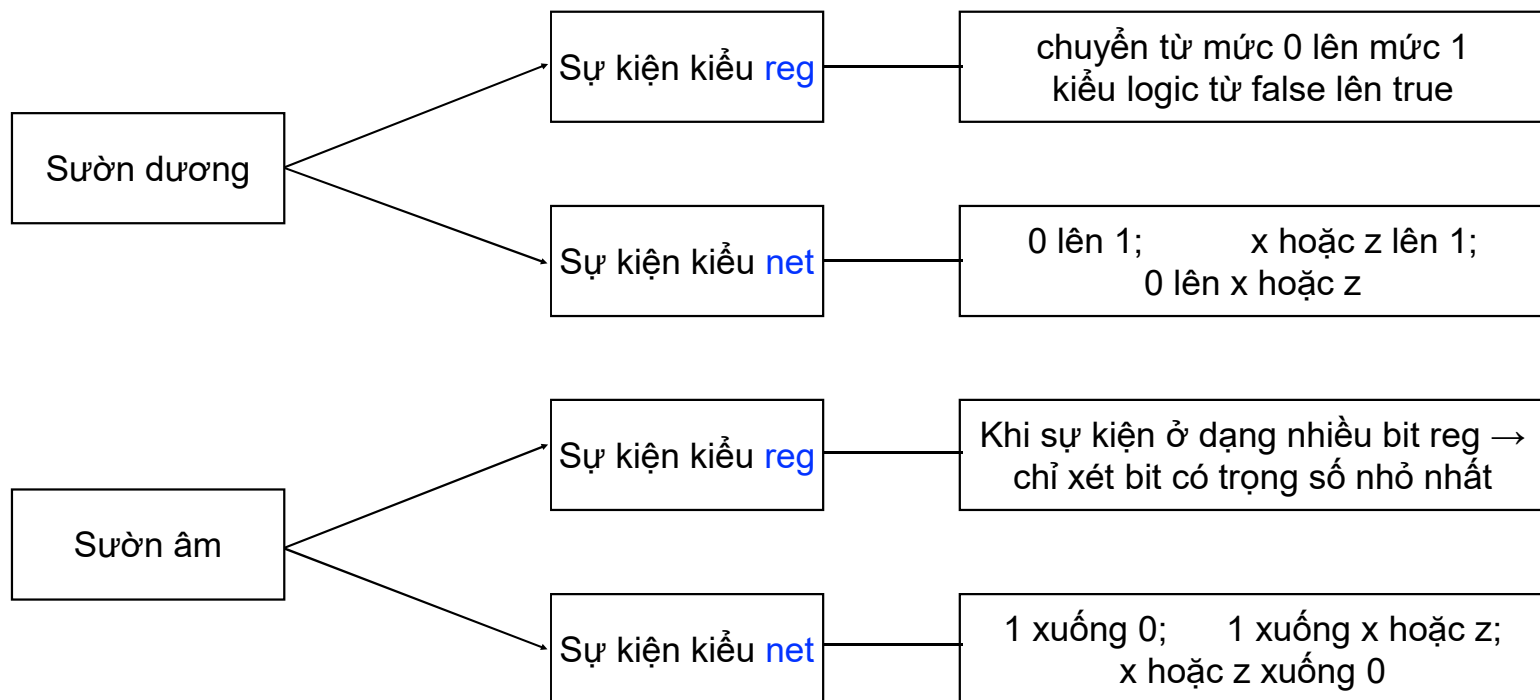
□ Ví dụ

```
module always_example();  
    reg clk, reset, enable, q_in, data;  
    always @(posedge clk)  
        if (reset) begin  
            data <= 0;  
        end else if (enable) begin  
            data <= q_in;  
        end  
endmodule
```


Các kiểu sự kiện - event Control

- event control không chấp nhận toán tử logic và số học trong event
 - @ (posedge clk): tại sườn dương của xung clock
 - @ (negedge clk): tại sườn âm của xung clock
 - @ (clk): khi clk thay đổi trạng thái (tại cả 2 sườn)
 - @ (posedge clk1 or clk2):
 - ▶ tại sườn dương của clk1 hoặc khi clk2 thay đổi trạng thái
 - ▶ “or” thực hiện khi có ít nhất 1 sự kiện xảy ra, có thể thay “or” thành dấu “,”. Ví dụ: @(a,b,c) tương đương với @(a or b or c)

Các kiểu sự kiện - event Control



Kiểu dữ liệu gán trong khối

```
module initial_bad();  
    reg clk,reset;  
    wire enable,data;  
    initial  
    begin  
        clk = 0;  
        reset = 0;  
        enable = 0;  
        data = 0;  
    end  
endmodule
```

```
module initial_good();  
    reg clk,reset,enable,data;  
    initial  
    begin  
        clk = 0;  
        reset = 0;  
        enable = 0;  
        data = 0;  
    end  
endmodule
```

Nhóm trong khối

- Nếu trong 1 khối thủ tục có nhiều câu lệnh thì các câu lệnh này phải nằm trong:
 - Khối nối tiếp `begin-end`
 - Khối song song `fork-join`
- Khi dùng `begin-end` có thể đặt tên cho nhóm đó → khối được đặt tên

Khối **begin-end**

```
module initial_begin_end();
reg clk,reset,enable,data;
initial
begin
    $monitor( "%g clk=%b reset=%b
enable=%b data=%b", $time, clk,
reset, enable, data);
    #1 clk = 0;
    #10 reset = 0;
    #5 enable = 0;
    #3 data = 0;
    #1 $finish;
end
endmodule
```

- **begin:**

- clk = 0 sau 1 đơn vị thời gian
- reset = 0 sau 11 đơn vị thời gian
- enable = 0 sau 16 đơn vị thời gian
- data = 0 sau 19 đơn vị thời gian

- **Mô phỏng:**

0 clk clk=x reset=x enable=x data=x

1 clk clk=0 reset=x enable=x data=x

11 clk clk=0 reset=0 enable=x data=x

16 clk clk=0 reset=0 enable=0 data=x

19 clk clk=0 reset=0 enable=0 data=0

Khởi fork-join

```
module initial_fork_join();
    reg clk,reset,enable,data;
    initial
    begin
        $monitor("%g clk=%b reset=%b
enable=%b data=%b", $time, clk,
reset, enable, data);
        fork
            #1 clk = 0;
            #10 reset = 0;
            #5 enable = 0;
            #3 data = 0;
        join
            #1 $display ("%g Terminating
simulation", $time); $finish;
        end
    endmodule
```

- **begin:**

- clk = 0 sau 1 đơn vị thời gian
- reset = 0 sau 10 đơn vị thời gian
- enable = 0 sau 5 đơn vị thời gian
- data = 0 sau 3 đơn vị thời gian

- **Mô phỏng:**

0 clk clk=x reset=x enable=x data=x

1 clk clk=0 reset=x enable=x data=x

3 clk clk=0 reset=x enable=x data=0

5 clk clk=0 reset=x enable=0 data=0

10 clk clk=0 reset=0 enable=0 data=0

11 Terminating simulation

Phép gán

- là cơ chế cơ bản để đưa giá trị vào nets và register
- Phép gán gồm
 - Vế trái: Chỉ định biến gán cho vế phải
 - Vế phải: Biểu thức bất kì cần đánh giá giá trị

Phép gán liên tục và gán thủ tục

Gán liên tục

- Mô hình luồng dữ liệu - assign
- Gán các giá trị vào biến net → vế trái kiểu net
- Vế trái sẽ cập nhật giá trị khi có bất kỳ sự thay đổi của biểu thức vế phải (sau một khoảng trễ xác định nếu có)

Gán thủ tục

- Trong mô hình hành vi, phép gán chỉ xảy ra bên trong thủ tục initial hoặc always
- Gán các giá trị vào biến reg → vế trái là kiểu reg hoặc phần tử nhớ
- Vế trái sẽ giữ nguyên giá trị cho đến khi có phép gán thủ tục khác cập nhật giá trị cho biến

Phép gán liên tục và gán thủ tục

Gán liên tục

```
module la (a,b,c,d);  
input b,c,d;  
output a;  
wire a;  
assign a = b | (c & d);  
endmodule
```

Gán thủ tục

```
module la1;  
reg a;  
wire b,c,d;  
always @(*)  
begin  
a = b | (c & d);  
end  
endmodule
```

Phép gán liên tục - gán trong khối

- Gán trong khối, thực hiện liên tiếp

- Thực hiện bởi dấu “=”

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
Initial
begin
x = 0;
y = 1;
z = 1;           //Scalar assignments
count = 0;       //Assignment to integer variables
reg_a = 16'b0;
reg_b = reg_a; //initialize vectors
#15 reg_a[2] = 1'b1;
                //Bit select assignment with delay
#10 reg_b[15:13] = {x, y, z}
                //Assign result of concatenation to
                // part select of a vector
count = count + 1;
                //Assignment to an integer (increment)
end
```

- Các thời điểm thực hiện các phát biểu như sau

- Các phát biểu từ $x=0$ đến $reg_b=reg_a$ được thực hiện tại $t=0$
- $reg_a[2]=1'b1$ thực hiện tại thời điểm 15
- $reg_b[15:13]=\{x,y,z\}$ thực hiện tại thời điểm $15+10=25$
- $count=count+1$ được thực hiện tại thời điểm 25

Phép gán liên tục - gán không trong khối

- Gán không trong khối, thực hiện song song
- Thực hiện bởi dấu “<=”
- Có 3 phép gán không trong khối. Các thời điểm thực hiện các phát biểu như sau

```
reg x, y, z;  
reg [15:0] reg_a, reg_b;  
integer count;  
//All behavioral statements must be inside an initial or  
//always block  
Initial  
begin  
x = 0;  
y = 1;  
z = 1; //Scalar assignments  
count = 0; //Assignment to integer variables  
reg_a = 16'b0;  
reg_b = reg_a; //Initialize vectors  
reg_a[2] <= #15 1'b1; //Bit select assignment with delay  
reg_b[15:13] <= #10 {x, y, z};  
//Assign result of concatenation to part select of a vector  
count <= count + 1;  
//Assignment to an integer (increment)  
end
```

- Các phát biểu từ $x=0$ đến $reg_b=reg_a$ được thực hiện tại $t=0$
- $reg_a[2]=1'b1$ thực hiện tại thời điểm 15
- $reg_b[15:13]=\{x,y,z\}$ thực hiện tại thời điểm $15+10=25$
- $count=count+1$ được thực hiện tại thời điểm 25

Non-blocking VS Blocking

Non-Blocking

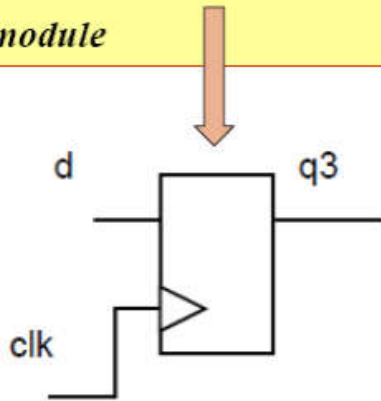
- Toán tử \leq
- Các phát biểu thực thi song song
- Thứ tự các phát biểu không ảnh hưởng đến kết quả cuối cùng
- Khi thực hiện hành vi vòng bộ mô phỏng, tính giá trị biểu thức bên phải trước khi gán cho vế trái
- Dùng trong khối **always** để thực hiện **mạch dãy**

Blocking

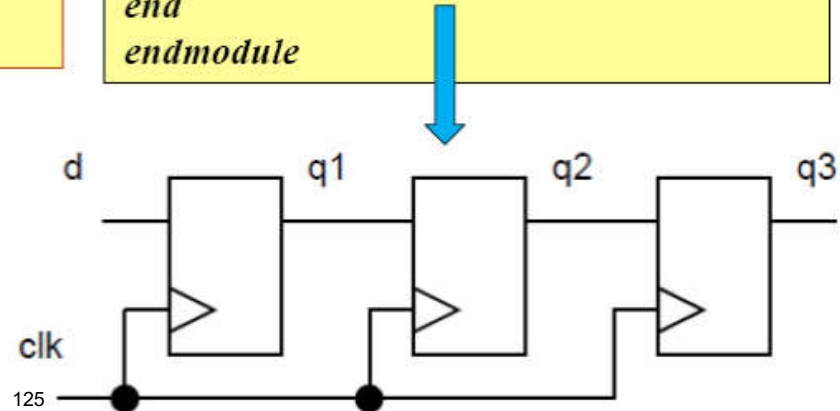
- Toán tử $=$
- Các phát biểu thực hiện tuần tự
- Thứ tự các phát biểu có thể ảnh hưởng đến kết quả cuối
- Khi thực hiện hành vi vòng bộ mô phỏng, chỉ tính giá trị biểu thức bên phải ngay sau khi phát biểu trước đó hoàn tất
- Dùng trong khối **always** để thực hiện **mạch tổ hợp**

Non-blocking VS Blocking

```
// Bad code - potential simulation race  
module pipeb1 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
    always @(posedge clk) begin  
        q1 = d;  
        q2 = q1;  
        q3 = q2;  
    end  
endmodule
```



```
// Good code  
module pipen1 (q3, d, clk);  
    output [7:0] q3;  
    input [7:0] d;  
    input clk;  
    reg [7:0] q3, q2, q1;  
    always @(posedge clk) begin  
        q1 <= d;  
        q2 <= q1;  
        q3 <= q2;  
    end  
endmodule
```



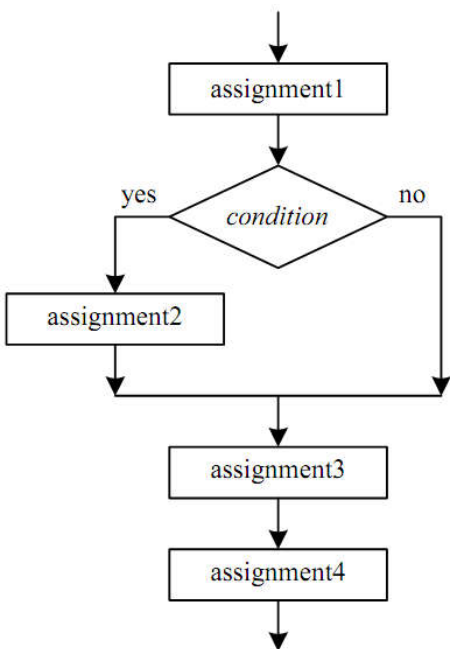
Non-blocking VS Blocking

- Trong thực tế khi có 1 phép chuyển giao dữ liệu xảy ra sau 1 sự kiện chung thì nên dùng gán trong khối

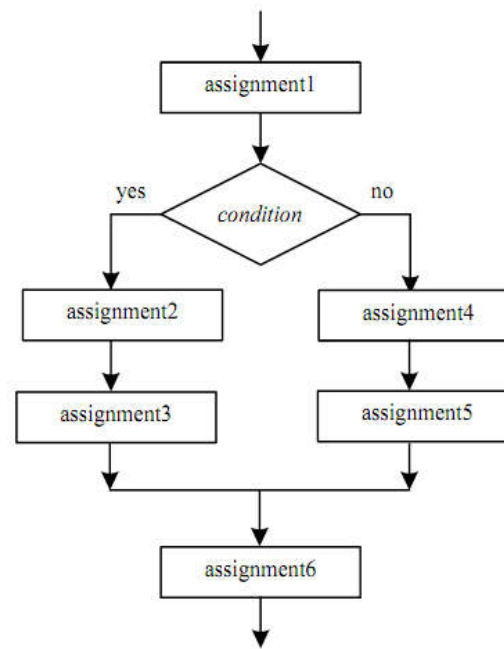
<pre>always @ (posedge clk) A=B; always @ (posedge clk) B=A;</pre>	<pre>always @ (posedge clk) begin A<=B; B<=A; end</pre>
→ Xung đột	→ A nhận giá trị trước đó của B, B nhận giá trị trước đó của A

Câu điều kiện if-then-else

- Được dùng để điều khiển các lệnh khác nhau
- Nếu có nhiều hơn 1 lệnh được thực hiện với **if** thì cần đặt trong **begin-end**



```
. . . .  
assignment1;  
if (condition) assignment2;  
assignment3;  
assignment4;  
. . . .
```



```
. . . .  
assignment1;  
if (condition)  
begin  
assignment2;  
assignment3;  
end  
else  
begin  
assignment4;  
assignment5;  
end  
assignment6;  
. . . .
```

Ví dụ if

```
module simple_if();

reg latch;
wire enable,din;

always @ (enable or din)
if (enable) begin
    latch <= din;
end

endmodule
```

```
module if_else();

reg dff;
wire clk,din,reset;

always @ (posedge clk)
if (reset) begin
    dff <= 0;
end else begin
    dff <= din;
end

endmodule
```


Ví dụ if

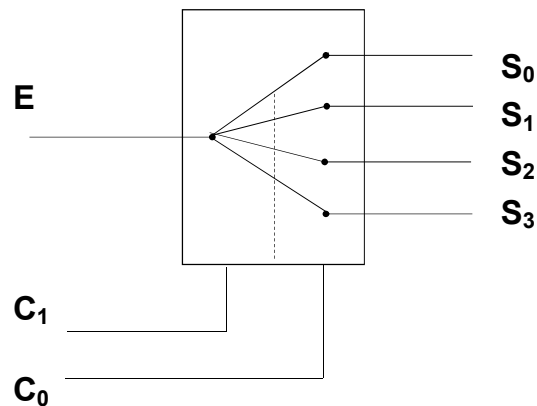
```
module nested_if();

reg [3:0] counter;
reg clk,reset,enable, up_en, down_en;

always @ (posedge clk)
// If reset is asserted
if (reset == 1'b0) begin
    counter <= 4'b0000;
// If counter is enable and up count is asserted
end else if (enable == 1'b1 && up_en == 1'b1) begin
    counter <= counter + 1'b1;
// If counter is enable and down count is asserted
end else if (enable == 1'b1 && down_en == 1'b1) begin
    counter <= counter - 1'b1;
// If counting is disabled
end else begin
    counter <= counter; // Redundant code
end
endmodule
```

Ví dụ bộ DEMUX 1:4

- Có 2 đầu vào và có 4 đầu ra
- Chức năng: đưa tín hiệu từ đầu vào đến 1 trong những đầu ra



C1	C0	S0	S1	S2	S3
0	0	E	z	z	z
0	1	z	E	z	z
1	0	z	z	E	z
1	1	z	z	z	E

Ví dụ bộ DEMUX 1:4

```
module demux4_1_beh(e,s,c);
input e;
input [1:0] c;
output s;
reg [3:0] s;
always @(c or e)
begin
    if (c==2'b00)
    begin
        s[0]=e;
        s[3:1]=3'bzzz;
    end
    else if (c==2'b01)
    begin
        s[1]=e;
        {s[3],s[2],s[0]}=3'bzzz;
    end
    else if (c==2'b10)
    begin
        s[2]=e;
        {s[3],s[1],s[0]}=3'bzzz;
    end
    else if (c==2'b11)
    begin
        s[3]=e;
        {s[2],s[1],s[0]}=3'bzzz;
    end
    else s=4'bzzzz;
end
endmodule
```

```
module demux4_1_beh_bad(e,s,c);
input e;
input [1:0] c;
output s;
reg [3:0] s;
always @(c or e)
begin
    if (c==2'b00)
    begin
        s[0]=e;
        s[3:1]=3'bzzz;
    end
    if (c==2'b01)
    begin
        s[1]=e;
        {s[3],s[2],s[0]}=3'bzzz;
    end
    if (c==2'b10)
    begin
        s[2]=e;
        {s[3],s[1],s[0]}=3'bzzz;
    end
    if (c==2'b11)
    begin
        s[3]=e;
        {s[2],s[1],s[0]}=3'bzzz;
    end
    else s=4'bzzzz;
end
endmodule
```

Ví dụ bộ DEMUX 1:4

- S hiển thị ở hệ nhị phân

```
module demux4_1_beh_tb();
    reg e;
    reg [1:0] c;
    wire [3:0] s;
    demux4_1_beh ff1 (e,s,c);
    initial
e =1'b1;
always
begin
    #2 c=2'b00;e=1'b1;
    #2 c=2'b00;e=1'b0;
    #2 c=2'b01;e=1'b0;
    #2 c=2'b10;e=1'b1;
    #2 c=2'b11;e=1'b0;
end
    initial #50 $stop;
endmodule
```

Câu điều kiện **case**

- Dùng để lựa chọn 1 biểu thức trong 1 loạt các trường hợp và thực hiện lệnh (nhóm lệnh) tương ứng
- Lệnh hỗ trợ lệnh đơn hoặc nhiều câu lệnh (kết hợp với begin-end)

```
case (expression)
  option 1: statement1;
  option 2 : statement2;
  option 3 : statement3;
  ...
  default: default_statement;    // optional,
  but recommended
endcase
```

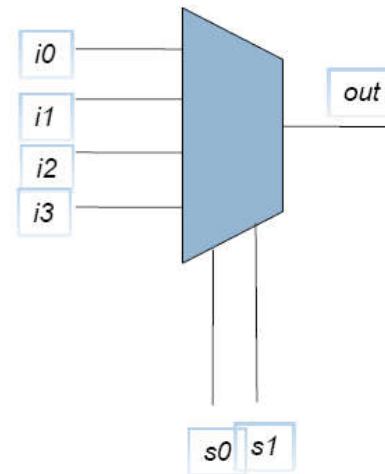
Notes:

- Luôn dùng lệnh *default* đặc biệt trong trường hợp kiểm tra giá trị x hoặc z.
- Mỗi lệnh *case* chỉ được phép có 1 lệnh *default*.

Câu điều kiện **case**

- Ví dụ bộ MUX 4:1

```
module mux4to1 (out, i0, i1, i2, i3, s1, s0);  
    output out;  
    input i0, i1, i2, i3, s0, s1;  
    reg out;  
  
    always @(s1 or s0 or i0 or i1 or i2 or i3)  
    case ({s1, s0}) // concatenated controls  
        2'd0 : out = i0;  
        2'd1 : out = i1;  
        2'd2 : out = i2;  
        2'd3 : out = i3;  
        default: out = i0;  
    endcase  
endmodule
```



casez và casex

- casez sẽ không quan tâm vị trí có giá trị z
- casex không quan tâm vị trí có giá trị x hoặc z

```
reg [3:0] encoding;  
integer state;  
casex (encoding) //logic value x  
    represents a don't care bit.  
4'b1xxx : next_state = 3;  
4'bx1xx : next_state = 2;  
4'bxx1x : next_state = 1;  
4'bxxx1 : next_state = 0;  
default : next_state = 0;  
endcase
```

Vòng lặp - **loop**

- Có 4 kiểu lặp loop tương tự C trong verilog
- Tất cả các lệnh loop chỉ xuất hiện trong khối **initial** hoặc **always**
- Vòng lặp có thể chứa các biểu thức trề