# 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

Phương pháp thiết kế truyền thống

Phương pháp thiết kế dùng HDL

3

๏ Dựa trên sơ đồ nguyên lý

๏ Không phù hợp với các thiết kế phức tạp và độ tích hợp cao như ASIC và FPGA

๏ Ngôn ngữ mô mả phần cứng (HDL)



๏ Mô tả thiết kế độc lập với công nghệ

๏ Dễ thiết kế và debug

๏ Dễ đọc hơn sơ đồ mạch điện (với những mạch có quy mô lớn)

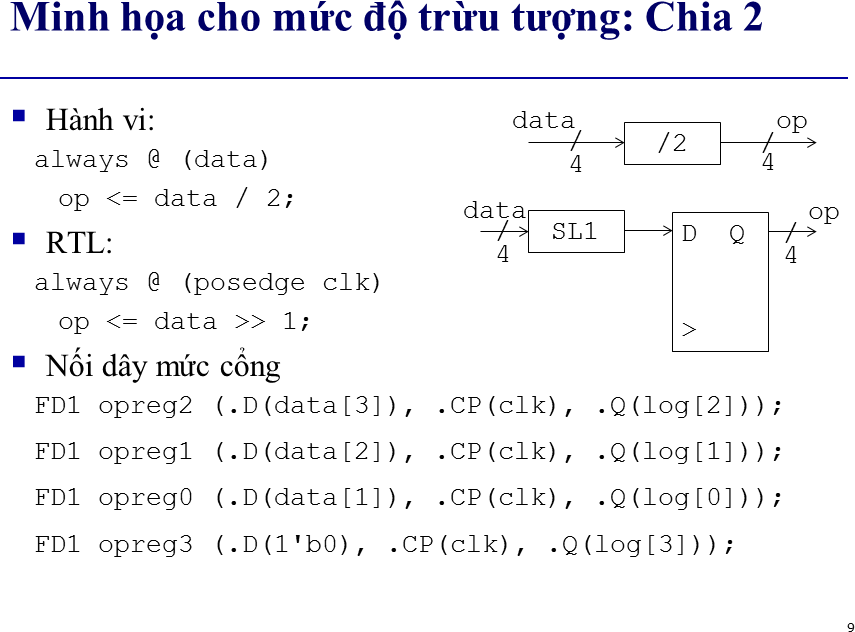
๏ Phù hợp với thiết kế ASIC và FPGA

# Verilog và VHDL

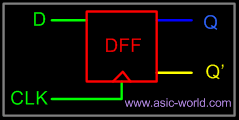
* + Verilog
    - Đơn giản, dễ sử dụng
    - Thiếu các cấu trúc cần thiết cho các tham số ở mức hệ thống
    - Mô tả đơn giản, hiệu quả, trực quan cho các mạch số
    - Cung cấp thiết kế mô hình chuyển mạch
* VHDL
  + Phức tạp hơn
  + Thích hợp với các thiết kế rất phức tạp
  + Phù hợp cho mô hình mức hành vi
  + Cung cấp nhiều hàm, thủ tục, thư viện
  + Câu lệnh phức tạp
  + Verilog 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 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

7

# Ví dụ các mức độ trừu tượng: mạch chia 2



**Ví dụ Verilog Code**

// D flip-flop Code

module d\_ff ( d, clk, q, q\_bar); input d ,clk;

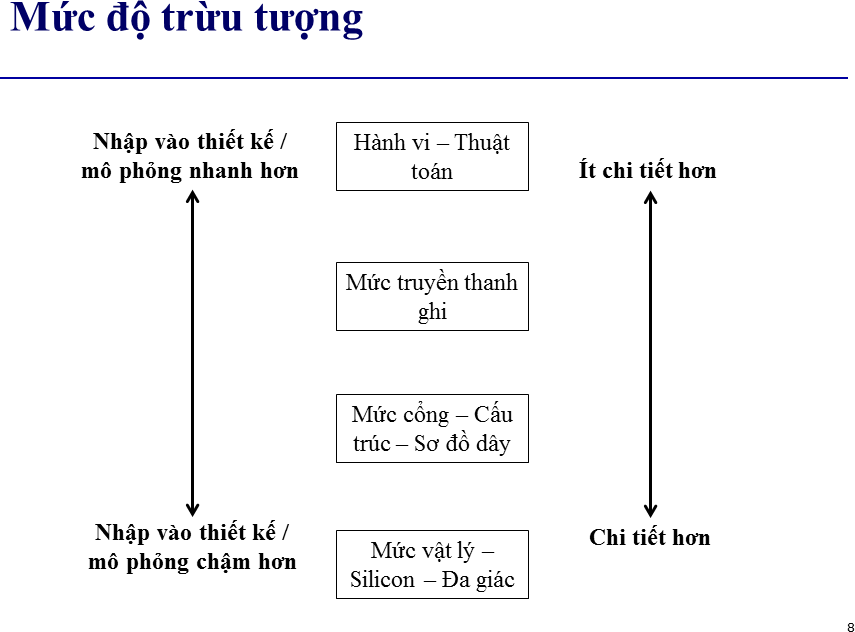
output q, q\_bar; wire d ,clk; reg q, q\_bar;

always @ (posedge clk) begin

q <= d; q\_bar <= ! d; end

endmodule

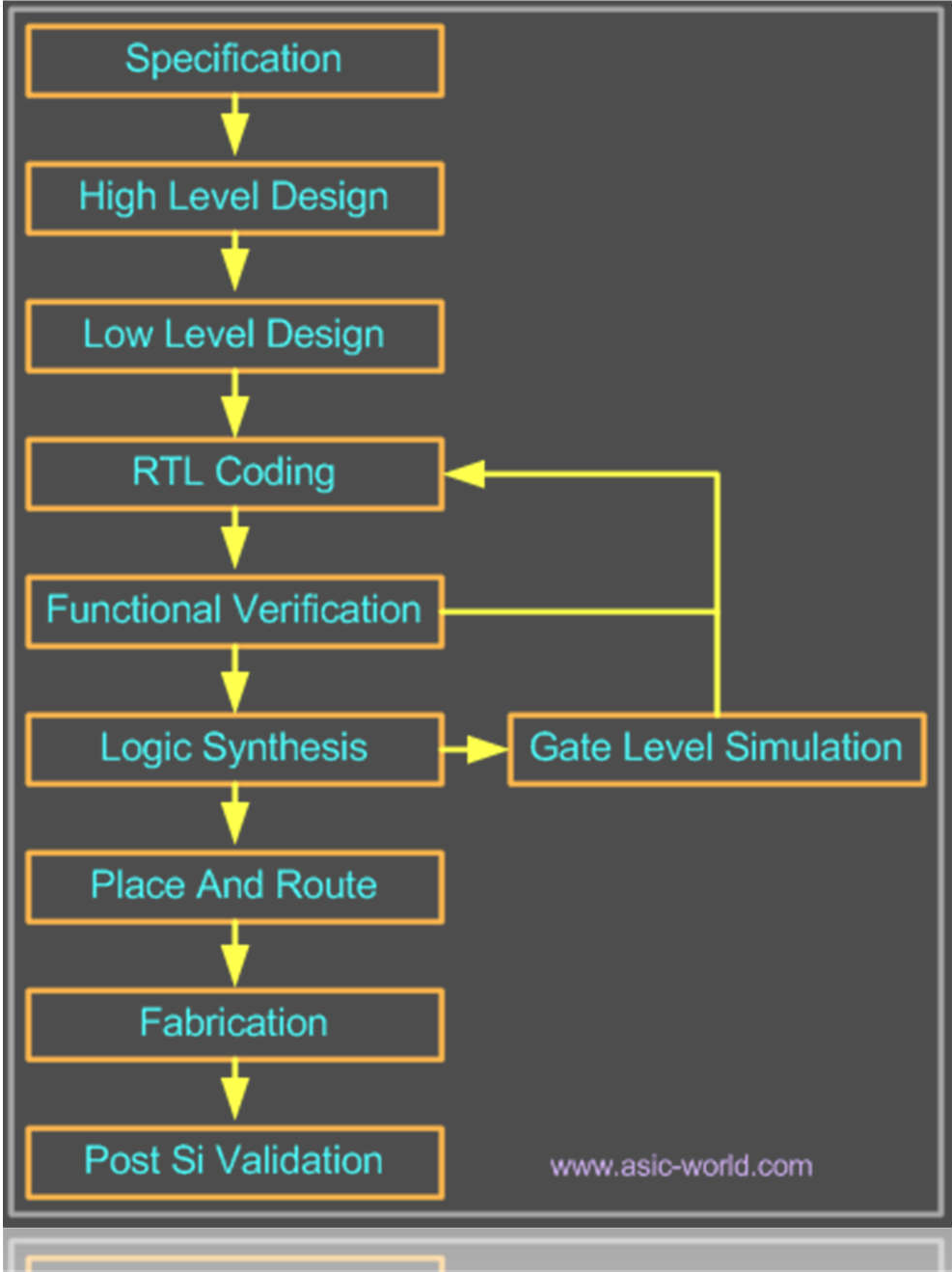
# Mức độ trừu tượng



9

# Các kiểu thiết kế

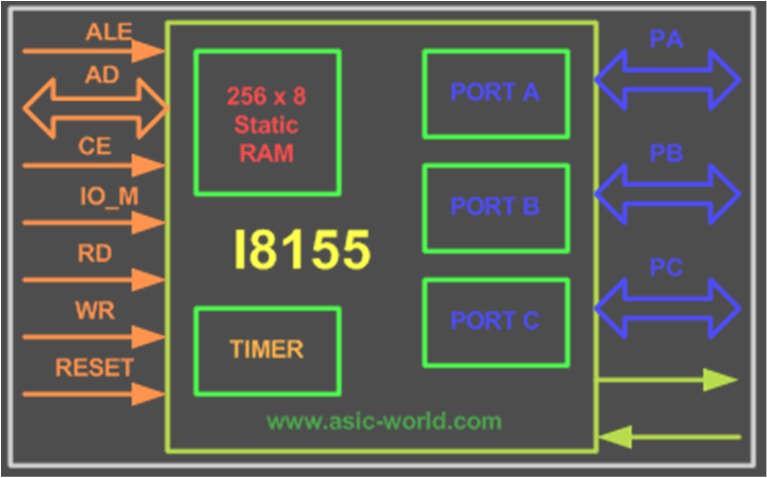
* Bottom-up Design
  + Phương pháp thiết kế truyền thống
  + Thực hiện tại mức Gate
  + Không phù hợp với các thiết kế phức tạp cho các mạch ASIC hoặc Vi xử lý
  + Cần thêm các cấu trúc mới và phương pháp thiết kế phân cấp
* Top-down Design
  + Là phương pháp mong muốn của các nhà thiết kế
  + Cho phép 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



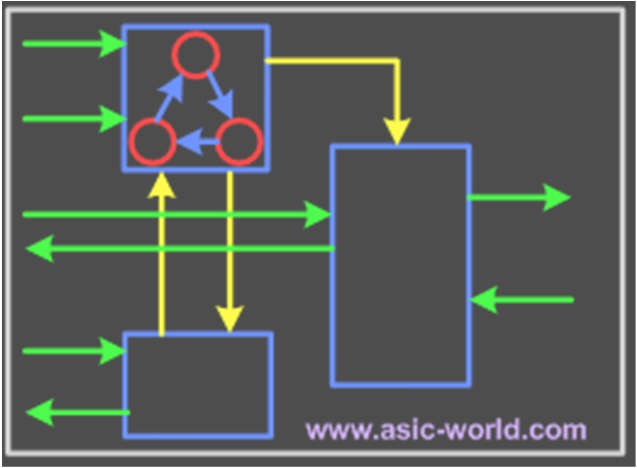
# Top-down Design

* + - 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

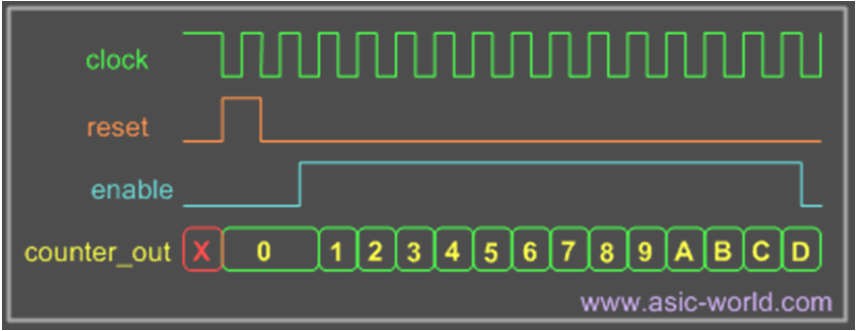
# Top-down Design - Low Level Design

* + - 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

# 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

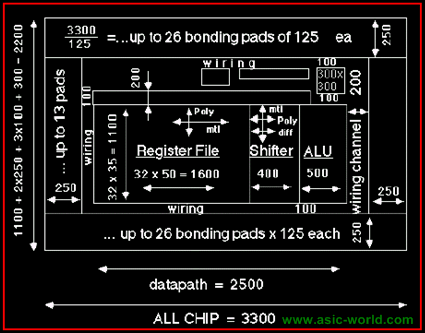
**Timing Simulation**

* 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

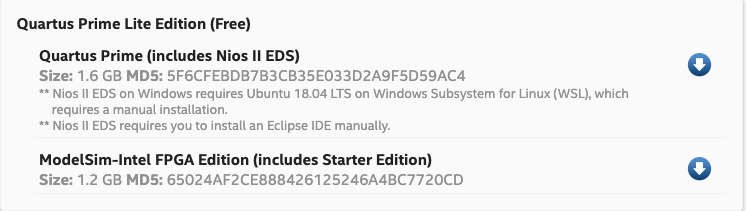
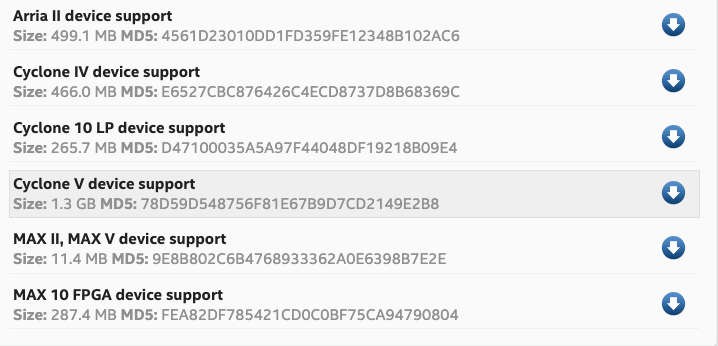
**Kiểm tra sau chế tạo - Post Silicon Validation**

* 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 Cài đặt Cyclone V device support



# 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,…

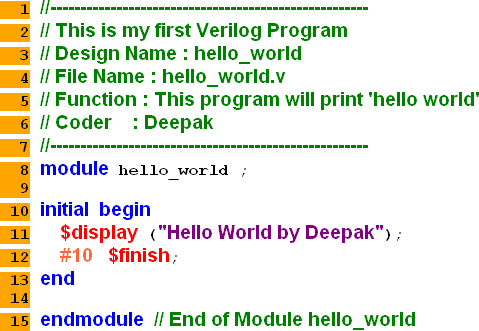
# Mô hình mức hành vi

* 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
* 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
* Khai báo các câu tiền xử lý của trình biên dịch
  + Include
  + Define
* Bắt đầu module bằng từ khoá module <tên\_module> (*dòng 8*)
* Khối 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 dài mô phỏng tại t=0
* Kết thúc bởi từ khoá endmodule

# Cấu trúc module

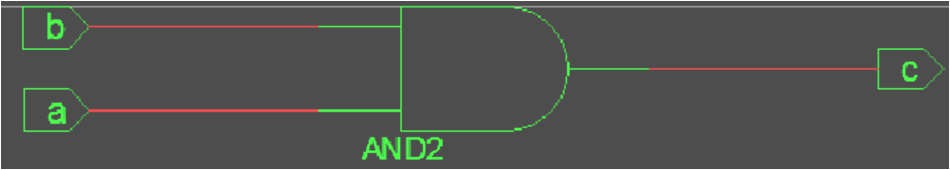
* Interface: khai báo các Ports là danh sách các tín hiệu vào, tín hiệu ra hoặc tín hiệu vào/ra để kết nối giữa các module

|  |  |
| --- | --- |
| **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 | Thân (body) |
| Các hàm và tác vụ |  |
| **endmodule** | Kết thúc |

* Body: thông số các thành phần bên trong module
* Add-on: tuỳ chọn

# 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

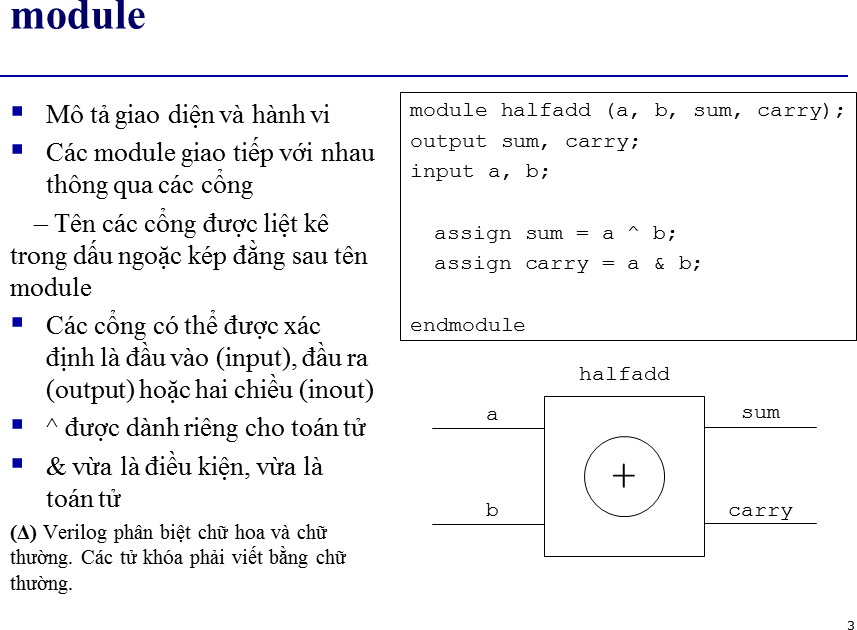


* Ngôn ngữ Verilog sẽ mô tả như sau module AND2\_1(a, b, c);

input a, b; output c;

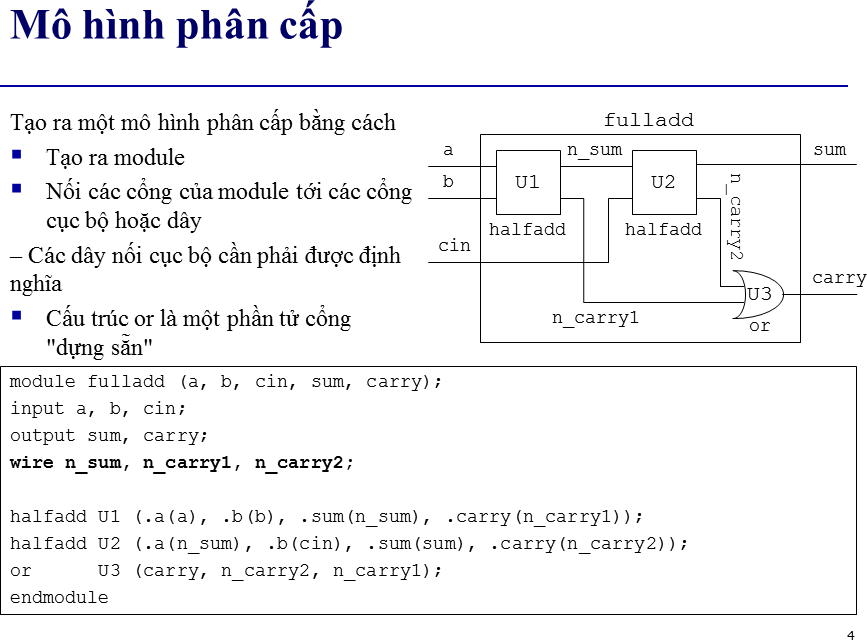
assign c = a & b; endmodule

# Module



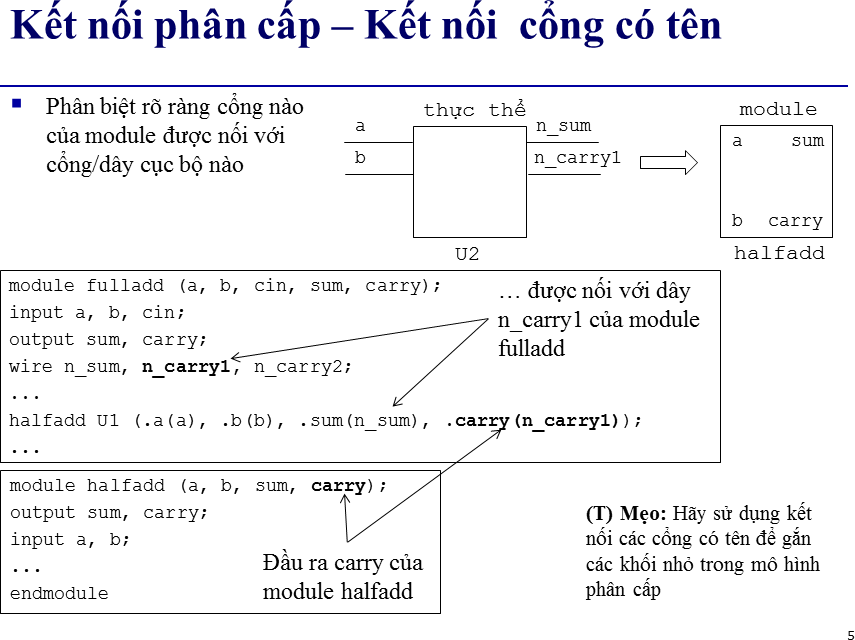
29

# Mô hìn h phân cấp



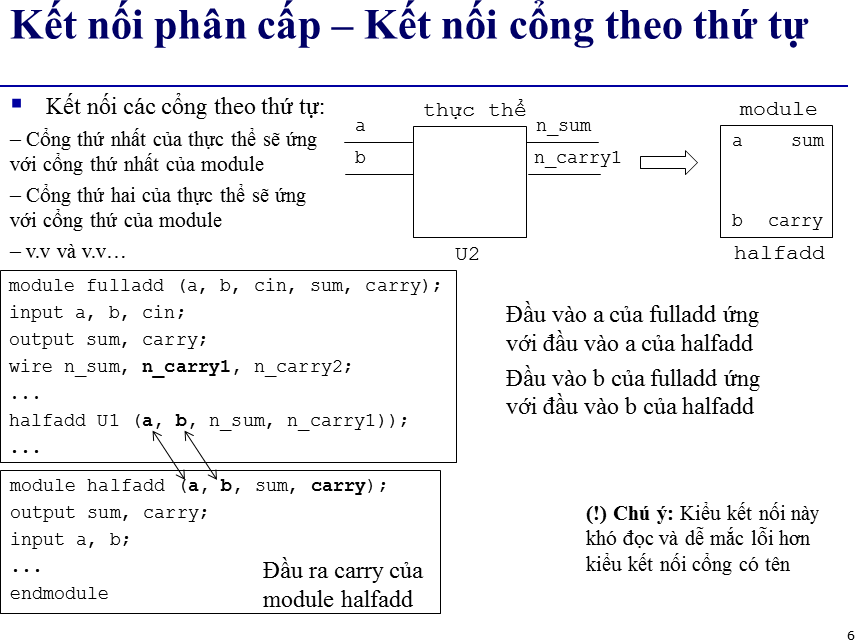
30

# Kết nối phân cấp - Kết nối cổng có tên



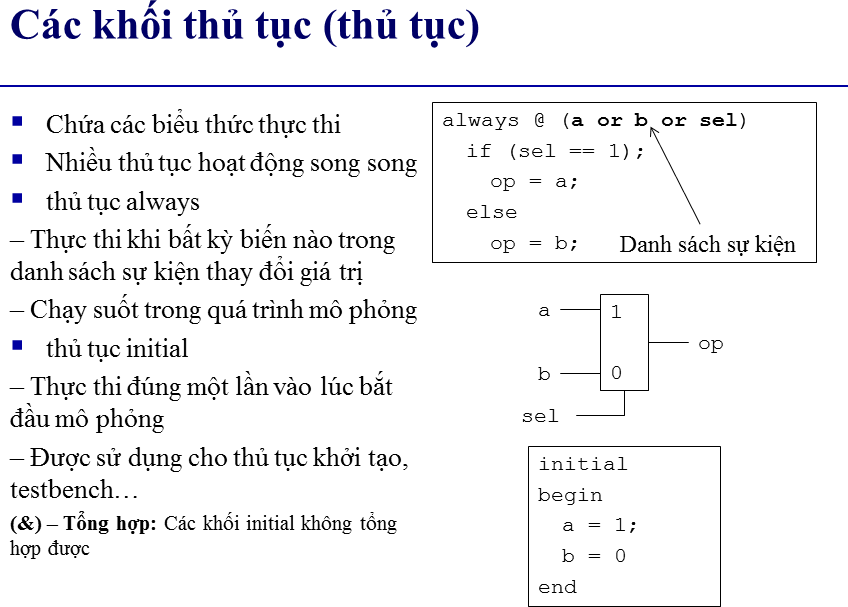
31

# Kết nối phân cấp - Kết nối cổng theo thứ tự

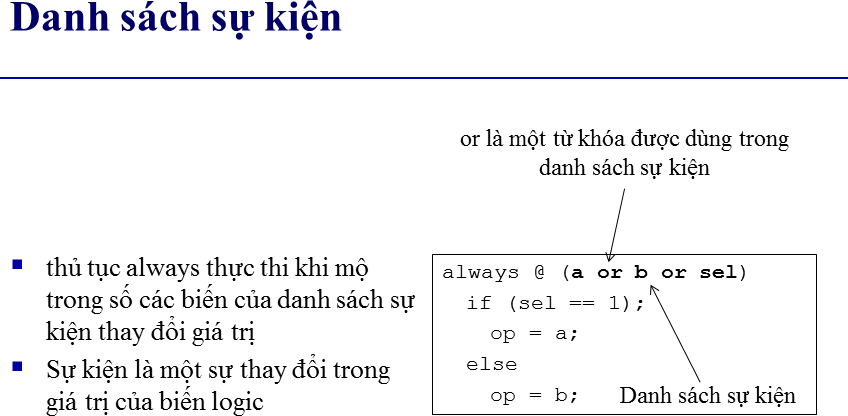


32

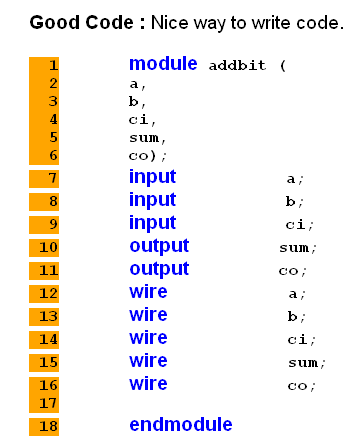
# Các khối thủ tục



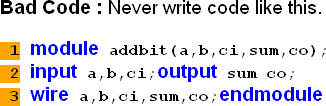
**Danh** **sách sự kiện**

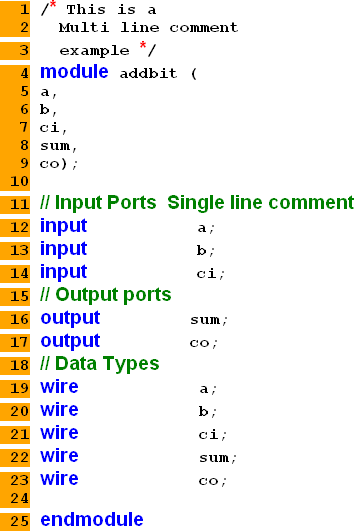


* Các quy ước cơ bản
  + Verilog HDL dùng tương tự như ngôn ngữ C
  + Có phân biệt CHỮ HOA và chữ thường
  + Tất cả các khoảng trắng là chữ thường



## 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

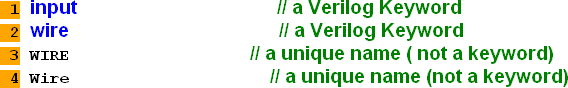


## Chú thích

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

## 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



## 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

## 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

## Chữ số

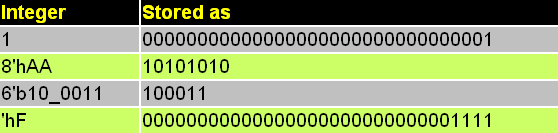
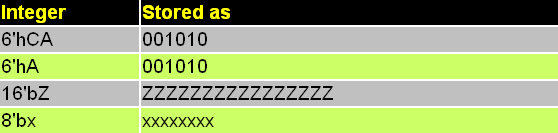
* + Dùng lưu giá trị các con số nhị phân, thập phân, hecta,…
  + Cú pháp: <size>’<base format> <number> Ví dụ: 8’hAA

Độ dài dữ liệu Kiểu dữ liệu Giá trị ban đầu

D hoặc d: hệ thập phân B hoặc b: hệ nhị phân

* + mặc định là 32 bit

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
* **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
* **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

X - Bit chưa biết Z - Trở kháng cao

## 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

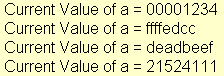
begin

a=3e10; // a gán giá trị dưới dạng khoa học a=3.14; // a gán giá trị = 3.14

end

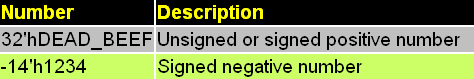
integer i; // định nghĩa biến integer i initial

i=a; //i nhận giá trị =3 (làm tròn)



## Số có dấu và không dấu

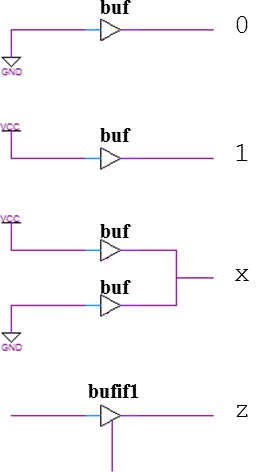
* + Bất kì số nào không có dấu (-) đứng trước đều là số dương hoặc số không dấu
  + các số có dấu (-) đứng trước size của số
  + Nội bộ verilog sẽ biểu diễn số âm dưới dạng số bù 2



# 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 điều 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

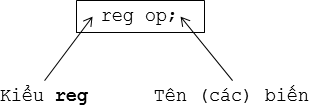
# Giá trị logic trong Verilog - 4 giá trị logic

* 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 cảm

# Khái niệm kiểu dữ liệu

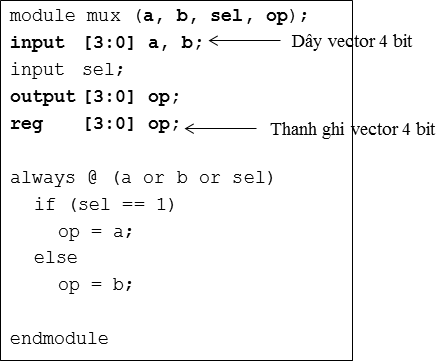
* Cần xác định kiểu dữ liệu khi định nghĩa một biến
* Các cổng của module được mặc định kiểu win
* Mặc định các kiểu biến win, vô hướng
* Việc sử dụng các kiểu dữ liệu được quy định chặt chẽ



# Vector

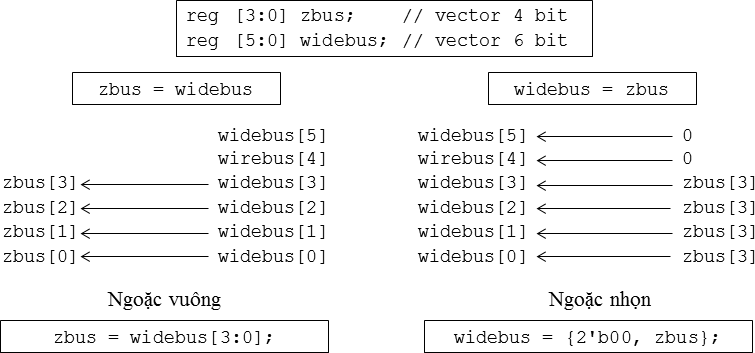
* là một biến có độ dài từ 2 bit trở lên
* Kích cỡ của biến được định nghĩa khi khai báo

# 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

# 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



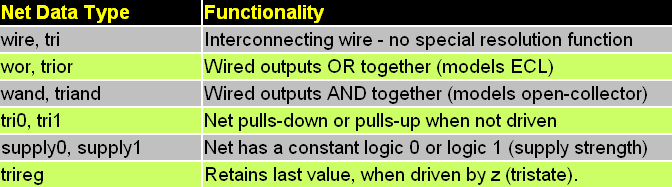
51

# Kiểu dữ liệu register và Net

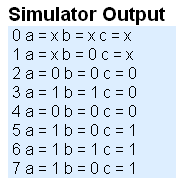
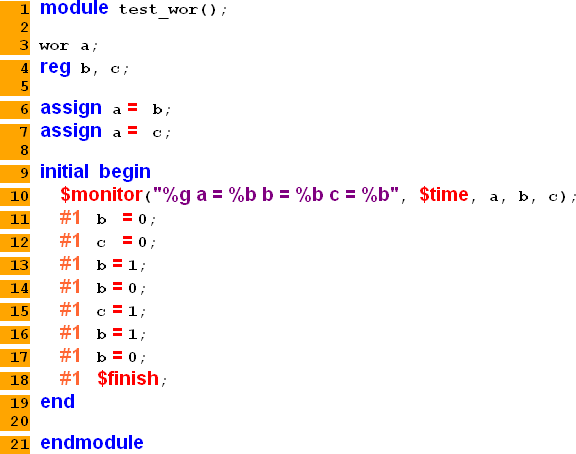
|  |  |  |
| --- | --- | --- |
|  | **Register (Thanh ghi)** | **Net (Dây)** |
| Từ khoá | **reg, integer, time, real** | **wire, wand, wor, tri, triand, trior, supply0, supply1** |
| Điều khiển | Theo sự kiện |  |
| Lưu trữ | Dữ liệu | Không lưu dữ liệu, chỉ là một kết nối |
| Sử dụng trong | khối “**always**”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



# Ví dụ - wor

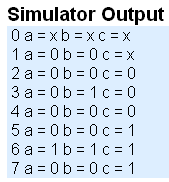
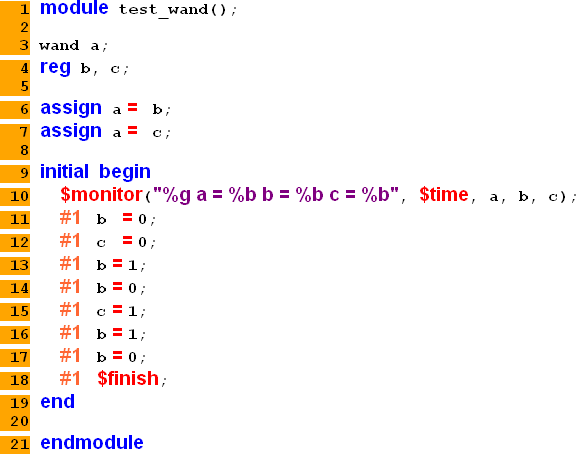


// giá trị a là mức logic của phép OR a và b

* + %g: hiển thị số thực theo số luỹ thừa hoặc hệ 10
  + %b: hệ nhị phân

54

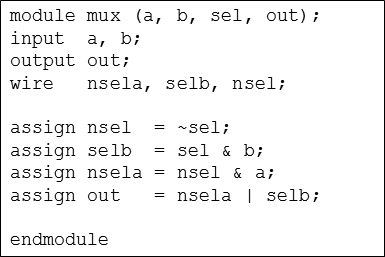
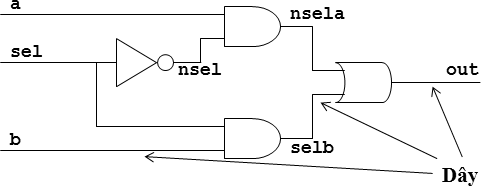
# Ví dụ - wand



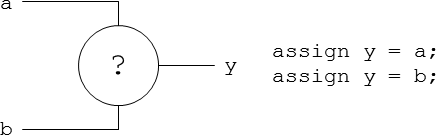
55

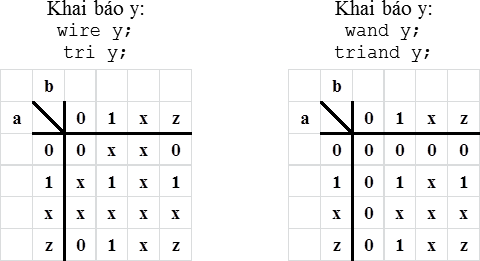
**wire và assign**

* + - 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



# 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



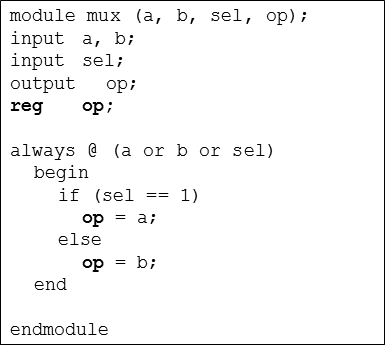
# Dữ liệu kiểu Register

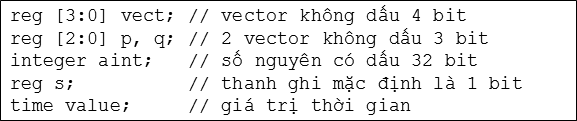
* + - Register lưu trữ giá trị cuối cùng được gán cho đến khi có câu lệnh làm thay đổi giá trị của nó
    - Register biểu diễn các cấu trúc lưu trữ dữ liệu (reg, integer, real, time)
    - Có thể tạo mảng regs → memories
    - Được dùng như các biến trong các khối thủ tục
    - Các khối thủ tục bắt đầu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

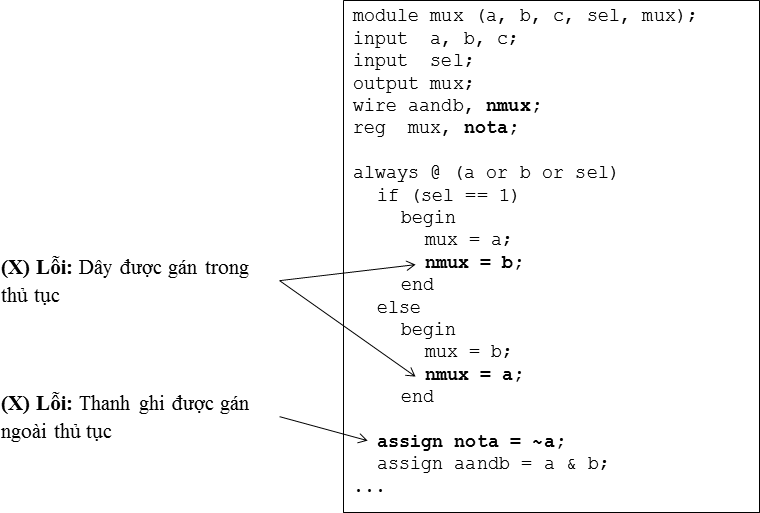
reg [MSB:LSB] reg\_name

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

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



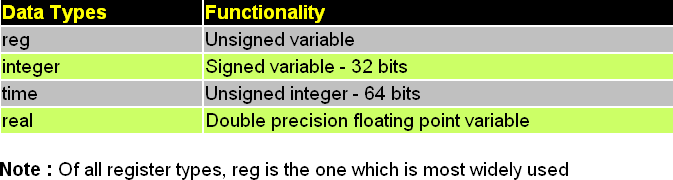


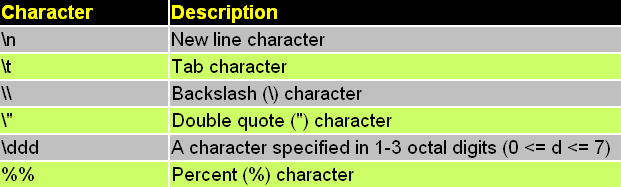
* + - Thanh ghi chỉ cập nhật giá trị trong các thủ tục
    - Các thủ tục chỉ cập nhật giá trị của thanh ghi
    - Vế phải của phép toán có thể là reg hoặc wire

# 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

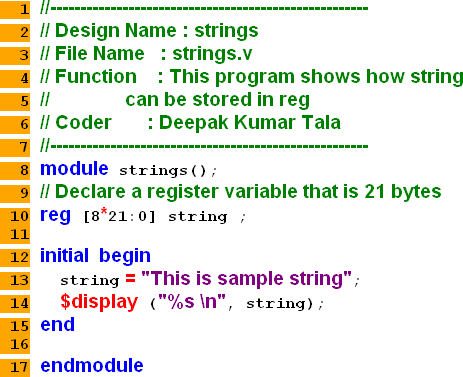
**time** c;

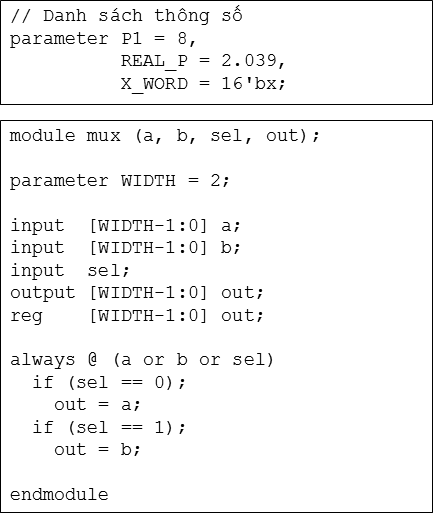
c = **$time**; // c = thời gian mô phỏng mạch điện

* + - * 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

* + - * Ví dụ

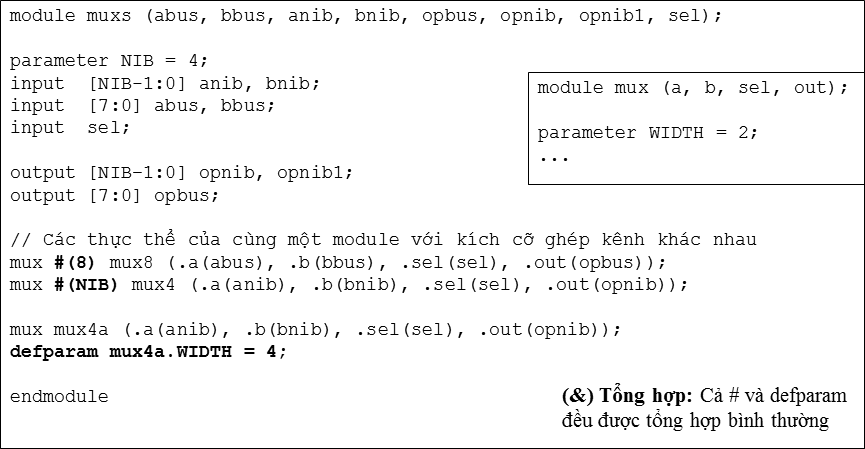


# Thông số - parameter

* + - * 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)

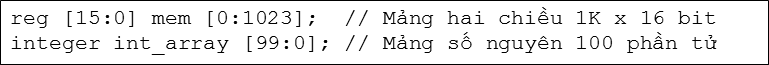
# Ghi đè giá trị của thông số

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

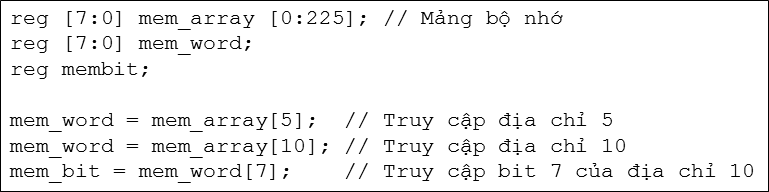


# Mảng 2 chiều

* + - * Verilog hỗ trợ mảng thanh ghi 2 chiều



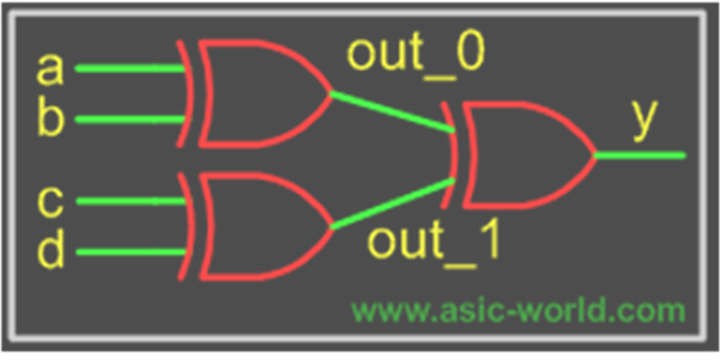
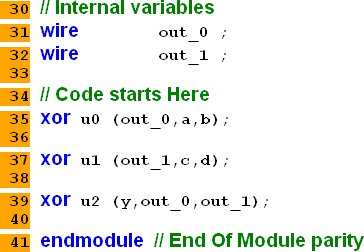
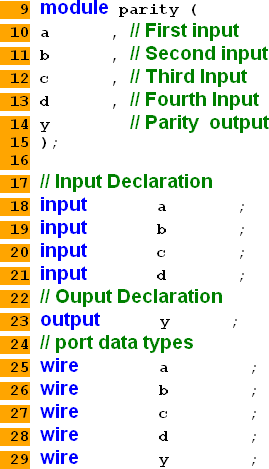
* + - * Mỗi phần tử của mảng được đánh địa chỉ bởi một chỉ số trong mảng 2 chiều
        + Chỉ được tham chiếu 1 phần tử mảng tại một thời điểm
        + Truy cập vào nhiều phần tử cần nhiều câu lệnh
        + Truy cập đến 1 bit đơn lẻ cần một biến trung gian

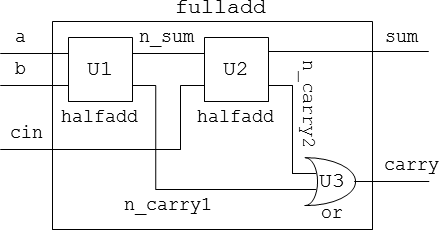


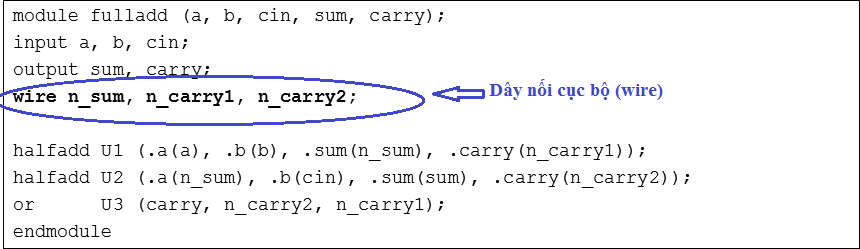
66

# 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 giai tiếp với nhau thông qua các cổng



* 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

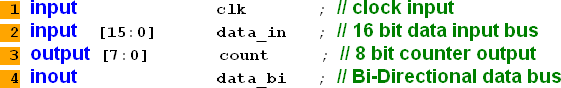


69

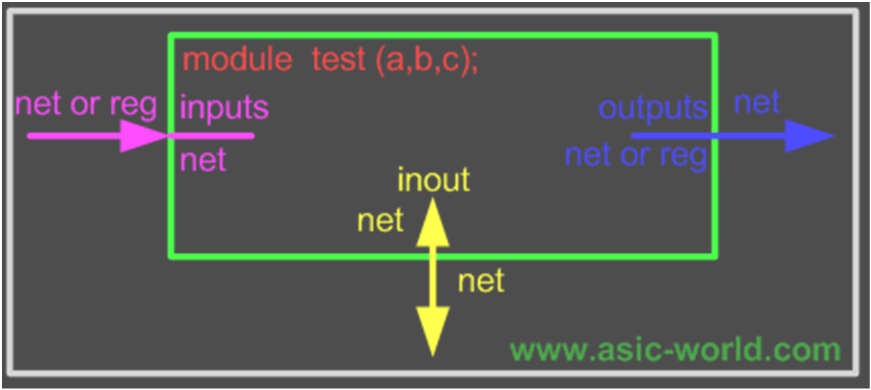
# Port

* 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;



# 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 xoá 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 );

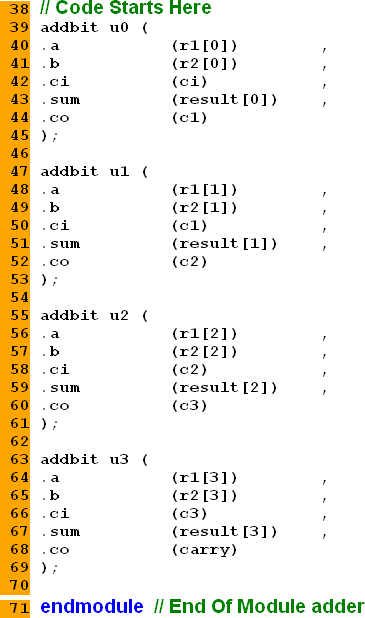
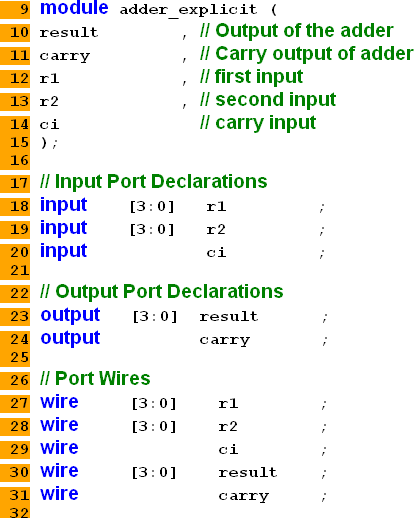
addbit u2 ( r1[2] , r2[2] , c2 , result[2] , c3 ); addbit u3 ( r1[3] , r2[3] , c3 , result[3] , carry

); endmodule

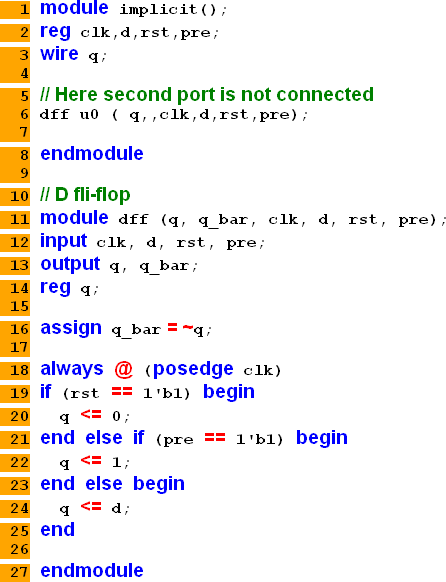
// End Of Module adder

72

# Kết nối các module theo tên Port

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

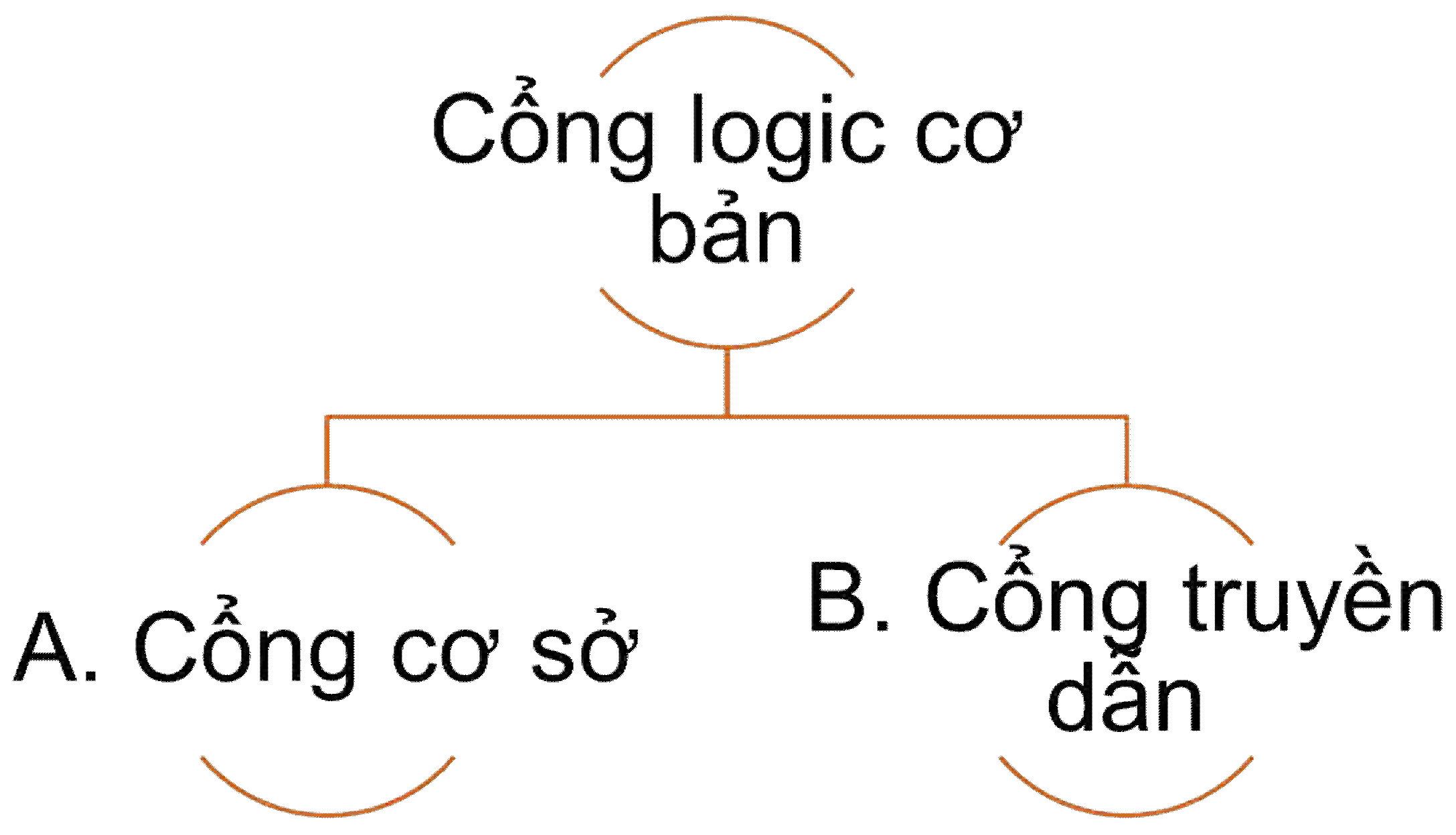
73



74

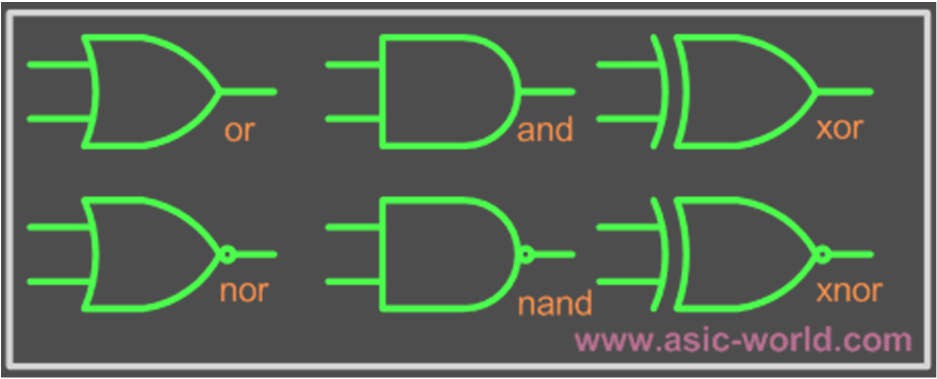
* + - Dùng để giám sát và điều khiển quá trình mô phỏng
    - Ký tự $ chỉ định hàm hệ thống
    - **$monitor**
      * $monitor($time, “%d %d %d”, address, sinout, cosout);
      * Hiển thị giá trị biến khi biến thay đổi giá trị
      * Mỗi lần thay đổi hiển thị giá trị trên 1 dòng
    - **$display**
* $display ("%d %d %d”,address,sinout,cosout);
* hiển thị giá trị hiện tại của biến theo định dạng xác định ở một dòng mới
* **$finish**: Thoát khỏi mô phỏng
* **$stop**: Dừng mô phỏng, có thể tiếp tục bởi us7e5 r

# 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
  + 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
  + Các kiểu cổng
  + Giá trị logic các cổng

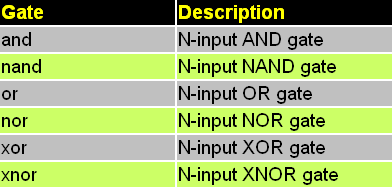
|  |  |
| --- | --- |
| **Logic Value** | **Description** |
| **0** | zero, low, false |
| **1** | one, high, true |
| **z** or **Z** | high impedance, floating |
| **x** or **X** | unknown, uninitialized, contention |

# Các cổng cơ sở

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

wire OUT, IN1, IN2;

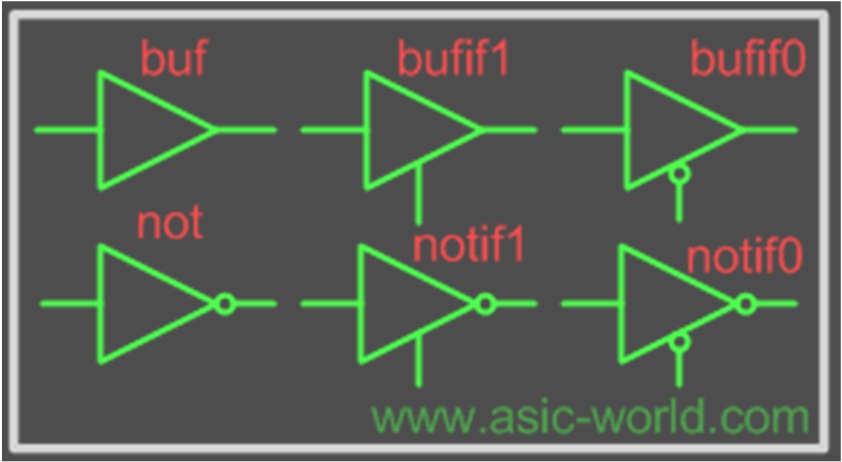
* + Sử dụng

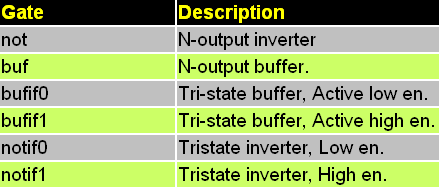
and a1 (OUT, IN1, IN2); nand na1 (OUT, IN1, IN2);

nand na1\_3in (OUT, IN1, IN2, IN3);

// cổng 3 đầu vào and (OUT, IN1, IN2);

// cổng không cần tên khởi tạo

* + 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

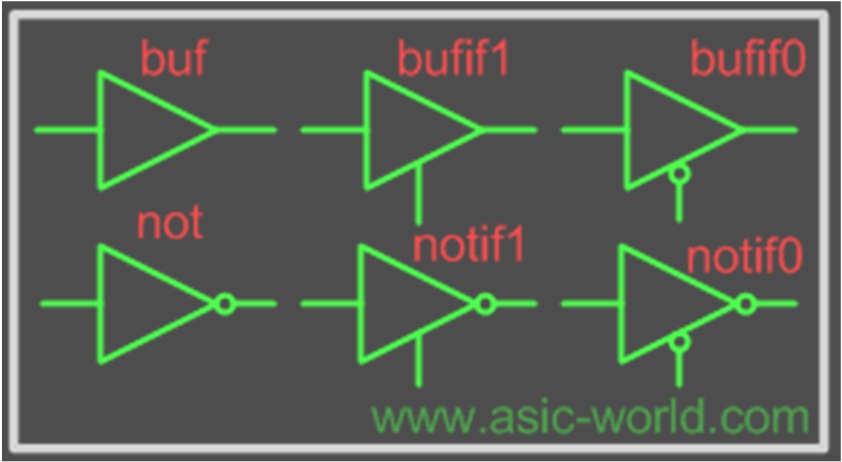
buf b1 (OUT, IN); not n1 (OUT1, IN);

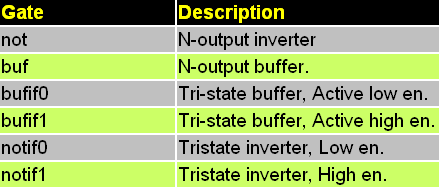
buf b1\_2output (OUT1, OUT2, IN);

* + Chuỗi cổng

wire [7:0] OUT, IN1, IN2;

nand n\_gate [7:0] (OUT, IN1, IN2);

* + 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);

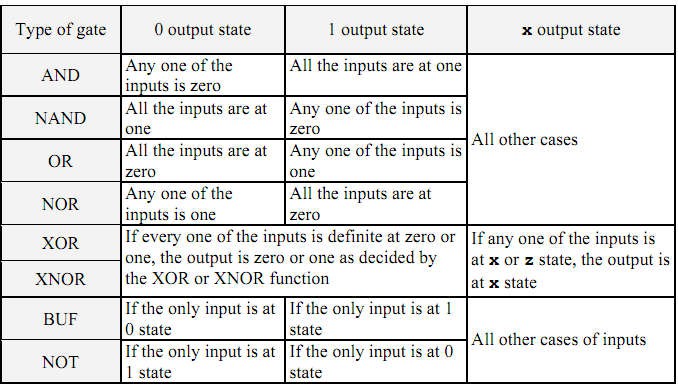
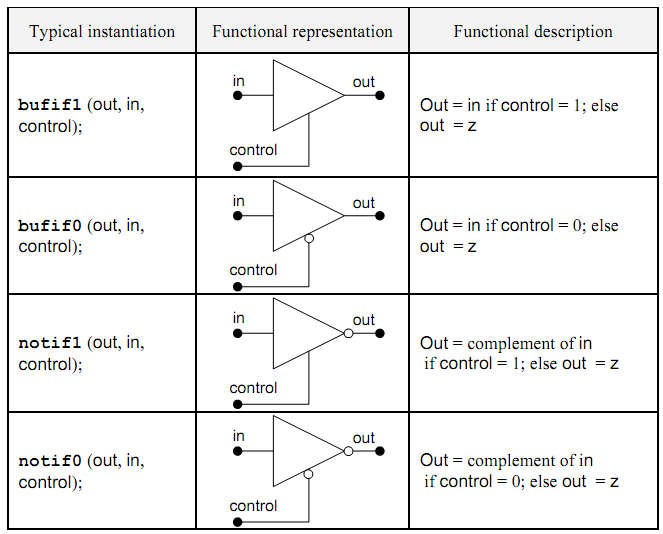
buf b1\_2output (OUT1, OUT2, IN);

* + Chuỗi cổng

wire [7:0] OUT, IN1, IN2;

nand n\_gate [7:0] (OUT, IN1, IN2);

# Thông tin thêm

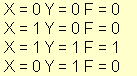
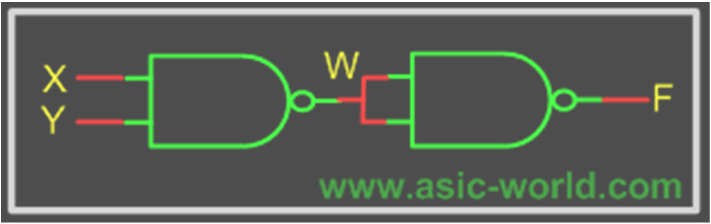


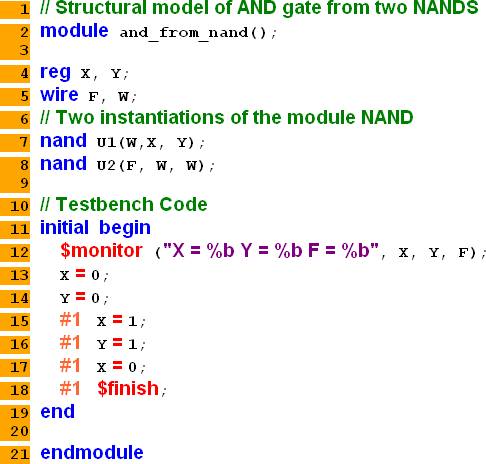
**Thiết kế dùng cổng**

* + 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

83

* Yêu cầu sinh viên
  + Bài 1: Xâu 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





84

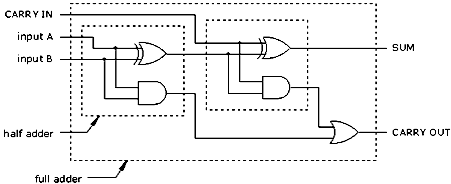
* + Giới thiệu chung
* Phương pháp thiết kế dùng Verilog HDL
* Thiết kế top-down
* Phần mềm hỗ trợ thiết kế
  + Các mô hình thiết kế Verilog
* Mô hình mức cổng (gate-level)
* Mô hình mức luồng dữ liệu
* Mô hình hành vi
  + Một số ví dụ cụ thể

# 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 Dataflow level**

module fulladder (sum,c\_out,a, b, c\_in); output sum,c\_out; input a, b, c\_in;

endmodule

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

module full\_adder

(sum, c\_out, a, b, c\_in); output sum, c\_out;

in0, in1, c\_in;

input

assign

{c\_out,sum} = a + b + c\_in;

* + 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
  + 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 ngầm định

//gán liên tục thông thường wire out;

assign out = in1&in2;

// 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;

* + Thực hiện phép toán số học: công (+), trừ (-), nhân (\*), chia (/), luỹ 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 hơ so với phép toán

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

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

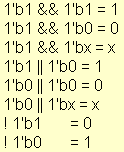
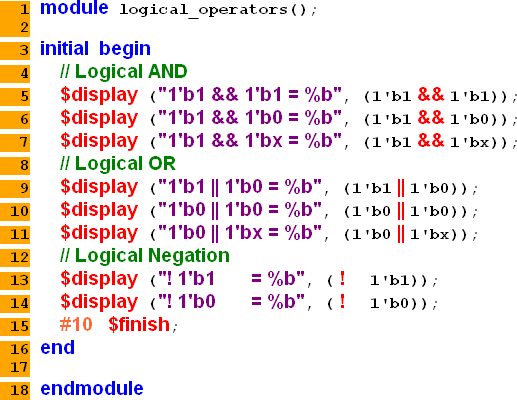
//modulus: lấy phần dư

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 13 | % | 3 | // | Evaluates | to | 1 |
| 16 | % | 4 | // | Evaluates | to | 0 |
| -7 | % | 2 | // | Evaluates | to | -1, takes sign |

of the first operand

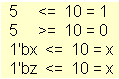
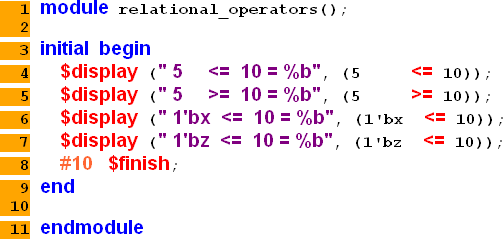
7 % -2 // Evaluates to +1, takes sign of the first operand

* 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 giạ 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



94

|  |  |
| --- | --- |
| **Operator** | **Description** |
| !A | **not A** |
| A && B | A **and** B |
| A || B | A **or** B |



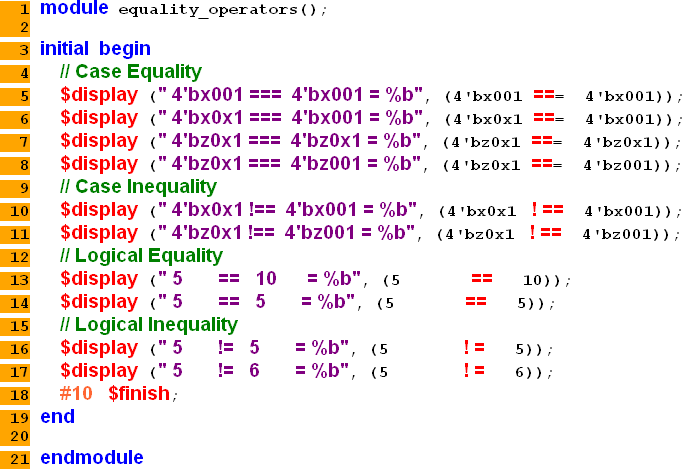
# Toán tử quan hệ

* 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

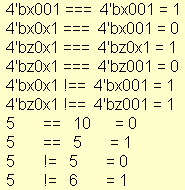
|  |  |
| --- | --- |
| **Operator** | **Description** |
| a < b | a nhỏ hơn b |
| a > b | a lớn hơn b |
| a <= b | a nhỏ hơn hoặc bằng b |
| a >= b | a lớn hơn hoặc bằng b |

* 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 cà 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 |



97

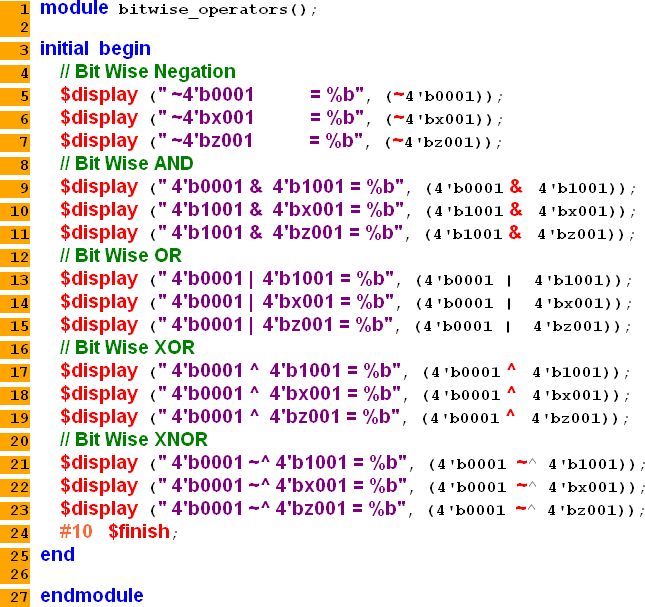


* Thực hiện phép toán logic theo từng bit của 2 toán hạng theo đúng thứ tự
* Nếu 1 toán hạng có chiều dài nhỏ hơn, nó sẽ thêm vào các bit 0 để 2 toán hạng có độ dài bằng nhau
* Sự khác biệt đối với phép toán logic: Kết quả phép toán logic là 1 bit, kết quả của phép bitwise là nhiều bit phụ thuộc độ dài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

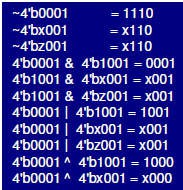
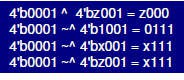
|  |  |
| --- | --- |
| **Operator** | **Description** |
| ~ | negation |
| & | and |
| | | inclusive or |
| ^ | exclusive or |
| ^~ or ~^ | exclusive nor (equivalence) |

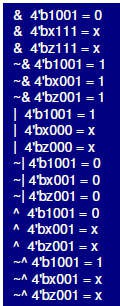
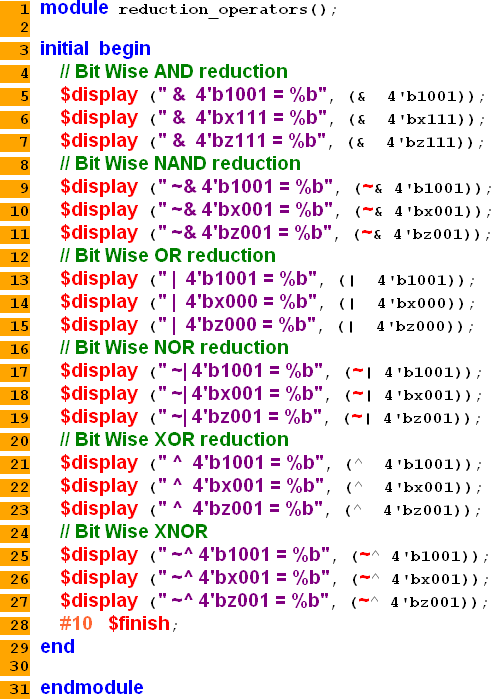
* 1&x = x&x = x
* 1|x = 1
* 0|x = x|x = x
* 0^x = 1^x = x^x = x
* 0^~x = 1^~x = x^~x = x

98



99





# 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 |

# 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;

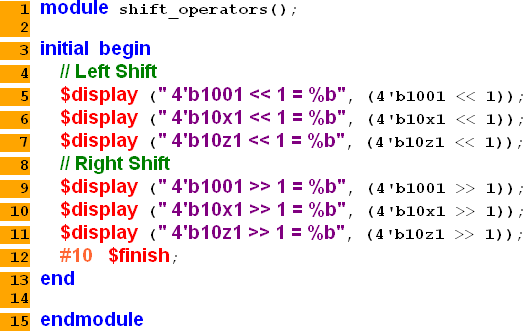
in0

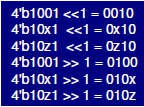
MUX 2:1

out

in1

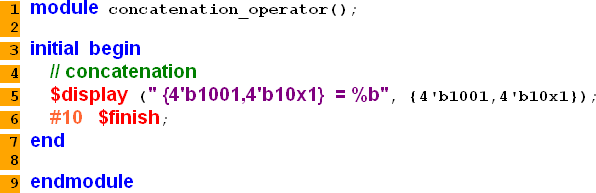
control

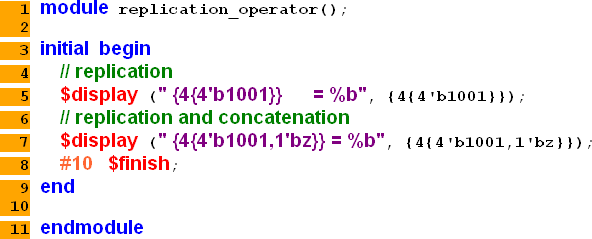


* Phép dịch
* Khi các bit được dịch, vị trí các bit bỏ trống sẽ bằng 0
* Phép dịch sẽ không quay vòng
* Toán hạng bên trái sẽ dịch số bit tương ứng với toán hạng bên phải

|  |  |
| --- | --- |
| **Ký hiệu** | **Mô tả** |
| << | Dịch trái |
| >> | Dịch phải |

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



* Lặp toán hạng
  + dùng để lặp n lần một nhóm bit

Lặp lại *n* lần giá trị *m*

{n{m}}

**Mô tả**

**Ký hiệu**

- 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}

# Thứ tự ưu tiên

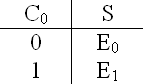
|  |  |
| --- | --- |
| **Operator** | **Symbols** |
| Unary, Multiply, Divide, Modulus | !, ~, \*, /, % |
| Add, Subtract, Shift | +, - , <<, >> |
| Relation, Equality | <,>,<=,>=,==,!=,===,!== |
| Reduction | &, !&,^,^~,|,~| |
| Logic | &&, || |
| Conditional | ? : |

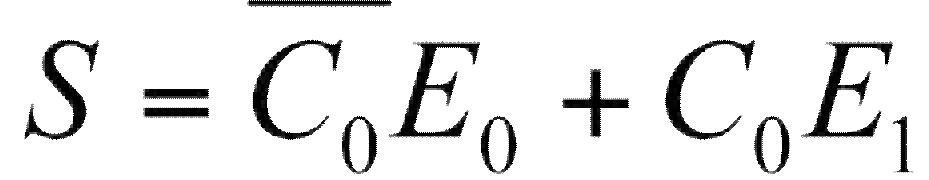
**Bài tập**

* 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

**E1**

**S**

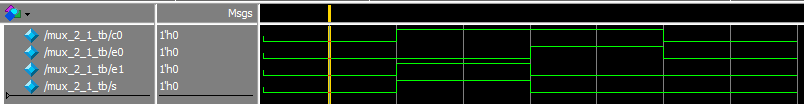
**E0**

**C0**

**Logic equation Conditional operator**

module mux\_2\_1(c0,e0,e1,s); input c0,e0,e1;

output s; assign

s=!c0 && e0||c0 && e1; endmodule

module mux\_2\_1(c0,e0,e1,s); input c0,e0,e1;

output s;

assign s = (c0)?e1 :e0; endmodule

# 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

* 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)

Sự kiện kiểu reg

chuyển từ mức 0 lên mức 1 kiểu logic từ false lên true

Sườn dương

Sự kiện kiểu net

0 lên 1; x hoặc z lên 1; 0 lên x hoặc z

Sự kiện kiểu reg

Khi sự kiện ở dạng nhiều bit reg → chỉ xét bit có trọng số nhỏ nhất

Sườn âm

Sự kiện kiểu net

1 xuống 0;

1 xuống x hoặc z;

x hoặc z xuống 0

# 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

117

* **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:**

1. clk clk=x reset=x enable=x data=x
2. 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

# Khối 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

## Gán liên tục

* Mô hình luồng dữ liệu - assign
* Gán các giá trị vào biến net → vế trái kiểu net
* Vế traí 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

## 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)

* 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 “<=“

reg x, y, z;

reg [15:0] reg\_a, reg\_b; integer count;

* Có 3 phép gán không trong khối. 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

//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)

* 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 =10
* count=count+1 được thực hiện tại thời điểm ~~25~~ 0 (không có trễ)

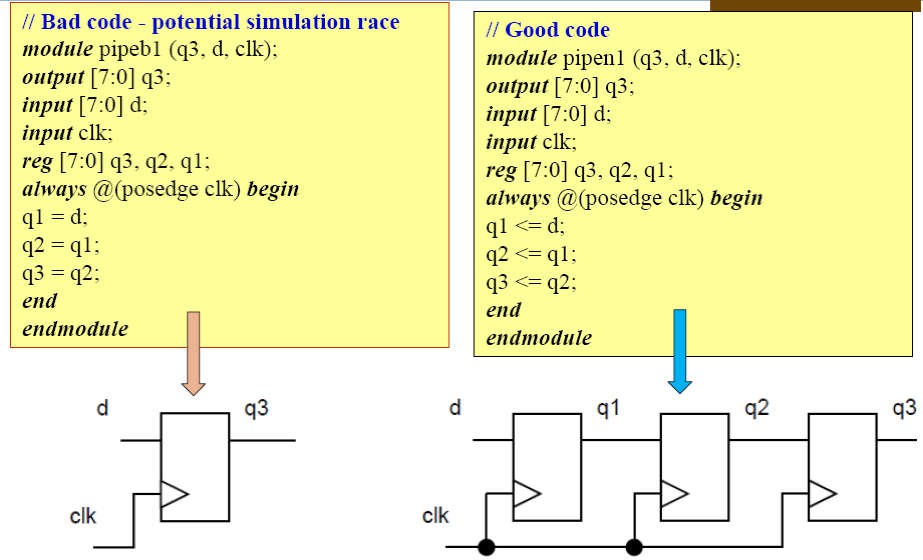
## Non-Blocking

* + Toán tử <=
  + 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

124

**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 đo hoàn tất
* Dùng trong khối always để thực hiện mạch tổ hợp



125

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

**always @ (posedge clk) A=B;**

nhận giá trị trước đó của A

 A nhận giá trị trước đó của B, B

 Xung đột

**always @ (posedge clk) B=A;**

**always @ (posedge clk) begin**

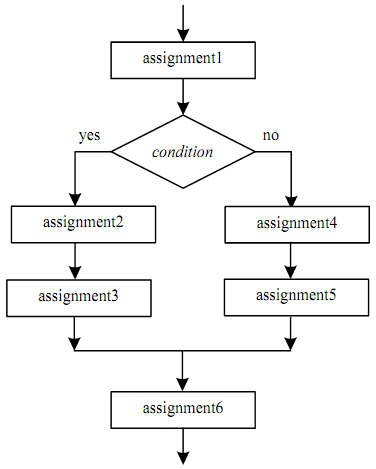
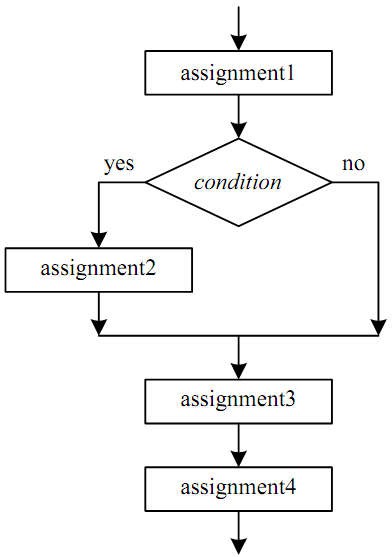
**A<=B; B<=A;**

**end**

126

# 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;

. . .

127

. . .

assignment1; if (condition)

begin assignment2; assignment3; end

else

begin assignment4; assignment5; end

assignment6;

. . .

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

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

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

**S0**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |

**E S1**

**S2 S3**

**C1**

**C0**

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

131

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

* 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;

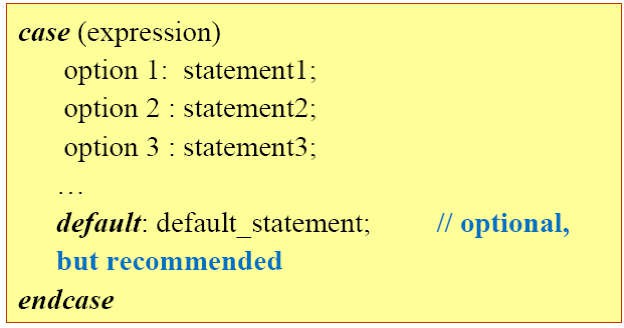
#2 c=2'b10;e=1'b1;

#2 c=2'b11;e=1'b0;

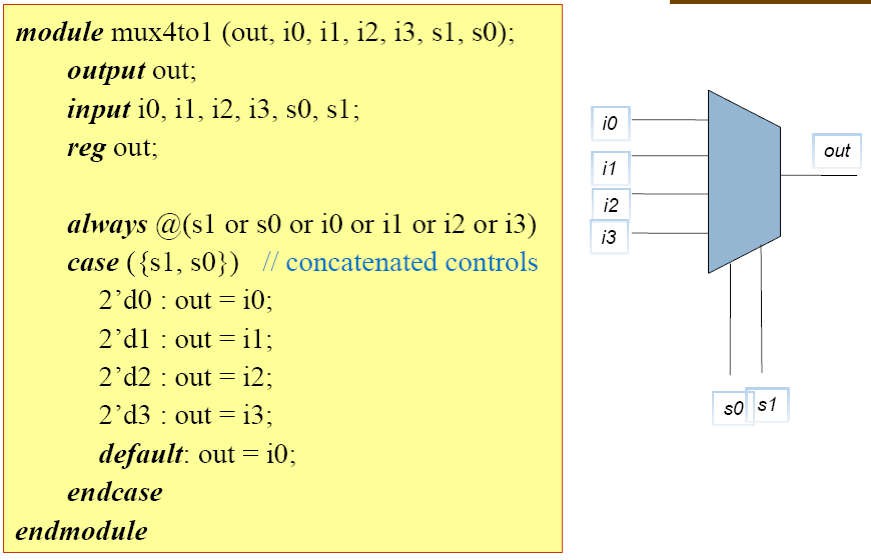
end

initial #50 $stop; endmodule

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

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.
* Ví dụ bộ MUX 4:1



# 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 trễ