

Contents

Chương 1: Tổng quan điện tử số.....	7
Biến và Hàm logic (Các phép toán cơ bản).....	7
Các phần tử logic cơ bản	8
AND IC	8
OR IC.....	8
NAND IC.....	9
NOR IC	9
XOR và XNOR IC	9
Hệ tổ hợp	9
Một số hệ tổ hợp cơ bản.....	9
Ví dụ - bộ mã hoá bàn phím	10
Hệ logic dãy.....	10
Mô hình Mealy	10
Mô hình Moore	10
Phần tử cơ bản của mạch logic dãy.....	12
Các loại trigger	13
RS trigger	13
D trigger.....	14
JK trigger	14
T trigger.....	15
Một số ứng dụng hệ dãy.....	15
- Bộ đếm module 16 không đồng bộ	15
- Bộ đếm module 10 không đồng bộ	15
- Bộ đếm module 8 đồng bộ	15
- Bộ đếm module 16 đồng bộ.....	16
• Thanh ghi	16
- Thanh ghi 4 bit vào nối tiếp - ra song song dùng D trigger.....	17
Chương 2: Giới thiệu công nghệ IC lập trình	18
• PLD: Programable Logic Device - IC số khả trình.....	18
IC số truyền thống.....	18
IC Số khả trình	18
Thuật ngữ ASIC	19
PHÂN LOẠI IC KHẢ TRÌNH	19
Full Custom ICs - ASIC đặc chế hoàn toàn	19
Standard-cell-based-ICs - ASIC dựa trên các tế bào chuẩn.....	20
Gate-array-based-ICs - ASIC dựa trên mảng công logic	20

Programmable Logic Device - PLD - các vi mạch lập trình.....	20
Simple PLD – SPLD.....	21
Complex PLD – CPLD.....	21
Field Programmable Gate Array – FPGA	21
FPGA vs Vi điều khiển	22
FPGA vs CPLD	22
KIẾN TRÚC IC KHẢ TRÌNH	22
Kiến trúc SPLD-PROM.....	22
Kiến trúc SPLD-PAL.....	23
So sánh PALs và PLAs	25
Bộ chuyển mã BCD sang mã Gray	26
Bộ so sánh số 2 bit	27
Kiến trúc CPLD	27
Board mạch CPLD - Altera Max7000.....	27
EPM7000 Series Block Diagram	28
Macrocell.....	28
Phần mở rộng chia sẻ (Shareable Logic Expanders)	29
Field-Programmable Gate Array – FPGA.....	29
• Look-Up Table (LUT).....	30
• Công nghệ lập trình chip.....	31
• Phân loại công nghệ lập trình chip	31
Lập trình dùng SRAM, EPROM, EEPROM.....	31
Lập trình dùng cầu chì nghịch (anti-fuse)	32
Cấu trúc FPGA của một số hãng.....	32
Xilinx FPGA Spartan II.....	33
Altera FPGA Cyclone	34
QUY TRÌNH THIẾT KẾ IC KHẢ TRÌNH.....	35
Các hướng tiếp cận thiết kế ASIC	35
Full-custom design	36
Standard-cell based design.....	36
Gate Array based design.....	36
FPGA based design	37
Ngôn ngữ lập trình Verilog HDL	40
Phương pháp thiết kế dùng HDL (Hardware Description Languages)	40
So sánh giữa phương pháp thiết kế truyền thống với HDL	40
So sánh Verilog và VHDL	41
Verilog HDL	41

Ví dụ các mức độ trừu tượng: Mạch chia 2	42
Ví dụ Verilog code.....	43
Mức độ trừu tượng.....	43
Các kiểu thiết kế	43
Top-down Design	44
Top-down Design - High level Design	44
Top-down Design - Low level Design	44
Top-down Design - RTL Coding.....	45
Mô phỏng	45
Synthesis.....	45
Đặt khối và định tuyến – Place and Route	46
Kiểm tra sau chế tạo – Post Silicon Validation.....	46
Các mô hình thiết kế Verilog	47
Mô hình mức hành vi	47
Mô hình mức thanh ghi RTL.....	47
Mô hình mức cổng.....	47
Cấu trúc chương trình Verilog	48
Cấu trúc module	48
Giao diện module	49
Module.....	49
Mô hình phân cấp.....	50
Kết nối phân cấp.....	50
Kết nối cổng có tên	50
Kết nối cổng theo thứ tự	51
Các khối thủ tục.....	51
Danh sách sự kiện	52
Ngôn ngữ Verilog HDL	52
Các quy ước cơ bản.....	52
Khoảng trắng	52
Chú thích	53
Chữ hoa và chữ thường.....	53
Từ định danh – tên riêng	53
Từ khóa.....	54
Chữ số	54
Số nguyên	54
Số thực	55
Số có dấu và không dấu	55

Kiểu dữ liệu	56
Giá trị logic trong Verilog – 4 giá trị logic	56
Khái niệm kiểu dữ liệu	56
Vector	57
Phép gán vector và thứ tự bit.....	57
Phép gán vector và độ dài bit	58
Kiểu dữ liệu register và Net	58
Kiểu dữ liệu Net.....	59
Ví dụ - wor	59
Ví dụ - wand.....	60
Wire và assign.....	60
Wire: cách giải quyết xung đột logic	61
Kiểu dữ liệu Register	61
Gán dữ liệu cho thanh ghi	61
Time.....	62
Chuỗi - String	62
Thông số - parameter.....	63
Ghi đè giá trị của thông số	63
Mảng 2 chiều	64
Module (Khái niệm).....	64
Khởi tạo một module	65
Mô hình phân cấp.....	65
Port.....	66
Kết nối Port	66
Kết nối module theo thứ tự Port	67
Kết nối module theo tên Port	67
Ví dụ Port không kết nối ngầm định.....	68
Hàm hệ thống – System Tasks.....	68
Mô hình thiết kế Verilog mức cổng.....	69
Các cổng cơ sở.....	69
Các cổng buf/not	70
Xây dựng cổng AND từ cổng NAND	71
Mô hình mức luồng dữ liệu	72
Mức cổng vs mức luồng dữ liệu	72
Phép gán liên tục - assign	73
Phương trình, toán tử và toán hạng.....	73
Toán tử số học	74

Toán tử logic	74
Toán tử quan hệ	75
Toán tử so sánh	76
Toán tử bitwise.....	77
Thuật toán Reduction	78
Toán tử điều kiện.....	78
Một số toán tử khác	79
Phép dịch	79
Ghép nối toán hạng	79
Lắp toán hạng	80
Thứ tự ưu tiên của các toán tử	80
Thiết kế bộ ghép kênh Mux 2:1.....	81
Mô hình hành vi	81
Các khối thủ tục.....	82
Khối initial	82
Khối always.....	83
Các kiểu sự kiện – event Control.....	83
Kiểu dữ liệu gán trong khối.....	84
Nhóm trong khối.....	85
Khối begin - end.....	85
Khối fork - join	86
Phép gán.....	86
Phép gán liên tục và gán thủ tục	86
Phép gán liên tục – gán trong khối.....	87
Phép gán liên tục – gán không trong khối	88
Non – blocking vs Blocking	88
Câu điều kiện if – then – else	90
Ví dụ if	90
Ví dụ bộ DEMUX 1:4	91
Câu điều kiện case	93
Casez và casex.....	94
Vòng lặp – loop.....	94
Bài tập	95
Bài1:Xây dựng cổng AND từ cổng NAND	95
Bài2:Xây dựng cổng D-FF từ cổng NAND	95
Bài3: Bộ cổng đày đủ 1 bit và 4 bit	96
Bài 4: Bộ ghép kênh 4:1	97

Bài 5: bộ so sánh 8 bit.....	98
Bài 6: Bộ mã hoá BCD	99
.....	104
.....	104
Bài 2: Xây dựng D-FF từ cổng NAND bằng cách viết chương trình với ngôn ngữ verilog.....	107
Bài 6: Xây dựng bộ chuyển mã BCD.....	108

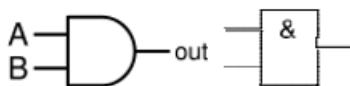
Chương 1: Tổng quan điện tử số

Biến và Hàm logic (Các phép toán cơ bản)

- Biến logic: đặc trưng cho trạng thái logic của các đối tượng. Được biểu diễn bởi các ký tự, trạng thái logic có giá trị là 0 hoặc 1
- Hàm logic: là biểu diễn của nhóm các biến logic, liên hệ với nhau thông qua các phép toán logic
- 3 phép toán cơ bản và (AND) , hoặc (OR) , phủ định - đảo (NOT)

Phép AND và cổng AND (AND gate)

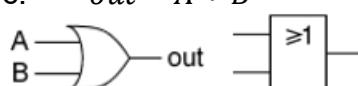
- Chức năng: thực hiện phép toán AND logic
- Cổng AND 2 đầu vào
- Biểu thức $out = A \cdot B$



A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

Phép OR và cổng OR (OR gate)

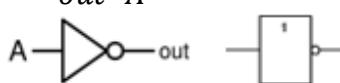
- Chức năng: thực hiện phép toán OR logic
- Cổng OR 2 đầu vào
- Biểu thức: $out = A + B$



A	B	out
0	0	0
0	1	1
1	0	1
1	1	1

Phép NOT và cổng NOT (NOT gate)

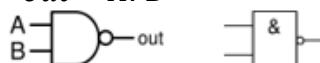
- Chức năng: thực hiện phép toán NOT logic
- Cổng NOT chỉ có 1 đầu vào
- Biểu thức $out = \bar{A}$



A	out
0	1
1	0

Phép VÀ ĐẢO (NAND gate)

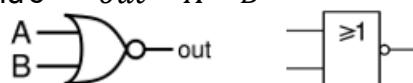
- Chức năng: thực hiện phép toán NOT logic của phép toán AND
- Cổng NAND có 2 đầu vào
- Biểu thức $out = \bar{A} \cdot \bar{B}$



A	B	out
0	0	1
0	1	1
1	0	1
1	1	0

Phép HOẶC ĐẢO (NOR gate)

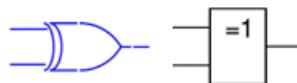
- Chức năng: thực hiện phép toán NOT logic của phép toán OR
- Cổng NOR có 2 đầu vào
- Biểu thức $out = \bar{A} + \bar{B}$



A	B	out
0	0	1
0	1	0
1	0	0
1	1	0

Phép XOR (XOR gate)

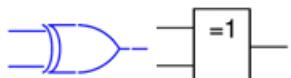
- Chức năng: thực hiện phép toán HOẶC CÓ LOẠI TRÙ, đầu ra bằng 0 khi có đầu vào giống nhau
- Cổng XOR có 2 đầu vào
- Biểu thức: $out = A \cdot B + A \cdot B = A \oplus B$



A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

Phép XNOR (XNOR gate)

- Chức năng: thực hiện phép toán NOT của phép XOR, đầu ra bằng 1 khi có đầu vào giống nhau
- Cổng XOR có 2 đầu vào
- Biểu thức: $out = A \oplus B = A \cdot B + A \cdot B$

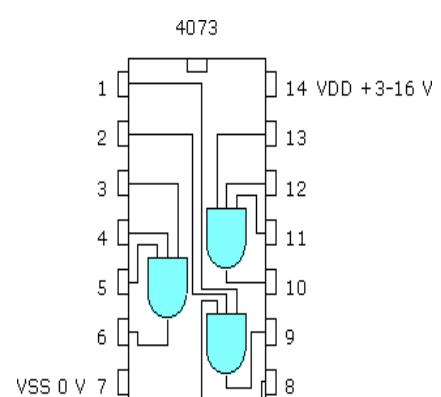
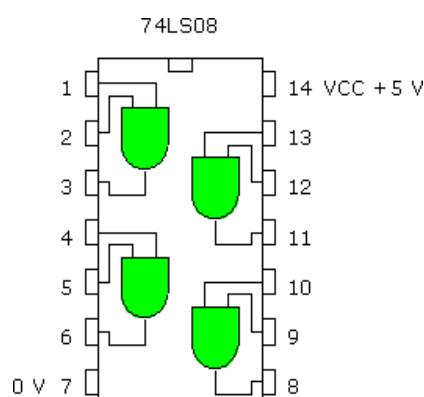
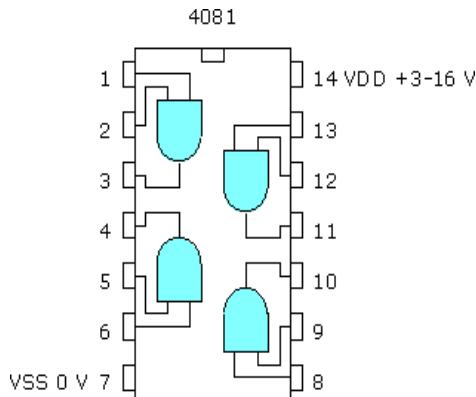


A	B	out
0	0	1
0	1	0
1	0	0
1	1	1

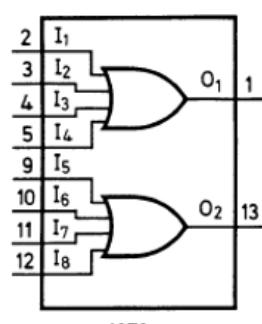
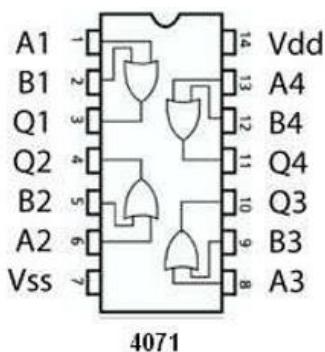
Các phần tử logic cơ bản

- | | | |
|------------------------------|-------------------------------|------------------------------------|
| •AND: 74LS08, 4081, 4073,... | •NOT: 74LS04/05 | •NOR: 74LS02, 4000, 4001, 4002,... |
| •OR: 74LS32, 4071, 4072,... | •NAND: 74LS00, 4011, 4012,... | •XOR: 74LS136, 4070/4030,... |
| | | •XNOR: 74LS266, 4077,... |

AND IC

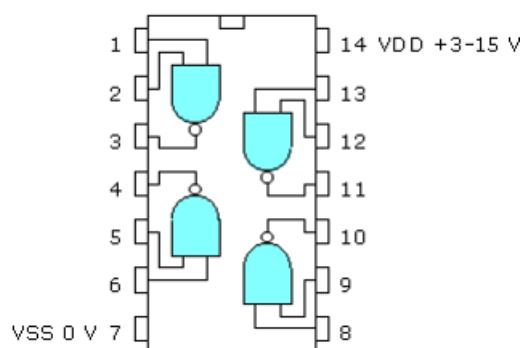


OR IC

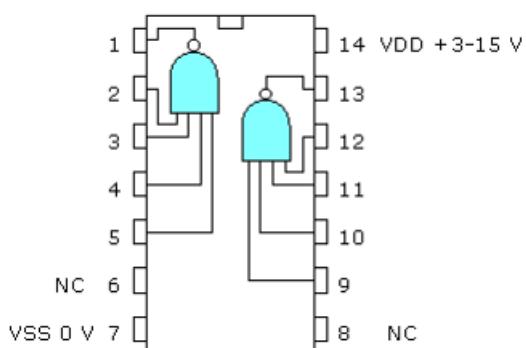


NAND IC

4011

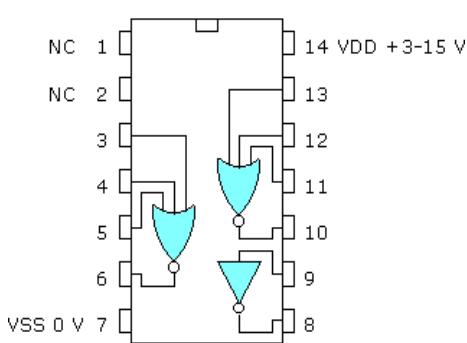


4012

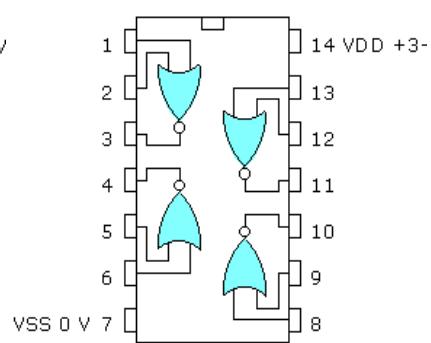


NOR IC

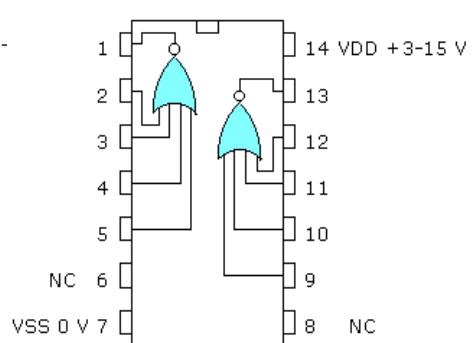
4000



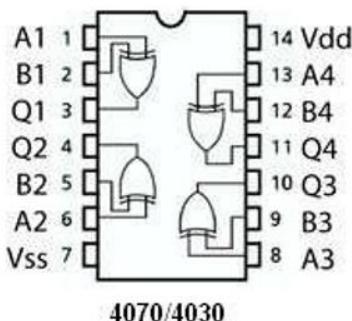
4001



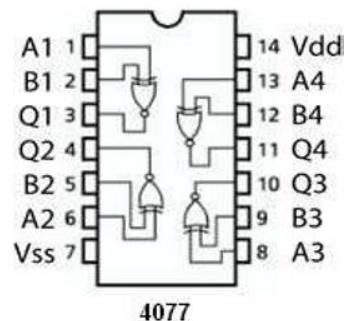
4002



XOR và XNOR IC



4070/4030



4077

Hệ tổ hợp

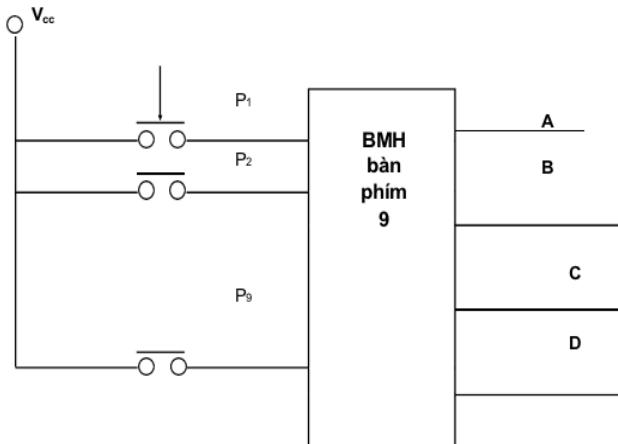
- là hệ mà tín hiệu ra chỉ phụ thuộc vào tín hiệu vào tại thời điểm hiện tại
- còn được gọi là hệ không có nhớ
- có thể được tổng hợp từ các phần tử logic cơ bản

Một số hệ tổ hợp cơ bản

bộ mã hoá | bộ giải mã | bộ chọn kênh | bộ phân kênh | các mạch số học

Ví dụ - bộ mã hóa bàn phím

- Mã hóa bàn phím
- Mỗi phím được gán với 1 mã riêng biệt
- Khi phím được tác động, bộ mã hóa sẽ đưa ra ở đầu ra mã tương ứng
- Bàn phím gồm 9 nút ấn, thiết kế bộ mã hóa với giả sử tại một thời điểm chỉ có một phím được ấn
-
- Sơ đồ
- Có 9 phím nên cần phải sử dụng mã 4 bit để mã hóa
- Bộ giải mã sẽ có 9 đầu vào và 4 bit đầu ra



Hệ logic dây

- Là mạch có tính chất nhớ
- Trạng thái đầu ra của mạch phụ thuộc trạng thái của các biến đầu vào + trạng thái hiện tại của mạch
- 2 loại mô hình : Mealy , Moore



Mô hình Mealy

- Mô hình mô tả hệ dây thông qua 5 tham số
- $X = \{x_1, x_2, x_3, \dots, x_n\}$ - tập n tín hiệu đầu vào
- $Y = \{y_1, y_2, y_3, \dots, y_l\}$ - tập l tín hiệu đầu ra
- $S = \{S_1, S_2, S_3, \dots, S_m\}$ - tập m trạng thái trong hệ
- $FS(S, X)$ - hàm biến đổi trạng thái
- $FY(S, X)$ - hàm tính trạng thái đầu ra - trong mô hình Mealy, FY phụ thuộc S và X

Mô hình Moore

- Mô hình mô tả hệ dây thông qua 5 tham số
- $X = \{x_1, x_2, x_3, \dots, x_n\}$ - tập n tín hiệu đầu vào
- $Y = \{y_1, y_2, y_3, \dots, y_l\}$ - tập l tín hiệu đầu ra
- $S = \{S_1, S_2, S_3, \dots, S_m\}$ - tập m trạng thái trong hệ
- $FS(S, X)$ - hàm biến đổi trạng thái
- $FY(S)$ - hàm tính trạng thái đầu ra - trong mô hình Moore, FY chỉ phụ thuộc S

Bảng chuyển trạng thái

- Mô hình Mealy

S	X		
	X ₁	...	X _n
S ₁	F _S (S ₁ , X ₁), F _Y (S ₁ , X ₁)	...	F _S (S ₁ , X _n), F _Y (S ₁ , X _n)
...
S _m	F _S (S _m , X ₁), F _Y (S _m , X ₁)	...	F _S (S _m , X _n), F _Y (S _m , X _n)

- Mô hình Moore

S	X			Y
	X ₁	...	X _n	
S ₁	F _S (S ₁ , X ₁)	...	F _S (S ₁ , X _n)	F _Y (S ₁)
...
S _m	F _S (S _m , X ₁)	...	F _S (S _m , X _n)	F _Y (S _m)

Ví dụ:

Sử dụng mô hình Mealy và mô hình Moore để mô tả hệ dãy thực hiện phép tính cộng

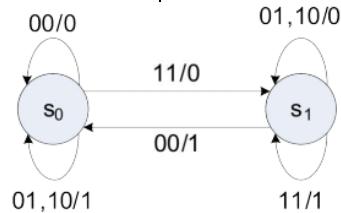
	t ₄	t ₃	t ₂	t ₁	t ₀
A	0	1	1	0	0
B	0	1	1	1	0
S	1	1	0	1	0

- Mô hình Mealy

- X = {00, 01, 10, 11} - do có 2 đầu vào A và B
- Y = {0,1} - có 1 đầu ra S (sum)
- S = {s0, s1} - có 2 trạng thái s0 là không nhớ, s1 là có nhớ

S	X				- F _Y (S,X)	F _S (s0,00) = 0 F _S (s0,10) = 1 F _S (s1,00) = 1 F _S (s1,10) = 0
	00	01	10	11		
s ₀	s ₀ , 0	s ₀ , 1	s ₀ , 1	s ₁ , 0		
s ₁	s ₀ , 1	s ₁ , 0	s ₁ , 0	s ₁ , 1		

S	X			
	00	01	10	11
s ₀	s ₀ , 0	s ₀ , 1	s ₀ , 1	s ₁ , 0
s ₁	s ₀ , 1	s ₁ , 0	s ₁ , 0	s ₁ , 1



Bảng chuyển trạng thái

Đồ hình chuyển trạng thái

- Mô hình Moore

- $X = \{00, 01, 10, 11\}$ - do có 2 đầu vào A và B
- $Y = \{0, 1\}$ - có 1 đầu ra S (sum)
- $S = \{s_{00}, s_{01}, s_{10}, s_{11}\}$ - có 4 trạng thái

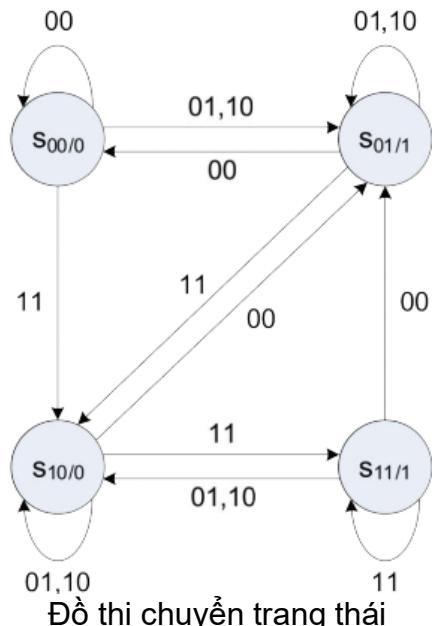
s_{ij} i - là có nhớ (1) hoặc không nhớ (0)
j - là tín hiệu ra

- $F_S(S, X)$

$$\begin{array}{ll} F_S(s_{00}, 00) = s_{00} & F_S(s_{00}, 01) = s_{01} \\ F_S(s_{00}, 10) = s_{01} & F_S(s_{00}, 11) = s_{10} \\ F_S(s_{01}, 00) = s_{00} & F_S(s_{01}, 01) = s_{01} \\ F_S(s_{01}, 10) = s_{01} & F_S(s_{01}, 11) = s_{10} \\ F_S(s_{10}, 00) = s_{01} & F_S(s_{10}, 01) = s_{10} \\ F_S(s_{10}, 10) = s_{10} & F_S(s_{10}, 11) = s_{11} \\ F_S(s_{11}, 00) = s_{01} & F_S(s_{11}, 01) = s_{10} \\ F_S(s_{11}, 10) = s_{10} & F_S(s_{11}, 11) = s_{11} \end{array}$$

- $F_Y(S)$

$$\begin{array}{ll} F_Y(s_{00}) = 0 & F_Y(s_{10}) = 1 \\ F_Y(s_{01}) = 0 & F_Y(s_{11}) = 1 \end{array}$$



Bảng chuyển trạng thái:

S	X				Y
	00	01	10	11	
s ₀₀	s ₀₀	s ₀₁	s ₀₁	s ₁₀	0
s ₀₁	s ₀₀	s ₀₁	s ₀₁	s ₁₀	1
s ₁₀	s ₀₁	s ₁₀	s ₁₀	s ₁₁	0
s ₁₁	s ₀₁	s ₁₀	s ₁₀	s ₁₁	1

Phần tử cơ bản của mạch logic dãy

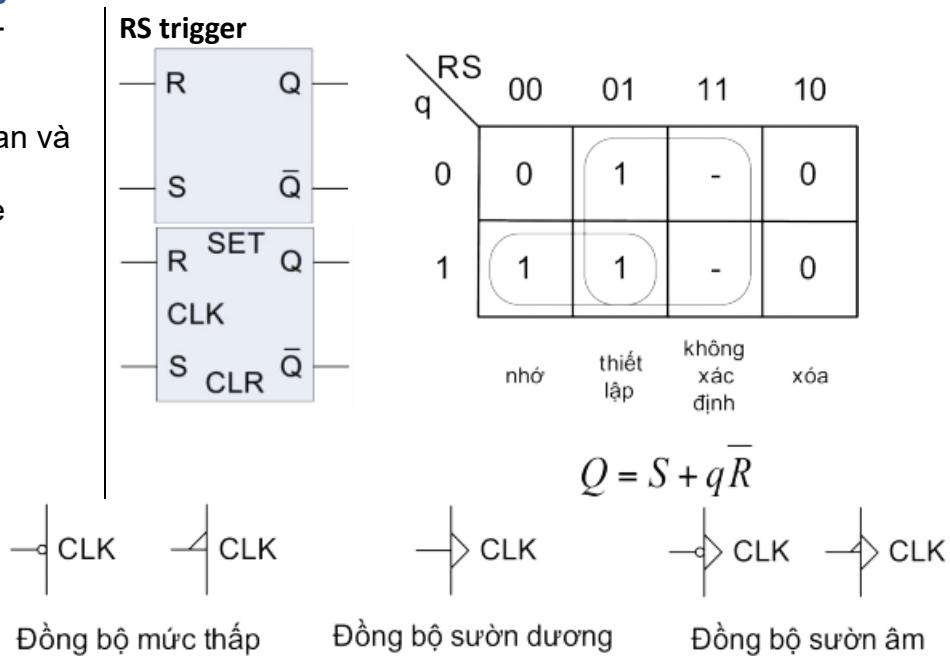
- Phần tử nhớ - trigger
- Đầu ra của trigger là trạng thái của nó
- Phân loại theo trạng thái làm việc
- Không đồng bộ: trạng thái đầu ra chỉ phụ thuộc trạng thái tín hiệu vào
- Đồng bộ: trạng thái đầu ra phụ thuộc trạng thái tín hiệu đầu vào và tín hiệu đồng bộ

Các kiểu đồng bộ

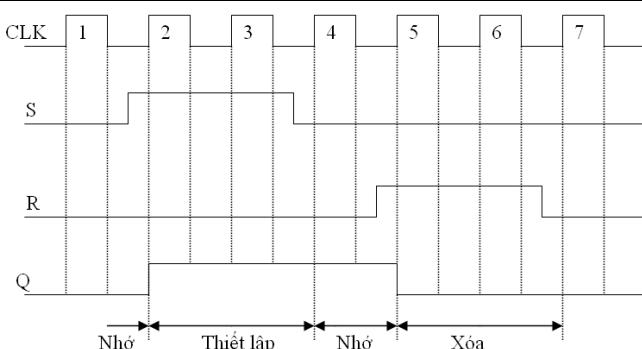
Kiểu đồng bộ		Trạng thái làm việc	Trạng thái nghỉ - giữ nguyên trạng thái
Theo mức	Mức cao	Tín hiệu đồng bộ ở mức cao H	Tín hiệu đồng bộ ở mức thấp L
	Mức thấp	Tín hiệu đồng bộ ở mức thấp L	Tín hiệu đồng bộ ở mức cao H
Theo sườn	Sườn lên	Xung sườn lên của tín hiệu đồng bộ	trường hợp còn lại
	Sườn xuống	Xung sườn xuống của tín hiệu đồng bộ	trường hợp còn lại
Theo xung đồng bộ	 Đồng bộ kiểu xung	Có xung	Không có xung

Các loại trigger

- RS - Set - Reset
- D - Delay
- JK - Jordan và Kelly
- T - Toggle

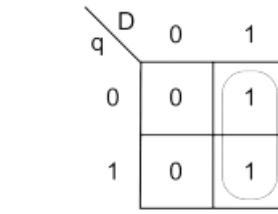
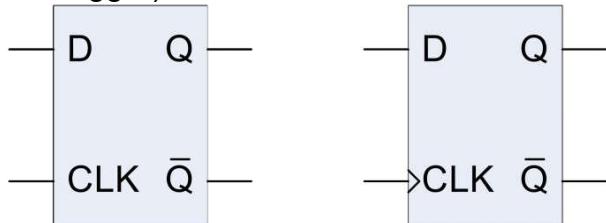
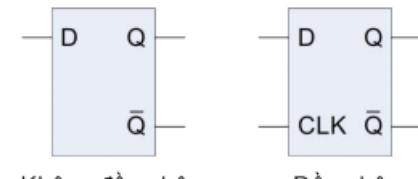


-Cho Trigger RS đồng bộ mức cao và đồ thị các tín hiệu R, S như hình vẽ. Hãy vẽ đồ thị tín:



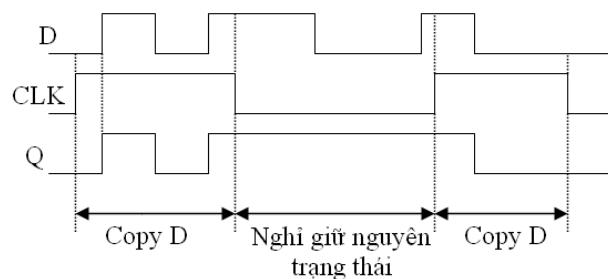
D trigger

- D trigger có 1 đầu vào D và hoạt động ở chế độ đồng bộ và không đồng bộ
- Chỉ xét D trigger ở chế độ đồng bộ
- Theo mức - chốt D (Latch)
- Theo sườn - xúc phát sườn (edge trigger)

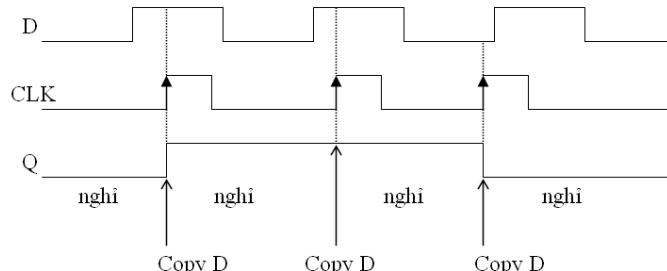


$$Q = D$$

- Cho chốt D kích hoạt mức cao. Hãy vẽ tín hiệu ra Q tương ứng trên cùng trực thời gian với tín hiệu vào D:

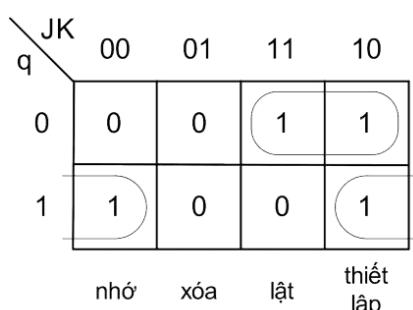


- Cho chốt D kích hoạt sườn dương. Hãy vẽ tín hiệu ra Q tương ứng trên cùng trực thời gian với tín hiệu vào D:



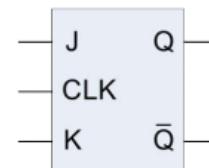
JK trigger

- JK trigger chỉ hoạt động ở chế độ đồng bộ.

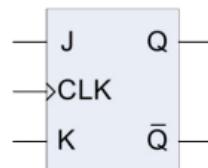


$$Q = \bar{q}J + q\bar{K}$$

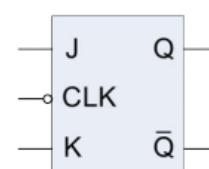
- J ~ S
- K ~ R



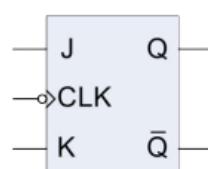
Tích cực mức cao



Tích cực sườn dương



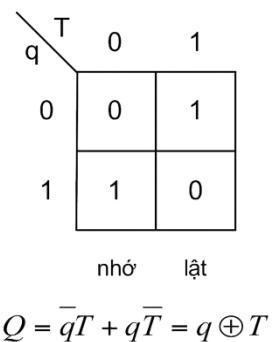
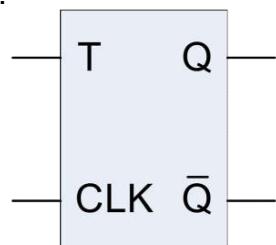
Tích cực mức thấp



Tích cực sườn âm

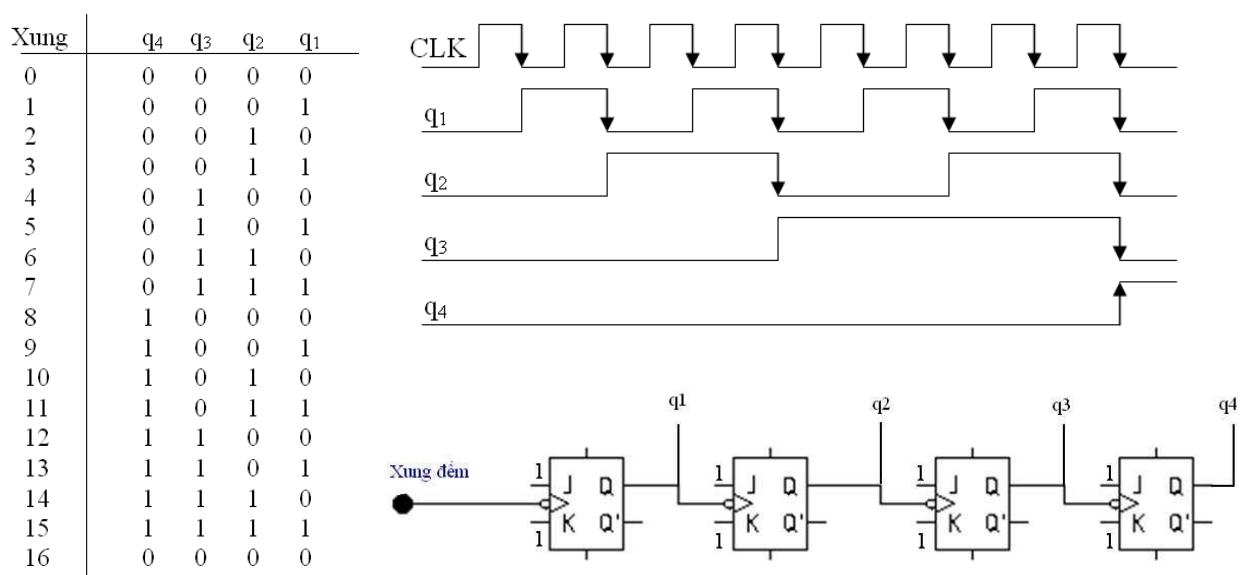
T trigger

T trigger chỉ hoạt động ở chế độ đồng bộ

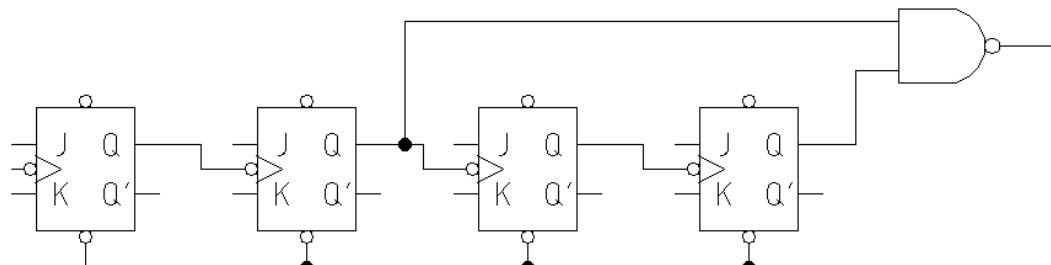


Một số ứng dụng hệ dãy

- Bộ đếm - đếm xung, bộ chia tần
- Module n nếu đếm n xung từ 0 đến n-1
- Đếm không đồng bộ và đếm đồng bộ
- Bộ đếm module 16 không đồng bộ



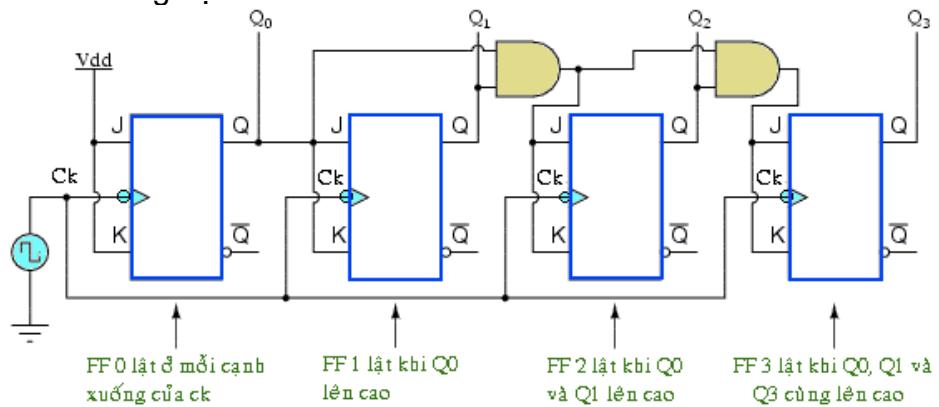
- Bộ đếm module 10 không đồng bộ
đếm đến xung 10 thì xoá về 0 bằng cách tạo xung tích cực vào chân CLR



- Bộ đếm module 8 đồng bộ

xung	q ₃	q ₂	q ₁
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

- Bộ đếm module 16 đồng bộ



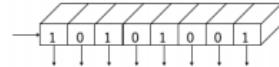
• Thanh ghi

- Có cấu tạo từ các trigger nối tiếp với nhau
- Chức năng: lưu và dịch chuyển thông tin
- Lưu ý: thanh ghi có thể được sử dụng làm bộ nhớ nhưng bộ nhớ không thể làm thanh ghi vì chỉ thanh ghi mới có chức năng dịch chuyển thông tin

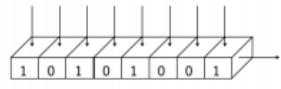
- vào nối tiếp - ra nối tiếp



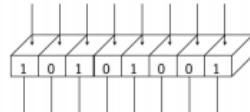
- vào nối tiếp - ra song song



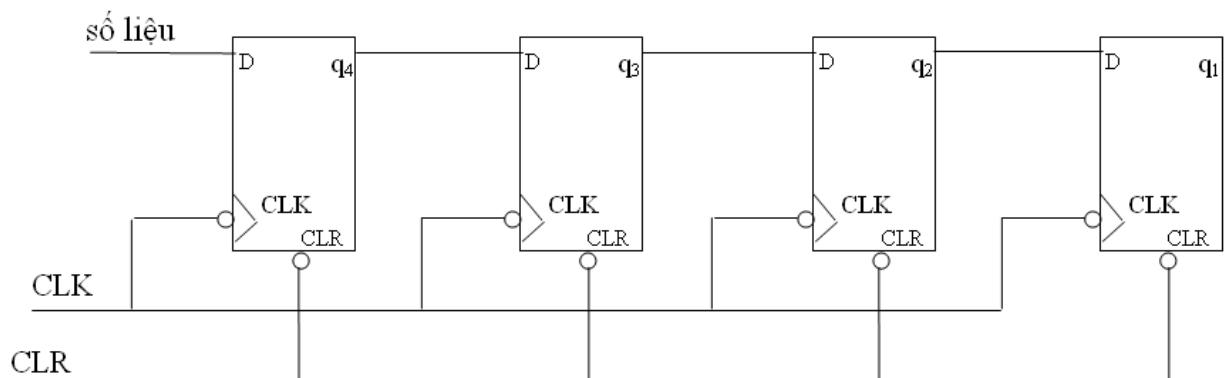
- vào song song - ra nối tiếp



- vào song song - ra song song



- Thanh ghi 4 bit vào nối tiếp - ra song song dùng D trigger



Dòng	Vào			Ra			
	CLR	số liệu	CLK	A	B	C	D
1	0	0	0	0 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
2	1	1	0	0 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
3	1	1	1	1 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
4	1	1	2	1 ↗ 1	0 ↗ 0	0 ↗ 0	0 ↗ 0
5	1	1	3	1 ↗ 1	1 ↗ 1	1 ↗ 1	0 ↗ 0
6	1	0	4	0 ↗ 1	1 ↗ 1	1 ↗ 1	1 ↗ 1
7	1	0	5	0 ↗ 0	1 ↗ 1	1 ↗ 1	1 ↗ 1
8	1	0	6	0 ↗ 0	0 ↗ 0	0 ↗ 0	1 ↗ 1
9	1	0	7	0 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
10	1	0	8	0 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
11	1	1	9	1 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0
12	1	0	10	0 ↗ 1	0 ↗ 0	0 ↗ 0	0 ↗ 0
13	1	0	11	0 ↗ 0	0 ↗ 1	0 ↗ 0	0 ↗ 0
14	1	0	12	0 ↗ 0	0 ↗ 0	0 ↗ 1	0 ↗ 0
15	1	0	13	0 ↗ 0	0 ↗ 0	0 ↗ 0	0 ↗ 0

Chương 2: Giới thiệu công nghệ IC lập trình

- **PLD: Programable Logic Device - IC số khả trình**

- Chứa các cấu trúc mạch có quy luật
- Gồm dãy (ma trận) các ô nhớ đồng dạng, cho phép truy xuất tới từng ô nhớ
- Có thể lập trình để thực hiện các hàm logic khác nhau

- Phân loại:

Phân loại theo mức độ tích hợp		
SSI - Small Scale Intergration	Mạch tích hợp cỡ nhỏ	N<10
MSI - Medium Scale Intergration	Mạch tích hợp cỡ trung bình	10<N<100
LSI - Large Scale Intergration.	Mạch tích hợp cỡ lớn	100<N<1000
VLSI - Very Large Scale Intergration	Mạch tích hợp cỡ rất lớn	>10 ³

- IC số truyền thống

- Sử dụng cổng logic cơ bản
- Phương pháp thiết kế logic truyền thống
- Các IC số tiêu chuẩn: IC74xx họ TTL hoặc 40xx họ CMOS

Đặc điểm:

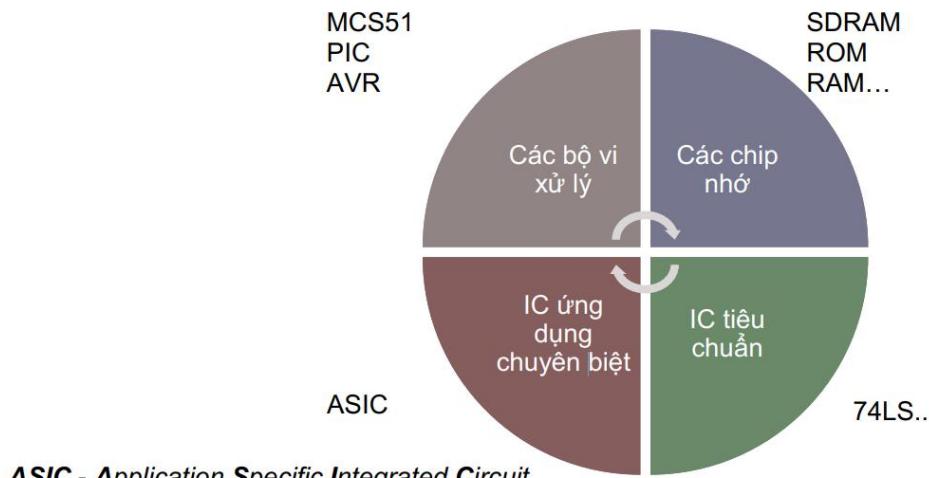
- Dễ dàng ghép nối IC chuẩn
- Thiết kế nhỏ và vừa
- Tốc độ làm việc thấp

- IC số khả trình

- Gồm dãy hoặc ma trận các ô nhớ đồng dạng
- Lập trình để thực hiện các hàm khác nhau
- Chứa các cấu trúc mạch có quy luật

Đặc điểm:

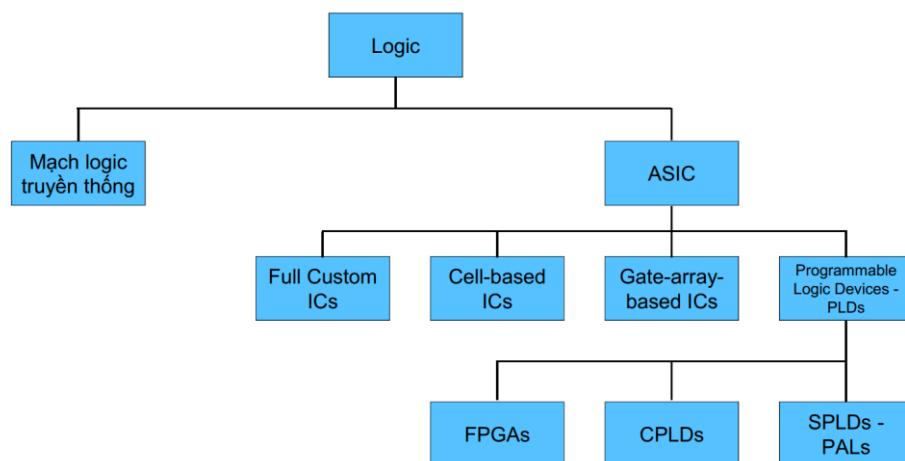
- Tăng khả năng tích hợp
- Công suất thấp
- Dễ thay đổi



Thuật ngữ ASIC

- Ra đời do nhu cầu hoạt động thiết kế vi mạch tăng cao
- ASIC = Application Specific Integrated Circuit - IC chuyên dụng
- Lưu ý: Nếu một IC xuất hiện trong sách tra cứu thì đó không phải là ASIC
- Đặc điểm chung
 - Dùng trong các ứng dụng chuyên biệt
 - Chế tạo chuyên biệt
 - Hiệu quả hơn so với giải pháp dùng phần mềm trên bộ vi xử lý
 - Hầu hết các chip SoC (System-on-chip) là ASIC

PHÂN LOẠI IC KHẢ TRÌNH



Full Custom ICs - ASIC đặc chế hoàn toàn

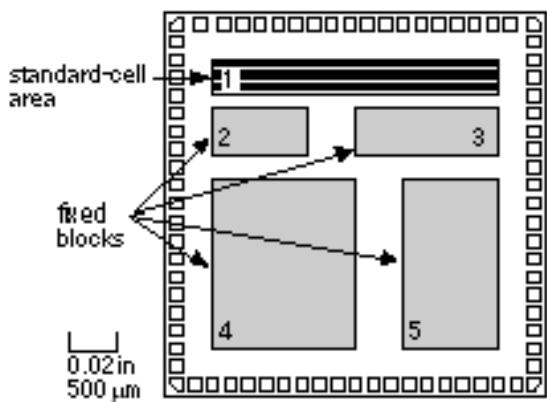
Card 1: Logic cell và các mask layers được thiết kế riêng biệt
Card 2: Mức bố trí Transistor và liên kết
Card 3: Cần thay đổi thiết kế khi công nghệ thay đổi

- Hiệu quả tối ưu
- Giá thành cao
- 8 tuần chế tạo

Each card contains a small image related to its content: Card 1 shows a cross-section of a logic cell with colored layers; Card 2 shows a circuit diagram with labels for transistors and connections; Card 3 shows a small image of an IC die and a text box explaining the design process.

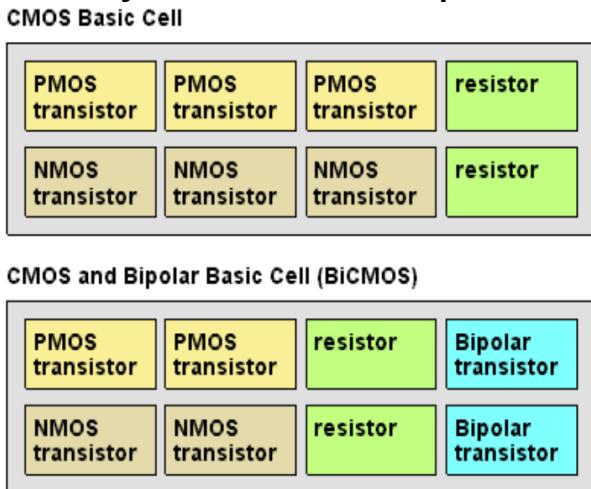
- Logic cell là một đơn vị linh kiện được sử dụng để lưu trữ thông tin 1 bit.
VD: 1 flip-flop
- Mask layers là các mạch hoặc nền tạo mạch điện trong IC, được sử dụng để tạo các kết nối giữa các logic cell

Standard-cell-based-ICs - ASIC dựa trên các tế bào chuẩn



- Cấu trúc gồm các khối
 - Khối tế bào chuẩn (khối 1)
 - Khối cố định (khối 2,3,4 và 5)
- Các khối chứa các hàng của các tế bào chuẩn
- Thiết kế gần giống với các viên gạch xây tường
- Ưu: tiết kiệm thời gian, giảm rủi ro
- Nhược: Chi phí mua thư viện cell, cần thời gian chế tạo mask layer

Gate-array-based-ICs - ASIC dựa trên mảng công logic



- Cấu trúc gồm các mảng các công logic giống nhau (mỗi cell có thể là các transistor và điện trở)
- Dùng phần mềm của hãng để kết nối các khối - dựa vào các thư viện cell và các macro
- Trong thư viện có thể có sẵn một số mask tùy biến

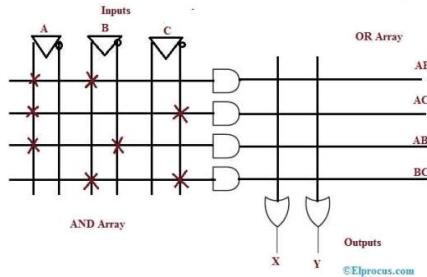
Programmable Logic Device - PLD - các vi mạch lập trình

- Là mạch kỹ thuật số có thể cấu hình lại bằng phần mềm
- Tại thời điểm sản xuất, mạch PLD mới chỉ gồm các thành phần logic, chưa được liên kết với nhau nên chưa thể hoạt động
- Trước khi sử dụng, PLD sẽ được lập trình bằng phần mềm chuyên biệt
- Phân loại
 - Simple PLD – SPLD
 - Complex PLD – CPLD
 - Field Programmable Gate Array – FPGA

Simple PLD – SPLD

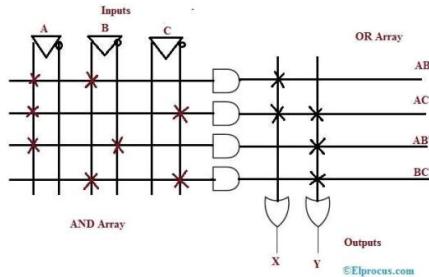
- Cấu tạo gồm từ vài chục - vài trăm cổng logic được tổ hợp dưới dạng ma trận các cổng AND và OR
- Hoạt động tốc độ cao, mạch cõi nhỏ có thể lập trình 1 hoặc nhiều lần
- Phân chia thành 2 loại:

PAL - Programmable Array Logic



AND khả trình, OR cố định

PLA - Programmable Logic Array

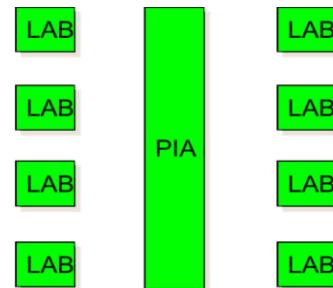


AND và OR đều khả trình

12

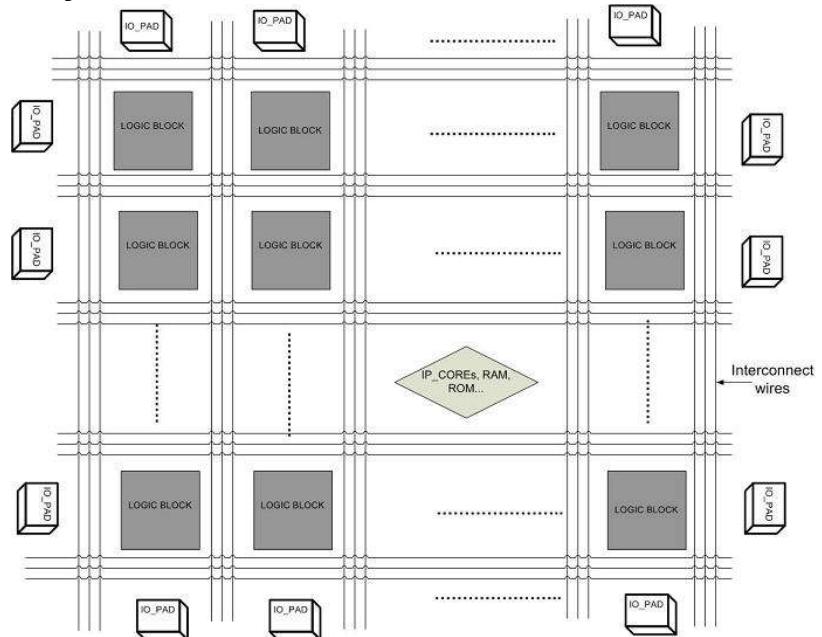
Complex PLD – CPLD

- Gồm nhiều mảng Simple PLD
- Cấu tạo gồm khối SPLD được gọi là Logic Array Block LAB và
- Khối Programmable Interconnect Array - PIA



Field Programmable Gate Array – FPGA

- Gồm các khối khả trình, đúc sẵn các khối logic và các kết nối
- Các khối được kết nối với nhau bằng cách lập trình
- Đặc điểm**
 - Không cần chế tạo
 - Phù hợp với thiết kế có độ phức tạp trung bình (<1M gate)



Kiến trúc tổng quan FPGA

FPGA vs Vi điều khiển

- Vi điều khiển
- Chỉ xử lý được 1 luồng lệnh
- Xung nhịp trong khoảng 10 - 100MHz
- Xử lý luồng lệnh đơn
- Hoạt động với các dữ liệu 8,16 or 32bit

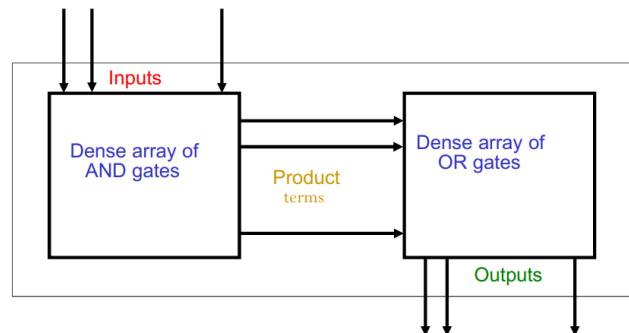
- FPGA
- Có khả năng xử lý đồng thời nhiều luồng dữ liệu song song
- Các hàm có thể hoạt động với các dữ liệu có độ lớn lên tới hàng trăm bit

FPGA vs CPLD

	FPGA	CPLD
Khối logic	Logic cell nằm ngoài, chia chung nguồn tài nguyên	Logic cell nằm giữa các nguồn tài nguyên
Kết nối	Phức tạp	Đơn giản hơn
Công nghệ lập trình	SRAM	EPROM, EEPROM
Cấu trúc logic	LUT	Mảng AND
Ứng dụng	Vừa và lớn	nhỏ

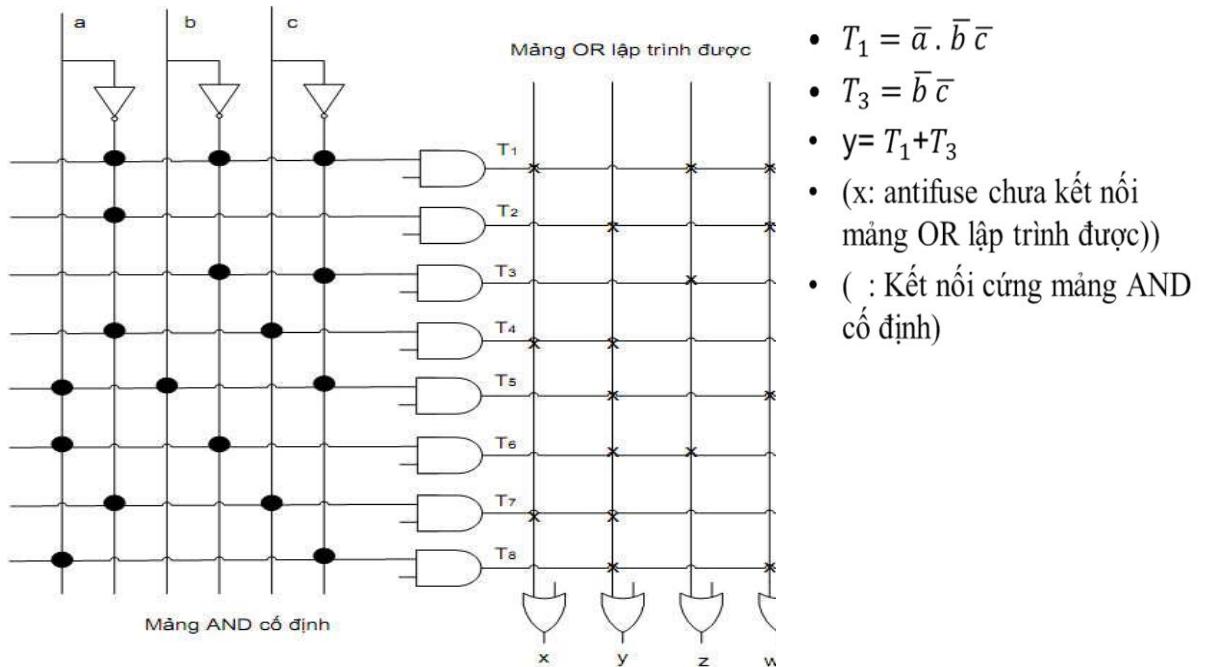
KIẾN TRÚC IC KHẢ TRÌNH

- Thực hiện hàm bằng cách ghép 2 mảng AND và OR
- Một trong 2 mảng khả trình, được gọi là IC khả trình



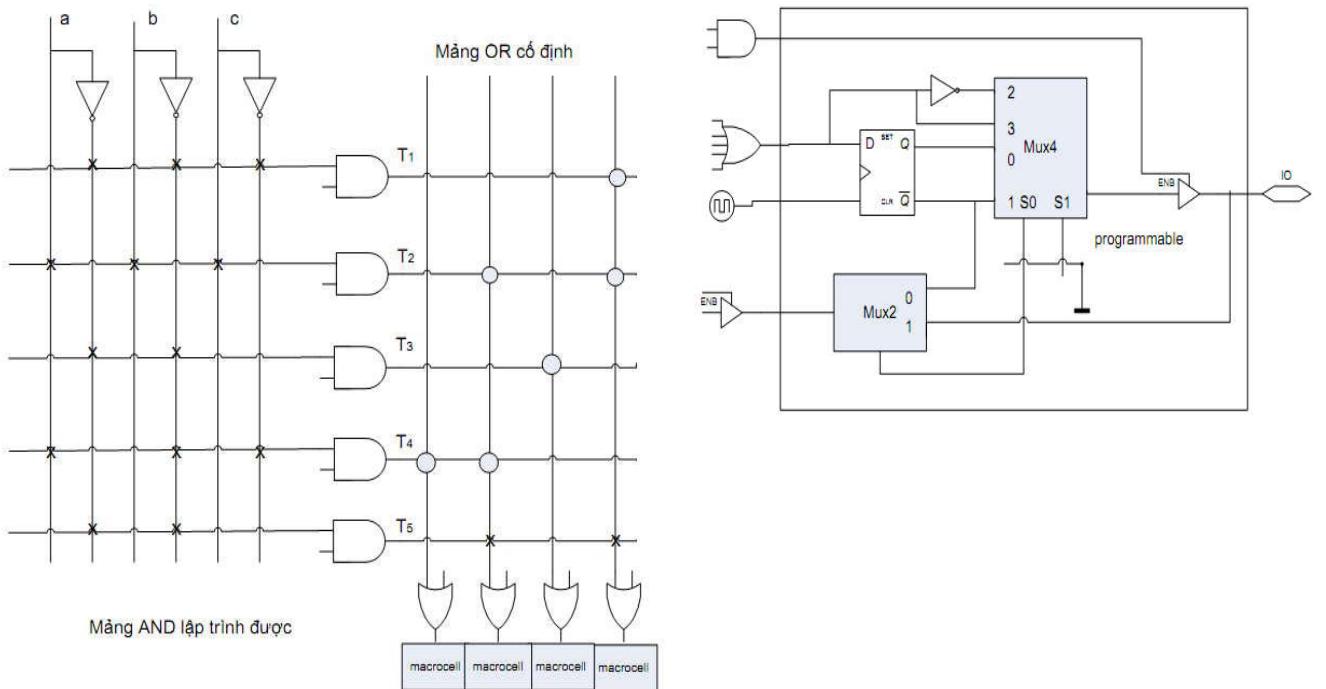
Kiến trúc SPLD-PROM

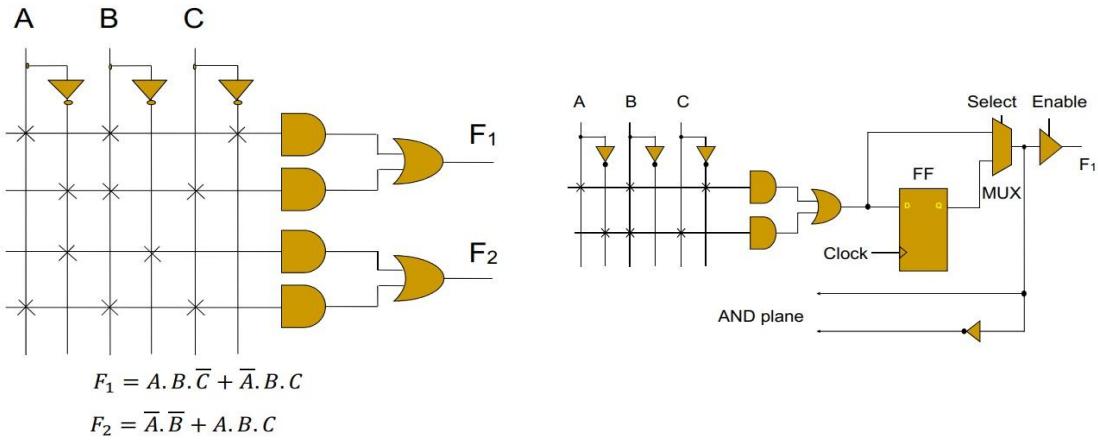
- PROM - Programmable Read Only Memory - 1956
- 16 đến 32 input. Do đó chỉ thực hiện các hàm đơn giản
- Cấu trúc: AND cố định, mảng OR khả trình
- Thực hiện thông qua kết nối antifuse
- PROM có khả năng tái lập trình gồm
 - UEPROM - Ultraviolet-Erasable PROM
 - EEPROM - Electric-Erasable PROM



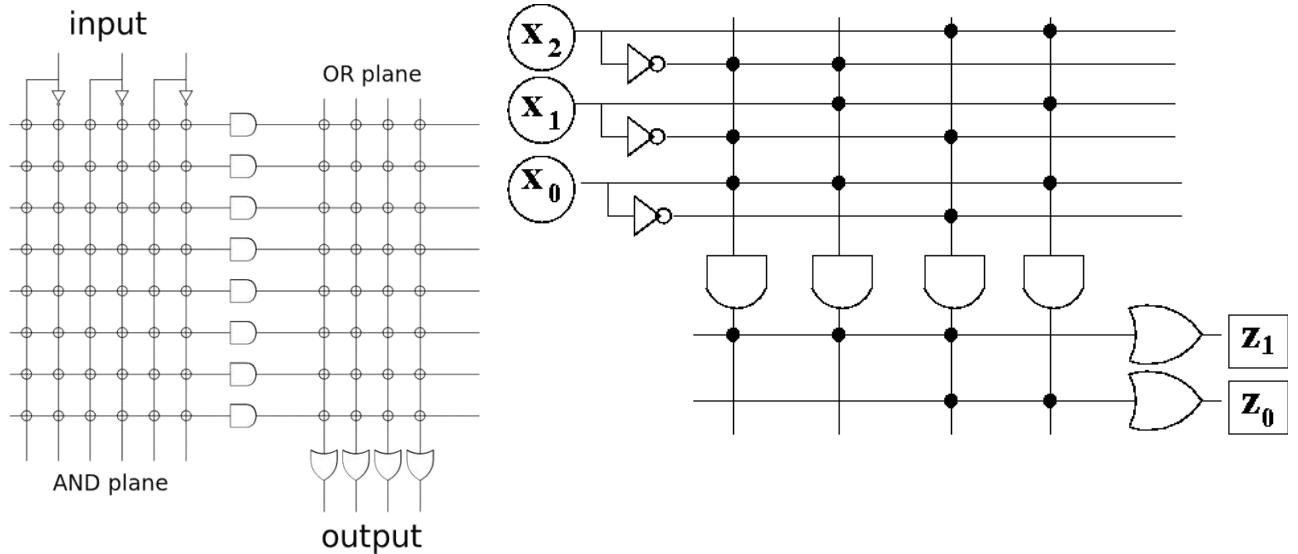
Kiến trúc SPLD-PAL

- PROM - Programmable Array Logic - 1970
- Cấu trúc: AND khả trình, mảng OR cố định. Do vậy linh hoạt hơn
- Mỗi đầu ra của mảng OR lập trình được gắn với macro-cell
- Macro-cell = 1 flip-flop + 1 MUX2 + 1 MUX4, được dùng để thực hiện các mạch logic dãy





- Cả 2 ma trận AND và OR đều khả trình



Thực hiện PLA logic

- Cần thực hiện các hàm:

$$F_0 = A + \bar{B}C$$

$$F_1 = A + \bar{B} \cdot C$$

$$F_1 = A \cdot B + A \cdot \bar{C}$$

$$F_2 = A \cdot B + \bar{B} \cdot \bar{C}$$

Personality Matrix

Product term	Inputs			Outputs			
	A	B	C	F_0	F_1	F_2	F_3
AB	1	1	-	0	1	1	0
$\bar{B}C$	-	0	1	0	0	0	1
$A\bar{C}$	1	-	0	0	1	0	0
$\bar{B}\bar{C}$	-	0	0	1	0	1	0
A	1	-	-	1	0	0	1

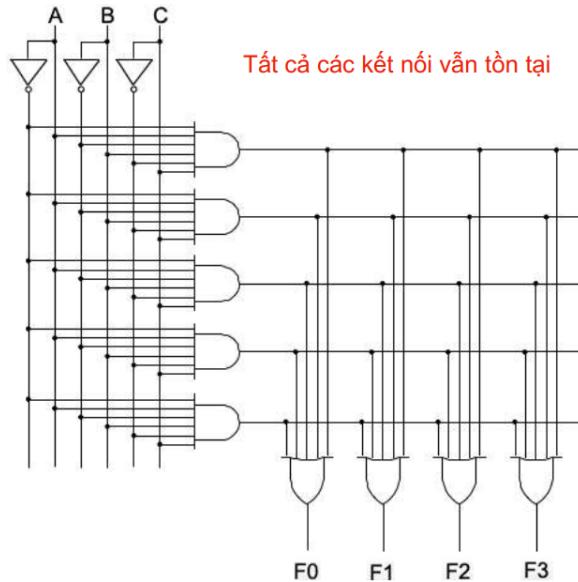
Input Side:

1 = asserted in term
0 = negated in term
- = does not participate

Output Side:

1 = term connected to output
0 = no connection to output

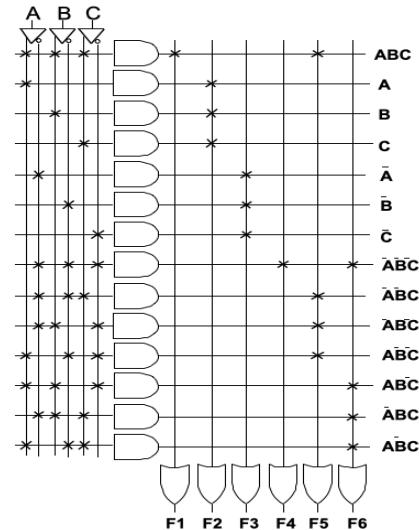
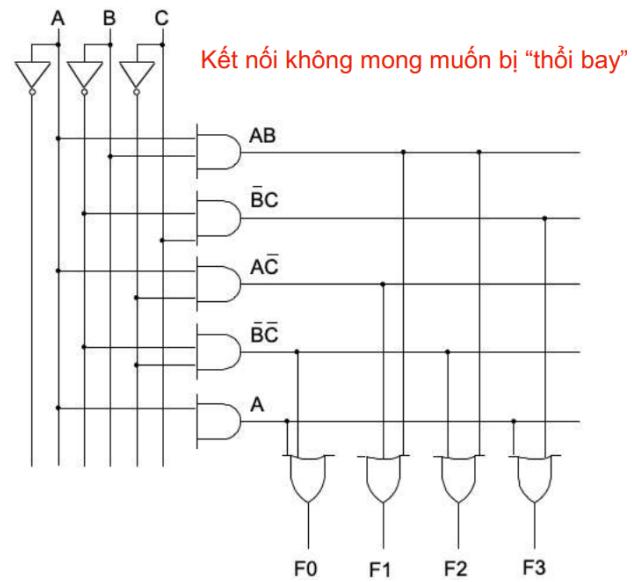
- Trước khi lập trình



Multiple functions of A, B, C

$$\begin{aligned}
 F1 &= ABC \\
 F2 &= A + B + C \\
 F3 &= \overline{ABC} \\
 F4 &= \overline{A} + \overline{B} + C \\
 F5 &= A \oplus B \oplus C \\
 F6 &= \overline{A \oplus B \oplus C}
 \end{aligned}$$

- Sau khi lập trình

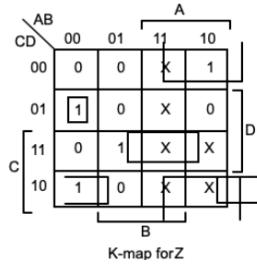
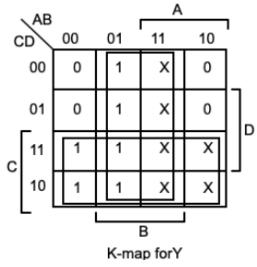
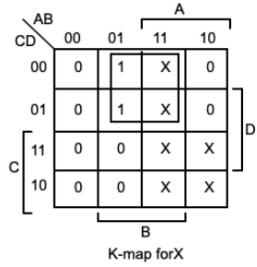
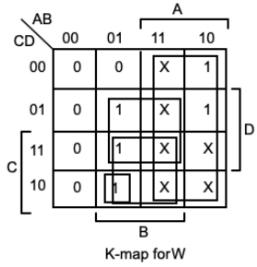


So sánh PALs và PLAs

PLAs	PALs
Cả mảng AND và OR đều lập trình được	Mảng AND lập trình được, mảng OR cố định
Linh hoạt hơn	Kém linh hoạt
Thực hiện được một số lượng lớn hàm logic, nhưng nhiều chân	Số lượng hàm output phụ thuộc số lượng cổng OR
Giá thành cao	Giá thành rẻ hơn so với PLAs

Bộ chuyển mã BCD sang mã Gray

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
0	1	1	1	1	0	1	1
1	0	0	0	1	0	0	1
1	0	0	1	1	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

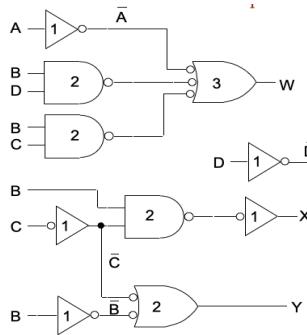


$$W = A + B \cdot D + B \cdot C$$

$$X = B \cdot \bar{C}$$

$$Y = B + C$$

$$Z = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{B} \cdot C \cdot \bar{D} + A \cdot \bar{D} + B \cdot C \cdot D$$



1: 7404 hex inverters
2,5: 7400 quad 2-input NAND
3: 7410 tri 3-input NAND
4: 7420 dual 4-input NAND

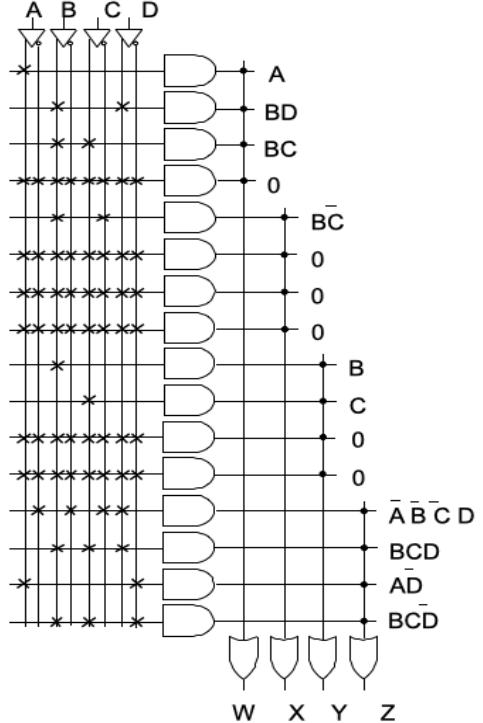
$$W = A + B \cdot D + B \cdot C$$

$$X = B \cdot \bar{C}$$

$$Y = B + C$$

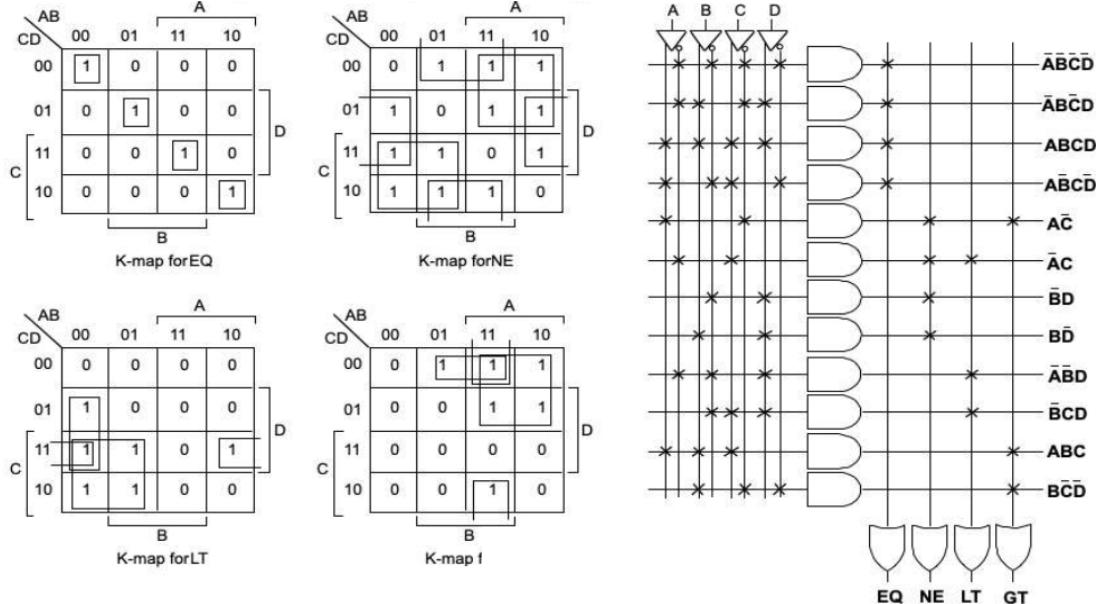
$$Z = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot D + \bar{B} \cdot C \cdot \bar{D} + A \cdot \bar{D} + B \cdot C \cdot D$$

cần 4 packages SSI



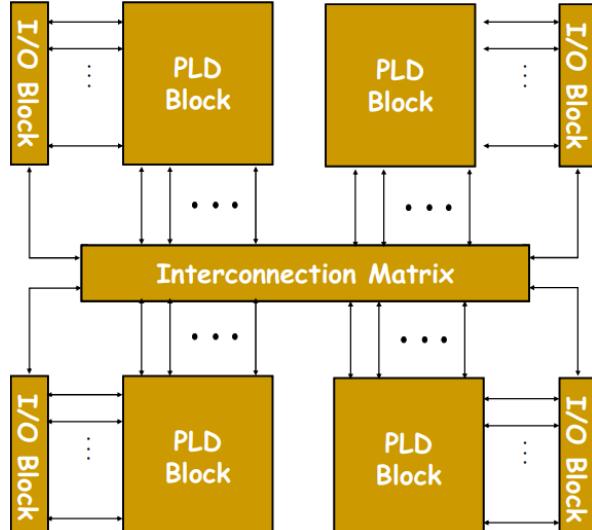
cần 1 package PLA/PAL

Bộ so sánh số 2 bit



Kiến trúc CPLD

- CPLD thường gồm logic tổ hợp PAL và các FF
- Cấu trúc thành các PLD Block – Logic Block là các SPLD cài sẵn, chứa từ 8 đến 16 macrocell
- Mã trận kết nối khả trinh, kết nối các khối PLD và I/O
- Dùng công nghệ lập trình EEPROM
- Kích thước mảng OR cố định
- Đầu ra tổ hợp hoặc dãy (registered)
- Ứng dụng: Bộ đếm đơn, State machines, decoders,...

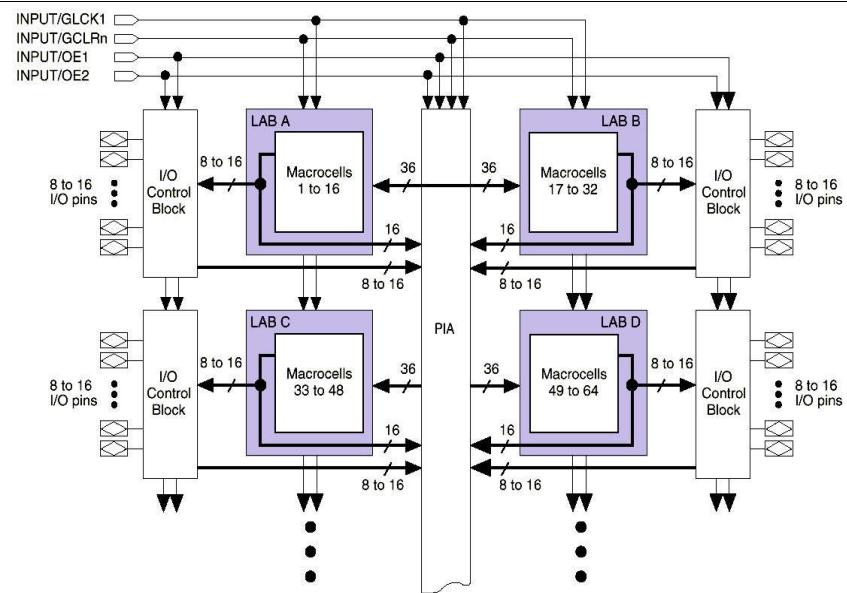


Board mạch CPLD - Altera Max7000

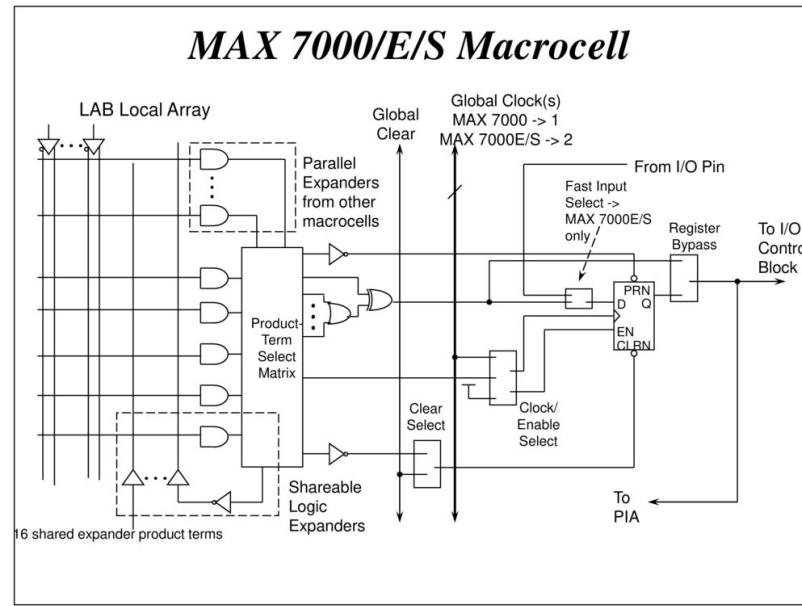
- Mật độ từ 2 LAB đến 16 LAB-Logic Array Block
- Mỗi LAB tương ứng với một SPLD dùng công nghệ EEPROM
- Mỗi LAB có 16 macrocell
- PIA - cấu trúc bú lập trình, kết nối các LAB, I/O, macrocell
- Kiểu lập trình ISP(In System Programmable)

Chức năng	EPM70 32	EPM70 64	EPM70 96	EPM71 28E	EPM71 60E	EPM71 92E	EPM72 56E
Useable gates	600	1250	1800	2500	3200	3750	5000
Macro-cells	32	64	96	128	160	192	256
LABs	2	4	6	8	10	12	16
Mã I/O pin	36	68	76	100	104	124	164

• EPM7000 Series
Block Diagram

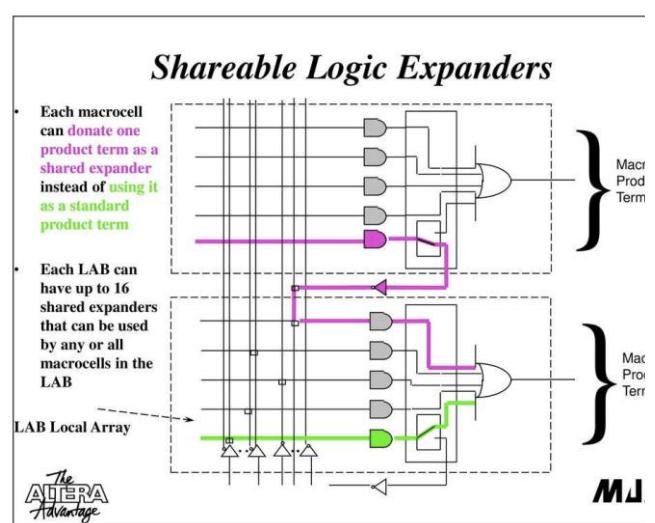
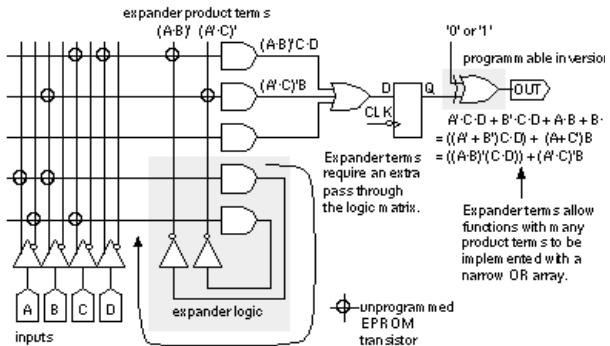


• Macrocell



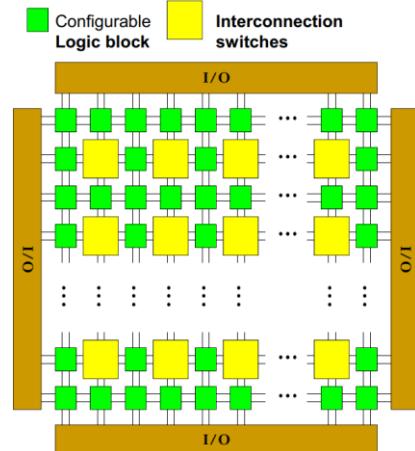
- Chứa 1 mảng cổng AND khả trình gồm: 5 cổng AND + 1 cổng OR, và ma trận lựa chọn tích
- Ma trận lựa chọn tích: kết nối output AND với input OR, mạch logic tổ hợp phục vụ lập trình cho đầu vào/ra
- Phần mở rộng chia sẻ: hồi tiếp các thành phần tích trở lại ma trận lập trình để chia sẻ với các macrocell khác
- Phần mở rộng song song: mượn các thành phần tích dùng từ các macrocell khác

• Phần mở rộng chia sẻ

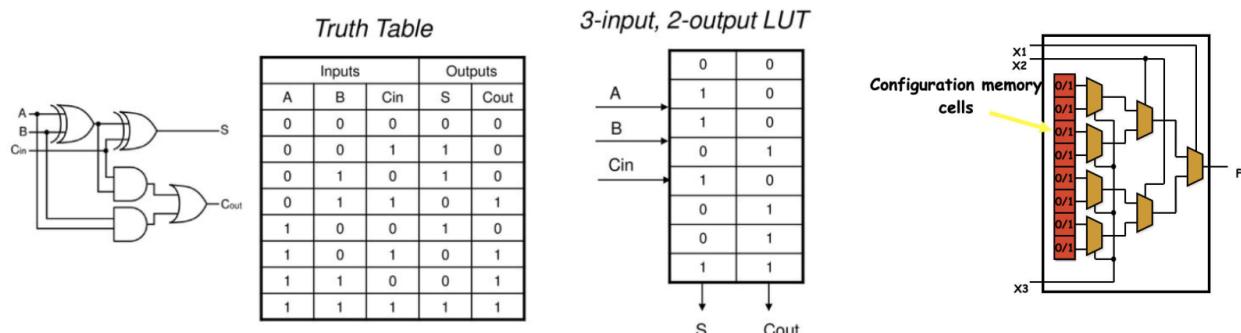
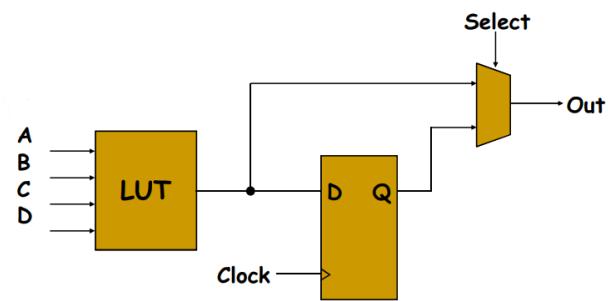


Field-Programmable Gate Array – FPGA

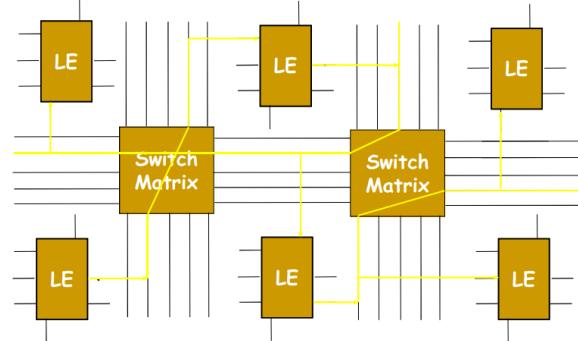
- FPGA: mảng cổng logic có thể lập trình được theo trường/miền
- Mật độ tích hợp cao hơn nhiều so với CPLD
- Cấu trúc mảng phần tử logic - CLB có thể lập trình được
- Kết nối theo kiểu hàng và cột
- Phản tử logic tạo hàm nhỏ hơn nhiều so với CPLD
- Công nghệ lập trình SDRAM
- FPGA có số lượng cổng logic lớn hơn rất nhiều so với CPLS
 - 2K đến 10M cổng
 - Đòi hỏi nhiều công nghệ khác nhau
 - FPGA chia làm 2 loại: RAM-based hoặc Flash-based:
- RAM based FPGA
 - Lập trình khi power-on
 - Cần bộ nhớ mở rộng để lập trình
 - Có thể cấu hình động
- Cấu trúc chung
 - Configurable Logic Block - CLB: Khả trinh, thực hiện chức năng logic dãy và tổ hợp
 - Interconnection: Khả trinh, kết nối đầu vào/ra với các logic Block CLB
 - Khối I/O khả trinh
 - Khối khác:
 - Phân phối Clock
 - Bộ nhớ nhúng
 - DSP, RAM, ROM, ...
- Flash based FPGA
 - Lưu chương trình trong bộ nhớ bền vững
 - Lập trình lại khó khăn
 - Cấu hình giữ nguyên khi mất điện



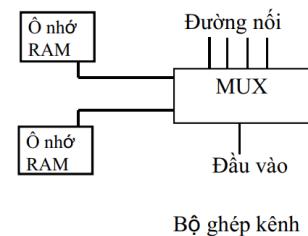
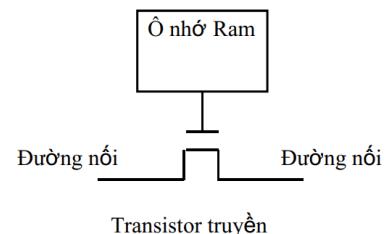
- Configurable Logic Block - CLB
- Chứa phần tử logic cơ bản Logic Element (LE)
- Kết nối bên trong CLB khả trinh, kết nối giữa LE với CLB
- Mỗi LE gồm:
 - Khối bảng tham chiếu LUT - mạch tổ hợp
 - Thanh ghi thực hiện mạch logic
 - Một số mạch logic khác: số học, mạch mở rộng cho các chức năng có hơn 4 input
- Look-Up Table (LUT)
- Bảng tham chiếu n-input có thể thực hiện bất kì hàm logic hay tổ hợp các đầu vào
- Lập trình bằng bảng chân lý



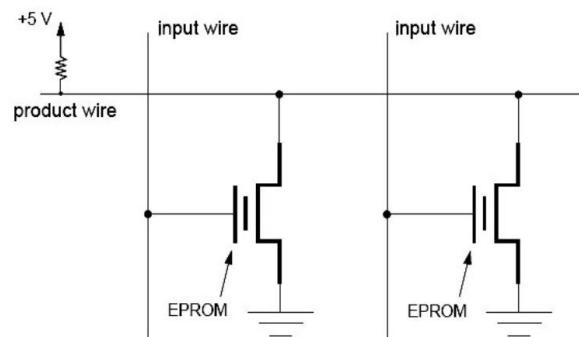
- Kết nối khả trinh
 - Có kiến trúc định tuyến, gồm các đoạn dây nối và các chuyển mạch lập trình được
 - Chuyển mạch lập trình được có thể có nhiều cấu tạo: anti-fuse, EPROM và EEPROM transistor
- Ưu điểm
 - Khả năng tái lập trình khi đang sử dụng
 - Công đoạn thiết kế đơn giản so với ASIC, giảm được chi phí, thời gian
 - Khả năng cấu hình động
- Ứng dụng
 - Nền tảng lý tưởng để tạo mẫu
 - Thiết kế và hoàn thiện nhanh chóng, giảm thời gian đưa ra thị trường
 - Giải pháp hiệu quả cho các sản phẩm không yêu cầu số lượng lớn
 - Thực hiện các hệ thống phần cứng yêu cầu khả năng tái lập trình
 - Thực hiện các hệ thống tái cấu hình động



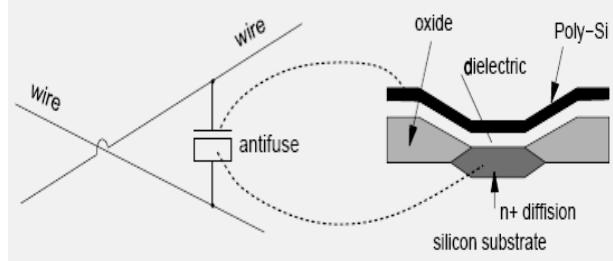
- Công nghệ lập trình chip
 - Các phần tử lập trình: cho phép các kết nối có thể lập trình được giữa các đoạn dây nối
 - Chức năng: thực hiện các kết nối lập trình được giữa các khối logic của FPGA
 - Được cấu hình ở trạng thái ON hoặc OFF
 - Chiếm diện tích của chip càng nhỏ càng tốt
 - Trở kháng khi ON thấp và khi OFF rất cao
 - Điện dung kí sinh thấp khi nối các đoạn dây
 - Có thể tích hợp một số lượng lớn các phần tử lập trình trên 1 chip
- Phân loại công nghệ lập trình chip
 - Dùng SRAM
 - Dùng EPROM (UV Light Erasable PROM) và EEPROM
 - Dùng cầu chì nghịch (anti-fuse)
- Lập trình dùng SRAM
 - Các kết nối lập trình được điều khiển bằng các cell nhớ SRAM
 - Mỗi chuyển mạch là một transistor truyền/MUX được điều khiển bằng trạng thái của một bit SRAM
 - Cần Power-on để nạp chương trình
 - Chip có diện tích lớp vì cần ít nhất 5 transistor cho mỗi RAM cell cũng như các transistor cần thêm cho mỗi cổng truyền hay MUX
 - Cho phép FPGA được tái cấu hình ngay trên mạch rất nhanh và có thể được chế tạo bằng công nghệ CMOS chuẩn



- Lập trình dùng EPROM
 - Thường dùng cho CPLD và SPLD
 - Tái lập trình không cần bộ nhớ ngoài
 - Không tái cấu hình trực tiếp trên mạch
 - EPROM hay EEPROM transistor được đặt giữa 2 dây thực hiện các chức năng nối AND
- EEPROM transistor 2 cổng
 - ON: Không có điện tích giữa 2 cổng
 - OFF: cho dòng lớn chạy giữa nguồn và kênh, điện tích được giữ lại cổng treo, Transistor OFF



- Lập trình dùng cầu chì nghịch (anti-fuse)
- Cấu trúc gồm 3 lớp: Silic mang nhiều điện tích dương (n^+), điện môi và Poly-Silic
- Lập trình bằng cách đặt điện áp cao (~18V) giữa 2 đầu antifuse, dòng điều khiển làm điện môi nóng chảy tạo lớp liên kết giữa Poly-Silic và n^+ . Hai lớp này được nối với các dây kim loại (300-500 Ω)
- Chỉ thực hiện 1 lần, không có khả năng tái lập trình
- Diện tích chip nhỏ. Tuy nhiên cần có không gian lớn cho các transistor điện thế cao để giữ cho dòng và áp cao lúc lập trình

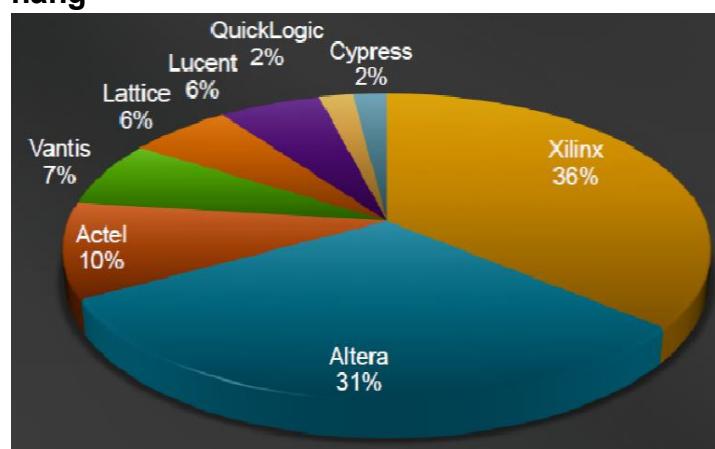


- Tóm tắt:

Name	Re-programmable	Volatile	Technology
Fuse	no	no	Bipolar
EPROM	yes out of circuit	no	UVCMS
EEPROM	yes in circuit	no	EECMOS
SRAM	yes in circuit	yes	CMOS
Antifuse	no	no	CMOS+

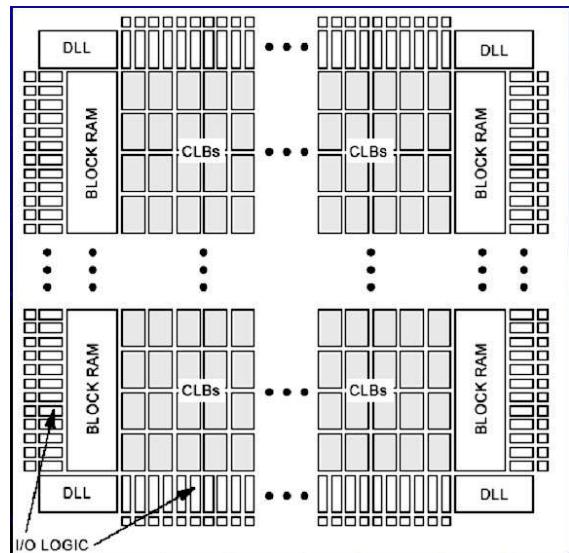
Cấu trúc FPGA của một số hãng

- **Xilinx**
- **Altera**
- **Actel**
- **Lattice**
- **QuickLogic**

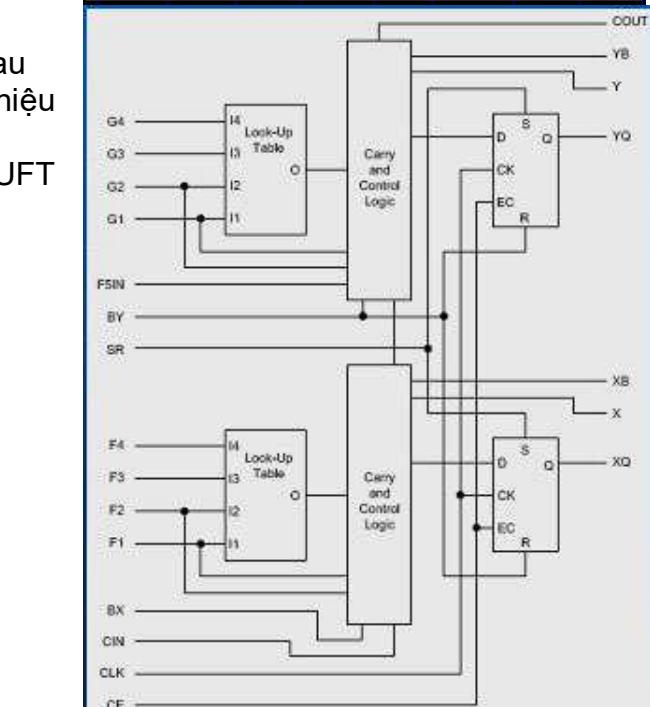


Xilinx FPGA Spartan II E

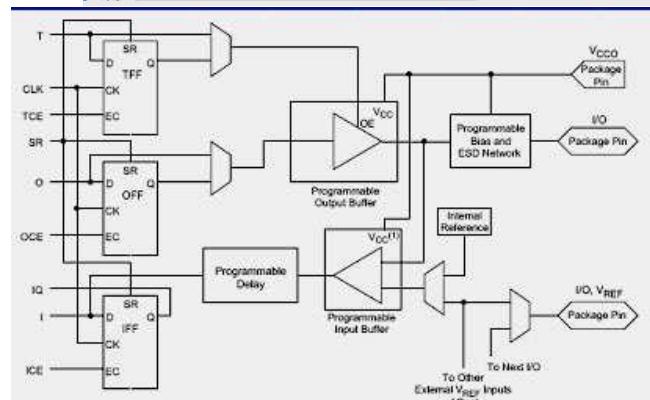
- Gồm mảng 2 chiều các khối lập trình được CLB
- Một CLB gồm 4 logic-cell
- Điều khiển lập trình bằng SRAM
- I/O logic block
- DLL(Delay Lock Loops-vòng khoá trễ): điều khiển xung Clock để giảm trễ, tạo trễ, đồng bộ giữa các tín hiệu Clock



- Logic cell
- Mỗi Logic-cell gồm 2 LUT giống nhau
- Mỗi LUT gồm 4 chân đầu vào, tín hiệu điều khiển và các D-FlipFlop
- 2 bộ điều khiển ngõ ra 2 trạng thái BUFT

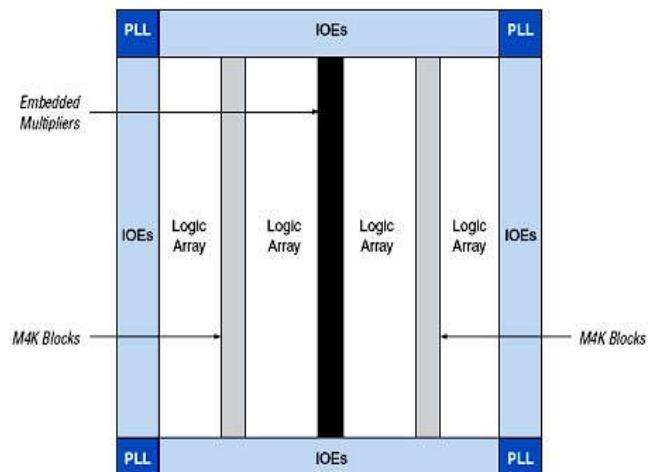


- I/O Logic Block
- Gồm 3 FF chung Clock và các tín hiệu Clock Enable (CE) điều khiển độc lập cho từng FF
- Tín hiệu vào qua 1 bộ đệm, tín hiệu ra qua bộ đệm 3 trạng thái theo các chuẩn bộ nhớ/giao tiếp Bus

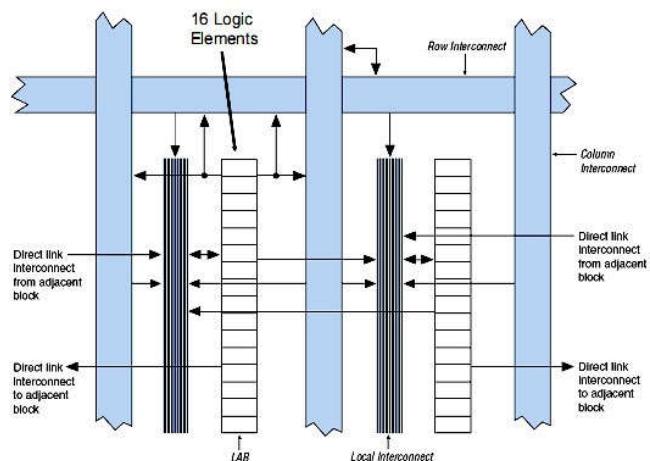


Altera FPGA Cyclone

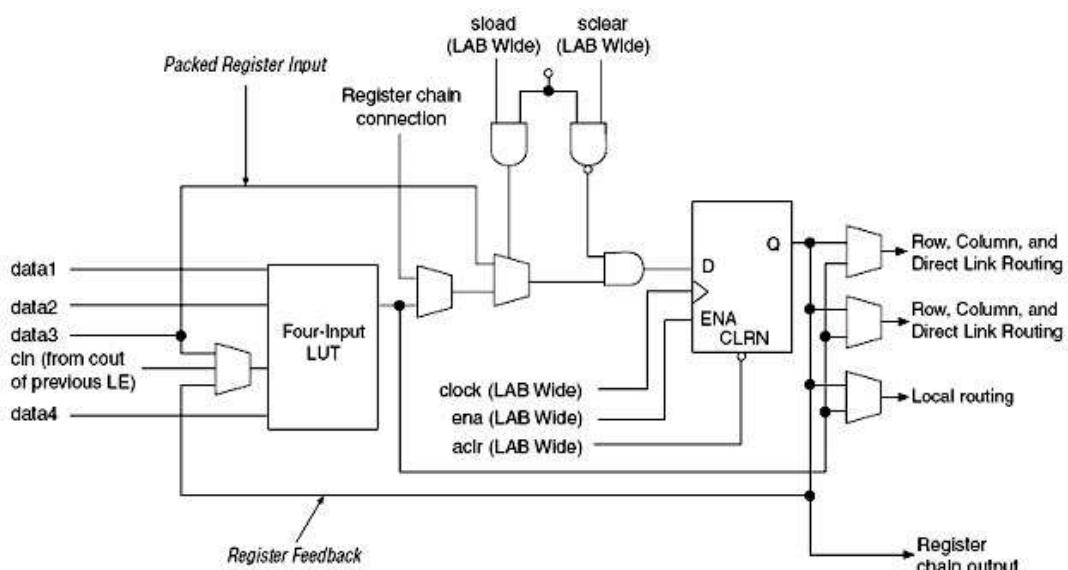
- Gồm một mảng lớn các dãy Block lập trình LAB, kết nối với nhau bởi PIA – Programmable Interconnect Array
- Gồm các khối
 - Các dãy logic chứa các bảng LUT
 - Bộ nhớ dạng khối - M4K
 - Bộ ghép kênh tích hợp sẵn
 - Khối vào ra IOE
 - Vòng khóa pha PLL cung cấp xung nhịp, tạo sự dịch pha cho các yêu cầu cần hỗ trợ đầu ra tốc độ cao



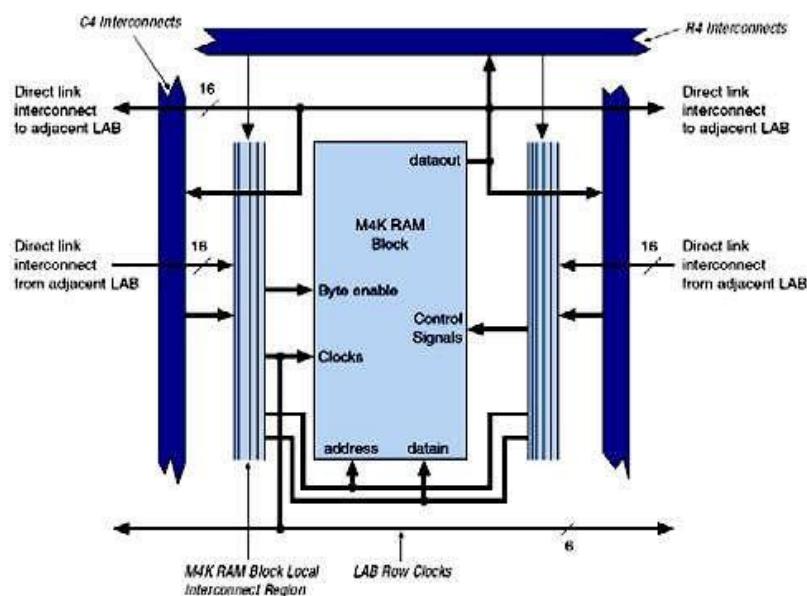
- LAB
 - Kích thước 1 LAB tương đương dung lượng nhớ 4K
 - Các đường kết nối hàng và cột giúp đẩy nhanh tốc độ kết nối giữa các LE trong LAB và giữa các LAB với nhau



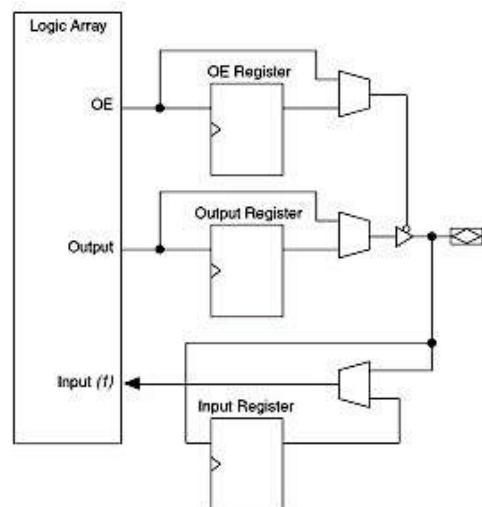
- LE - Logic Elements
 - 16 khối LE tạo nên một khối LAB



- Bộ nhớ M4K
- Tích hợp sẵn thường có 1 hoặc 2 khối bộ nhớ dung lượng M4K
- Thực hiện thanh ghi dịch và nhiều kiểu bộ nhớ khác nhau như RAM, ROM, các bộ đệm FIFO, thanh ghi dịch

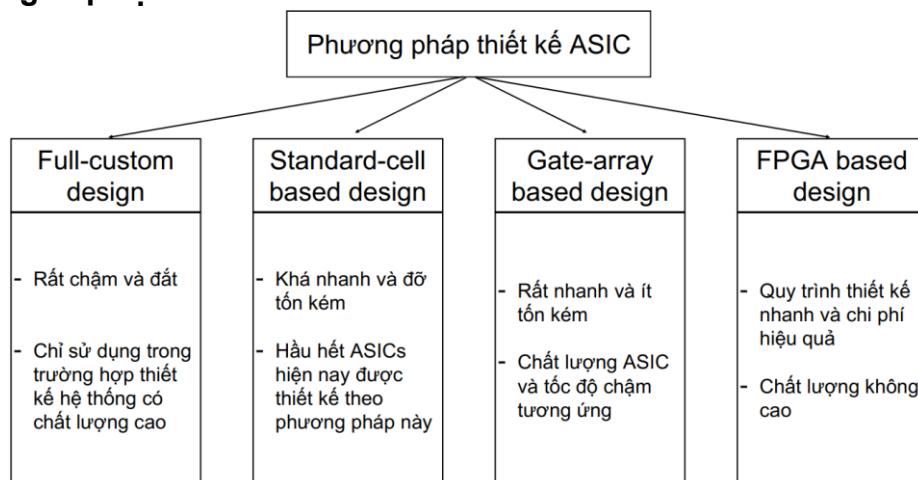


- Khối vào ra IOE
- Mỗi phần tử vào ra IOE chứa 1 bộ đệm vào ra 2 hướng và 3 thanh ghi để truyền dẫn tín hiệu theo 2 hướng



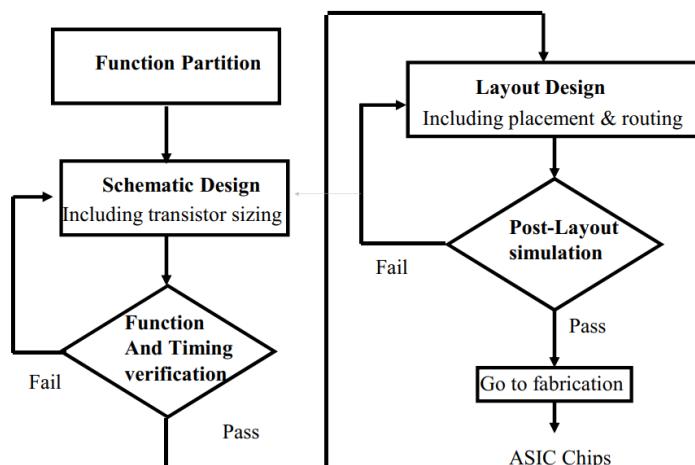
QUY TRÌNH THIẾT KẾ IC KHẢ TRÌNH

Các hướng tiếp cận thiết kế ASIC



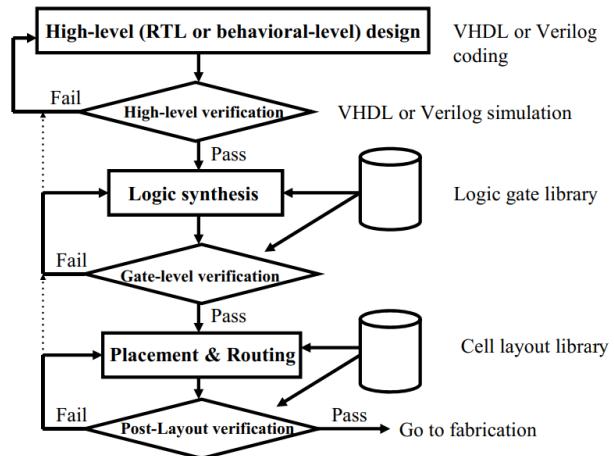
Full-custom design

- Thiết kế chip từ những phần tử đơn giản nhất
- Thiết kế được từ các phần tử nhỏ đến cả IC
- Rất linh hoạt, có khả năng tối ưu
- Tốn nhiều công sức, chi phí cao



Standard-cell based design

- Có khả năng tự động thực hiện
- Cần sự hỗ trợ từ thư viện
- Do sử dụng thư viện nên tiết kiệm được thời gian thiết kế, giảm nhiều công sức



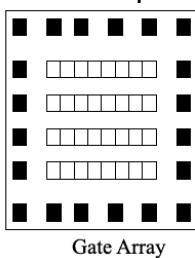
Gate Array based design

-Hướng tiếp cận này nhanh hơn so với Standard-cell based design do một số thành phần của IC đã được thực hiện xong

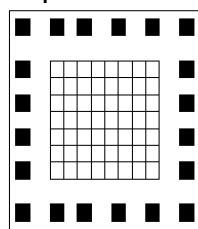
-Một số thành phần của chip (transistor) đã có sẵn. Thành phần còn lại (wire) sẽ được thiết kế, chế tạo thêm để hoàn thiện mạch

-Ưu điểm là tiết kiệm

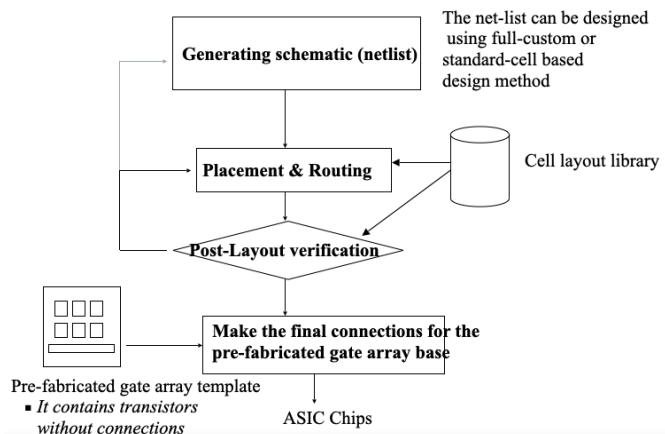
-Nhược điểm là hệ thống không được tối ưu như 2 phương pháp đầu



Gate Array

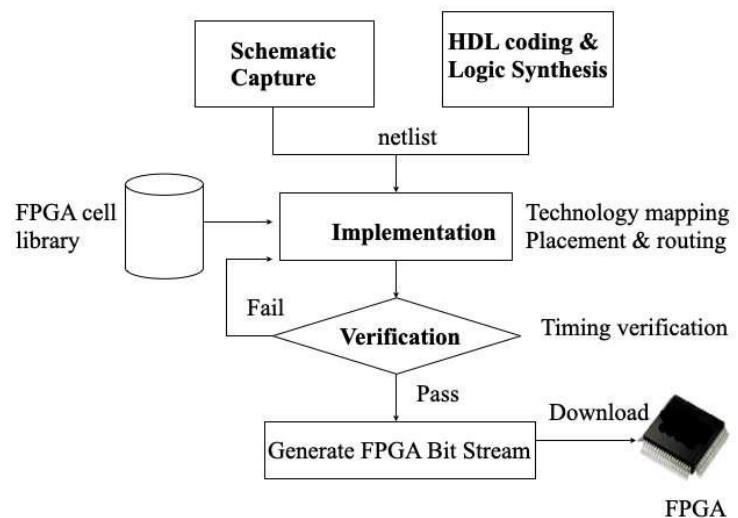


Sea-of-Gates



FPGA based design

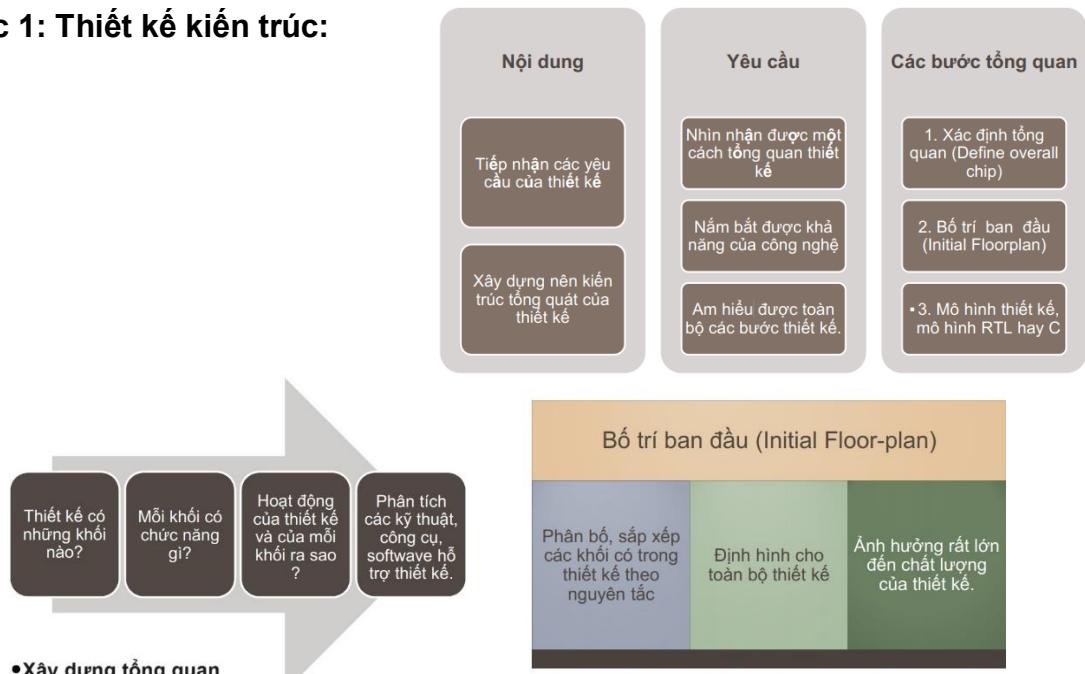
- Hướng tiếp cận nhanh nhất do phần lớn các thành phần của IC đã được chế tạo xong
- Việc thay đổi các kết nối được thực hiện bằng phần mềm



Quy trình thiết kế

- Xây dựng mạch nguyên lý, lưu đồ VHDL hoặc Verilog
- Thực hiện mạch bao gồm xác định khái niệm chức năng, đi dây nối, kết hợp phân tích lưu đồ thời gian, sắp xếp vị trí,...
- Download xuống mạch phần cứng
 - Bước 1: Thiết kế kiến trúc (Architecture design)
 - Bước 2: Thiết kế logic (Logic Design)
 - Bước 3: Thiết kế mạch (Circuit Design)
 - Bước 4: Thiết kế vật lý cho các cell đặc trưng (Mask design)
 - Bước 5: Thiết kế vật lý (Physical design)

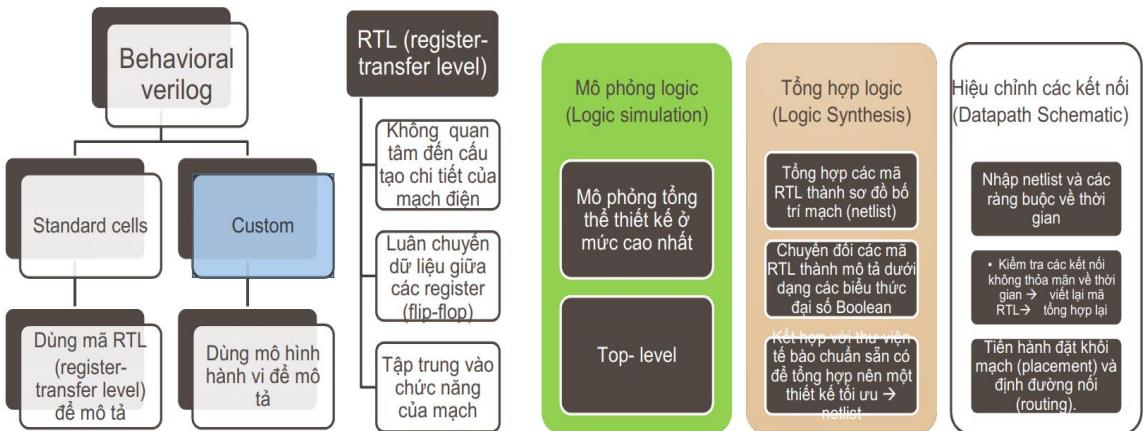
Bước 1: Thiết kế kiến trúc:



Bước 2: Thiết kế logic:



- Thiết kế mô hình hành vi



Bước 3: Thiết kế mạch



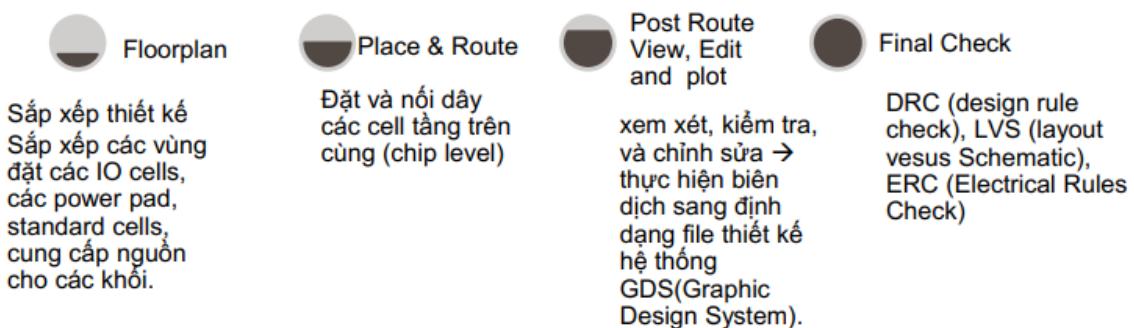
Bước 4: Thiết kế vật lý cho các cell đặc trưng

- Thiết kế ở cấp độ transistor
- Layer design
- Chuyển từ sơ đồ lý luận sang sơ đồ mạch vật lý
- Nhà sản xuất sẽ đỡ các lớp mẫu đã được quy định trên cơ sở các lớp mặt nạ (mask layers) để được các lớp vật lý biểu thị các transistor, các cổng logic và các kết nối giữa chúng.



Bước 5: Thiết kế vật lý (Physical design)

- Thực hiện ở cấp độ chip trong quy trình SoC
- Sắp xếp các mạch thiết kế trên một vùng điện tích nhất định dựa và thiết kế lý luận của chip (gate-level netlist)
- File GDS của chip được đưa xuống nhà sản xuất để làm ra thành phẩm



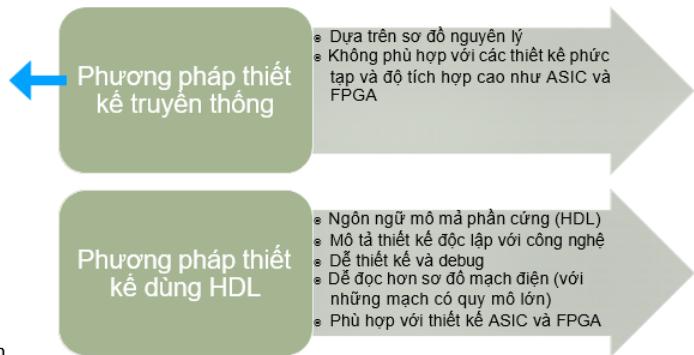
Ngôn ngữ lập trình Verilog HDL

- **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



3

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

4

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 kỳ mức nào

5

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:

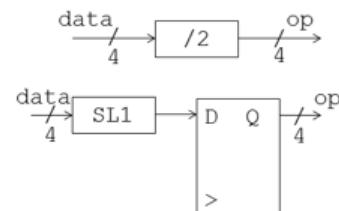
```
always @ (data)
    op <= data / 2;
```

- RTL:

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

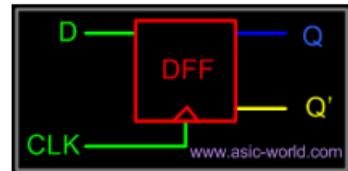
- Nối dây mức cổng

```
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]));
```

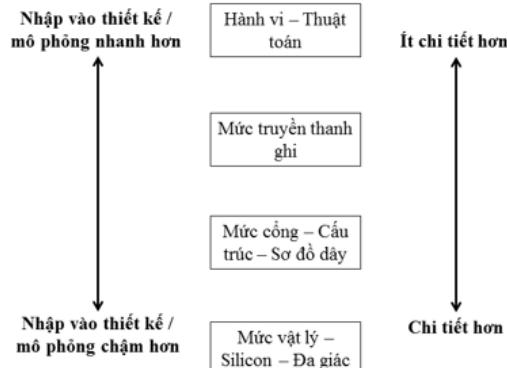


Ví dụ Verilog Code

```
// 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

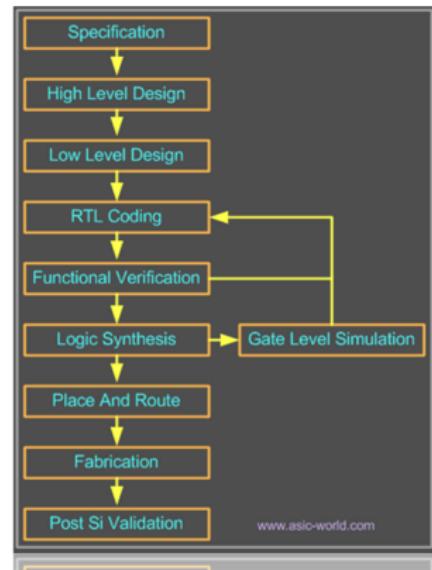


Các kiểu thiết kế

- 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 cỗng 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



11

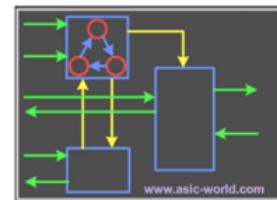
Top-down Design - High level Design

- 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

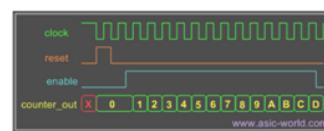
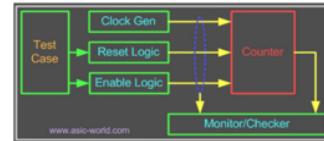
- Chuyển thiết kế mức thấp
- Dùng các cấu trúc của ngôn ngữ mô tả phần cứng HDL
- Cần kết nối các đoạn mã trước khi kiểm tra và tổng hợp

```
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
```

14

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

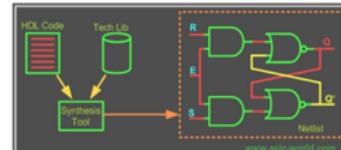


Timing Simulation

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

15

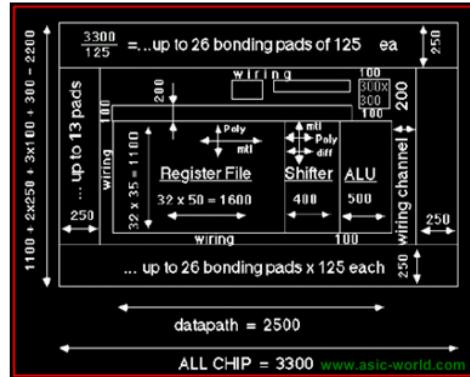
Tổng hợp - Synthesis



- 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 cồng 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 cồng
 - 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 khói và định tuyến - Place and Route

- Đặt khói 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



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

- 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 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	Giao diện (Interface)
Khai báo các tham số	
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

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
```

28

Module

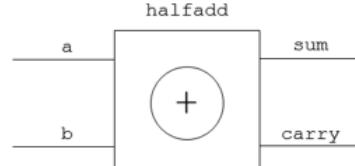
- Mô tả giao diện và hành vi
- Các module giao tiếp với nhau thông qua các cổng
 - Tên các cổng được liệt kê trong dấu ngoặc kép紧跟 sau tên module
- Các cổng có thể được xác định là đầu vào (input), đầu ra (output) hoặc hai chiều (inout)
- \wedge được dành riêng cho toán tử
- $\&$ vừa là điều kiện, vừa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



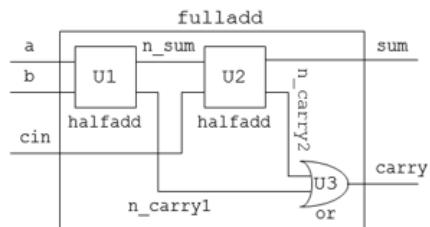
29

3

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 "dụng sẵn"



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

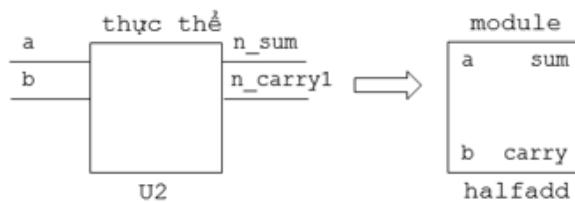
    halfadd U1 (.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
    halfadd U2 (.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));
    or     U3 (carry, n_carry2, n_carry1);
endmodule
```

30

4

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

31

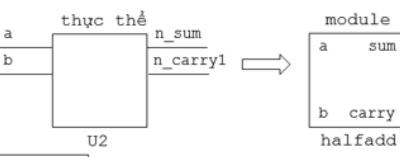
5

Kết nối phân cấp - Kết nối công theo thứ tự

- Kết nối các công theo thứ tự:

- Công thứ nhất của thực thể sẽ ứng với công thứ nhất của module
- Công thứ hai của thực thể sẽ ứng với công 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);
        input a, b;
        ...
        endmodule
    
```



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 công có tên

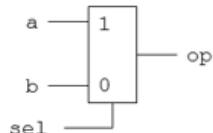
32

6

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
```

33

Danh sách sự kiện

or là một từ khóa được dùng trong 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

```
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 trắng, 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
```

37

• 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	// a Verilog Keyword
2 wire	// a Verilog Keyword
3 WIRE	// a unique name (not a keyword)
4 wire	// a unique name (not a keyword)

38

• Từ định danh - tên riêng

- dùng cho tên biến, hàm, khôi, 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
H hoặc h: hệ thập lục phân
O hoặc o: hệ bát phân
- mặc định là 32 bit
- 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

Integer	Stored as
1	0001
8'hAA	10101010
6'b10_0011	100011
'hF	0001111

Integer	Stored as
6'CA	001010
6'A	001010
16'bZ	zzzzzzzzzzzzzzz
8'bx	xxxxxxx

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 nhân 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

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 = fffffedcc
 Current Value of a = deadbeef
 Current Value of a = 21524111

45

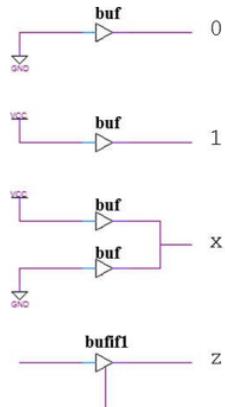
Kiểu dữ liệu

- 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": trờ kháng cao, 3 trạng thái, chưa điều khiển, chưa nối, chưa biết bộ điều khiển

Giá trị "x" khác với don't care



47

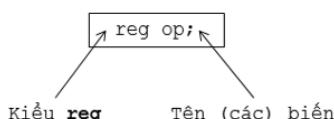
Khái niệm kiểu dữ liệu

- 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
```



48

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
```

49

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
```

50

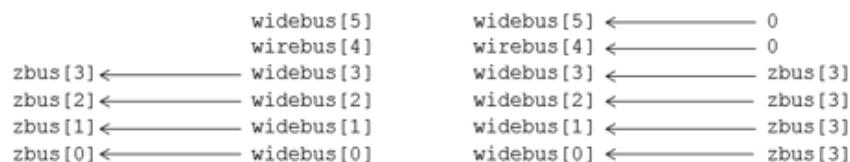
Phép gán vector và độ dài bit

- 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 = zbus



Ngoặc vuông

zbus = widebus[3:0];

51

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		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

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)
trireg	Retains last value, when driven by z (tristate).

53

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 luỹ 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
```

54

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
```

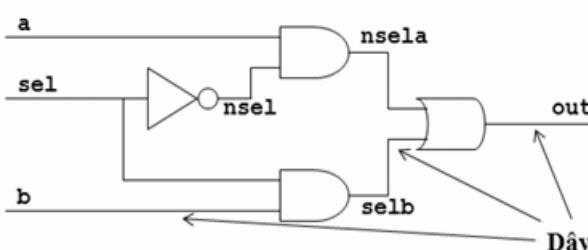
Simulator Output

```
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
```

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



```
module mux (a, b, sel, out);
  input a, b;
  output out;
  wire nsela, selb, nsel;

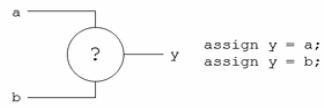
  assign nsel = ~sel;
  assign selb = sel & b;
  assign nsela = nsel & a;
  assign out = nsela | selb;

endmodule
```

56

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;

a	b	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;

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

57

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
```

58

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
```

- Thanh ghi chỉ cập nhật giá trị trong các thủ tục
- Các thủ tục chỉ cập nhật giá trị của thanh ghi
- Vết phai 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
    → assign nota = ~a;
    assign aandb = a & b;
  ...
endmodule
```

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

61

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

62

- 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 rõ các khai báo cục bộ (các công 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
```

64

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 kenh 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

65

Mảng 2 chiều

- Verilog hỗ trợ mảng thanh ghi 2 chiều

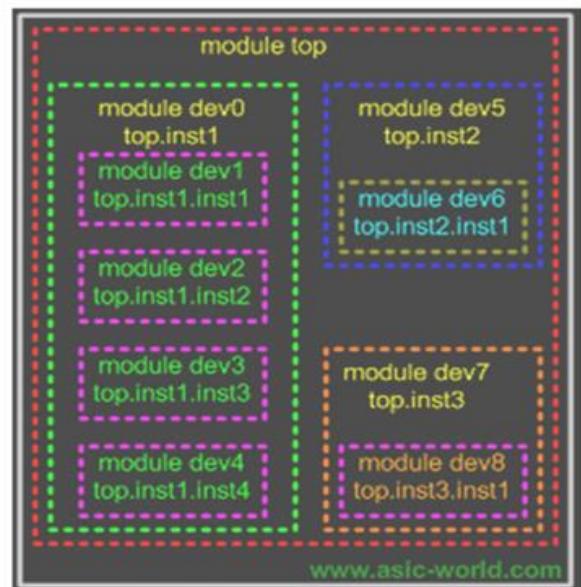
```
reg [15:0] mem [0:1023]; // Mảng hai chiều 1K x 16 bit  
integer int_array [99:0]; // Mảng số nguyên 100 phần tử
```

- 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

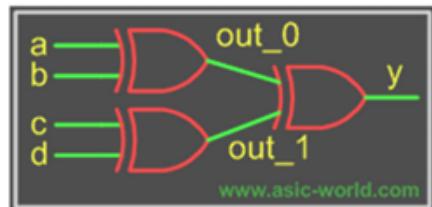


67

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 // Ouput Declaration
23 output y ;
24 // port data types
25 wire a ;
26 wire b ;
27 wire c ;
28 wire d ;
29 wire y ;
```

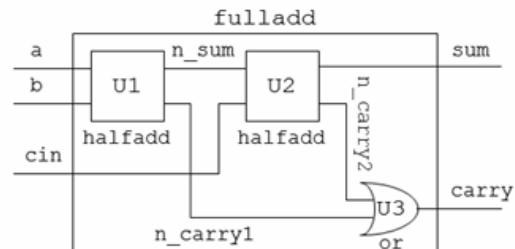
```
30 // Internal variables
31 wire out_0 ;
32 wire out_1 ;
33
34 // Code starts Here
35 xor u0 (out_0,a,b);
36
37 xor u1 (out_1,c,d);
38
39 xor u2 (y,out_0,out_1);
40
41 endmodule // End Of Module parity
```



68

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;
```

Dây nối cục bộ (wire)

```
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
```

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;
```

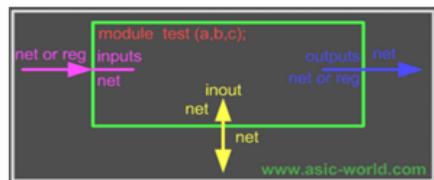
1	input	clk	; // clock input	
2	input	[15:0]	data_in	; // 16 bit data input bus
3	output	[7:0]	count	; // 8 bit counter output
4	inout		data_bi	; // Bi-Directional data bus

70

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



71

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

```
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 ;
```

73

```
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 flip-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
```

74

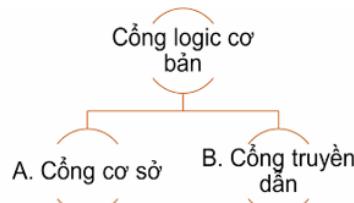
Hàm hệ thống - System Tasks

- 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

Mô hình thiết kế Verilog mức công

- Là mức thấp nhất trong 4 mô hình của Verilog
- Mạch được mô tả trên cơ sở các cỗng 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 cỗng



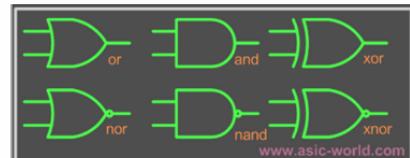
- Giá trị logic các cỗng

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

Các cỗng cơ sở

- Nhóm cỗng thực hiện phép toán - 6 cỗng: and, nand, or, nor, xor, xnor
- Khai báo
 - wire OUT, IN1, IN2;
- Sử dụng

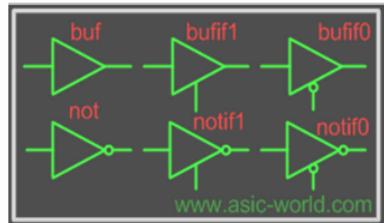
```
and a1 (OUT, IN1, IN2);  
nand nal (OUT, IN1, IN2);  
nand nal_3in (OUT, IN1, IN2, IN3);  
// cỗng 3 đầu vào  
and (OUT, IN1, IN2);  
// cỗng không cần tên khởi tạo
```



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

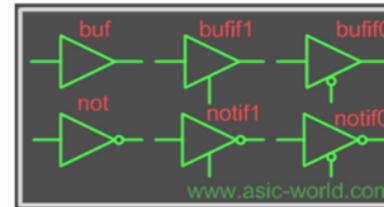
Các cổng buf/not

- Nhóm cổng **đệm/dảo** - 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);`



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.

- Nhóm cổng **đệm/dảo** - 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);`



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.

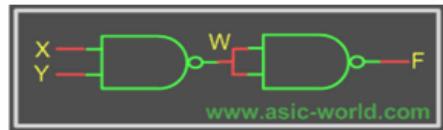
Thông tin thêm

Typical instantiation	Functional representation	Functional description
<code>bufif1 (out, in, control);</code>		Out = in if control = 1; else out = z
<code>bufif0 (out, in, control);</code>		Out = in if control = 0; else out = z
<code>notif1 (out, in, control);</code>		Out = complement of in if control = 1; else out = z
<code>notif0 (out, in, control);</code>		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	All other cases of inputs
XNOR	If the only input is at 0 state	If the only input is at 1 state	
BUF	If the only input is at 1 state	If the only input is at 0 state	
NOT	If the only input is at 1 state	If the only input is at 0 state	

Bài 1: Xây dựng công AND từ công 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

84

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 cồng trong các thiết kế phức tạp, số lượng cồng lớn

Mức cồng VS mức luồng dữ liệu

Gate level

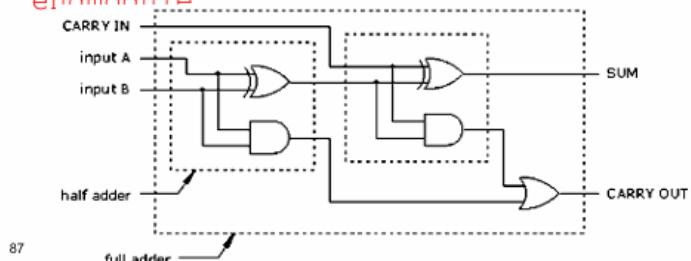
```
module fulladder
(sum,c_out,a, b, c_in);
output sum,c_out;
input a, b, c_in;
wire s1, c1, c2;

//khai báo cồng 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
```



87

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 công 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 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 (/), luỹ thừa (**), chia lấy phần dư modulus (%)
- Nếu có bất kỳ 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
```

91

```
A = 4'b0011; B = 4'b0100;          // Nếu có bất kỳ một toán hạng nào là  
// A and B are register vectors   'x' → kết quả là 'x'  
D = 6; E = 4; F=2                in1 = 4'b101x;  
// D and E are integers           in2 = 4'b1010;  
A * B // Multiply A and B. Evaluates to sum = in1 + in2; // sum will be  
4'b1100                           evaluated to the value 4'bx  
D / E // Divide D by E. Evaluates to 1. //modulus: lấy phần dư  
Truncates any fractional part.      13 % 3 // Evaluates to 1  
A + B // Add A and B. Evaluates to 16 % 4 // Evaluates to 0  
4'b0111                           -7 % 2 // Evaluates to -1, takes sign  
B - A // Subtract A from B. Evaluates to 4'b0001 of the first operand  
7 % -2 // Evaluates to +1, takes  
F = E ** F; //E to the power F, yields 16 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

```

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 <= b	a nhỏ hơn hoặc bằng b
a >= 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
10 end
11 endmodule

```

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

95

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

```
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
```

97

```
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

- $\sim x = x$
- $0 \& x = 0$
- $1 \& x = x \& x = x$
- $1|x = 1$
- $0|x = x|x = x$
- $0^x = 1^x = x^x = x$
- $0^{\sim}x = 1^{\sim}x = x^{\sim}x = x$

Operator	Description
\sim	negation
$\&$	and
$ $	inclusive or
$^$	exclusive or
$^{\sim}$ or $\sim^$	exclusive nor (equivalence)

98

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
```

99

~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
$\sim\&$	nand
	or
$\sim $	nor
\wedge	xor
$\wedge\sim$ or $\sim\wedge$	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 ("%b & %b = %b", 4'b1001, 4'b1001);
6   $display ("%b & %b = %b", 4'b1001, 4'bx111);
7   $display ("%b & %b = %b", 4'b1001, 4'bz111);
8   // Bit Wise NAND reduction
9   $display ("%b ~& %b = %b", 4'b1001, 4'b1001);
10  $display ("%b ~& %b = %b", 4'b1001, 4'bx001);
11  $display ("%b ~& %b = %b", 4'b1001, 4'bz001);
12  // Bit Wise OR reduction
13  $display ("%b | %b = %b", 4'b1001, 4'b1001);
14  $display ("%b | %b = %b", 4'b1001, 4'bx000);
15  $display ("%b | %b = %b", 4'b1001, 4'bz000);
16  // Bit Wise NOR reduction
17  $display ("%b ~| %b = %b", 4'b1001, 4'b1001);
18  $display ("%b ~| %b = %b", 4'b1001, 4'bx001);
19  $display ("%b ~| %b = %b", 4'b1001, 4'bz001);
20  // Bit Wise XOR reduction
21  $display ("%b ^ %b = %b", 4'b1001, 4'b1001);
22  $display ("%b ^ %b = %b", 4'b1001, 4'bx001);
23  $display ("%b ^ %b = %b", 4'b1001, 4'bz001);
24  // Bit Wise XNOR
25  $display ("%b ~^ %b = %b", 4'b1001, 4'b1001);
26  $display ("%b ~^ %b = %b", 4'b1001, 4'bx001);
27  $display ("%b ~^ %b = %b", 4'b1001, 4'bz001);
28  #10 $finish;
29
30
31 endmodule
  
```

Toán tử điều kiện

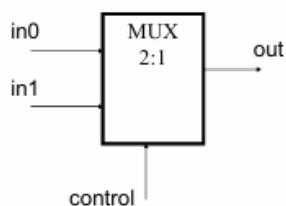
- Có 3 toán hạng
- Cú pháp

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;



101

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
```

102

- 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

- 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'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

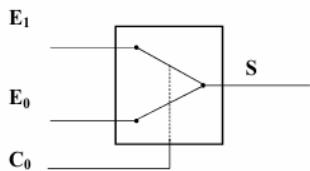
Operator	Symbols
Unary, Multiply, Divide, Modulus	!, ~, *, /, %
Add, Subtract, Shift	+, -, <<, >>
Relation, Equality	<, >, <=, >=, ==, !=, ===, !===
Reduction	&, !&, ^~, , ~
Logic	&&,
Conditional	? :

105

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

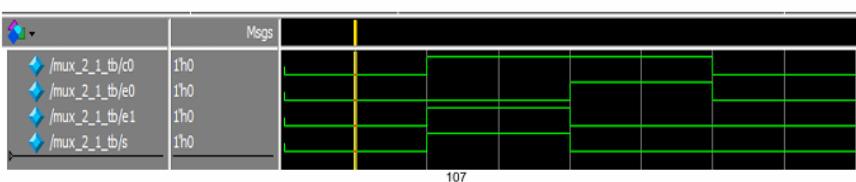
$$S = \overline{C_0}E_0 + C_0E_1$$

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
```



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

109

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
```

110

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
```

111

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 hóa 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
```

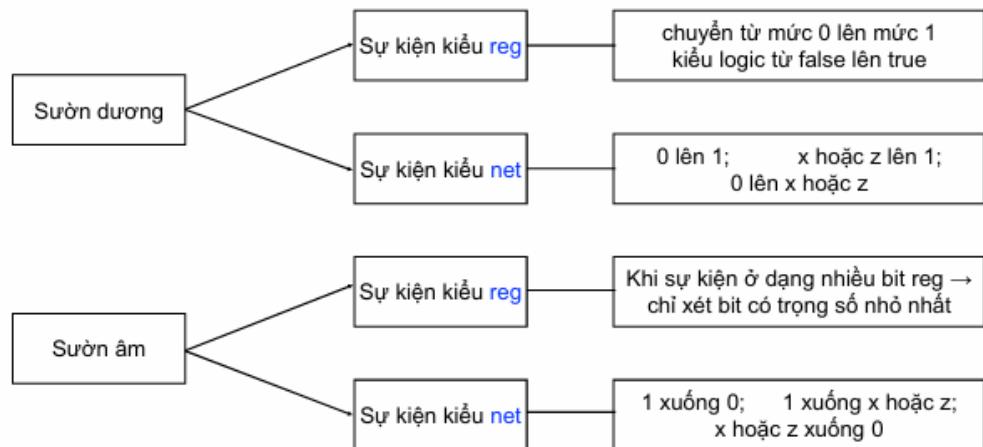
112

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)**

113

Các kiểu sự kiện - event Control



114

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
```

115

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
```

117

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
```

118

Phép gán

- là cơ chế cơ bản để đưa giá trị vào nets và register
- Phép gán gồm
 - Về trái: Chỉ định biến gán cho về phải
 - Về 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 → về trái kiểu net
- Về trái sẽ cập nhật giá trị khi có bất kỳ sự thay đổi của biểu thức về 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 → về trái là kiểu reg hoặc phần tử nhớ
- Về 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 lal;
reg a;
wire b,c,d;
always @(*)
begin
a = b | (c & d);
end
endmodule
```

Phép gán liên tục - gán trong khối

- Gán trong khói, 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

122

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;
//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
```

- 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 $t=0$
 - $reg_a[2]=1'b1$ 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)

123

Non-blocking VS Blocking

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 biến thức bên phải trước khi gán cho về trái
- Dùng trong khối **always** để thực hiện **mạch dãy**

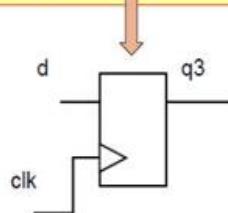
Blocking

- Toán tử $=$
- Các phát biểu thực hiện tuân 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 biến thức bên phải ngay sau khi phát biểu trước đó hoàn tất
- Dùng trong khối **always** để thực hiện **mạch tổ hợp**

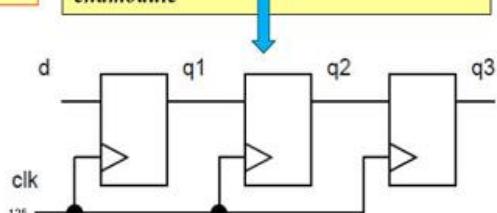
124

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 thi nên dùng gán trong khối

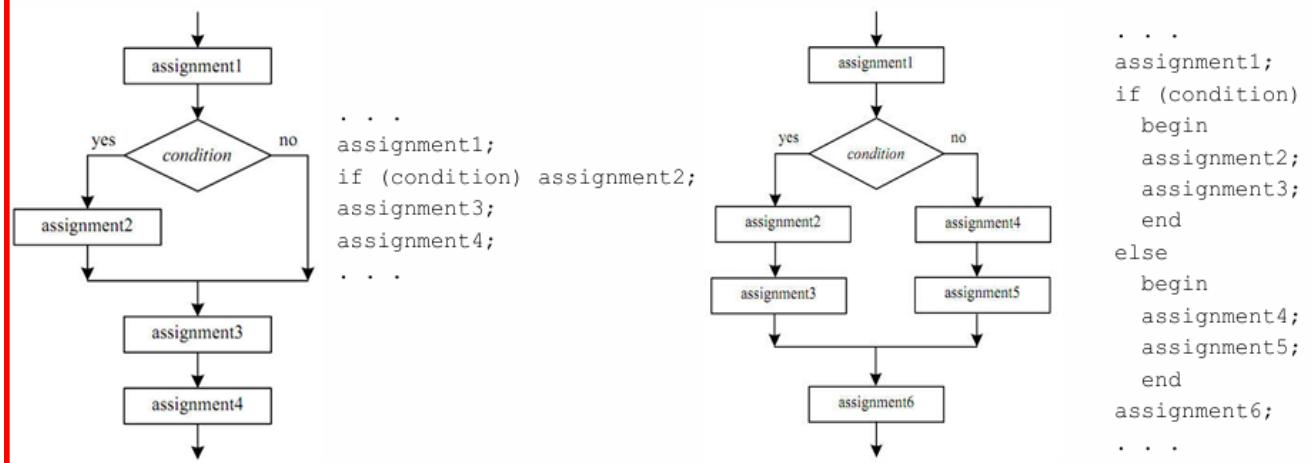
always @ (posedge clk) A=B; always @ (posedge clk) B=A;	always @ (posedge clk) begin A<=B; B<=A; end
--	--

→ 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**



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
```

```

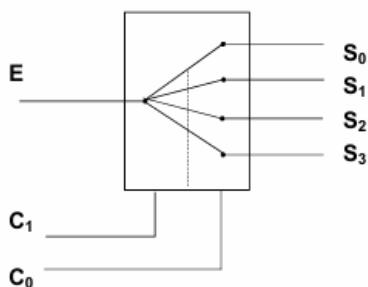
module nested_if();
reg [3:0] counter;
reg clk,reset,enable, up_en, down_en;

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



c ₁	c ₀	s ₀	s ₁	s ₂	s ₃
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
```

131

Ví dụ bộ DEMUX 1:4

- S hiến 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
```

132

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.

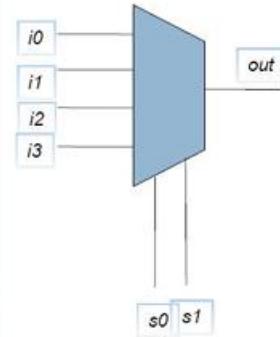
133

Câu điều kiện case

- Ví dụ bộ MUX 4:1

```
module mux4to1 (out, i0, i1, i2, i3, s0, s1);
    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
```



134

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
```

135

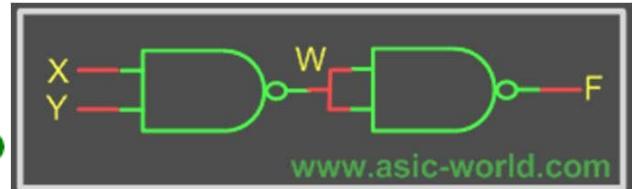
Vòng lặp - loop

- 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 trẽ

Bài tập

Bài 1: Xây dựng cổng AND từ cổng NAND

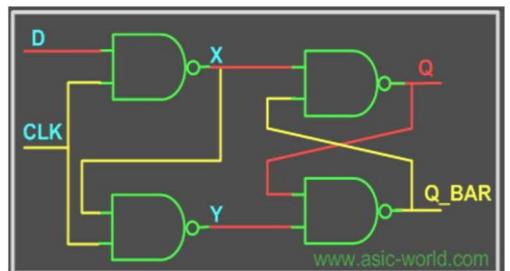
```
1 // Structural model of AND gate from two NANDS
2 module and_from_nand();
3
4 reg X, Y;
5 wire F, W;
6 // Two instantiations of the module NAND
7 nand U1(W,X, Y);
8 nand U2(F, W, W);
9
10 // Testbench Code
11 initial begin
12 $monitor ("X = %b Y = %b F = %b", X, Y, F);
13 X = 0;
14 Y = 0;
15 #1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

Bài 2: Xây dựng cổng D-FF từ cổng NAND

```
1 module dff_from_nand();
2 wire Q,Q_BAR;
3 reg D,CLK;
4
5 nand U1 (X,D,CLK) ;
6 nand U2 (Y,X,CLK) ;
7 nand U3 (Q,Q_BAR,X) ;
8 nand U4 (Q_BAR,Q,Y) ;
9
10 // Testbench of above code
11 initial begin
12 $monitor("CLK = %b D = %b Q = %b Q_BAR = %b",CLK, D, Q, Q_BAR);
13 CLK = 0;
14 D = 0;
15 #3 D = 1;
16 #3 D = 0;
17 #3 $finish;
18 end
19
20 always #2 CLK = ~CLK;
21
22 endmodule
```

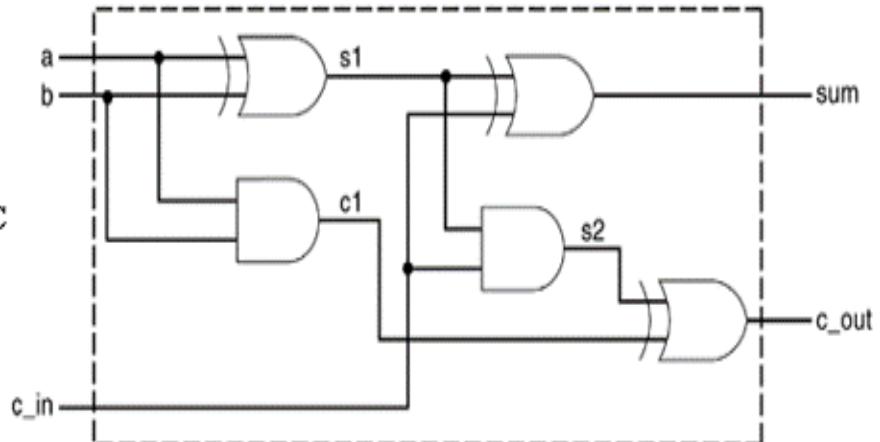


Bài 3: Bộ cộng đầy đủ 1 bit và 4 bit

```
module fulladder (sum,c_out,a, b, c_in);
//khai báo cổng vào ra

output sum,c_out;
input a, b, c_in;
wire s1, c1, c2;

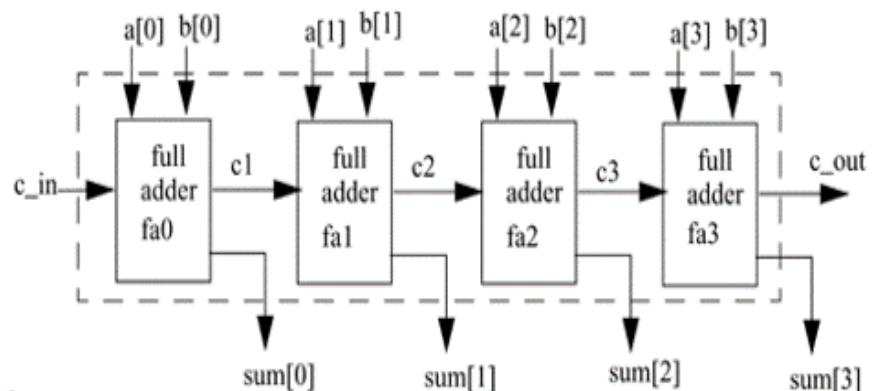
//khai báo cổng logic
xor (s1,a,b);
and (c1,a,b);
xor (sum,s1,c_in);
and (s2,s1,c_in);
xor (c_out, s2,c1);
endmodule
```



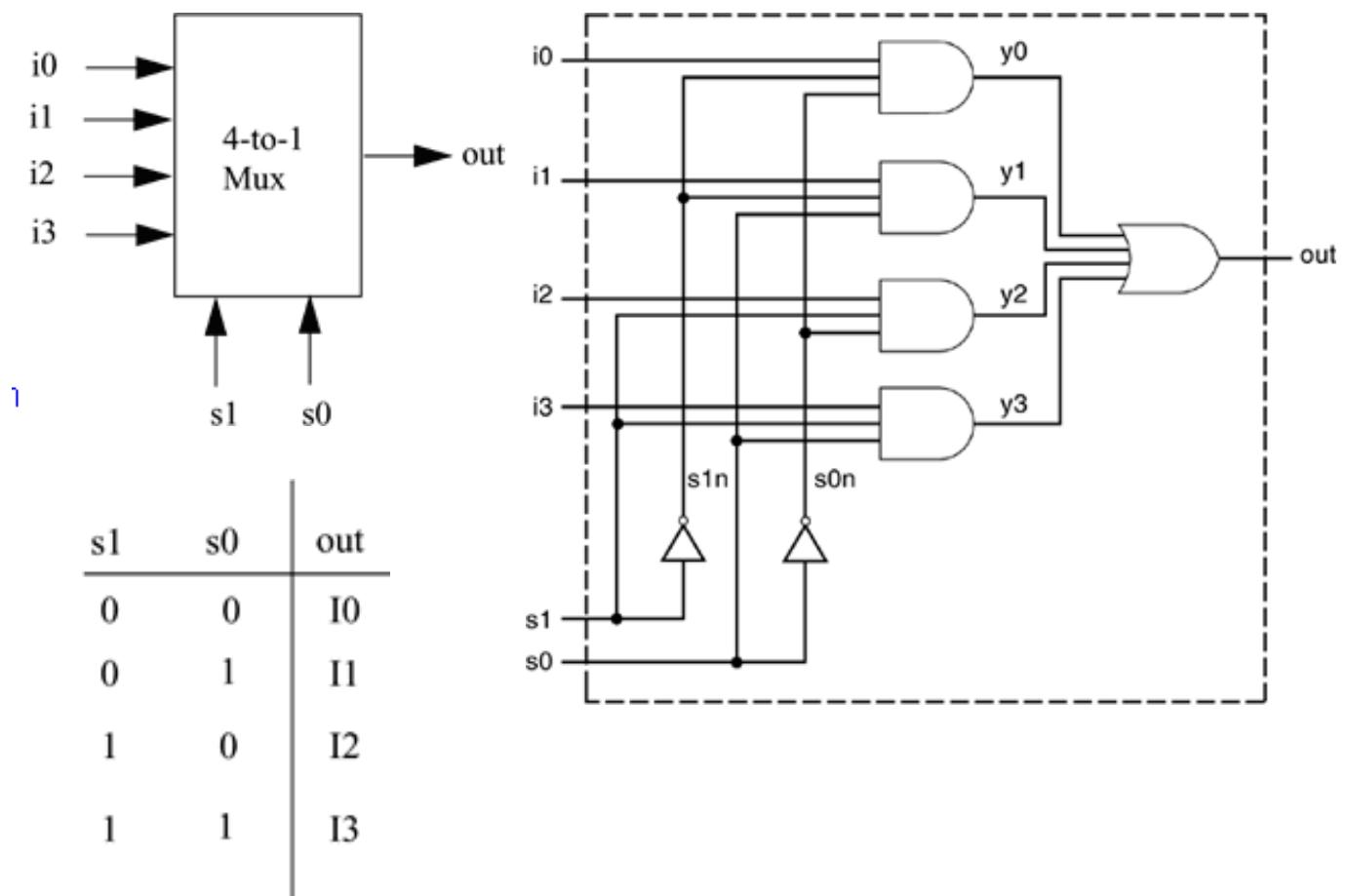
```
module fulladder4 (sum, c_out, a, b, c_in);
//khai báo I/O port
```

```
output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;
wire a1, c2, c3;

//ghép 4 bộ cộng 1 bit
fulladder fa0(sum[0], c1, a[0], b[0],c_in);
fulladder fa1(sum[1], c2, a[1], b[1],c1);
fulladder fa2(sum[2], c3, a[2], b[2],c2);
fulladder fa3(sum[3], c_out, a[3], b[3],c3);
endmodule
```



Bài 4: Bộ ghép kênh 4:1



```

module mux4_to_1 (out, i0, i1, i2, i3, s1, s0)

output out;
input i0, i1, i2, i3;
input s1,s0;
wire s1n,s0n;
wire y0,y1, y2, y3;

not (s1n,s1);
not (s0n;s0);
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);
endmodule

```

```

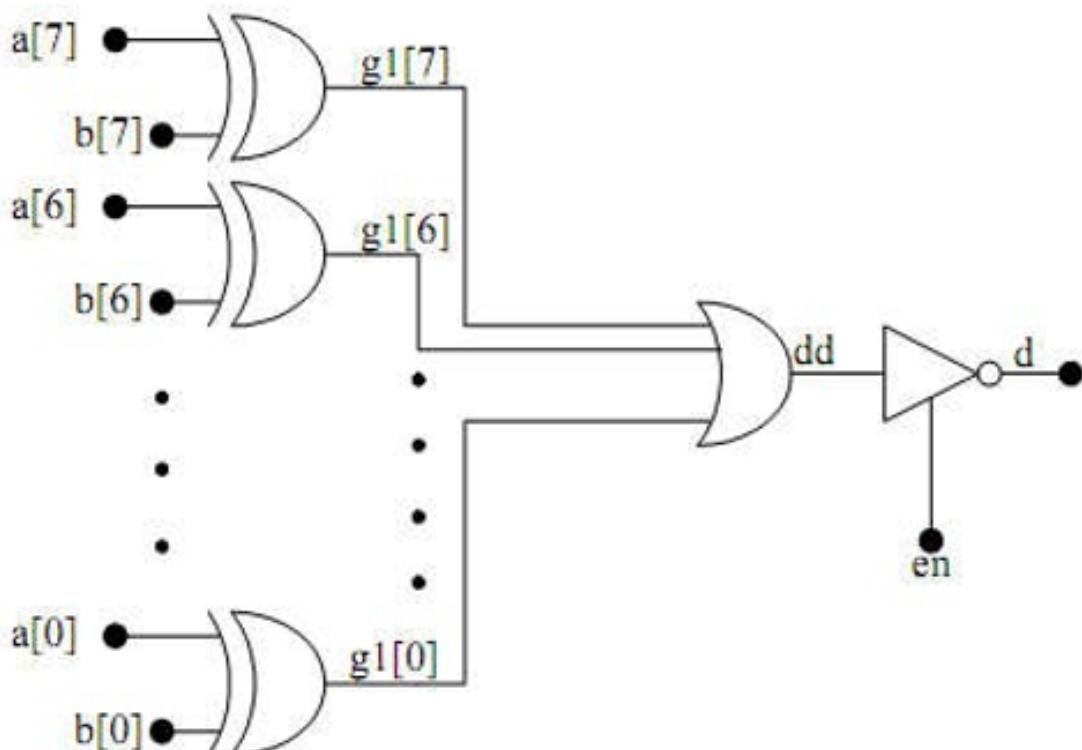
1 module mux_from_gates ();
2 reg c0,c1,c2,c3,A,B;
3 wire Y;
4 //Invert the sel signals
5 not (a_inv, A);
6 not (b_inv, B);
7 // 3-input AND gate
8 and (y0,c0,a_inv,b_inv);
9 and (y1,c1,a_inv,B);
10 and (y2,c2,A,b_inv);
11 and (y3,c3,A,B);
12 // 4-input OR gate
13 or (Y, y0,y1,y2,y3);
14
15 // Testbench Code goes here
16 initial begin
17 $monitor (
18 "c0 = %b c1 = %b c2 = %b c3 = %b A = %b B = %b Y = %b",
19 c0, c1, c2, c3, A, B, Y);

```

c0 = 0	c1 = 0	c2 = 0	c3 = 0	A = 0	B = 0	Y = 0	20	c0 = 0;
c0 = 1	c1 = 0	c2 = 0	c3 = 0	A = 1	B = 0	Y = 0	21	c1 = 0;
c0 = 0	c1 = 1	c2 = 0	c3 = 0	A = 1	B = 0	Y = 0	22	c2 = 0;
c0 = 1	c1 = 1	c2 = 1	c3 = 0	A = 1	B = 1	Y = 0	23	c3 = 0;
c0 = 0	c1 = 0	c2 = 1	c3 = 1	A = 1	B = 1	Y = 1	24	A = 0;
c0 = 1	c1 = 0	c2 = 1	c3 = 1	A = 1	B = 1	Y = 1	25	B = 0;
c0 = 0	c1 = 1	c2 = 0	c3 = 1	A = 1	B = 1	Y = 1	26	#1 A = 1;
c0 = 0	c1 = 0	c2 = 0	c3 = 0	A = 0	B = 1	Y = 0	27	#2 B = 1;
c0 = 1	c1 = 0	c2 = 1	c3 = 0	A = 0	B = 1	Y = 0	28	#4 A = 0;
c0 = 0	c1 = 1	c2 = 1	c3 = 0	A = 0	B = 1	Y = 1	29	#8 \$finish;
c0 = 0	c1 = 0	c2 = 0	c3 = 1	A = 0	B = 0	Y = 0	30	end
c0 = 1	c1 = 0	c2 = 0	c3 = 1	A = 0	B = 0	Y = 0	31	always #1 c0 = ~c0;
c0 = 0	c1 = 1	c2 = 0	c3 = 1	A = 0	B = 0	Y = 0	32	always #2 c1 = ~c1;
c0 = 0	c1 = 0	c2 = 1	c3 = 1	A = 0	B = 0	Y = 1	33	always #3 c2 = ~c2;
c0 = 1	c1 = 0	c2 = 1	c3 = 1	A = 0	B = 0	Y = 1	34	always #4 c3 = ~c3;
c0 = 0	c1 = 1	c2 = 0	c3 = 0	A = 0	B = 1	Y = 0	35	36
c0 = 0	c1 = 0	c2 = 0	c3 = 0	A = 0	B = 1	Y = 0	37	endmodule

Bài 5: Bộ so sánh 8 bit

- Viết module Verilog và testbench mô phỏng mô tả bộ so sánh 2 giá trị byte a và b theo mô hình mức cổng
 - Giá trị đầu ra d sẽ =1 nếu $a=b$, =0 trong các trường hợp còn lại.
 - Đầu ra được kích hoạt nếu $en=1$; nếu $en=0$ thì $d=z$.

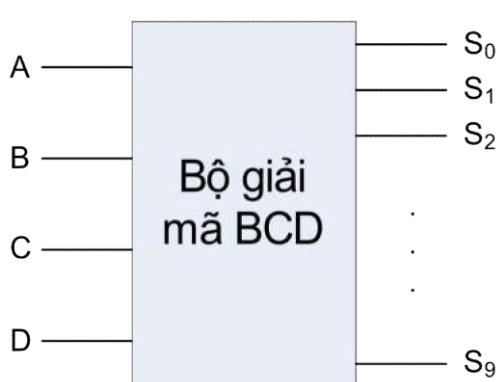
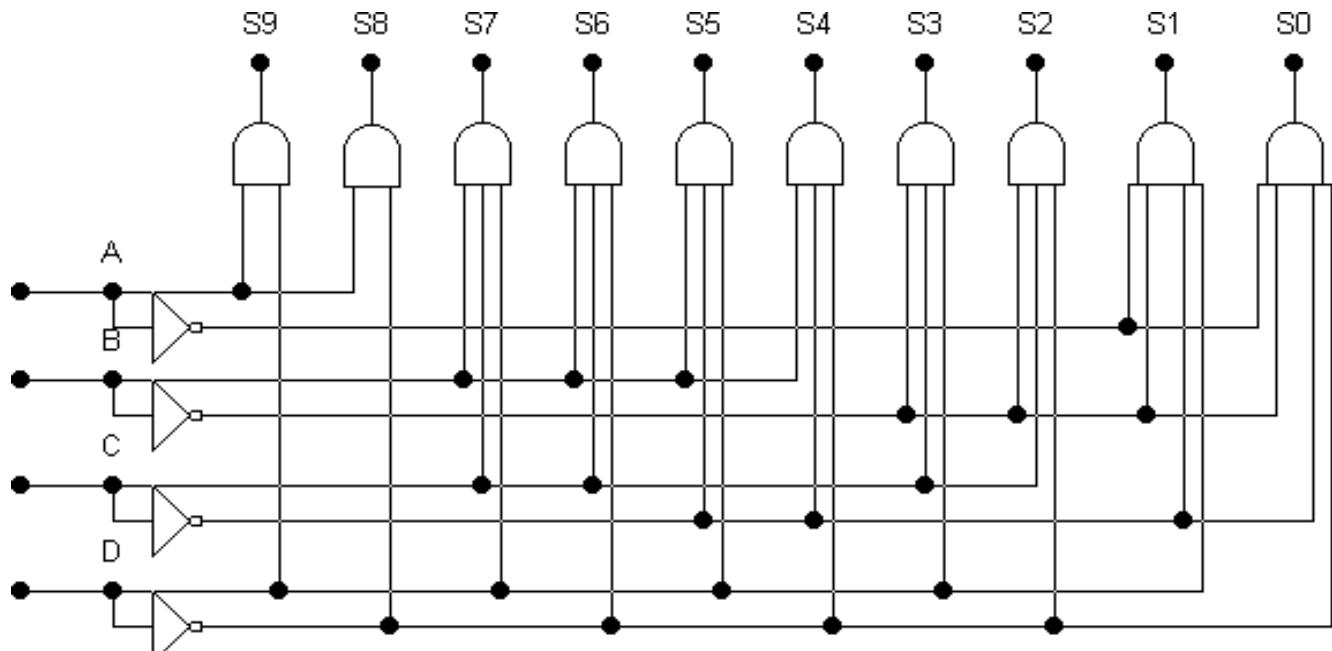


```

module comp(d,a,b,en); initial
  input en;
  input[7:0]a,b;
  output d;
  wire [7:0]c;
  wire dd;
  xor g1[7:0](c,b,a);
  or(dd,c);
  notif1(d,dd,en);
Endmodule
//=====
module comp_tb;
reg[7:0]a,b;
reg en;
comp gg(d,a,b,en);
initial $monitor($time," en = %b , a = %b , b = %b ,d = %b ",en,a,b,d);
initial #30 $stop;
endmodule

```

Bài 6: Bộ mã hoá BCD



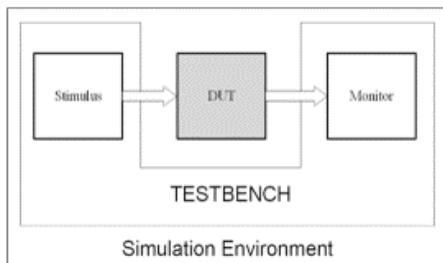
Chữ số thập phân	Tù mã nhị phân
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Code file và Testbench file

- Giống nhau
 - Có cùng định dạng .v (verilog) hoặc .vhdl (VHDL)
- Khác nhau

Code file

- Mô tả khối DUT (Design Under Test)
tức là khối chức năng



Testbench file

- Tạo kết nối và các tín hiệu cung cấp cho khối DUT
- Gọi thiết kế DUT
- Quan sát các thành phần tín hiệu vào/ ra của khối DUT

Mô tả DUT

```
1 module mux_2x1 ( input a, b, sel,
2                     output out);
3     wire sel_n;
4     wire out_0;
5
6     not (sel_n, sel);
7
8     and (out_0, a, sel);
9     and (out_1, b, sel_n);
10
11    or (out, out_0, out_1);
12 endmodule
```

- Bắt đầu và kết thúc bởi 2 từ khóa **module** (dòng 1) và **endmodule** (dòng 12)
- Tên module tùy chọn
- Các biến **input** và **output** có thể khai báo bên trong tên module như hình hoặc định nghĩa ở bên ngoài
- Khai báo các kết nối (dòng 3,4)
- Thiết lập khối DUT (dòng 6-11)

Mô tả testbench file

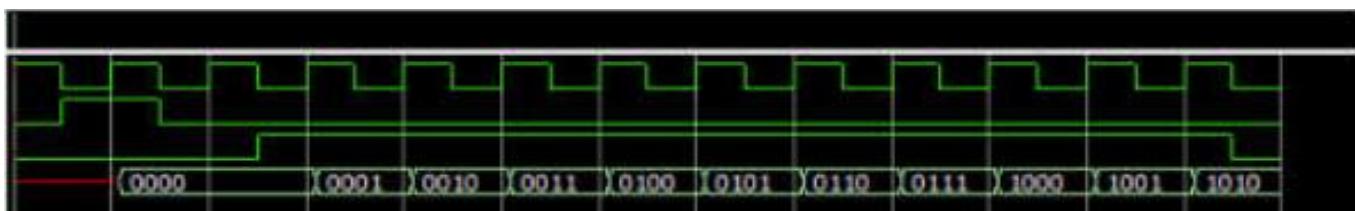
```
1 module tb;
2     reg a, b, sel;
3     wire out;
4     integer i;
5
6     mux_2x1 u0 ( .a(a), .b(b), .sel(sel), .out(out));
7
8     initial begin
9         {a, b, sel} <= 0;
10
11        $monitor ("T=%0t a=%0b b=%0b sel=%0b out=%0b", $time, a, b, sel, out);
12
13        for (int i = 0; i < 10; i = i+1) begin
14            #1 a <= $random;
15            b <= $random;
16            sel <= $random;
17        end
18    end
19 endmodule
```

- Bắt đầu và kết thúc bởi 2 từ khóa **module** (dòng 1) và **endmodule** (dòng 19)
- Tên module tùy chọn, thường có chữ tb để phân biệt
- Khai báo các giao tiếp, tín hiệu kết nối với DUT (dòng 2-4)
- Gọi khối DUT để chạy
- Thiết lập các tín hiệu hoặc xây dựng dạng giám sát tín hiệu kết nối với DUT (dòng 8-18)

Mạch đếm counter

```
-----  
// Design Name : first_counter  
// File Name : first_counter.v  
// Function : This is a 4 bit up-counter with  
// Synchronous active high reset and  
// with active high enable signal  
-----  
module first_counter (  
    clock , // Clock input of the design  
    reset , // active high, synchronous Reset input  
    enable , // Active high enable signal for counter  
    counter_out // 4 bit vector output of the counter  
) ; // End of port list  
-----Input Ports-----  
input clock ;  
input reset ;  
input enable ;  
-----Output Ports-----  
output [3:0] counter_out ;  
-----Input ports Data Type-----  
// By rule all the input ports should be wires  
wire clock ;  
wire reset ;  
wire enable ;  
  
----- Output Ports Data Type -----  
// Output port can be a storage element (reg) or a wire  
reg [3:0] counter_out ;  
-----Code Starts Here-----  
// Since this counter is a positive edge triggered one,  
// We trigger the below block with respect to positive  
// edge of the clock.  
always @ (posedge clock)  
begin : COUNTER // Block Name  
// At every rising edge of clock we check if reset is active  
// If active, we load the counter output with 4'b0000  
if (reset == 1'b1) begin  
    counter_out <= #1 4'b0000;  
end  
// If enable is active, then we increment the counter  
else if (enable == 1'b1) begin  
    counter_out <= #1 counter_out + 1;  
end  
end // End of Block COUNTER  
endmodule // End of Module counter
```

```
`include "first_counter.v"  
module first_counter_tb();  
// Declare inputs as regs and outputs as wires reg  
clock, reset, enable;  
wire [3:0] counter_out;  
  
// Initialize all variables initial  
begin  
    $display ("time\t clk reset enable counter");  
    $monitor ("%g\t %b\t %b\t %b\t %b",  
             $time, clock, reset, enable, counter_out);  
  
    clock = 1; // initial value of clock  
    reset = 0; // initial value of reset  
    enable = 0; // initial value of enable  
    #5 reset = 1; // Assert the reset  
    #10 reset = 0; // De-assert the reset  
    #10 enable = 1; // Assert enable #100  
    enable = 0; // De-assert enable #5  
    $finish; // Terminate simulation  
end  
  
// Clock generator always  
begin  
    #5 clock = ~clock;  
    // Toggle clock every 5 ticks end  
  
// Connect DUT to test bench  
first_counter U_counter ( clock,  
    reset,  
    enable,  
    counter_out  
);  
endmodule
```



Testbench

- Tesbench là chương trình để kiểm tra hoạt động thiết kế.
- Viết Testbech phức tạp tương tự RTL code của chương trình cần kiểm tra.
- Hiện nay các ASIC ngày càng phức tạp do đó việc kiểm tra hoạt động của nó cũng phức tạp.
- Trong thiết kế ASIC, thông thường 70% thời gian cần dùng cho công việc kiểm tra ở các công đoạn (verification/validation/Testing)
- Để viết Testbench, điều quan trọng cần có thông số thiết kế và hiểu rõ các thông số đó. Sau đó đưa ra Test plan và Test case cụ thể.

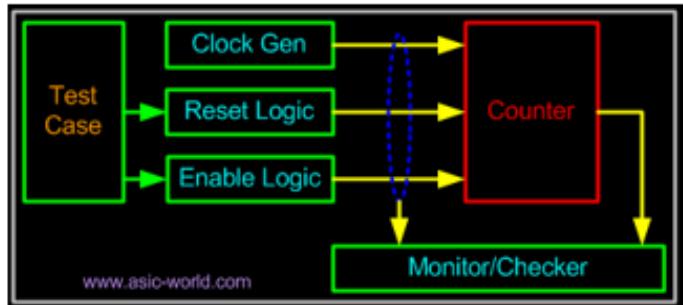
Ví dụ: Bộ đếm lên 4 bit

- Xét bộ đếm lên 4 bit. Bộ đếm sẽ tăng khi enable=1 và reset về 0 khi tín hiệu reset=1. tín hiệu reset được đồng bộ với xung clock.
- Code như sau

```
module counter (clk, reset, enable, count);
    input clk, reset, enable;
    output [3:0] count;
    reg [3:0] count;
    always @ (posedge clk)
        if (reset == 1'b1)
            begin
                count <= 0;
            end
        else if ( enable == 1'b1)
            begin
                count <= count + 1;
            end
    endmodule
```

Test Plan/ Test Case

- **Test Plan**
 - Môi trường testbench như hình bên.
 - DUT (Design-under-Test) được thực hiện trong testbench.
 - Testbench gồm khối tạo xung clock, khối tạo xung reset, khối tạo tín hiệu logic enable và mạch logic so sánh. Mạch so sánh này sẽ tính giá trị mong muốn của bộ đếm và so sánh với giá trị output của bộ đếm.
- **Test Cases**
 - Reset Test : đặt reset = 1 trong vài nhịp clock sau đó đưa reset= 0 xem bộ đếm có về 0 không
 - Enable Test: đặt enable=1/0 sau khi đặt reset.
 - Thay đổi ngẫu nhiên enable và reset..



Writing a TestBench

- Bước 1: xây dựng một tập mẫu giả định gồm các input (kiểu reg) và các output (kiểu wire) của DUT. Sau đó thực hiện DUT theo code sau. Chú ý testbench không có port list.
- Bước 2: thêm bộ tạo xung clock. Trước khi đưa bộ tạo xung clock cần đặt cho tất cả các input các giá trị xác định như code sau.
 - Initial: chỉ được thực hiện 1 lần duy nhất, simulator sẽ thiết lập các tín hiệu clk, reset, enable về 0 (disable).
 - Có nhiều cách tạo xung clock với tần số khác nhau
 - Kiểm tra xem testbench có tạo xung clock chính xác không.

```
module counter_tb;
    reg clk, reset, enable;
    wire [3:0] count;
    counter U0(.clk (clk),
               .reset (reset),
               .enable (enable),
               .count (count)
               );
// tạo xung clock
initial
begin
    clk = 0;
    reset = 0;
    enable = 0;
end
always
#5 clk = !clk;
// phần tiếp theo của testbench (cnt.)
endmodule
```

Test Bench continues..

- **\$dumpfile:** chỉ tên file sẽ lưu dạng sóng mô phỏng
- **\$dumpvars:** hướng trình biên dịch Verilog bắt đầu trùt toàn bộ tín hiệu vào file counter.vcd.
- **\$display:** in ra màn hình text hoặc các giá trị.
- \t: chèn tab.
- **.\$monitor:** theo dõi sự thay đổi của các biến (clk, reset, enable, count). Khi có bất kỳ sự thay đổi nào của các biến nó sẽ in các giá trị đó.
- **\$finish :** dùng để dừng mô phỏng sau 100 đơn vị time. (note: initial và always block bắt đầu thực thi tại time 0).

```
// phần tiếp theo của testbench
initial
begin
    $dumpfile("counter.vcd");
    $dumpvars;
end
initial
begin
    $display("\t\ttime,\tclk,\treset,
             \tenable,\tcount");
    $monitor("$d,\t$b,\t$b,\t$b,\t$b,\t$d",
            $time, clk,reset,enable,count);
end
initial #100
$finish;
//phần tiếp theo của testbench
```

Tín hiệu reset

- Phần trên đã minh họa cơ bản cách thực hiện testbench. Tiếp theo có thể thêm tín hiệu reset.
- Nhìn vào testcase, ta cần thêm một ràng buộc để có thể kích hoạt tín hiệu reset tại bất kỳ thời điểm mờ phỏng nào.
- Trong Verilog có các “sự kiện” (event), các event này có thể được kích hoạt và được giám sát xem có xảy ra không.
- Viết code theo event, tín hiệu reset sẽ đợi khi sự kiện được kích hoạt “reset_trigger”: khi sự kiện xảy ra, đặt reset =1 tại sườn âm của xung clock và sau đó lại cho reset =0 ở sườn âm clock tiếp theo. sau khi đặt reset=0, kích hoạt tín hiệu reset tại một sự kiện khác “reset_done_trigger”. Sự kiện này sẽ được dùng ở chỗ khác trong testbench để đồng bộ.

```
event reset_trigger;
event reset_done_trigger;
initial begin
    forever begin
        @ (reset_trigger);
        @ (negedge clk);
        reset = 1;
        @ (negedge clk);
        reset = 0;
    -> reset_done_trigger;
    end
end
```

Thêm các Test case

- Thêm một số tổ hợp test, xét 3 trường hợp
 - Reset Test: ban đầu đặt reset=0, sau đó đặt reset =1 trong vài chung kỲ clock tiếp. Quan sát xem bộ đếm có reset về 0 không.
 - Enable Test: đặt enable=1/0 sau khi đưa reset.
 - Thay đổi ngẫu nhiên Enable và freset.
- Chú ý: có rất nhiều cách để viết test case tùy thuộc vào sự sáng tạo của người thiết kế. Trong ví dụ này xét một số trường hợp đơn giản.

- **Test Case 3 – thay đổi reset/enable ngẫu nhiên.**
- Chú ý: 3 test case này không được cùng tồn tại trong 1 file vì sẽ dẫn đến hiện tượng chạy đua do 3 khối initial cùng tạo tín hiệu reset và enable.
- Thông thường , khi code các testbench, các test case được code riêng biệt và được đưa vào testbench với chỉ dẫn `include.

```

initial
begin : TEST_CASE #10
-> reset_trigger;
@ (reset_done_trigger);
fork
  repeat (10) begin
    @ (negedge clk);
    enable = $random;
  end
  repeat (10) begin
    @ (negedge clk);
    reset = $random;
  end
join
end

```

Test Case

- **Test Case 1 – thay đổi tín hiệu reset**

```
// -----
```

```
initial
```

```
begin: TEST_CASE
```

```
#10 -> reset_trigger;
```

```
end
```

```
//-----
```

- Trong test case 1 chỉ kích hoạt sự kiện reset_trigger sau 10 đơn vị time mô phỏng.
- Trong test case 2: kích hoạt tín hiệu reset, đợi đến khi hoàn tất reset đặt tín hiệu enable =1.

- **Test Case 2: thay đổi enable sau khi đặt reset.**

```
//-----
```

```
initial
```

```
begin: TEST_CASE
```

```
#10 -> reset_trigger; @
```

```
(reset_done_trigger); @
```

```
(negedge clk);
```

```
enable = 1;
```

```
repeat(10)begin @
```

```
(negedge clk); end
```

```
enable = 0;
```

```
end
```

```
//-----
```

Bài 2: Xây dựng D-FF từ cổng NAND bằng cách viết chương trình với ngôn ngữ verilog

```
module D_FF_NAND(input D, input Clk, input Reset, output reg Q);
```

```
wire nD, nClk, nReset;
```

```
wire S, R;
```

```
// Tạo tín hiệu nghịch đảo của D, Clk và Reset
```

```
assign nD = ~D;
```

```
assign nClk = ~Clk;
```

```
assign nReset = ~Reset;
```

```
// Tạo tín hiệu Set (S) và Reset (R)
```

```
assign S = nD & Clk;
```

```

assign R = D & Clk;
always @(posedge nClk or posedge nReset)
begin
if (nReset)
    Q <= 0;
else
    Q <= S | (Q & R);
end
endmodule

```

Giải thích code:

Trong mô-đun D_FF_NAND, chúng ta sử dụng cổng NAND để xây dựng D-FF. Tín hiệu D là tín hiệu dữ liệu đầu vào, Clk là tín hiệu xung clock, Reset là tín hiệu đặt lại, và Q là tín hiệu đầu ra. Trong mô-đun, chúng ta sử dụng các tín hiệu nghịch đảo nD, nClk và nReset bằng cách sử dụng toán tử \sim để tạo tín hiệu nghịch đảo của các tín hiệu đầu vào. Chúng ta cũng tạo tín hiệu S và R bằng cách kết hợp các tín hiệu nD và Clk hoặc D và Clk bằng cổng AND để tạo tín hiệu Set (S) và Reset (R). Trong khối always, chúng ta sử dụng xung clock nClk và tín hiệu đặt lại nReset để định nghĩa hoạt động của D-FF. Nếu tín hiệu nReset được kích hoạt (cao), đầu ra Q được đặt về 0. Trong trường hợp ngược lại, chúng ta sử dụng các tín hiệu S, Q và R để cập nhật giá trị của Q.

Bài 6: Xây dựng bộ chuyển mã BCD

```

module BCD_converter(input [3:0] binary, output [3:0] BCD);
always @(binary)
begin
case(binary)
  4'b0000: BCD = 4'b0000;
  4'b0001: BCD = 4'b0001;
  4'b0010: BCD = 4'b0010;

```

```

4'b0011: BCD = 4'b0011;
4'b0100: BCD = 4'b0100;
4'b0101: BCD = 4'b0101;
4'b0110: BCD = 4'b0110;
4'b0111: BCD = 4'b0111;
4'b1000: BCD = 4'b1000;
4'b1001: BCD = 4'b1001;
default: BCD = 4'b0000;

```

endcase

end

endmodule

Giải thích code: Trong mô-đun BCD_converter, chúng ta có một đầu vào binary với 4 bit và một đầu ra BCD cũng với 4 bit. Trong khối always, chúng ta sử dụng trạng thái của tín hiệu binary để xác định giá trị tương ứng trong mã BCD. Chúng ta sử dụng câu lệnh case để ánh xạ từng giá trị của binary sang giá trị tương ứng của BCD. Các trường hợp (cases) từ 4'b0000 đến 4'b1001 biểu thị giá trị từ 0 đến 9 trong mã BCD. Trường hợp mặc định (default) được sử dụng để gán giá trị mặc định là 0 cho BCD nếu giá trị binary không khớp với bất kỳ trường hợp nào.

```

module BCD_converter(input [3:0] binary, output reg [3:0] BCD);
    reg [3:0] count;
    always @(posedge binary)
        begin
            if (count == 4'b1001) // Nếu đang ở giá trị 9, chuyển sang 0
                count <= 4'b0000;
            else

```

```

count <= count + 1; // Tăng count lên 1
end

always @(posedge binary)
begin
  case(count)
    4'b0000: BCD <= 4'b0000;
    4'b0001: BCD <= 4'b0001;
    4'b0010: BCD <= 4'b0010;
    4'b0011: BCD <= 4'b0011;
    4'b0100: BCD <= 4'b0100;
    4'b0101: BCD <= 4'b0101;
    4'b0110: BCD <= 4'b0110;
    4'b0111: BCD <= 4'b0111;
    4'b1000: BCD <= 4'b1000;
    4'b1001: BCD <= 4'b1001;
    default: BCD <= 4'b0000;
  endcase
end

endmodule

```