

Kiến trúc máy tính & hợp ngữ

00 – Giới thiệu môn học

Thông tin GV

- * ThS. Vũ Minh Trí
- * Email: vmtri@fit.hcmus.edu.vn
 - * Subject khi gửi mail bắt đầu: *[KHMT_Tên Lớp]....*
 - * Nên bắt đầu gửi câu hỏi trên Moodle trước khi gửi email (trừ khi được yêu cầu)
- * Bộ môn MMT&VT, Khoa CNTT, Trường ĐHKHTN
 - * Phòng I74, Cơ sở 227 Nguyễn Văn Cừ Q5
 - * Khi cần thiết, hãy email hẹn trước nếu muốn gặp GV

Thông tin về môn học

- * Tên môn học: Kiến trúc máy tính & Hợp ngữ
- * Số tín chỉ: 4
 - * Lý thuyết: 45 tiết
 - * Thực hành: 30 tiết (Hình thức 2)

Mô tả nội dung môn học

- * Mô tả cấu tạo của máy tính
- * Giải thích nguyên tắc hoạt động của máy tính và của các thành phần bên trong
- * Tính toán số học trên máy tính
- * Sử dụng hợp ngữ để viết một đoạn chương trình thực hiện một công việc đơn giản. Giải thích cặn kẽ các công đoạn: biên dịch, liên kết, nạp, thực thi của chương trình đã viết.
- * Trình bày các bước trong một chu trình xử lý lệnh
- * Hình dung được các công đoạn thiết kế bộ vi xử lý
- * Nguyên tắc hoạt động của các bộ nhớ trên máy tính.

Phân bố thang điểm

Hình thức	Diễn giải	Thang điểm
Kiểm tra trên lớp	Trong 1 số buổi học LT ngẫu nhiên GV sẽ yêu cầu SV làm bài kiểm tra nhỏ về bài học (tối đa 30 phút)	2 điểm
Thi LT giữa kỳ	Dự kiến vào tuần thứ 8, kiểm tra kiến thức đã học của 7 tuần trước	2 điểm
Thi LT cuối kỳ	Thời gian theo lịch của trường, kiểm tra tất cả các kiến thức đã học của 15 tuần trước	4 điểm
Thực hành	GVTH sẽ upload các bài tập lên moodle cho SV làm. Cuối kỳ GV sẽ chọn ngẫu nhiên 3 bài nộp của SV để chấm lấy điểm TH	2 điểm

Phương pháp học

- * **Trên lớp:**
 - * Tập trung nghe giảng, không nói chuyện riêng
 - * Suy nghĩ
 - * Thảo luận (khi có yêu cầu)
 - * Đặt câu hỏi với GV
- * **Ở nhà:**
 - * Đọc thêm sách
 - * Tìm kiếm tài liệu trên Internet
 - * Tham gia tích cực trên moodle

Tài liệu đọc thêm

- * Patterson and Hennessy, ***Computer Organization and Design: The Hardware / Software Interface (3rd edition)***, Morgan Kaufmann, 2005
- * Nguyễn Minh Tuấn, ***Kiến trúc máy tính***, ĐHKHTN, TP.HCM
- * W.Stallings, ***Computer Architecture and Organization***, Prentice Hall, 2007

Một số quy định môn học

- * Không nói chuyện nhiều trong lớp, gây ảnh hưởng GV và không khí lớp học
- * Khi kiểm tra, thi cử:
 - * Không gian lận, copy bài nhau. Nếu vi phạm sẽ nhận 0 điểm cho tất cả các bài liên quan
 - * Không xin điểm, nộp bài trễ deadline
- * Thường xuyên theo dõi thông tin trên moodle

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Vũ Minh Trí – vmtri@fit.hcmus.edu.vn

01 – Tổng quan máy tính

Nội dung bài giảng

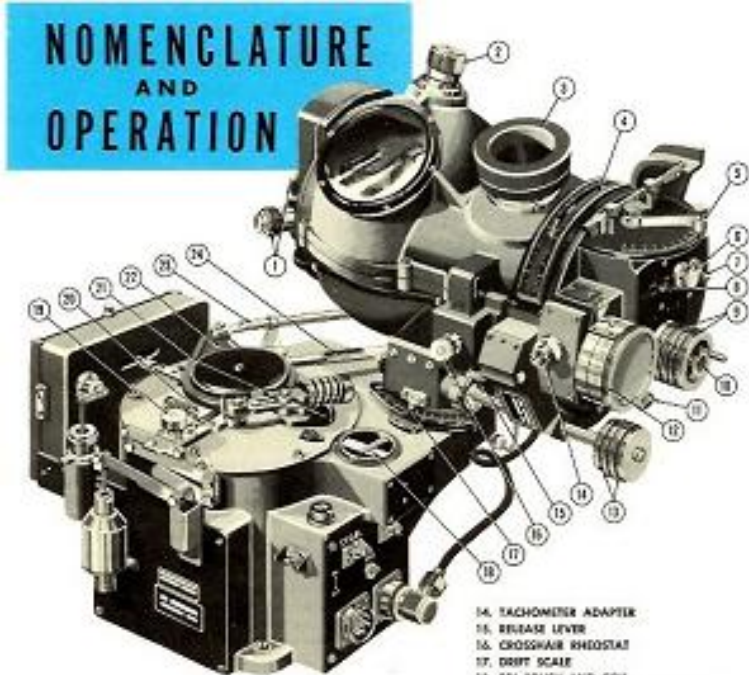
2

- Quá trình phát triển và một số nét đặc trưng của các thế hệ máy tính
- Định luật Moore
- Một số thành phần cơ bản của máy tính cá nhân ngày nay
- Giải thích các khái niệm wafer, chip, chipset
- Mô hình abstraction layers

Thế hệ 0

Non-digital computers

3



1. LEVELING KNOBS
2. CAGING KNOB
3. EYEPIECE
4. INDEX WINDOW
5. TRAIL ARM AND TRAIL PLATE
6. EXTENDED VISION KNOB
7. RATE MOTOR SWITCH
8. DISC SPEED GEAR SHIFT
9. RATE AND DISPLACEMENT KNOBS
10. MIRROR DRIVE CLUTCH
11. SEARCH KNOB
12. DISC SPEED DRUM
13. TURN AND DRIFT KNOBS

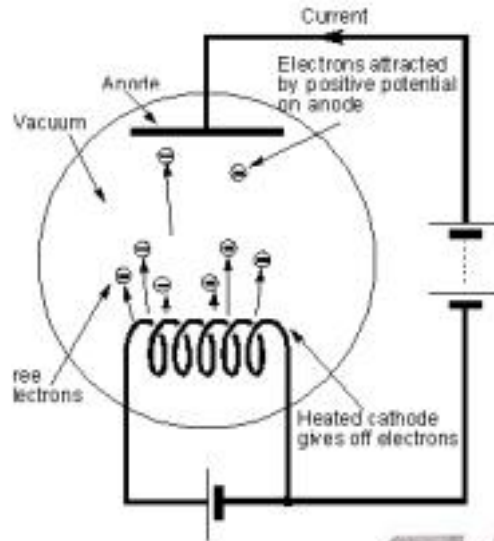
14. TACHOMETER ADAPTER
15. RELEASE LEVER
16. CROSSHAIR RHODOSTAT
17. DRIFT SCALE
18. PEN BRUSH AND COIL
19. AUTOPILOT CLUTCH ENGAGING KNOB
20. AUTOPILOT CLUTCH
21. BOMBIGHT CLUTCH ENGAGING LEVER
22. BOMBIGHT CLUTCH
23. BOMBIGHT CONNECTING ROD
24. AUTOPILOT CONNECTING ROD

The bombight has 2 main parts, sighthead and stabilizer. The sighthead pivots on the stabilizer and is locked to it by the dovetail locking pin. The sighthead is connected to the directional gyro in the stabilizer through the bombight connecting rod and the bombight clutch.

http://en.wikipedia.org/wiki/Analog_computer

Thế hệ 1 Vacuum-tube (Đèn chân không)

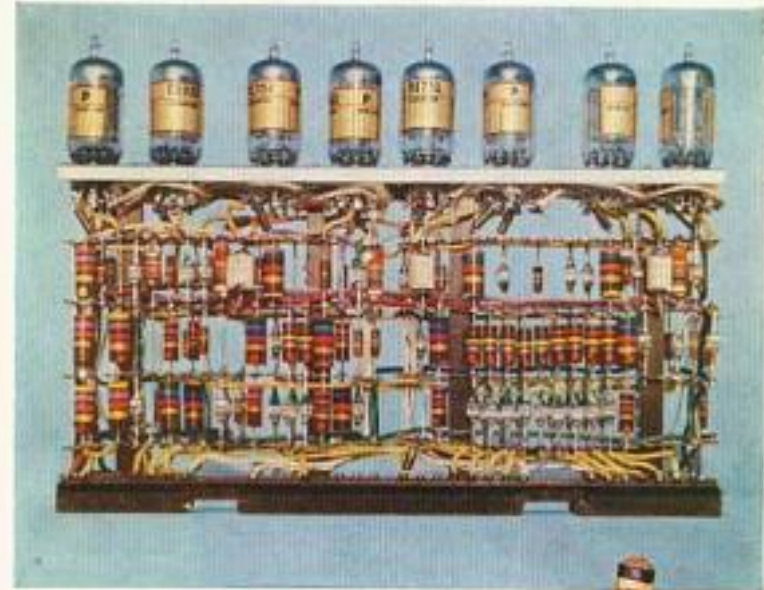
4



Vacuum tube



IBM 700



MODERN ABACUS

new tool for lightning-fast calculation



Resembling a Chinese abacus, this 12-inch electronic assembly operates at a speed of one million pulses a second. It is one of 274 similar electronic units that perform the computing and control functions of IBM's great new "701" Electronic Data Processing Machines.

These extraordinary machines are providing the nation's defense projects with the most flexible and productive computer ever manufactured in quantity.

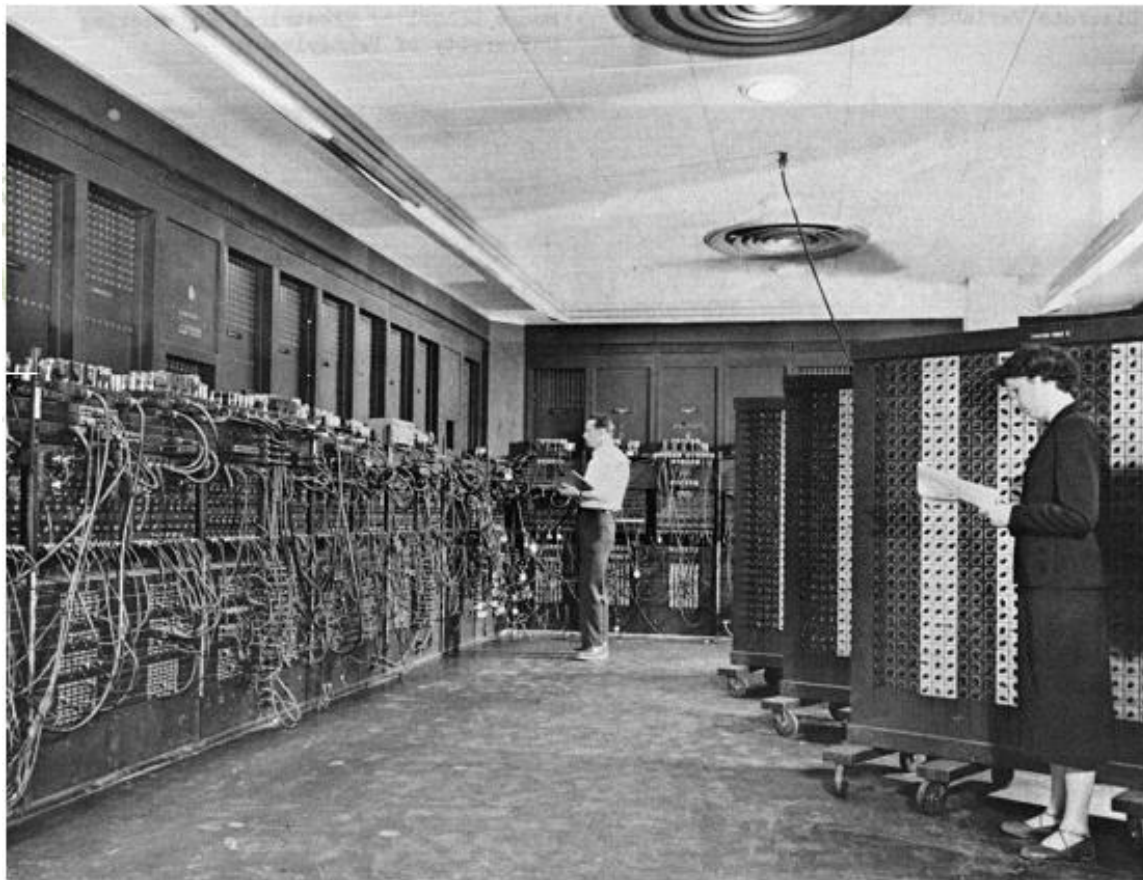
In every field of business, IBM machines reduce the drudgery and increase the speed and accuracy of computing and accounting operations.

IBM

Electronic Business Machines
INTERNATIONAL BUSINESS MACHINES

Hệ thống ENIAC (Electronic Numerical Intergrator and Computer)

5

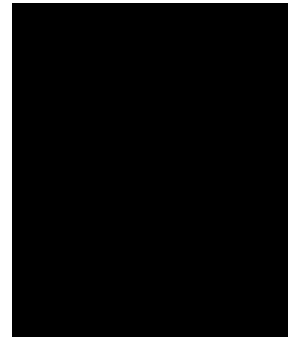
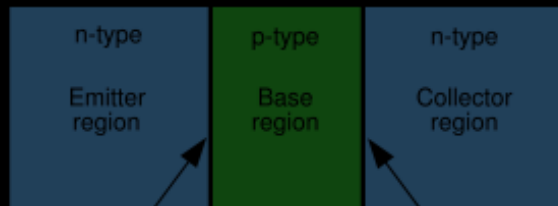
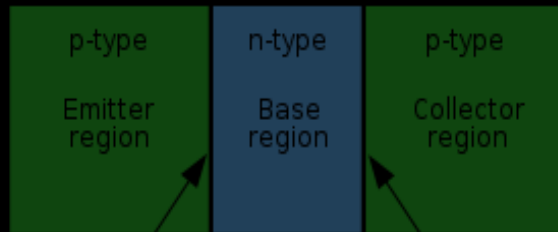


Detail of the back of a panel of ENIAC, showing vacuum tubes

Thế hệ 2

Transistor (Linh kiện bán dẫn)

6



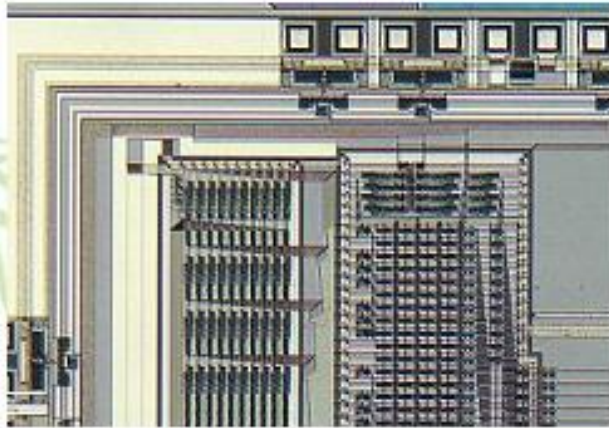
IBM 7094



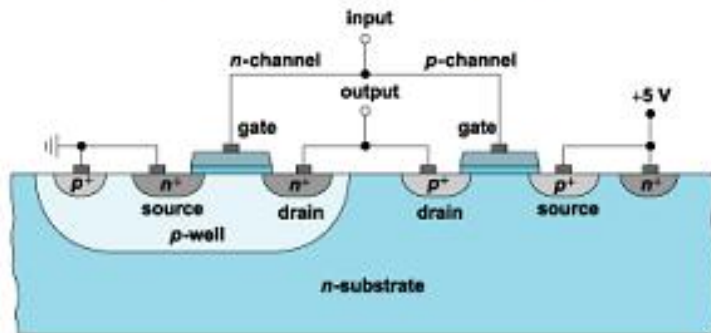
Thế hệ 3

Integrated Circuit (Vi mạch tích hợp)

7



Integrated circuit (IC)



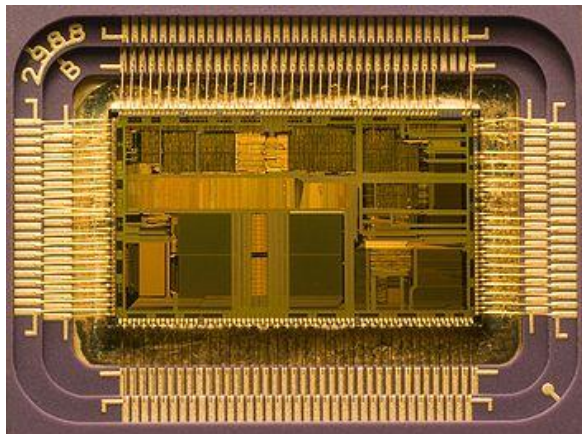
IBM 360

http://en.wikipedia.org/wiki/IBM_360

Thế hệ 4

Microprocessor (Vi xử lý)

8



Bộ vi xử lý Intel 80486DX2



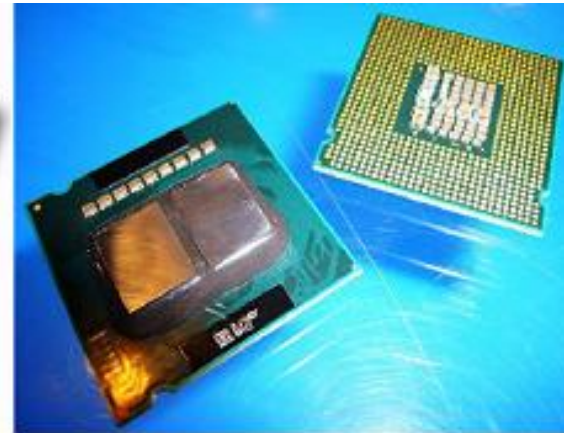
Intel 4004 with 2300 transistors inside



XT computer with Intel 8086 chip

Ngày nay (CPU đa nhân, Super-computer)

9



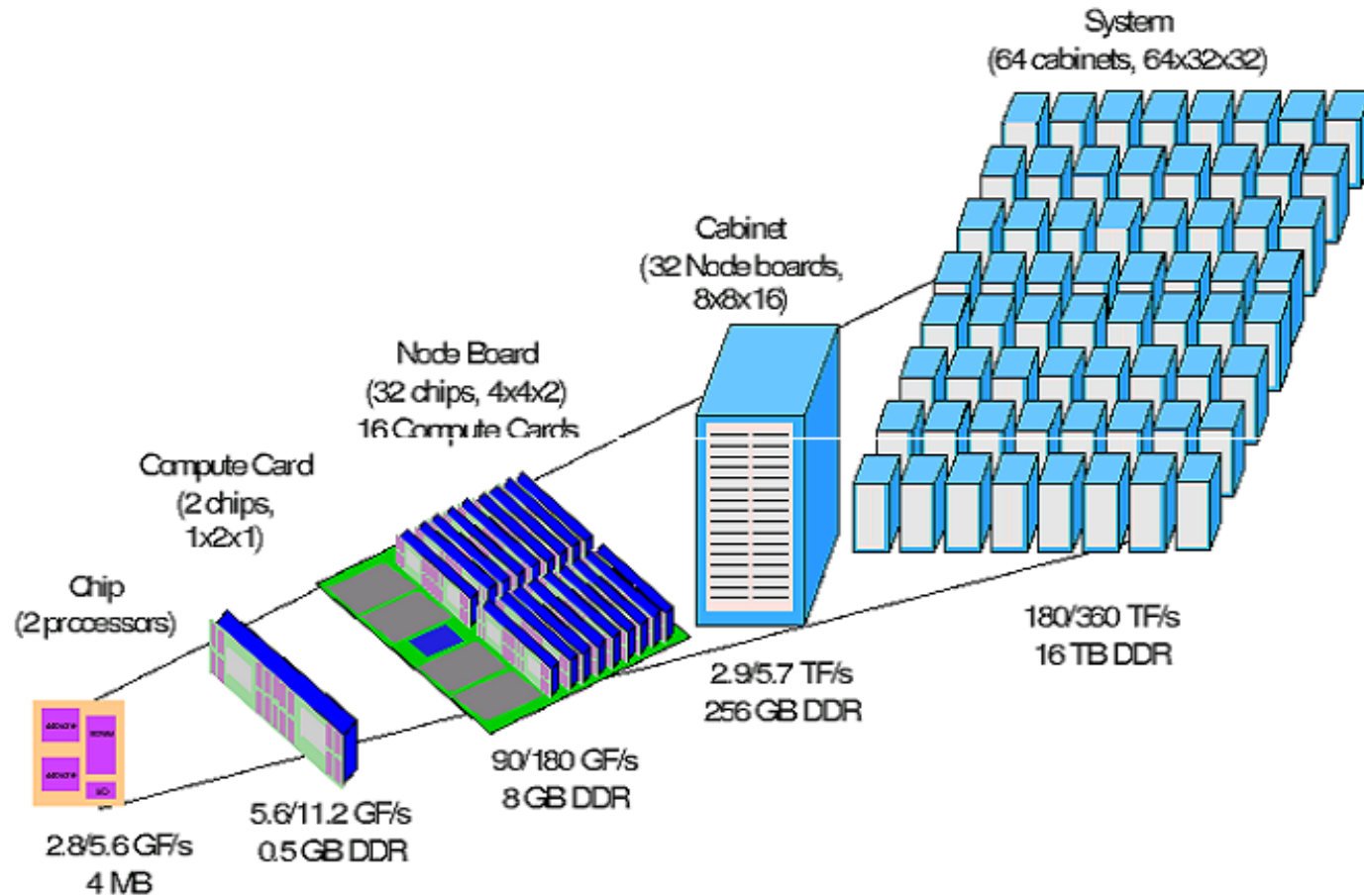
478.2 teraFLOPS

<http://www.top500.org/system/8968>



Thế hệ 5 Parallel Processing ?

10



Tổng kết

11

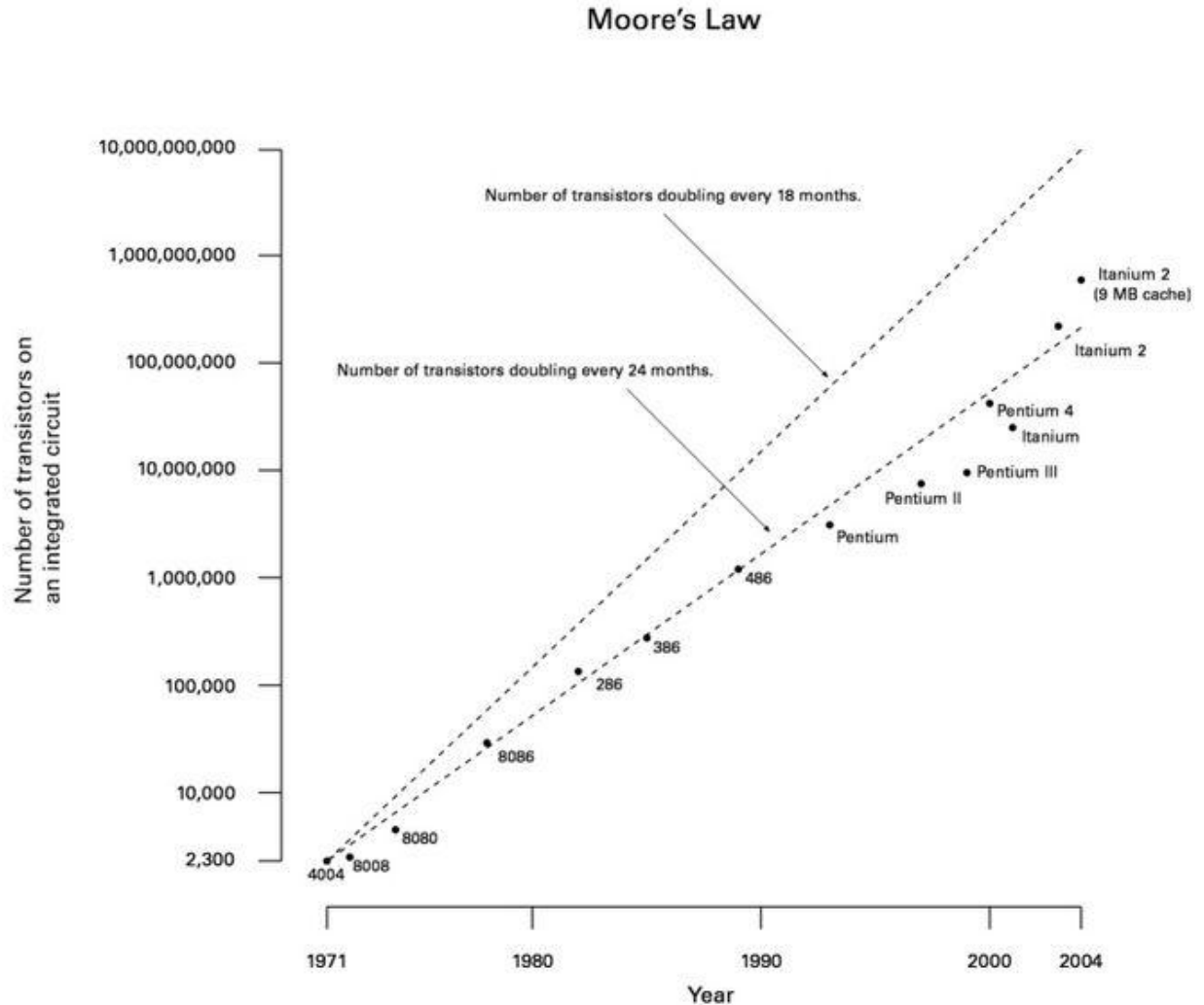
Thế hệ	Khoảng thời gian	Công nghệ
1	1940 – 1956	Vacuum tubes
2	1956 – 1963	Transistors
3	1964 – 1971	Integrated Circuits
4	1971 – nay	Microprocessors
5	Tương lai	Parallel Processing

Định luật Moore

12

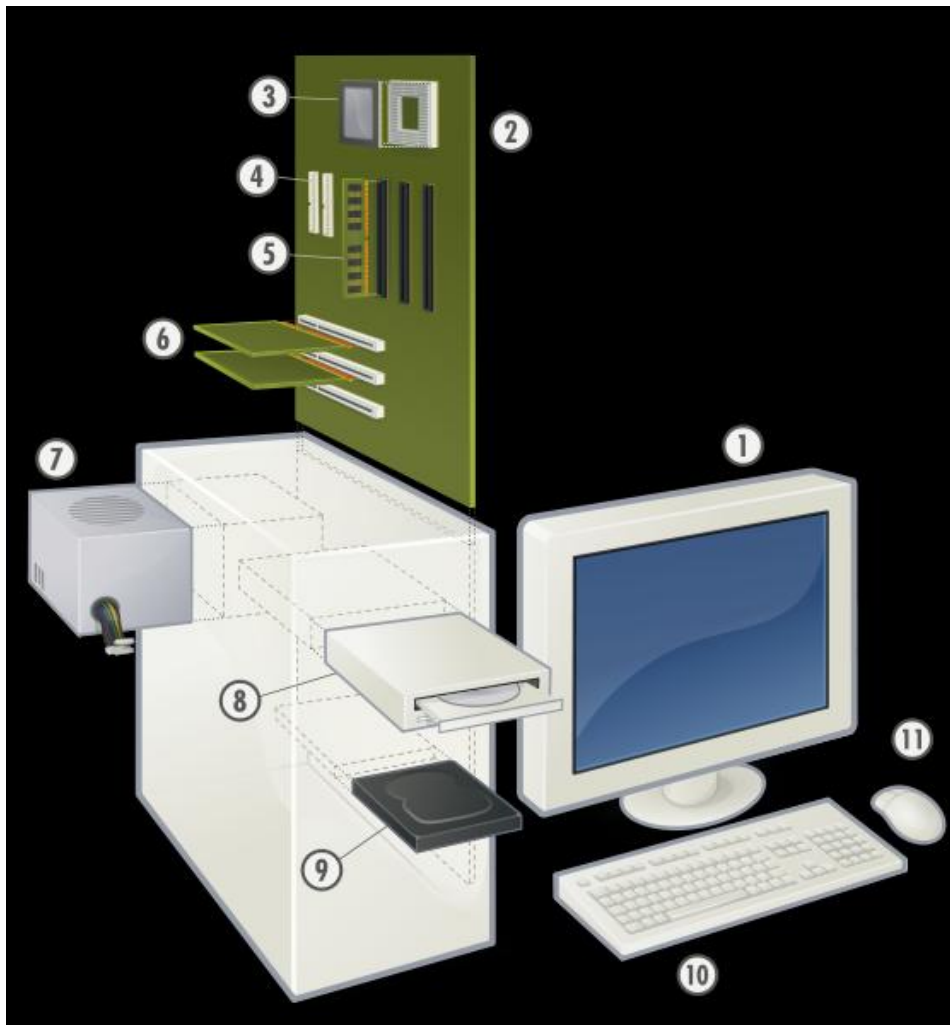


The number of transistors on a chip will double about every two years (18 months in some docs)



Một số thành phần cơ bản trên máy tính cá nhân ngày nay

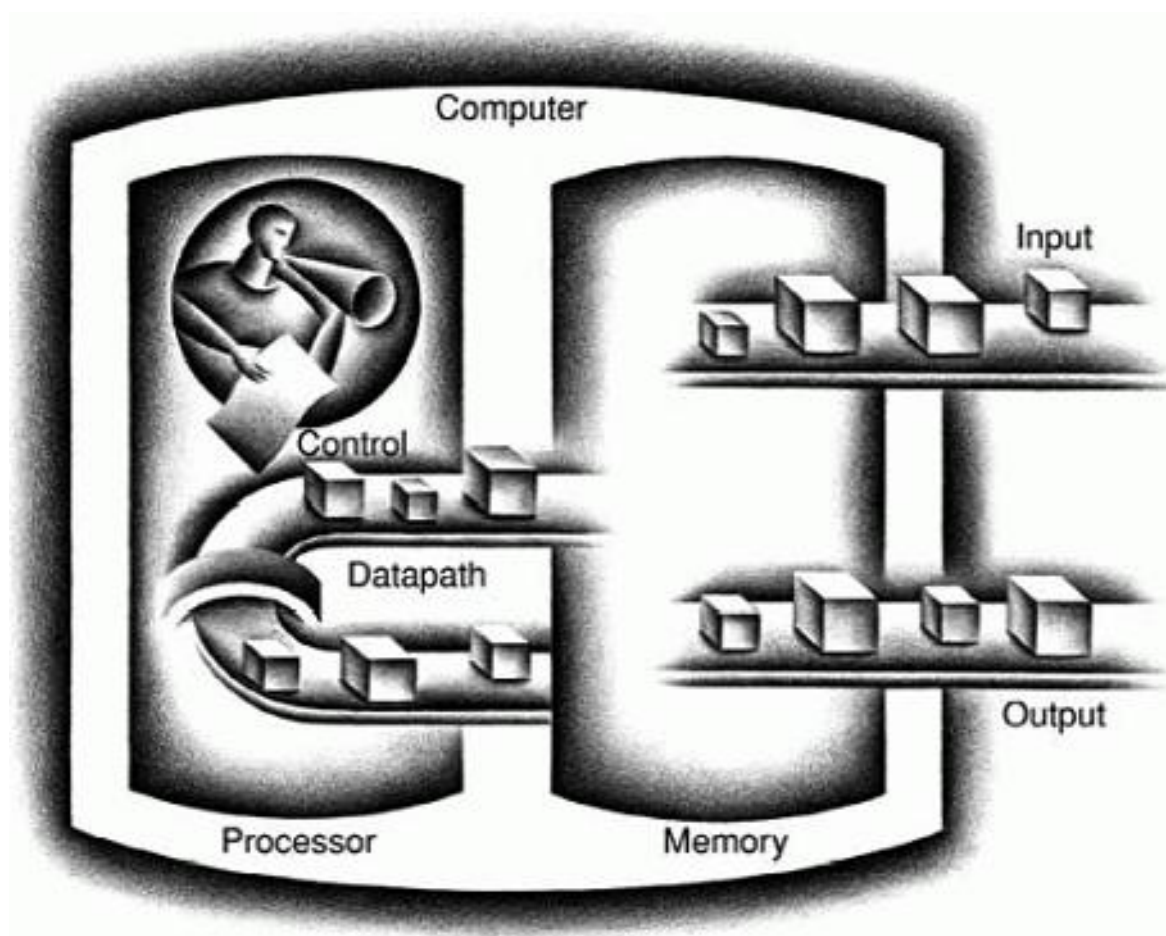
13



- 1: Màn hình
- 2: Mainboard
- 3: CPU
- 4: Chân cắm dây nối HDD
- 5: RAM
- 6: Chân cắm mở rộng PCI /
PCI Express
- 7: Nguồn điện
- 8: Ổ quang CD / DVD
- 9: Ổ đĩa cứng
- 10: Bàn phím
- 11: Chuột

...5 thành phần cơ bản

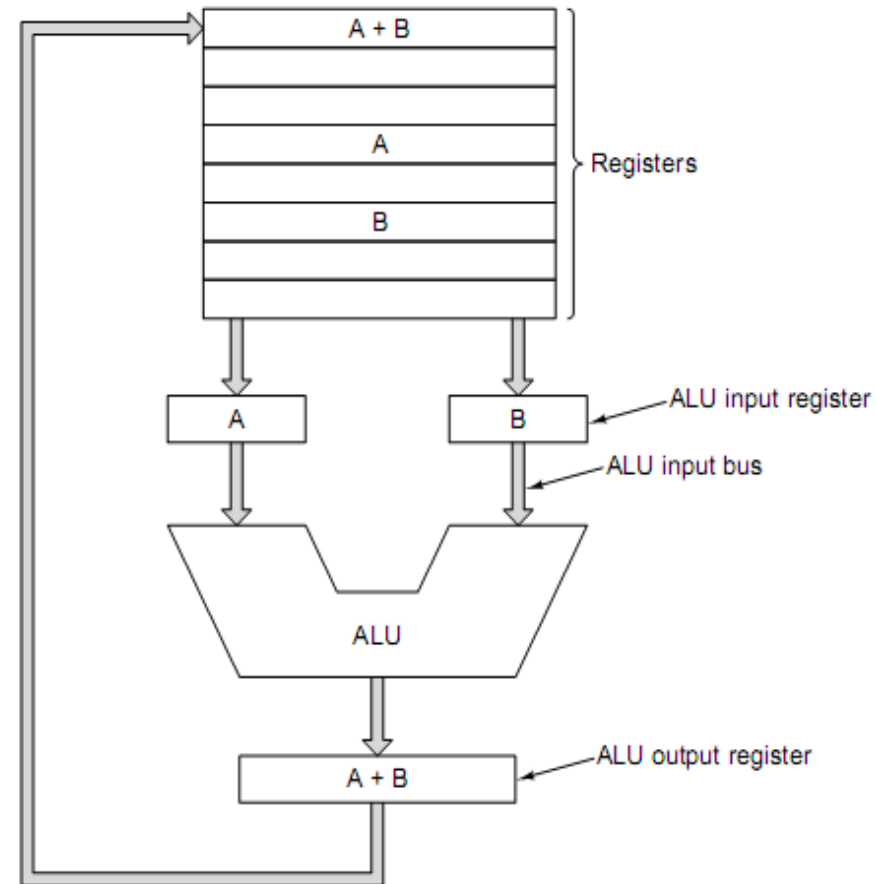
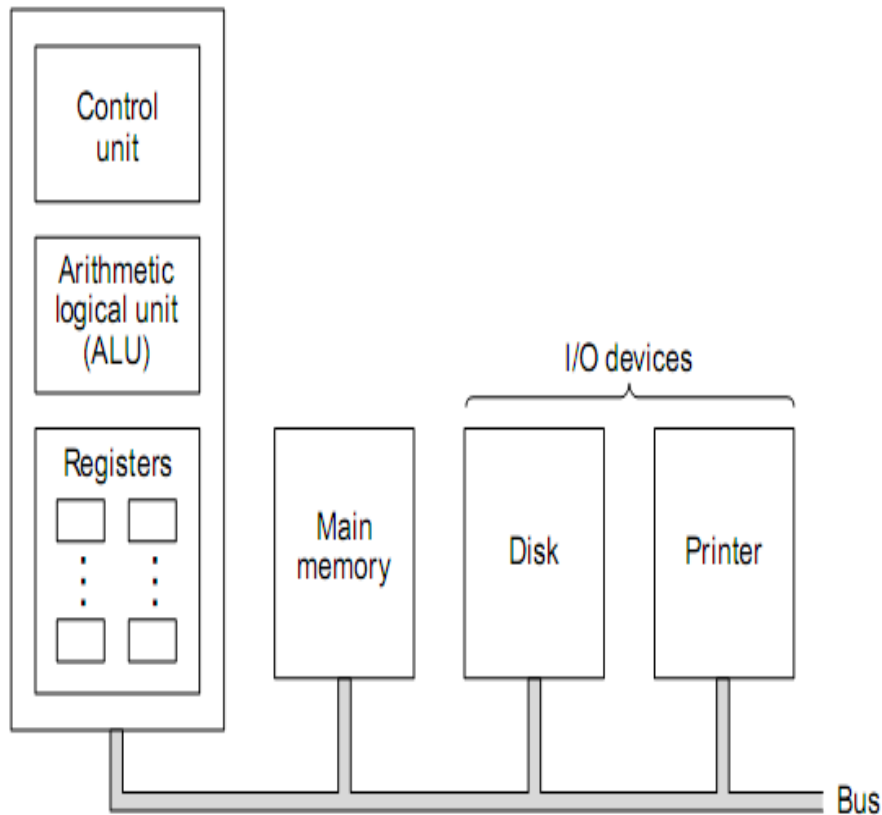
14



Control – Data path ?

15

Central processing unit (CPU)

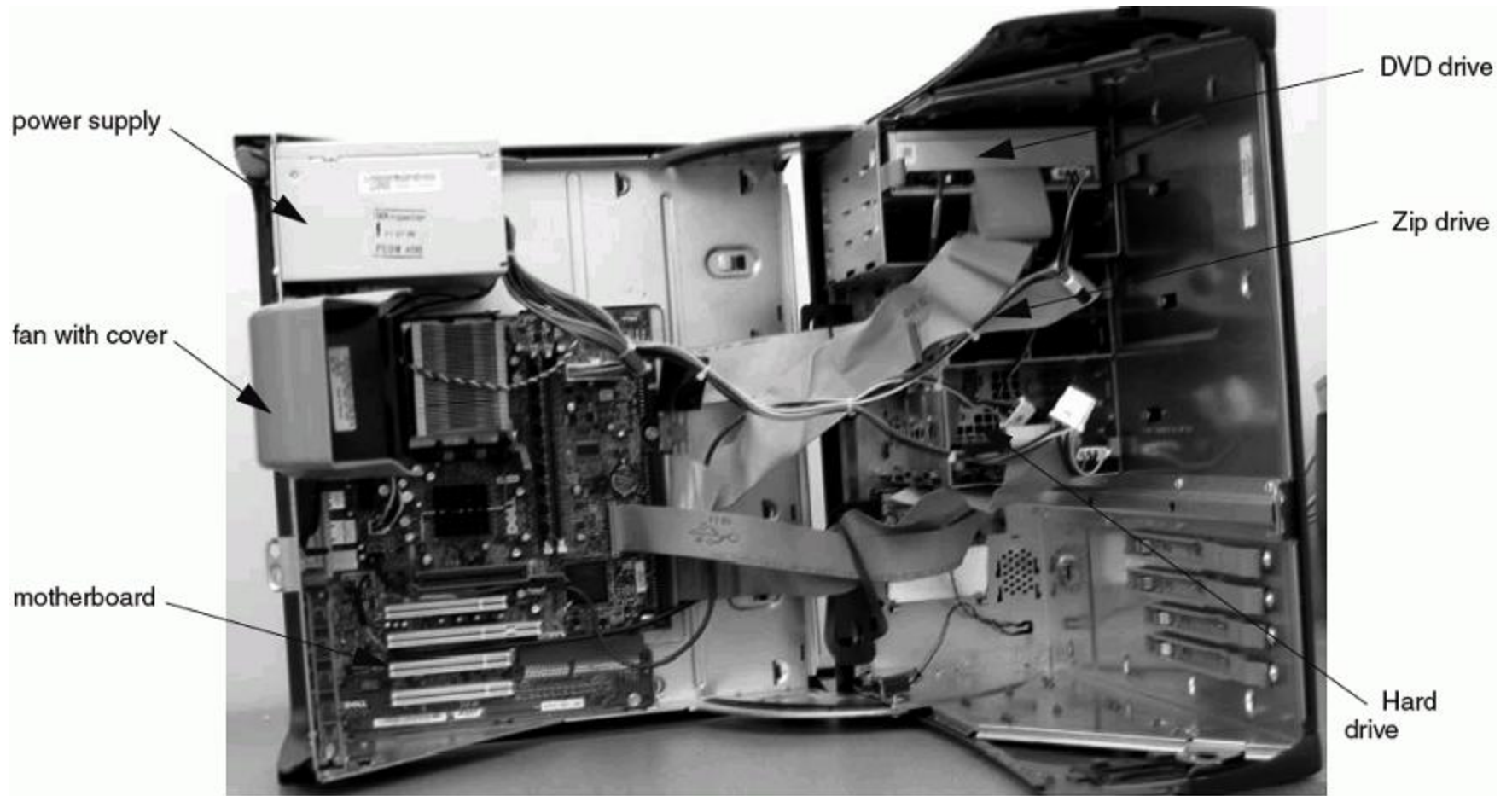


The organization of a simple computer with one CPU and two I/O devices.

The data path of a typical Von Neumann machine

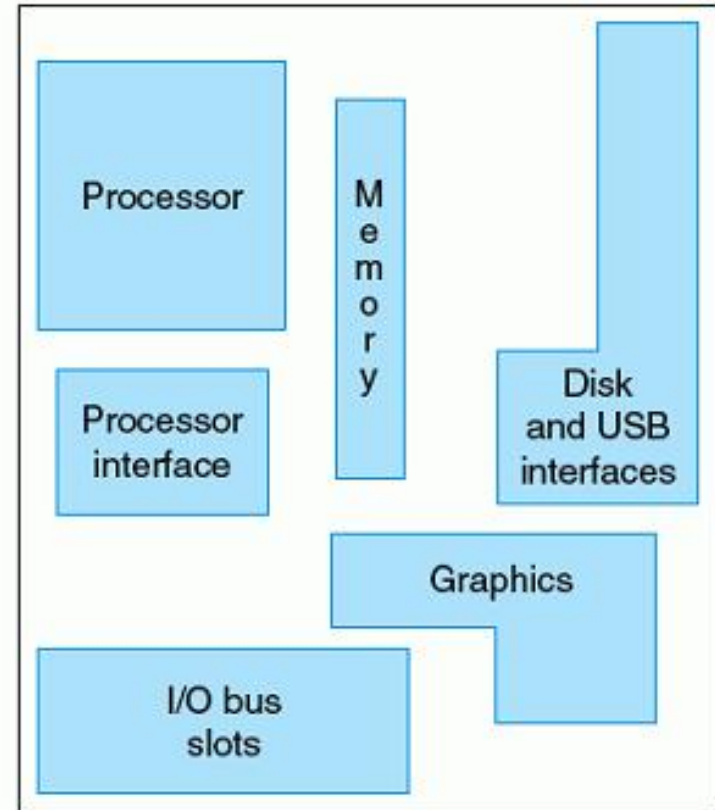
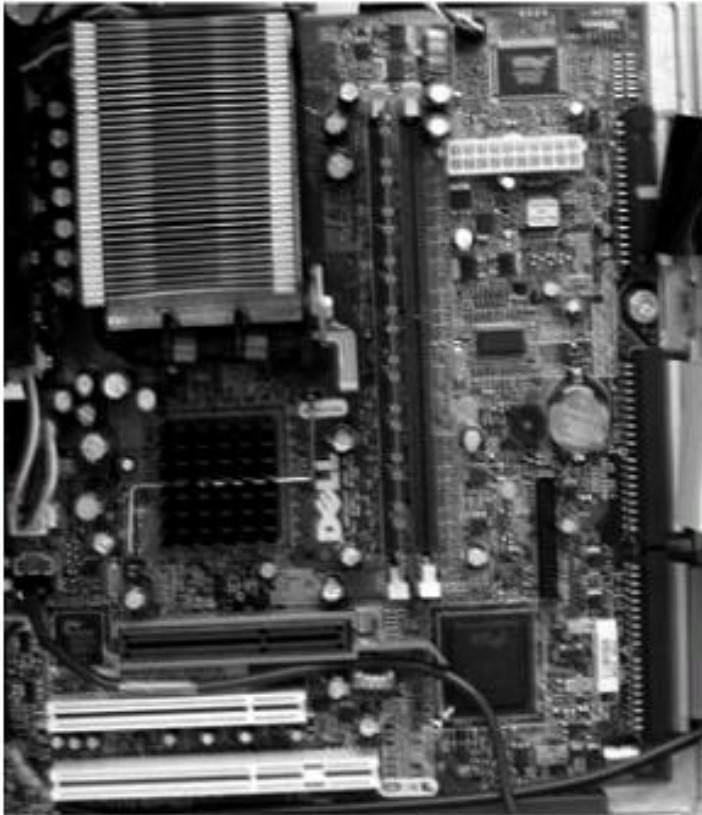
Inside PC

16



Mainboard (Motherboard)

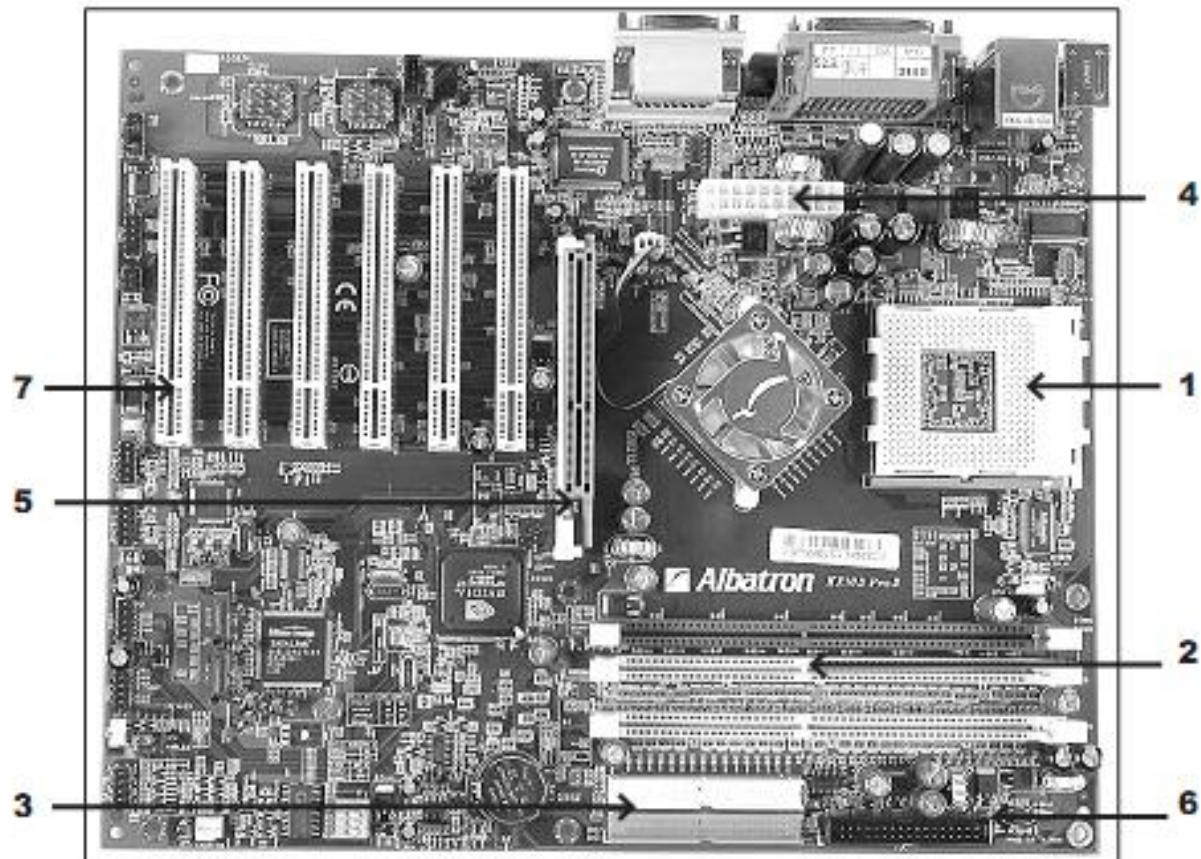
17



Inside mainboard

18

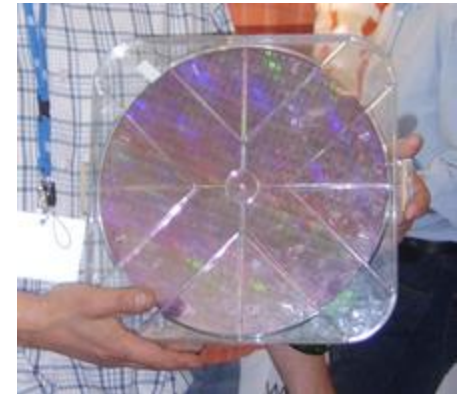
The motherboard is at the heart of the computer. All the various components of a computer are **connected** to the motherboard. 1. The processor (cpu) 2. Memory chips 3. Hard disk 4. Power supply (psu) 5. Graphics card (AGP) 6. Floppy drive 7. PCI slots for sound cards, modems etc.



Một số khái niệm cơ bản - Wafer

19

- Wafer (Để chip): Tấm silicon mỏng đã được cấy vật liệu khác nhau để tạo ra những vi mạch
- Có kích thước trung bình từ 25,4mm (1 inch) – 200mm (7.9 inch).
- Intel, TSMC hay Samsung đã nâng kích thước của wafer lên 300mm (12 inch), thậm chí lên 450mm (18 inch)
- Kích thước wafer được tăng lên đã làm giá thành của một vi mạch trở nên rất rẻ.



Một số khái niệm cơ bản - Chip

20

- Chip: Có thể hiểu là mạch tích hợp (Integrated Circuit) gắn trên đế chip (wafer) nhằm xử lý các công việc trên máy tính
- Chip có kích thước rất nhỏ nhưng có thể chứa hàng chục triệu transistor, số lượng transistor càng lớn thì tốc độ truyền và xử lý tín hiệu càng nhanh
- Hiện nay có các loại chip xử lý: 4, 8, 16, 32, 64 bit

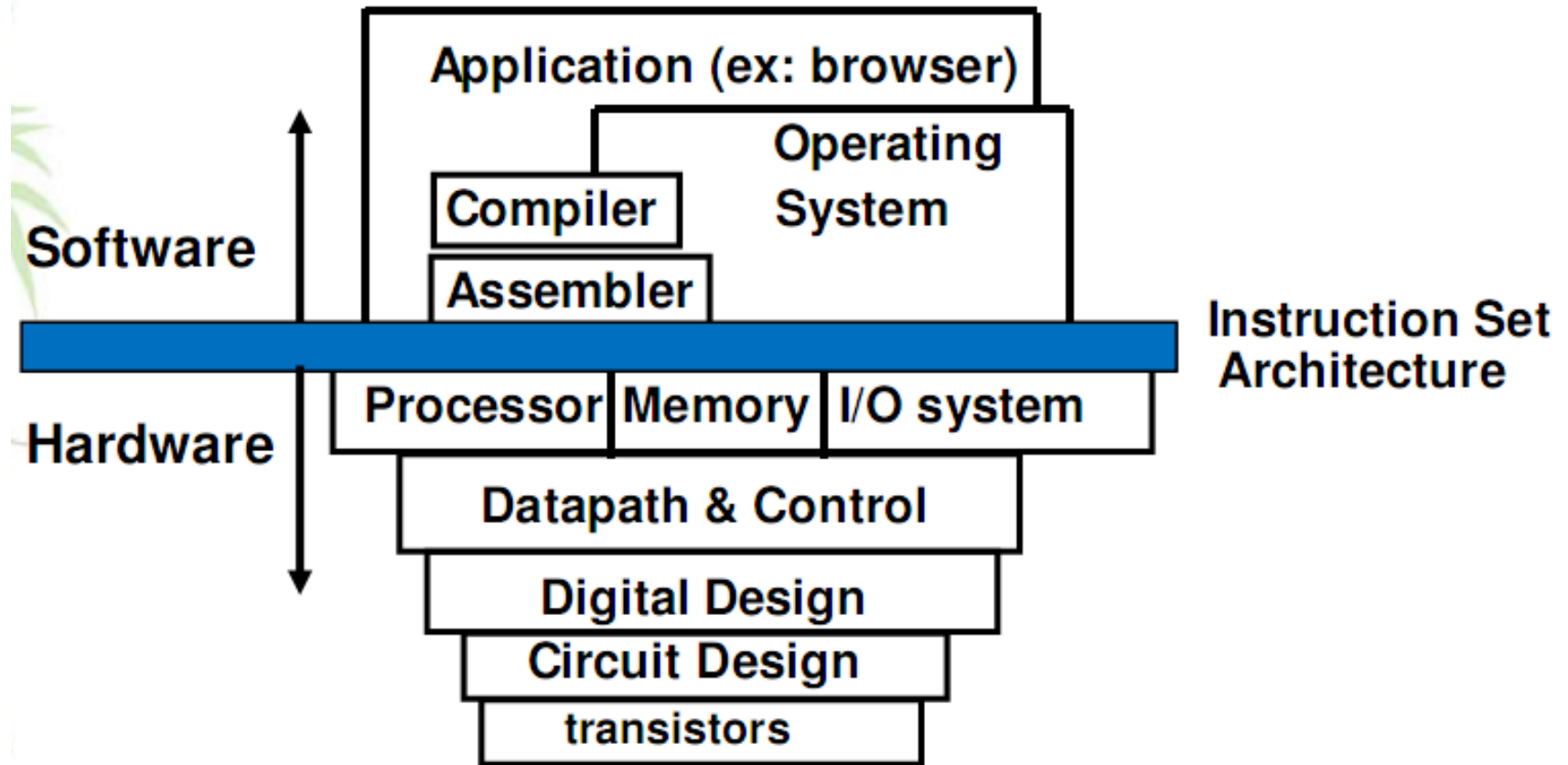
Một số khái niệm cơ bản - Chipset

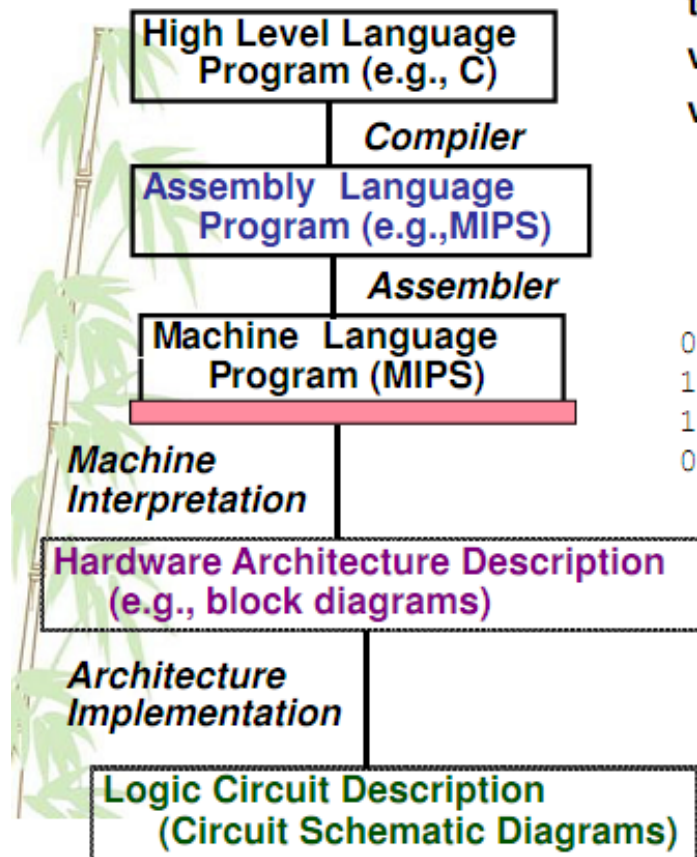
21

- Chipset là tập hợp nhiều chip gắn kết lại với nhau trên cùng 1 đế chip (wafer) để xử lý nhiều công việc trên máy tính
- Một số chipset thông dụng:
 - **CPU:** Đơn vị xử lý trung tâm
 - **GPU:** Đơn vị xử lý đồ họa trên máy
 - **RAM:** Bộ nhớ truy cập tức thời chuyên phục vụ cho CPU
 - **Bán cầu bắc** (tích hợp trên mainboard): Hỗ trợ truyền thông tin cho CPU, RAM, nằm sát CPU (Hệ thống Mainboard AMD không có chipset này vì được tích hợp ngay trên CPU)
 - **Bán cầu nam** (tích hợp trên mainboard): Quản lý thiết bị ngoại vi như HDD, Mouse, Keyboard...Nằm cuối mainboard

Abstraction layers

22

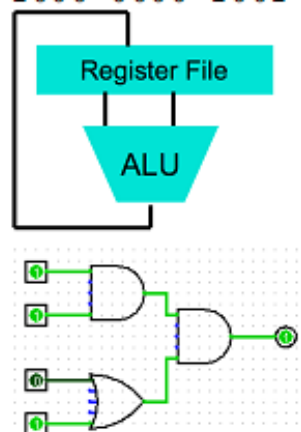




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



Homework

24

- Đọc tài liệu:
 - 01_Timeline.pdf
 - 02_Hardware.pdf
 - Patterson and Hennessy, ***Computer Organization and Design: The Hardware / Software Interface (3rd edition)***, **Chương 1**

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Võ Minh Trí – vmtri@fit.hcmus.edu.vn

02 – Biểu diễn số nguyên

Hệ cơ số q tổng quát

2

- Tổng quát số nguyên có n chữ số thuộc hệ cơ số q bất kỳ được biểu diễn:

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot q^{n-1} + \dots + x_1 \cdot q^1 + x_0 \cdot q^0$$

(mỗi chữ số x_i lấy từ tập X có q phần tử)

- Ví dụ:
 - Hệ cơ số 10: $A = 123 = 100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$
 - $q = 2, X = \{0, 1\}$: hệ nhị phân (binary)
 - $q = 8, X = \{0, 1, 2, \dots, 7\}$: hệ bát phân (octal)
 - $q = 10, X = \{0, 1, 2, \dots, 9\}$: hệ thập phân (decimal)
 - $q = 16, X = \{0, 1, 2, \dots, 9, A, B, \dots, F\}$: hệ thập lục phân (hexadecimal)
- Chuyển đổi: $A = 123$ **d** = 01111011 **b** = 173 **o** = 7B **h**
- Hệ cơ số thường được biểu diễn trong máy tính là hệ cơ số 2

Chuyển đổi giữa các hệ cơ số

3

□ Đặc điểm

- Con người sử dụng hệ thập phân
- Máy tính sử dụng hệ nhị phân, bát phân, thập lục phân

□ Nhu cầu

- Chuyển đổi qua lại giữa các hệ đếm ?
 - Hệ khác sang hệ thập phân (... \rightarrow dec)
 - Hệ thập phân sang hệ khác (dec \rightarrow ...)
 - Hệ nhị phân sang hệ khác và ngược lại (bin \leftrightarrow ...)
 - ...

Chuyển đổi giữa các hệ cơ số

[1] Decimal (10) → Binary (2)

4

- Lấy số cơ số 10 chia cho 2
 - Số dư đưa vào kết quả
 - Số nguyên đem chia tiếp cho 2
 - Quá trình lặp lại cho đến khi số nguyên = 0

□ Ví dụ: A = 123

- $123 : 2 = 61$ dư 1
- $61 : 2 = 30$ dư 1
- $30 : 2 = 15$ dư 0
- $15 : 2 = 7$ dư 1
- $7 : 2 = 3$ dư 1
- $3 : 2 = 1$ dư 1
- $1 : 2 = 0$ dư 1


Kết quả: 1111011, vì 123 là số dương,
thêm 1 bit hiển dấu vào đầu là 0 vào

→ Kết quả cuối cùng: **01111011**

Chuyển đổi giữa các hệ cơ số

[2] Decimal (10) → Hexadecimal (16)

5

- Lấy số cơ số 10 chia cho 16
 - Số dư đưa vào kết quả
 - Số nguyên đem chia tiếp cho 16
 - Quá trình lặp lại cho đến khi số nguyên = 0
 - Ví dụ: A = 123
 - $123 : 16 = 7$ dư 12 (B) 
 - $7 : 16 = 0$ dư 7
- Kết quả cuối cùng: **7B**

Chuyển đổi giữa các hệ cơ số

[3] Binary (2) → Decimal (10)

6

- Khai triển biểu diễn và tính giá trị biểu thức

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot 2^{n-1} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

- Ví dụ:

- $1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$

Chuyển đổi giữa các hệ cơ số

[4] Binary (2) → Hexadecimal (16)

7

- Nhóm từng **bộ 4 bit** trong biểu diễn nhị phân rồi chuyển sang ký số tương ứng trong hệ thập lục phân (0000 → 0, ..., 1111 → F)
- Ví dụ
 - $1001011_2 = 0100\ 1011 = 4B_{16}$

HEX	BIN	HEX	BIN	HEX	BIN	HEX	BIN
0	0000	4	0100	8	1000	C`	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Chuyển đổi giữa các hệ cơ số

[5] Hexadecimal (16) → Binary (2)

8

- Sử dụng bảng dưới đây để chuyển đổi:

HEX	BIN	HEX	BIN	HEX	BIN	HEX	BIN
0	0000	4	0100	8	1000	C`	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Ví dụ:

- $4B_{16} = 1001011_2$

Chuyển đổi giữa các hệ cơ số

[6] Hexadecimal (16) → Decimal (10)

9

- Khai triển biểu diễn và tính giá trị biểu thức

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot 16^{n-1} + \dots + x_1 \cdot 16^1 + x_0 \cdot 16^0$$

- Ví dụ:

- $7B_{16} = 7 \cdot 16^1 + 12 (B) \cdot 16^0 = 123_{10}$

Hệ nhị phân

10

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot 2^{n-1} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

- Được dùng nhiều trong máy tính để biểu diễn các giá trị lưu trong các thanh ghi hoặc trong các ô nhớ. Thanh ghi hoặc ô nhớ có kích thước 1 byte (8 bit) hoặc 1 word (16 bit).
- n được gọi là chiều dài bit của số đó
- Bit trái nhất x_{n-1} là bit có giá trị (nặng) nhất **MSB** (Most Significant Bit)
- Bit phải nhất x_0 là bit ít giá trị (nhẹ) nhất **LSB** (Less Significant Bit)

Ý tưởng nhị phân

11

- Số nhị phân có thể dùng để biểu diễn bất kỳ việc gì mà bạn muốn!
- Một số ví dụ:
 - Giá trị logic: 0 → False; 1 → True
 - Ký tự:
 - 26 ký tự (A → Z): 5 bits ($2^5 = 32$)
 - Tính cả trường hợp viết hoa/thường + ký tự lạ → 7 bits (ASCII)
 - Tất cả các ký tự ngôn ngữ trên thế giới → 8, 16, 32 bits (Unicode)
 - Màu sắc: Red (00), Green (01), Blue (11)
 - Vị trí / Địa chỉ: (0, 0, 1)...
 - Bộ nhớ: N bits → Lưu được tối đa 2^N đối tượng

Số nguyên không dấu

12

□ Đặc điểm

- Biểu diễn các đại lượng luôn dương
 - Ví dụ: chiều cao, cân nặng, mã ASCII...
- Tất cả bit đều được sử dụng để biểu diễn giá trị (không quan tâm đến dấu âm, dương)
- Số nguyên không dấu 1 byte lớn nhất là $1111\ 1111_2 = 2^8 - 1 = 255_{10}$
- Số nguyên không dấu 1 word lớn nhất là $1111\ 1111\ 1111\ 1111_2 = 2^{16} - 1 = 65535_{10}$
- Tùy nhu cầu có thể sử dụng số 2, 3... word.
- **LSB = 1** thì số đó là số đó là **số lẻ**

Số nguyên có dấu

13

- Lưu các số dương hoặc âm (số có dấu)
- Có 4 cách phổ biến:
 - [1] Dấu lượng
 - [2] Bù 1
 - [3] Bù 2
 - [4] Số quá (thừa) K
- Số có dấu trong máy tính được biểu diễn ở dạng số bù 2

Số nguyên có dấu

[1] Dấu lượng

14

- **Bit trái nhất (MSB):** bit đánh dấu âm / dương
 - 0: số dương
 - 1: số âm
- **Các bit còn lại:** biểu diễn độ lớn của số (hay giá trị tuyệt đối của số)
- **Ví dụ:**
 - **Một byte 8 bit:** sẽ có 7 bit (trừ đi bit dấu) dùng để biểu diễn giá trị tuyệt đối cho các số có giá trị từ 0000000 (0_{10}) đến 1111111 (127_{10})
 - Ta có thể biểu diễn các số từ -127_{10} đến $+127_{10}$
 - $-N$ và N chỉ khác giá trị bit MSB (bit dấu), phần độ lớn (giá trị tuyệt đối) hoàn toàn giống nhau

Số nguyên có dấu

[2] Bù 1

15

- Tương tự như phương pháp [1], bit MSB dùng làm bit dấu
 - 0: Số dương
 - 1: Số âm
- Các bit còn lại (*) dùng làm độ lớn
- **Số âm: Thực hiện phép đảo bit tất cả các bit của (*)**
- **Ví dụ:**
 - Dạng bù 1 của 00101011 (43) là 11010100 (-43)
 - **Một byte 8 bit: biểu diễn từ -127_{10} đến $+127_{10}$**
 - Bù 1 có hai dạng biểu diễn cho số 0, bao gồm: 00000000 (+0) và 11111111 (-0) (mẫu 8 bit, giống phương pháp [1])
 - Khi thực hiện phép cộng, cũng thực hiện theo quy tắc cộng nhị phân thông thường, tuy nhiên, **nếu còn phát sinh bit nhớ thì phải tiếp tục cộng bit nhớ này vào kết quả vừa thu được**

Số nguyên có dấu

[3] Bù 2

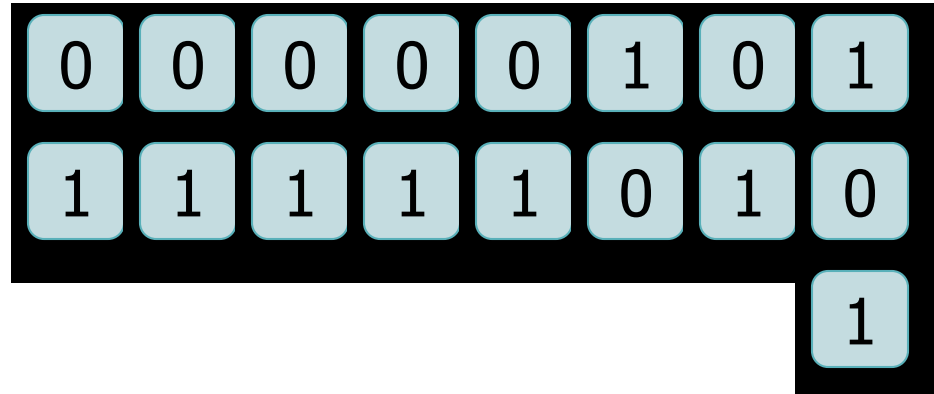
16

- Biểu diễn giống như số bù 1 + **ta phải cộng thêm số 1 vào kết quả (dạng nhị phân)**
- Số bù 2 ra đời khi người ta gặp vấn đề với hai phương pháp dấu lượng [1] và bù 1 [2], đó là:
 - Có hai cách biểu diễn cho số 0 (+0 và -0) → không đồng nhất
 - Bit nhớ phát sinh sau khi đã thực hiện phép tính phải được cộng tiếp vào kết quả → dễ gây nhầm lẫn**→ Phương pháp số bù 2 khắc phục hoàn toàn 2 vấn đề đó**
- **Ví dụ:**
 - **Một byte 8 bit:** biểu diễn từ -128_{10} đến $+127_{10}$ (được lợi 1 số vì chỉ có 1 cách biểu diễn số 0)

Số bù 1 và Số bù 2

17

Số 5 (8 bit)

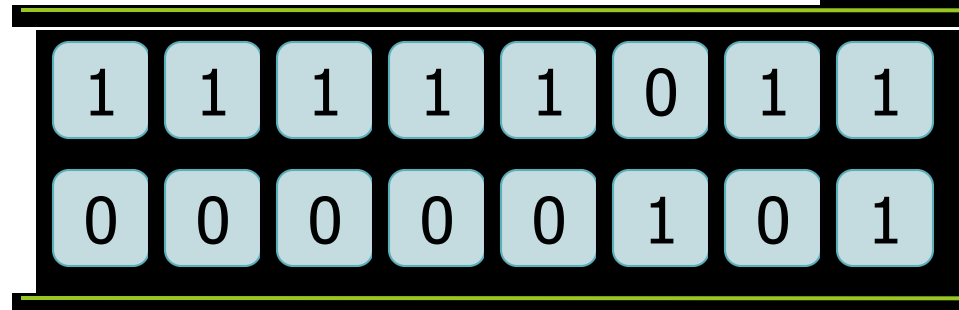


Số bù 1 của 5

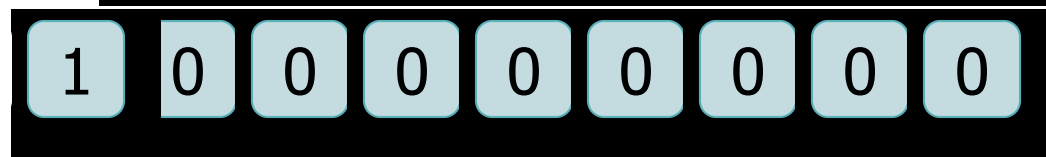
+

Số bù 2 của 5

+ Số 5



Kết quả



Nhận xét số bù 2

18

- (Số bù 2 của x) + x = một dãy toàn bit 0 (không tính bit 1 cao nhất do vượt quá phạm vi lưu trữ)
→ Do đó số bù 2 của x chính là giá trị âm của x hay $-x$ (Còn gọi là phép lấy đối)
- Đổi số thập phân âm -5 sang nhị phân?
→ Đổi 5 sang nhị phân rồi lấy số bù 2 của nó
- Thực hiện phép toán $a - b$?
→ $a - b = a + (-b)$ → Cộng với số bù 2 của b .

Số nguyên có dấu

[4] Số quá (thừa) K

19

- Còn gọi là biểu diễn số dịch (*biased representation*)
- Chọn một số nguyên dương K **cho trước** làm giá trị dịch
- Biểu diễn số N:
 - **+N (dương)**: có được bằng cách lấy $K + N$, với K được chọn sao cho **tổng của K và một số âm bất kỳ trong miền giá trị luôn luôn dương**
 - **-N (âm)**: có được bằng cách lấy $K - N$ (hay lấy bù hai của số vừa xác định)
- **Ví dụ:**
 - **Dùng 1 Byte (8 bit):** biểu diễn từ -128_{10} đến $+127_{10}$
 - Trong hệ 8 bit, biểu diễn $N = 25$, chọn số thừa $k = 128$, :
 - $+25_{10} = 10011001_2$
 - $-25_{10} = 01100111_2$
 - Chỉ có một giá trị 0: $+0 = 10000000_2$, $-0 = 10000000_2$

Nhận xét

20

- **Số bù 2 [3]** → lưu trữ số có dấu và các phép tính của chúng trên máy tính (**thường dùng nhất**)
 - Không cần thuật toán đặc biệt nào cho các phép tính cộng và tính trừ
 - Giúp phát hiện dễ dàng các trường hợp bị tràn.
- **Dấu lượng [1] / số bù 1 [2]** → dùng các thuật toán phức tạp và bất lợi vì luôn có hai cách biểu diễn của số 0 (+0 và -0)
- **Dấu lượng [1]** → phép nhân của số có dấu chấm động
- **Số thừa K [4]** → dùng cho số mũ của các số có dấu chấm động

Biểu diễn số âm (số bù 2)

21

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot (-2^{n-1}) + x_{n-2} \cdot 2^{n-2} \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$



Phạm vi lưu trữ: $[-2^{n-1}, 2^{n-1} - 1]$

□ Ví dụ:

$$\begin{aligned} \square 1101\ 0110_2 &= -2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 \\ &= -128 + 64 + 16 + 4 + 2 = \\ &= -42_{10} \end{aligned}$$

Ví dụ (số bù 2)

22

$$+123 = 01111011b$$

$$-123 = 10000101b$$

$$0 = 00000000b$$

$$-1 = 11111111b$$

$$-2 = 11111110b$$

$$-3 = 11111101b$$

$$-127 = 10000001b$$

$$-128 = 10000000b$$

Tính giá trị không dấu và có dấu

23

- Tính giá trị không dấu và có dấu của 1 số?
 - Ví dụ số word (16 bit): **1**100 1100 1111 0000
 - Số nguyên không dấu ?
 - Tất cả 16 bit lưu giá trị → giá trị là **52464**
 - Số nguyên có dấu ?
 - Bit **MSB = 1** do đó số này là **số âm**
 - Áp dụng công thức → giá trị là **-13072**

Tính giá trị không dấu và có dấu

24

- Nhận xét
 - Bit **MSB = 0** thì giá trị có dấu bằng giá trị không dấu.
 - Bit **MSB = 1** thì giá trị có dấu bằng giá trị không dấu trừ đi 256 (2^8 nếu tính theo byte) hay 65536 (2^{16} nếu tính theo word).
- Tính giá trị không dấu và có dấu của 1 số?
 - Ví dụ số word (16 bit): **1100 1100 1111 0000**
 - Giá trị không dấu = **52464**
 - Giá trị có dấu: vì bit **MSB = 1** nên giá trị có dấu = $52464 - 65536 = -13072$

Phép dịch bit và phép xoay

25

- **Shift left (SHL):** 1100 1010 → 1001 0100
 - Chuyển tất cả các bit sang trái, bỏ bit trái nhất, thêm 0 ở bit phải nhất
- **Shift right (SHR):** 1001 0101 → 0100 1010
 - Chuyển tất cả các bit sang phải, bỏ bit phải nhất, thêm 0 ở bit trái nhất
- **Rotate left (ROL):** 1100 1010 → 1001 0101
 - Chuyển tất cả các bit sang trái, bit trái nhất thành bit phải nhất
- **Rotate right (ROR):** 1001 0101 → 1100 1010
 - Chuyển tất cả các bit sang phải, bit phải nhất thành bit trái nhất

Phép toán Logic

AND, OR, NOT, XOR

26

AND	0	1
0	0	0
1	0	1

“Phép nhân”

OR	0	1
0	0	1
1	1	1

“Phép cộng”

XOR	0	1
0	0	1
1	1	0

“Phép so sánh khác”

NOT	0	1
	1	0

“Phép phủ định”

$$\begin{array}{r} \text{AND} \quad 11010011 \\ \quad \quad 00001111 \\ \hline \quad \quad 00000011 \end{array}$$

$$\begin{array}{r} \text{OR} \quad 00000011 \\ \quad \quad 01100000 \\ \hline \quad \quad 01100011 \end{array}$$

$$\begin{array}{r} \text{XOR} \quad 01100011 \\ \quad \quad 01100011 \\ \hline \quad \quad 00000000 \end{array}$$

$$\begin{array}{r} \text{NOT} \quad 11010011 \\ \hline = \quad \quad 00101100 \end{array}$$

Ví dụ

27

□ $X = 0000\ 1000b = 8d$

→ $X\ \text{shl}\ 2 = 0010\ 0000b = 32d = 8 \cdot 2^2$

→ $(X\ \text{shl}\ 2)\ \text{or}\ X = 0010\ 1000b = 40d = 32 + 8$

□ $Y = 0100\ 1010b = 74d$

→ $((Y\ \text{and}\ 0Fh)\ \text{shl}\ 4) = 1010\ 0000$

OR

OR

→ $((Y\ \text{and}\ F0h)\ \text{shr}\ 4) = 0000\ 0100$

= $1010\ 0100 = 164d$ (không dấu)

= $(164 - 2^8) = -92d$ (có dấu)

Một số nhận xét

28

- $x \text{ SHL } y = x \cdot 2^y$
- $x \text{ SHR } y = x / 2^y$
- **AND** dùng để tắt bit (AND với 0 luôn = 0)
- **OR** dùng để bật bit (OR với 1 luôn = 1)
- **XOR, NOT** dùng để đảo bit (XOR với 1 = đảo bit đó)
- $x \text{ AND } 0 = 0$
- $x \text{ XOR } x = 0$

- **Mở rộng:**
 - Lấy giá trị tại bit thứ i của x : $(x \text{ SHR } i) \text{ AND } 1$
 - Gán giá trị 1 tại bit thứ i của x : $(1 \text{ SHL } i) \text{ OR } x$
 - Gán giá trị 0 tại bit thứ i của x : $\text{NOT}(1 \text{ SHL } i) \text{ AND } x$
 - Đảo bit thứ i của x : $(1 \text{ SHL } i) \text{ XOR } x$

Các phép toán tử

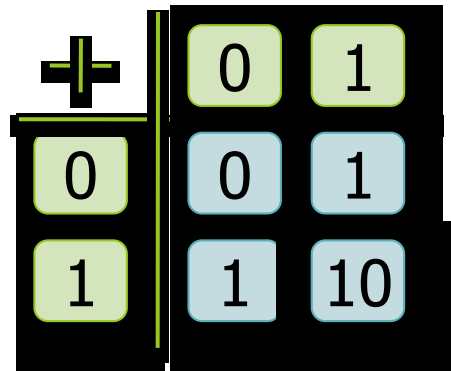
29

- Phép Cộng (+)
- Phép Trừ (-)
- Phép Nhân (*)
- Phép Chia (/)

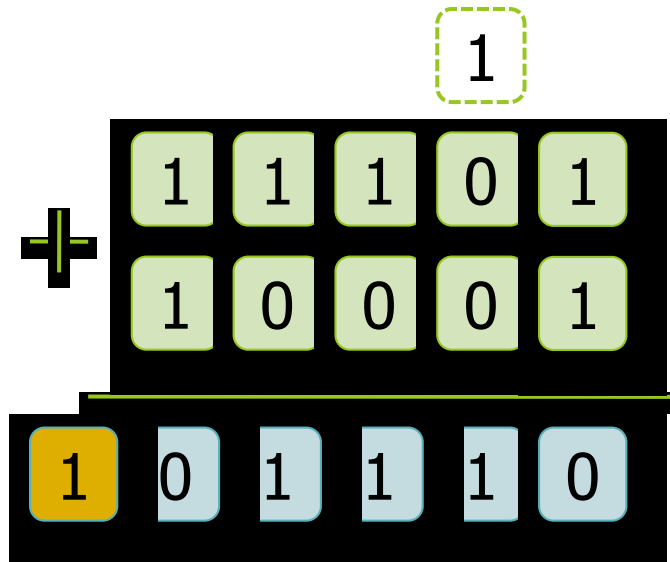
Phép cộng

30

- Nguyên tắc cơ bản:



- Ví dụ:



Phép cộng

31

$$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$$

(a) $(-7) + (+5)$

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline \underline{1}0000 = 0 \end{array}$$

(b) $(-4) + (+4)$

$$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$$

(c) $(+3) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline \underline{1}1011 = -5 \end{array}$$

(d) $(-4) + (-1)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$$

(e) $(+5) + (+4)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline \underline{1}0011 = \text{Overflow} \end{array}$$

(f) $(-7) + (-6)$

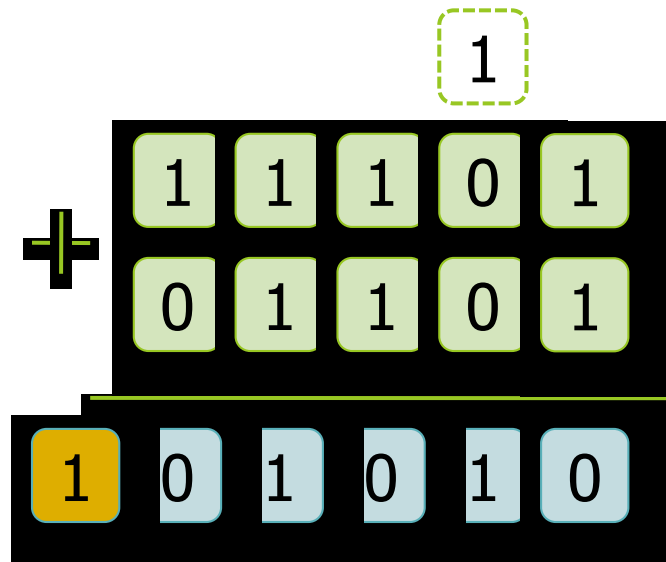
Phép trừ

32

- Nguyên tắc cơ bản: Đưa về phép cộng

$$A - B = A + (-B) = A + (\text{số bù 2 của } B)$$

- Ví dụ: $11101 - 10011 = 11101 + 01101$



Phép trừ

33

$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$$

(a) M = 2 = 0010
S = 7 = 0111
-S = 1001

$$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$$

(b) M = 5 = 0101
S = 2 = 0010
-S = 1110

$$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$$

(c) M = -5 = 1011
S = 2 = 0010
-S = 1110

$$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$$

(d) M = 5 = 0101
S = -2 = 1110
-S = 0010

$$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$$

(e) M = 7 = 0111
S = -7 = 1001
-S = 0111

$$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

(f) M = -6 = 1010
S = 4 = 0100
-S = 1100

Phép nhân

34

- Nguyên tắc cơ bản:

		0	1
0	0	0	0
1	0	1	

Phép nhân

36

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array} \quad \begin{array}{l} = 11 \\ = 13 \\ \\ \\ \\ = 143 \end{array}$$

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 00000000 \\ + 1011 \\ \hline 00001011 \\ + 0000 \\ \hline 00001011 \\ + 1011 \\ \hline 00110111 \\ + 1011 \\ \hline 10001111 \end{array}$$

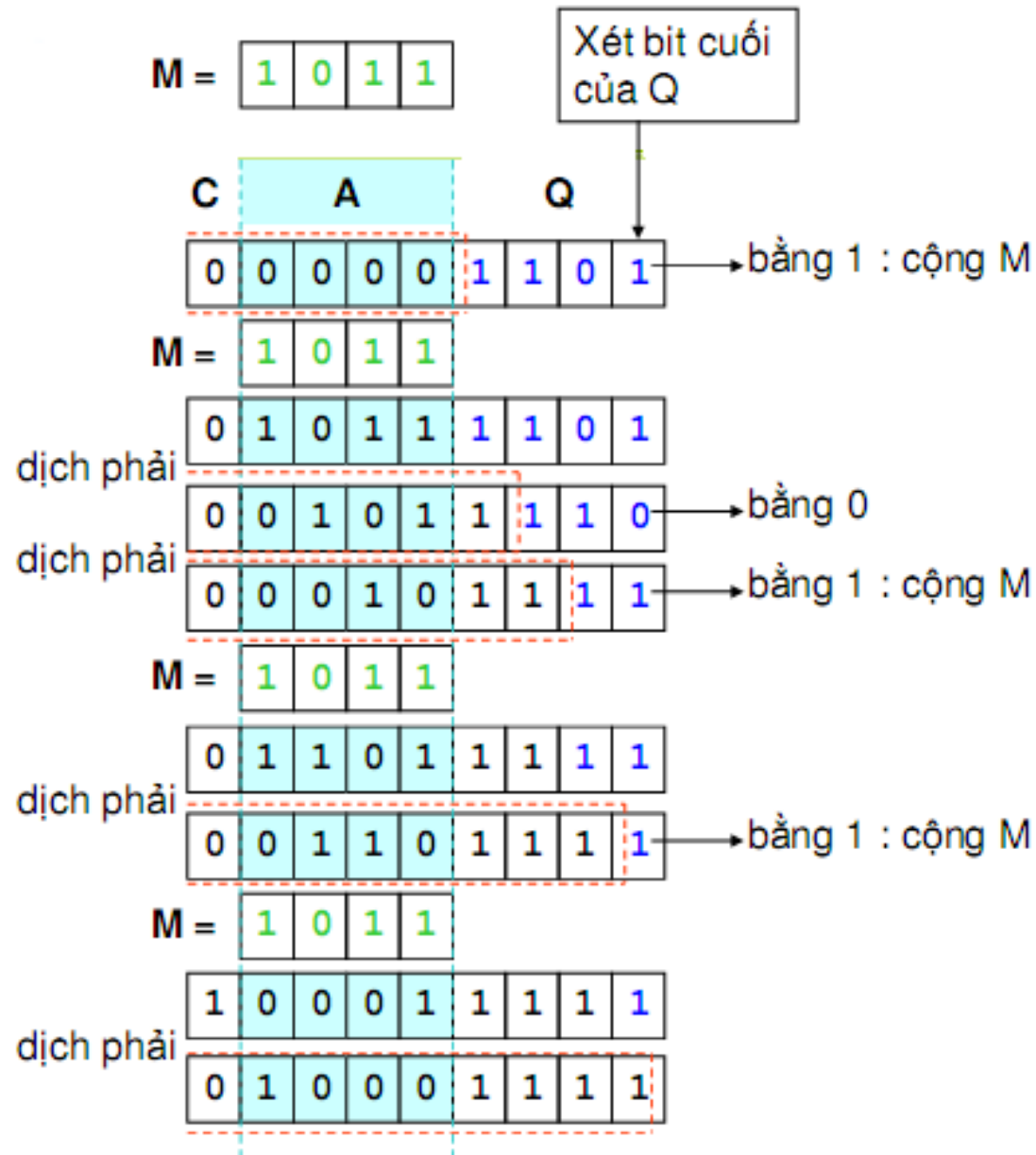
Thuật toán nhân

37

- Giả sử ta muốn thực hiện phép nhân $M \times Q$ với
 - Q có n bit
- Ta định nghĩa các biến:
 - C (1 bit): đóng vai trò bit nhớ
 - A (n bit): đóng vai trò 1 phần kết quả nhân ($[C, A, Q]$: kết quả nhân)
 - $[C, A]$ ($n + 1$ bit) ; $[C, A, Q]$ ($2n + 1$ bit): coi như các thanh ghi ghép
- Thuật toán:

```
Khởi tạo:  $[C, A] = 0$ ;  $k = n$ 
Lặp khi  $k > 0$ 
{
    Nếu bit cuối của  $Q = 1$  thì
        Lấy  $(A + M) \rightarrow [C, A]$ 

    Shift right  $[C, A, Q]$ 
     $k = k - 1$ 
}
```



Thuật toán nhân cải tiến (số không/có dấu)

39

Khởi tạo: $A = 0$; $k = n$; $Q_{-1} = 0$ (thêm 1 bit = 0 vào cuối Q)

Lặp khi $k > 0$

{

Nếu 2 bit cuối của Q_0Q_{-1}

{

= 10 thì $A - M \rightarrow A$

= 01 thì $A + M \rightarrow A$

= 00, 11 thì A không thay đổi

}

Shift right $[A, Q, Q_{-1}]$

$k = k - 1$

}

Kết quả: $[A, Q]$

Ví dụ

$$M = 7, Q = -3, n = 4$$

40

	A	Q	Q ₋₁	M
Khởi đầu	0000	1101	0	0111
Bước 0: $A=A-M$	1001	1101	0	0111
shift	1100	1110	1	0111
Bước 1: $A=A+M$	0011	1110	1	0111
shift	0001	1111	0	0111
Bước 2: $A=A-M$	1010	1111	0	0111
shift	1101	0111	1	0111
Bước 3: shift	1110	1011	1	0111

Kết quả 11101011 = -21

Phép chia

41

□ Giả sử ta muốn thực hiện Q / M với

Khởi tạo: $A = n$ bit 0 nếu $Q > 0$; $A = n$ bit 1 nếu $Q < 0$; $k = n$

Lặp khi $k > 0$

{

Shift left (SHL) $[A, Q]$

$A - M \rightarrow A$

Nếu $A < 0$: $Q_0 = 0$ và $A + M \rightarrow A$

Ngược lại: $Q_0 = 1$

$k = k - 1$

}

Kết quả: Q là thương, A là số dư

Ví dụ phép chia

42

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Initial value	0000	0111	Initial value
0000	1110	Shift	0000	1110	Shift
1101		Subtract	1101		Add
0000	1110	Restore	0000	1110	Restore
0001	1100	Shift	0001	1100	Shift
1110		Subtract	1110		Add
0001	1100	Restore	0001	1100	Restore
0011	1000	Shift	0011	1000	Shift
0000		Subtract	0000		Add
0000	1001	Set $Q_0 = 1$	0000	1001	Set $Q_0 = 1$
0001	0010	Shift	0001	0010	Shift
1110		Subtract	1110		Add
0001	0010	Restore	0001	0010	Restore

(a) (7)/(3)

(b) (7)/(-3)

Prefix in byte (Chuẩn IEC)

43

- International Electrotechnical Commission (IEC)

Name	Abbr	Factor
kibi	Ki	$2^{10} = 1,024$
mebi	Mi	$2^{20} = 1,048,576$
gibi	Gi	$2^{30} = 1,073,741,824$
tebi	Ti	$2^{40} = 1,099,511,627,776$
pebi	Pi	$2^{50} = 1,125,899,906,842,624$
exbi	Ei	$2^{60} = 1,152,921,504,606,846,976$
zebi	Zi	$2^{70} = 1,180,591,620,717,411,303,424$
yobi	Yi	$2^{80} = 1,208,925,819,614,629,174,706,176$

Prefix in byte (Chuẩn SI)

44

- International System of Units (SI)

Name	Abbr	Factor	SI size
Kilo	K	$2^{10} = 1,024$	$10^3 = 1,000$
Mega	M	$2^{20} = 1,048,576$	$10^6 = 1,000,000$
Giga	G	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$
Tera	T	$2^{40} = 1,099,511,627,776$	$10^{12} = 1,000,000,000,000$
Peta	P	$2^{50} = 1,125,899,906,842,624$	$10^{15} = 1,000,000,000,000,000$
Exa	E	$2^{60} = 1,152,921,504,606,846,976$	$10^{18} = 1,000,000,000,000,000,000$
Zetta	Z	$2^{70} = 1,180,591,620,717,411,303,424$	$10^{21} = 1,000,000,000,000,000,000,000$
Yotta	Y	$2^{80} = 1,208,925,819,614,629,174,706,176$	$10^{24} = 1,000,000,000,000,000,000,000,000$

- **Chú ý:** khi nói "kilobyte" chúng ta nghĩ là 1024 byte nhưng thực ra nó là 1000 bytes theo chuẩn SI, 1024 bytes là kibibyte (IEC)
- Hiện nay chỉ có các nhà sản xuất đĩa cứng và viễn thông mới dùng chuẩn SI
 - **30 GB** → $30 * 10^9 \sim \mathbf{28} * 2^{30}$ bytes
 - 1 Mbit/s → 10^6 b/s

Homework

45

- Đọc chương 9, sách của W.Stalling
- Đọc trước slide bài giảng số thực

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Võ Minh Trí – vmtri@fit.hcmus.edu.vn

03 – Biểu diễn số thực

Đặt vấn đề

2

- Biểu diễn số 123.375_{10} sang hệ nhị phân?
 - Ý tưởng đơn giản: Biểu diễn phần nguyên và phần thập phân riêng lẻ
 - Với phần nguyên: Dùng 8 bit ($[0_{10}, 255_{10}]$)
 $123_{10} = 64 + 32 + 16 + 8 + 2 + 1 = 0111\ 1011_2$
 - Với phần thập phân: Tương tự dùng 8 bit
 $0.375 = 0.25 + 0.125 = 2^{-2} + 2^{-3} = 0110\ 0000_2$
- $123.375_{10} = 0111\ 1011.0110\ 0000_2$
- Tổng quát công thức khai triển của số thập phân hệ nhị phân:

$$\underbrace{x_{n-1}x_{n-2}\dots x_0}_{\text{Phần nguyên}} \cdot \underbrace{x_{-1}x_{-2}\dots x_{-m}}_{\text{Phần thập phân}} = \underbrace{x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} \dots + x_0 \cdot 2^0}_{\text{Phần nguyên}} + \underbrace{x_{-1} \cdot 2^{-1} + x_{-2} \cdot 2^{-2} + \dots + x_{-m} \cdot 2^{-m}}_{\text{Phần thập phân}}$$

Đặt vấn đề

3

- **Tuy nhiên...với 8 bit:**
 - Phần nguyên lớn nhất có thể biểu diễn: 255
 - Phần thập phân nhỏ nhất có thể biểu diễn: $2^{-8} \sim 10^{-3} = 0.001$
- **Biểu diễn số nhỏ như $0.0001 (10^{-4})$ hay $0.000001 (10^{-5})$?**
- Một giải pháp: Tăng số bit phần thập phân
 - Với **16 bit** cho phần thập phân: $\text{min} = 2^{-16} \sim 10^{-5}$
 - Có vẻ không hiệu quả...Cách tốt hơn ?
- **Floating Point Number (Số thực dấu chấm động)**

Floating Point Number ?

4

- Giả sử ta có số (ở dạng nhị phân)

$$X = 0.\underbrace{0000000000000000}_{14 \text{ số } 0}11_2 = (2^{-15} + 2^{-16})_{10}$$

→ $X = 0.11_2 * (2^{-14})_{10} (= (2^{-1} + 2^{-2}).2^{-14} = 2^{-15} + 2^{-16})$

- Thay vì dùng 16 bit để lưu trữ phần thập phân, ta có thể chỉ cần 6 bit:

$$X = 0.11 \ 1110$$

- Cách làm: Di chuyển vị trí dấu chấm sang phải 14 vị trí, dùng 4 bit để lưu trữ số 14 này
- Đây là ý tưởng **cơ bản** của số thực dấu chấm động (floating point number)

Chuẩn hóa số thập phân

5

- Trước khi các số được biểu diễn dưới dạng số chấm động, chúng cần được chuẩn hóa về dạng: $\pm 1.F * 2^E$
 - F : Phần thập phân không dấu (định trị - Significant)
 - E : Phần số mũ (Exponent)
- Ví dụ:
 - $+0.09375_{10} = 0.00011_2 = +1.1 * 2^{-4}$
 - $-5.25_{10} = 101.01_2 = -1.0101 * 2^2$

Biểu diễn số chấm động

6

- Có nhiều chuẩn nhưng hiện nay chuẩn IEEE 754 được dùng nhiều nhất để lưu trữ số thập phân theo dấu chấm động trong máy tính, gồm 2 dạng:
(slide sau)

Biểu diễn số chấm động

7

- Số chấm động chính xác đơn (32 bits):



- Số chấm động chính xác kép (64 bits):



- **Sign:** Bit dấu (1: Số âm, 0: Số dương)
- **Exponent:** Số mũ (Biểu diễn dưới dạng số quá K (Biased) với
 - Chính xác đơn: $K = 127$ ($2^{n-1} - 1 = 2^{8-1} - 1$) với n là số bit lưu trữ Exponent
 - Chính xác kép: $K = 1023$ ($2^{n-1} - 1 = 2^{11-1} - 1$)
- **Significand (Fraction):** Phần định trị (phần lẻ sau dấu chấm)

Ví dụ

8

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **X = -5.25**

- **Bước 1:** Đổi X sang hệ nhị phân

$$X = -5.25_{10} = -101.01_2$$

- **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -5.25 = -101.01 = -1.0101 * 2^2$$

- **Bước 3:** Biểu diễn Floating Point

- Số âm: bit dấu Sign = 1

- Số mũ E = 2 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 2 + 127 = 129_{10} = 1000\ 0001_2$$

- Phần định trị = 0101 0000 0000 0000 0000 000 (Thêm 19 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 1 1000 0001 0101 0000 0000 0000 0000 000

Câu hỏi

9

- Vì sao phần số mũ exponent không giữ nguyên lại phải lưu trữ dưới dạng số quá K (Dạng biased)?

Đáp án

10

- Sở dĩ Exponent được lưu trữ dưới dạng Biased vì ta muốn chuyển từ miền giá trị **số có dấu** sang **số không dấu** (vì trong biased, số k được chọn để sau khi cộng số bất kỳ trong miền giá trị gốc, kết quả là số luôn dương)
→ Dễ dàng so sánh, tính toán

Câu hỏi

11

- Khi muốn biểu diễn số 0 thì ta không thể tìm ra bit trái nhất có giá trị = 1 để đẩy dấu chấm động, vậy làm sao chuẩn hóa về dạng $\pm 1.F * 2^E$?
 - Với số dạng $\pm 0.F * 2^{-127}$ thì chuẩn hóa được nữa không?
 - Với $K = 127$, exponent lớn nhất sẽ là 255
- Số mũ gốc ban đầu lớn nhất là $255 - 127 = +128$
- **Vô lý** vì với 8 bit có dấu ta không thể biểu diễn được số +128 ?

Đáp án

12

- Vì đó là những **số thực đặc biệt**, ta không thể biểu diễn bằng dấu chấm động 😊

Số thực đặc biệt

13

- Số 0 (zero)
 - Exponent = 0, Significand = 0
- Số không thể chuẩn hóa (denormalized)
 - Exponent = 0, Significand \neq 0
- Số vô cùng (infinity)
 - Exponent = 111...1 (toàn bit 1), Significand = 0
- Số báo lỗi (NaN – Not a Number)
 - Exponent = 111...1 (toàn bit 1), Significand \neq 0

Normalized number

14

- Largest positive normalized number: $+1.[23 \text{ số } 1] * 2^{127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	1111 1110	1111 1111 1111 1111 1111 111

- Smallest positive normalized number: $+1.[23 \text{ số } 0] * 2^{-126}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0001	0000 0000 0000 0000 0000 000

- Tương tự cho số negative (số âm)

Denormalized number

15

- Largest positive denormalized number: $+0.[23 \text{ số } 1] * 2^{-127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0000	1111 1111 1111 1111 1111 111

Tuy nhiên IEEE 754 quy định là $+0.[23 \text{ số } 1] * 2^{-126}$ vì muốn tiến gần hơn với "Smallest positive normalized number = $+1.[23 \text{ số } 0] * 2^{-126}$ "

- Smallest positive denormalized number: $+1.[22 \text{ số } 0]1 * 2^{-127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0000	0000 0000 0000 0000 0000 001

Tuy nhiên IEEE 754 quy định là $+0.[22 \text{ số } 0]1 * 2^{-126}$

- Tương tự cho số negative (số âm)

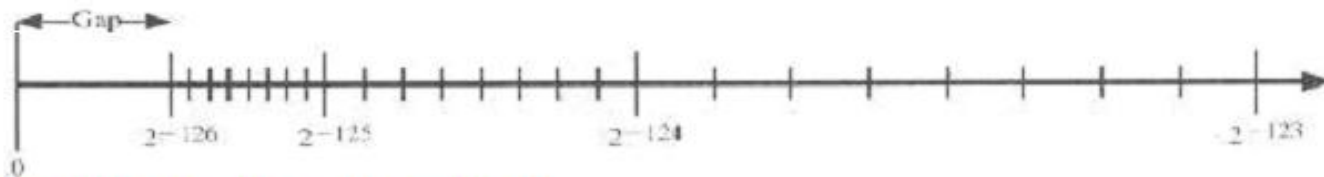
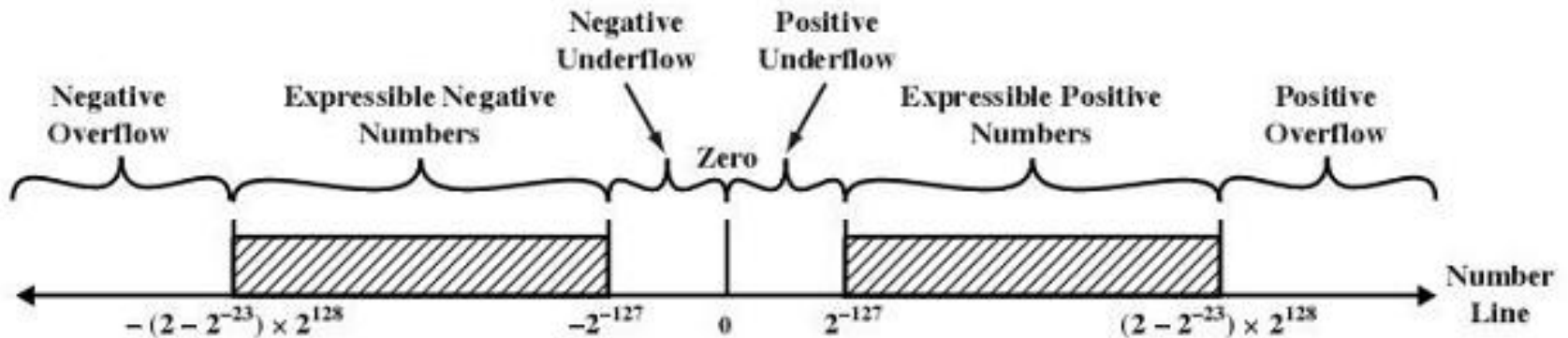
Ví dụ: $n = 4, m = 3, \text{bias} = 7$

16

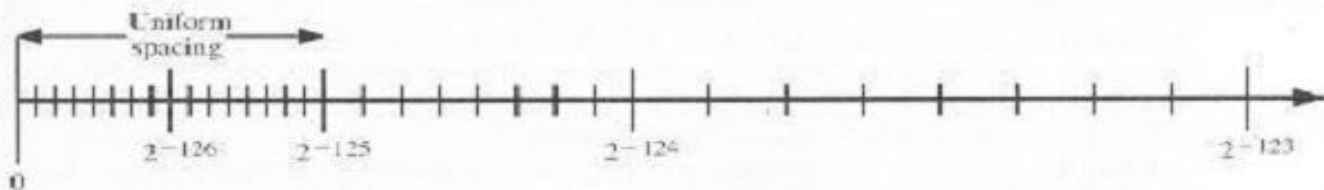
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	← closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	← largest denorm
					
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	← smallest norm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	← closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	← closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
0	1110	111	7	$15/8 * 128 = 240$	← largest norm	
.....						
0	1111	000	n/a	inf		

Phân bố các số thực (32 bits)

17



Without denormalized numbers



With denormalized numbers

Chuẩn IEEE 754

18

Parameter	Format			
	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥ 43	64	≥ 79
Exponent width (bits)	8	≥ 11	11	≥ 15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥ 1023	1023	≥ 16383
Minimum exponent	-126	≤ -1022	-1022	≤ -16382
Number range (base 10)	$10^{-38}, 10^{+38}$	unspecified	$10^{-308}, 10^{+308}$	unspecified
Significand width (bits)*	23	≥ 31	52	≥ 63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

* not including implied bit

Bài tập 1

19

□ Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **$X = +12.625$**

□ **Bước 1:** Đổi X sang hệ nhị phân

$$X = -12.625_{10} = -1100.101_2$$

□ **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -12.625_{10} = -1100.101_2 = -1.100101 * 2^3$$

□ **Bước 3:** Biểu diễn Floating Point

□ Số dương: bit dấu Sign = 0

□ Số mũ E = 3 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 3 + 127 = 130_{10} = 1000\ 0010_2$$

□ Phần định trị = 1001 0100 0000 0000 0000 000 (Thêm 17 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 0 1000 0010 1001 0100 0000 0000 0000

Bài tập 2

20

□ Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **X = -3050**

□ **Bước 1:** Đổi X sang hệ nhị phân

$$X = -3050_{10} = -1011\ 1110\ 1010_2$$

□ **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -3050_{10} = -1011\ 1110\ 1010_2 = -1.01111101010 * 2^{11}$$

□ **Bước 3:** Biểu diễn Floating Point

□ Số âm: bit dấu Sign = 1

□ Số mũ E = 11 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 11 + 127 = 138_{10} = 1000\ 1010_2$$

□ Phần định trị = 0111 1101 0100 0000 0000 000 (Thêm 12 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 1 1000 1010 0111 1101 0100 0000 0000

Bài tập 3

21

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): $X = +1.1 * 2^{-128}$
 - **Lưu ý:**
 - Số X: positive number
 - $X < \text{Smallest positive normalized number: } +1.[23 \text{ số } 0] * 2^{-126}$
 - số X là số không thể chuẩn hóa (denormalized number)
 - **Chuyển X về dạng: $X = +0.011 * 2^{-126}$**
 - **Bước 3:** Biểu diễn Floating Point
 - Số dương: bit dấu Sign = 0
 - Vì đây là số không thể chuẩn hóa → Phần mũ exponent được biểu diễn: $0000\ 0000_2$
 - Phần định trị = $0110\ 0000\ 0000\ 0000\ 0000\ 0000$
- Kết quả nhận được: $0\ 0000\ 0000\ 0110\ 0000\ 0000\ 0000\ 0000$

Homework

22

- Sách W.Stalling – Computer Arithmetic, đọc chương 9
- Đọc file 04_FloatingPoint.doc
- Trả lời các câu hỏi:
 - Overflow, underflow?
 - Cộng trừ nhân chia trên số thực?
 - Quy tắc làm tròn?
 - NaN: nguyên tắc phát sinh?
 - Quiet NaN và Signaling NaN?

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Vũ Minh Trí – vmtri@fit.hcmus.edu.vn

04 – Lập trình hợp ngữ (Phần 1)

Ngôn ngữ lập trình

2

- Là loại ngôn ngữ **nhân tạo** (Ví dụ: C/C++) được cấu thành bởi 2 yếu tố chính:
 - **Từ vựng:** là các keyword (struct, enum, if, int...)
 - **Ngữ pháp:** syntax (if(...){} else{}, do{} while()...)
- Ngôn ngữ lập trình giúp cho người sử dụng nó (gọi là lập trình viên) có thể diễn đạt và mô tả các hướng dẫn cho máy tính hoạt động theo ý muốn của mình
- Độ phức tạp (trừu tượng) của các hướng dẫn này quyết định thứ bậc của ngôn ngữ
 - **Độ phức tạp càng cao thì bậc càng thấp**
 - Ví dụ: C Sharp (C#) là ngôn ngữ bậc cao hơn C

Nhận xét

3

- Ngôn ngữ nào mà con người dễ hiểu nhất lại là ngôn ngữ máy tính “khó hiểu” nhất
 - Ngôn ngữ bậc càng cao thì con người càng dễ hiểu nhưng máy tính lại càng “khó hiểu”
- Nhưng máy tính lại là nơi chúng ta cần nó hiểu đúng và nhanh nhất để có thể thực thi những gì chúng ta muốn

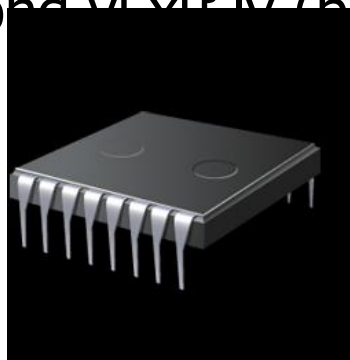
→ **Ngôn ngữ máy (Machine language)**



Ngôn ngữ máy (Machine Language)

4

- Ngôn ngữ máy cho phép người lập trình đưa ra các **hướng dẫn đơn giản** mà bộ vi xử lý (CPU) có thể **thực hiện được ngay**
- Các hướng dẫn này được gọi là chỉ thị / lệnh (**instruction**) hoặc mã máy (**machine code**)
- Mỗi bộ vi xử lý (CPU) có 1 ngôn ngữ riêng, gọi là bộ lệnh (**instruction set**)
- Trong cùng 1 dòng vi xử lý (processor family) bộ lệnh gần giống nhau



Instruction

5

- Là dãy bit chứa yêu cầu mà bộ xử lý trong CPU (ALU) phải thực hiện
- Instruction gồm 2 thành phần:
 - Mã lệnh (opcode): thao tác cần thực hiện
 - Thông tin về toán hạng (operand): các đối tượng bị tác động bởi thao tác chứa trong mã lệnh

ISA (Instruction Set Architecture)

6

- Tập lệnh dành cho những bộ vi xử lý có kiến trúc tương tự nhau
- Một số ISA thông dụng:
 - Dòng vi xử lý 80x86 (gọi tắt x86) của Intel
 - IA-16: Dòng xử lý 16 bit (Intel 8086, 80186, 80286)
 - IA-32: Dòng xử lý 32 bit (Intel 80368 – i386, 80486 – i486, Pentium II, Pentium III ...)
 - IA-64: Dòng xử lý 64 bit (Intel x86-64 như Pentium D...)
 - MIPS: Dùng rất nhiều trong hệ thống nhúng (embedded system)
 - PowerPC của IBM

Thiết kế ISA: CISC & RISC

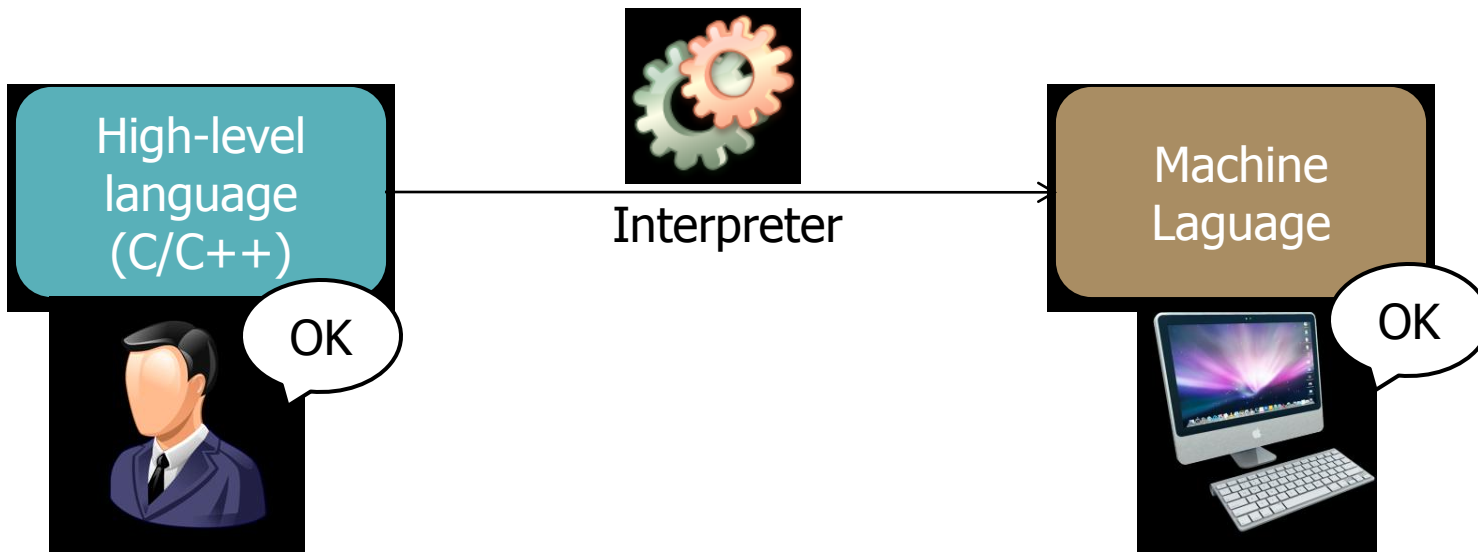
7

- Có 2 trường phái thiết kế bộ lệnh:
 - **Complete Instruction Set Computer (CISC):** bộ lệnh gồm rất nhiều lệnh, từ đơn giản đến phức tạp
 - **Reduced Instruction Set Computer (RISC):** bộ lệnh chỉ gồm các lệnh đơn giản
- Nên chọn kiểu nào?

Tuy nhiên

8

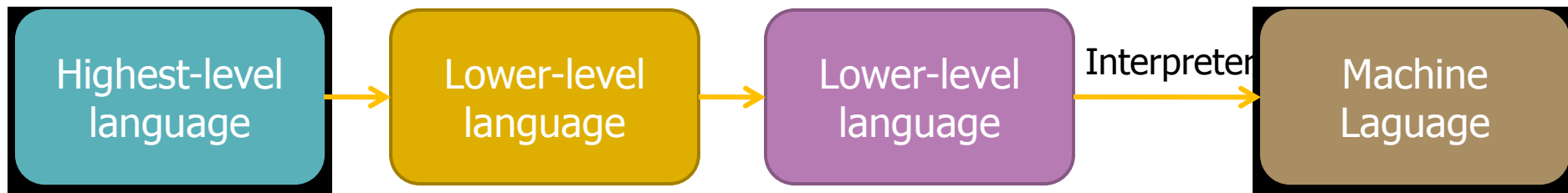
- Không phải ai cũng muốn / có thể lập trình ngôn ngữ máy vì quá khó hiểu so với ngôn ngữ bình thường của con người
- Nhu cầu cần có **bộ phận phiên dịch (interpreter)**



Nhận xét

9

- Trong 1 số trường hợp, việc viết bằng ngôn ngữ cấp “quá cao” trở nên chạy **khá chậm** vì phải phiên dịch **nhiều lần** để trở thành ngôn ngữ máy
→ **Hợp ngữ (Assembly language)**



Hợp ngữ

10

- Các mã máy chỉ là các con số (0 / 1)
- Trong ngôn ngữ máy không có khái niệm biến → thay vào đó là địa chỉ ô nhớ, thanh ghi (lưu trữ mã lệnh, dữ liệu)
- Để dễ dàng lập trình hơn → dùng ký hiệu **mã giả** thay cho các số biểu diễn địa chỉ ô nhớ, các tên (label, tên biến, tên chương trình)
- **Hợp ngữ rất gần với ngôn ngữ máy nhưng lại đủ để con người hiểu và sử dụng tốt hơn ngôn ngữ máy**

■ Ví dụ: Ghi giá trị 5 vào thanh ghi \$4

Ngôn ngữ máy: 00110100 0000100 00000000 00000101

Hợp ngữ : ori \$4, \$0, 5

Lưu ý

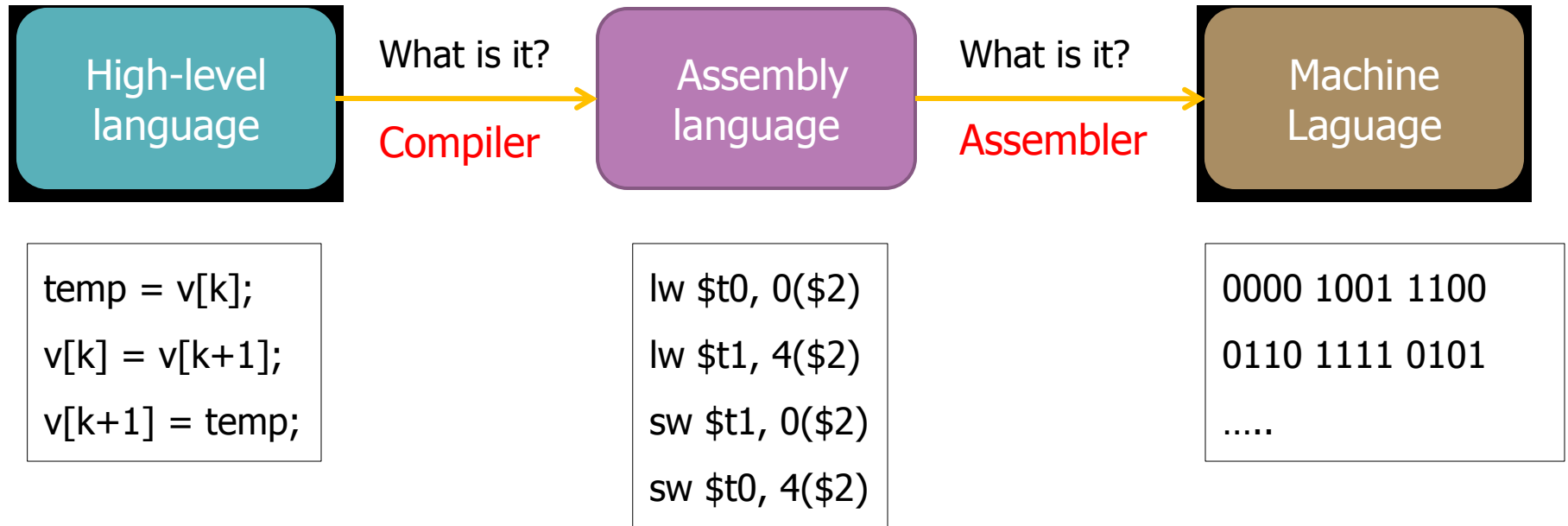
11

- Vì mỗi bộ vi xử lý có 1 cấu trúc thanh ghi và tập lệnh (ngôn ngữ) riêng nên khi lập trình hợp ngữ phải nói rõ là **lập trình cho bộ vi xử lý nào, hay dòng (family) vi xử lý nào**
 - Ví dụ:
 - Hợp ngữ cho MIPS
 - Hợp ngữ cho dòng vi xử lý Intel 80x86

Thảo luận

12

- Ta có thể hình dung như sau:



Compiler

13

- **Trình biên dịch ngôn ngữ cấp cao → hợp ngữ**
- **Compiler phụ thuộc vào:**
 - **Ngôn ngữ cấp cao được biên dịch**
 - **Kiến trúc hệ thống phần cứng bên dưới mà nó đang chạy**
 - **Ví dụ:**
 - **Compiler cho C <> Compiler cho Java**
 - **Compiler cho "C on Windows" <> "C on Linux"**

Assembler

14

- **Trình biên dịch hợp ngữ → ngôn ngữ máy**
- Một bộ vi xử lý (đi kèm 1 bộ lệnh xác định) có thể có nhiều Assembler của nhiều nhà cung cấp khác nhau chạy trên các OS khác nhau
 - *Ví dụ: Cùng là kiến trúc x86, nhưng có thể dùng A86, GAS, TASM, MASM, NASM*
- **Assembly program phụ thuộc vào Assembler mà nó sử dụng** (do các mở rộng, đặc điểm khác nhau giữa các Assembler)

Thảo luận

15

- Bản thân Compiler cũng là chương trình, vậy nó được biên dịch bằng gì?

→ **Assembler**

- Sau khi đã biên dịch tập tin mã nguồn ngôn ngữ cấp cao thành tập tin mã máy (machine language), làm sao để chạy những tập tin này trên máy tính?

→ **Linker & Loader**

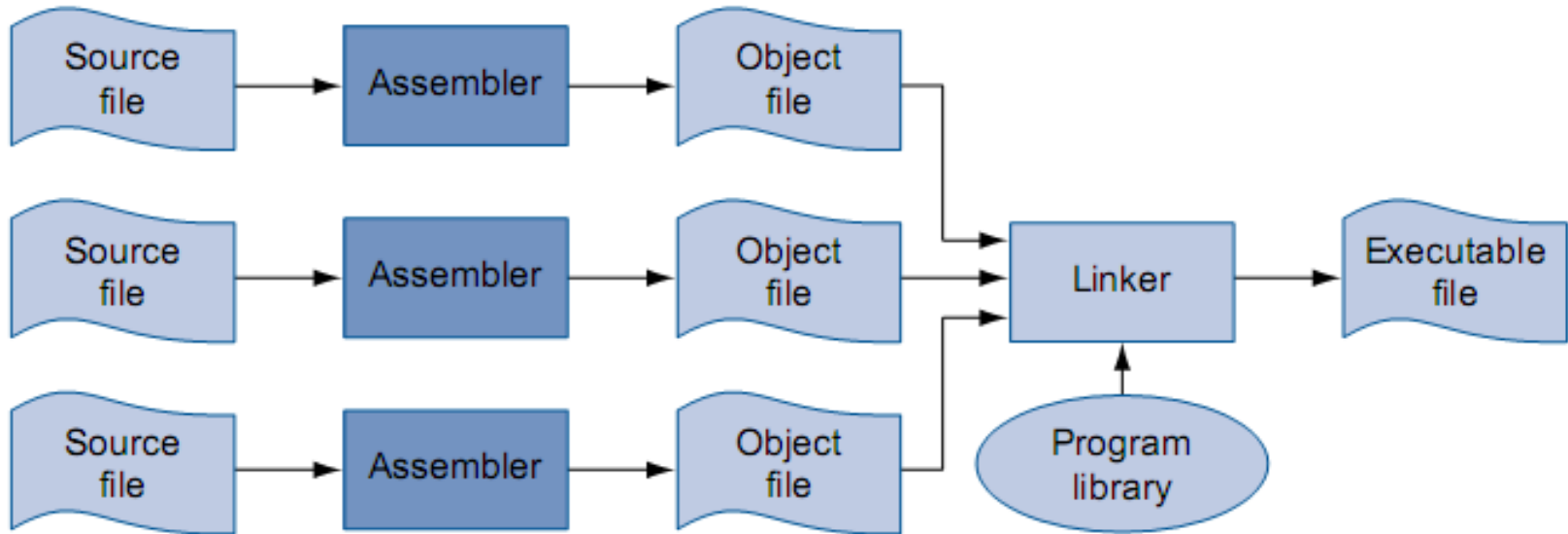
Linker

16

- Thực tế khi lập trình, ta sẽ dùng nhiều file (header / source) liên kết và kèm theo các thư viện có sẵn
- Cần chương trình Linker để **liên kết các file** sau khi đã biên dịch thành mã máy này (**Object file**)
- *Tập tin thực thi (ví dụ: .exe, .bat, .sh)*

Quá trình tạo file thực thi

17

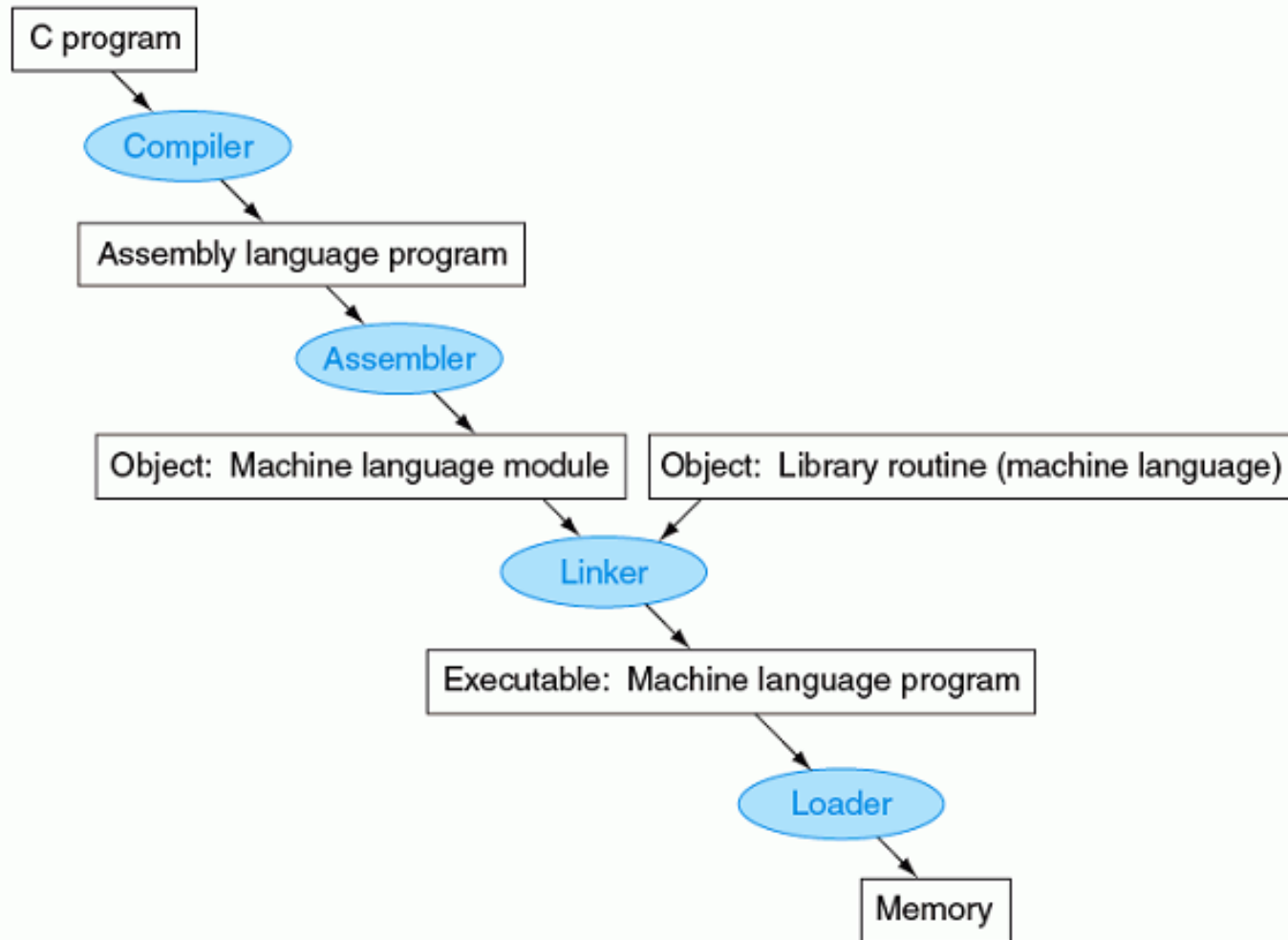


- Khi double click vào những tập tin thực thi, cần chương trình **tính toán và tải vào memory** để CPU xử lý

→ Loader

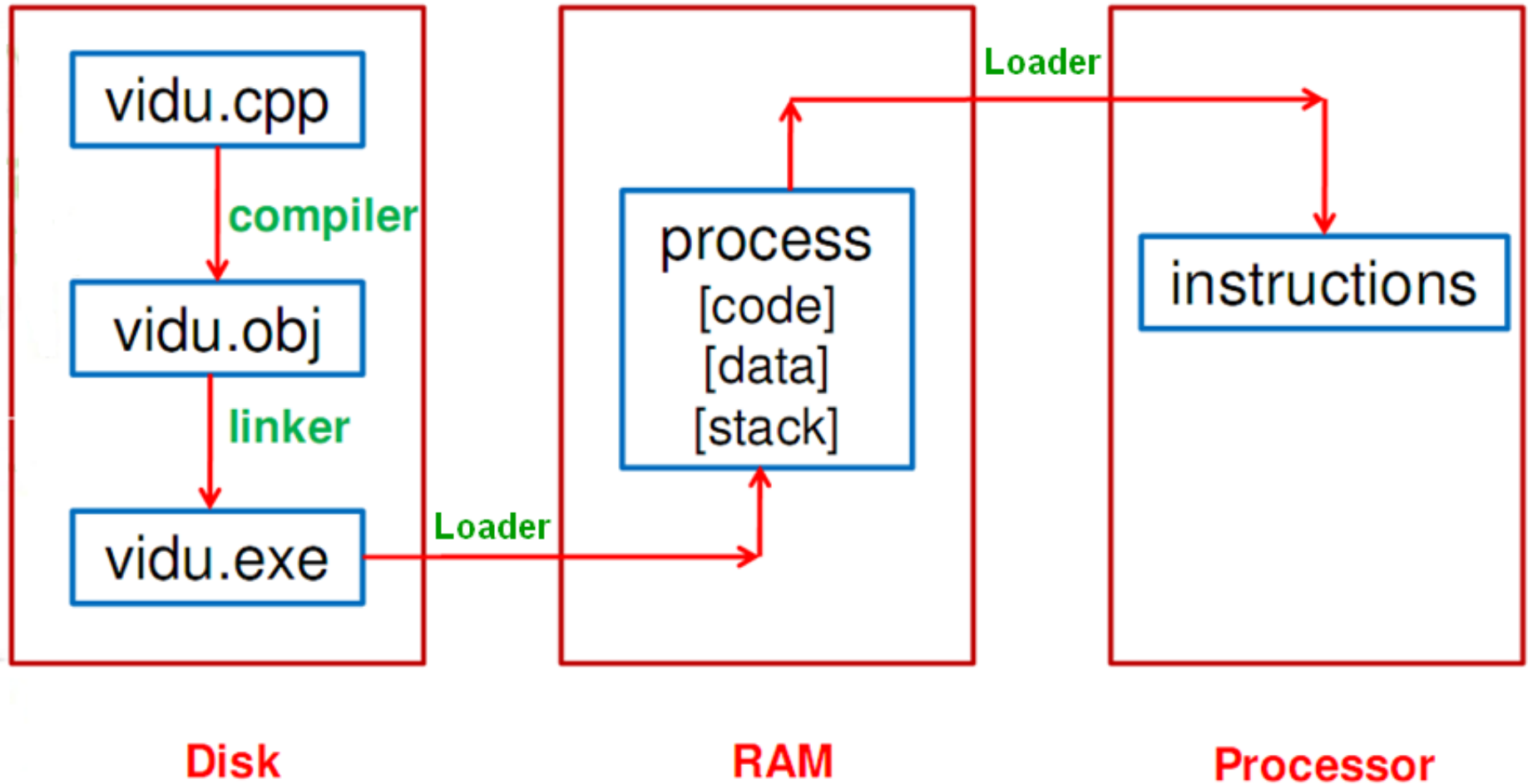
Quá trình thực thi file trên máy

18



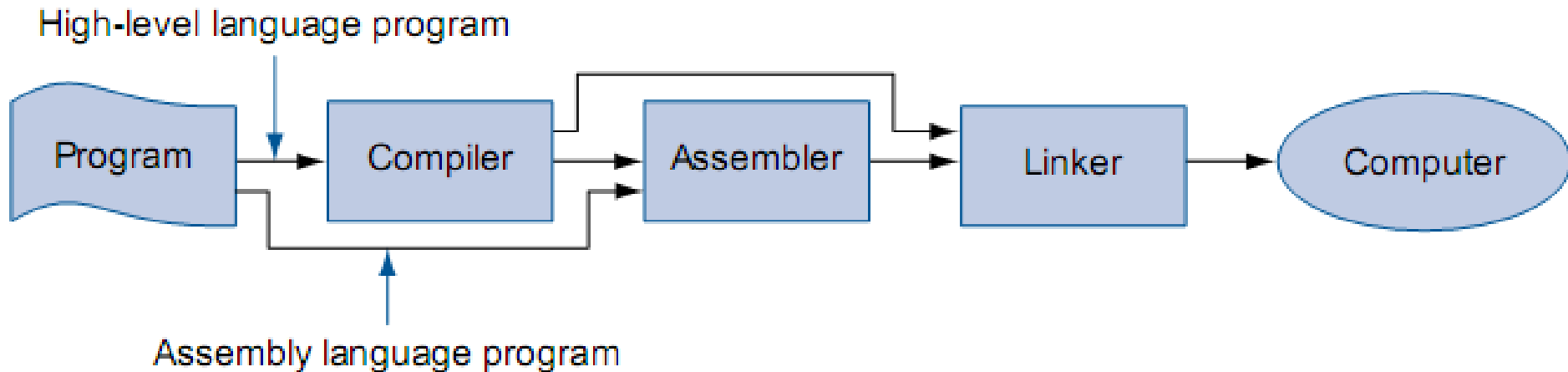
Ví dụ

19



Mô hình thực tế

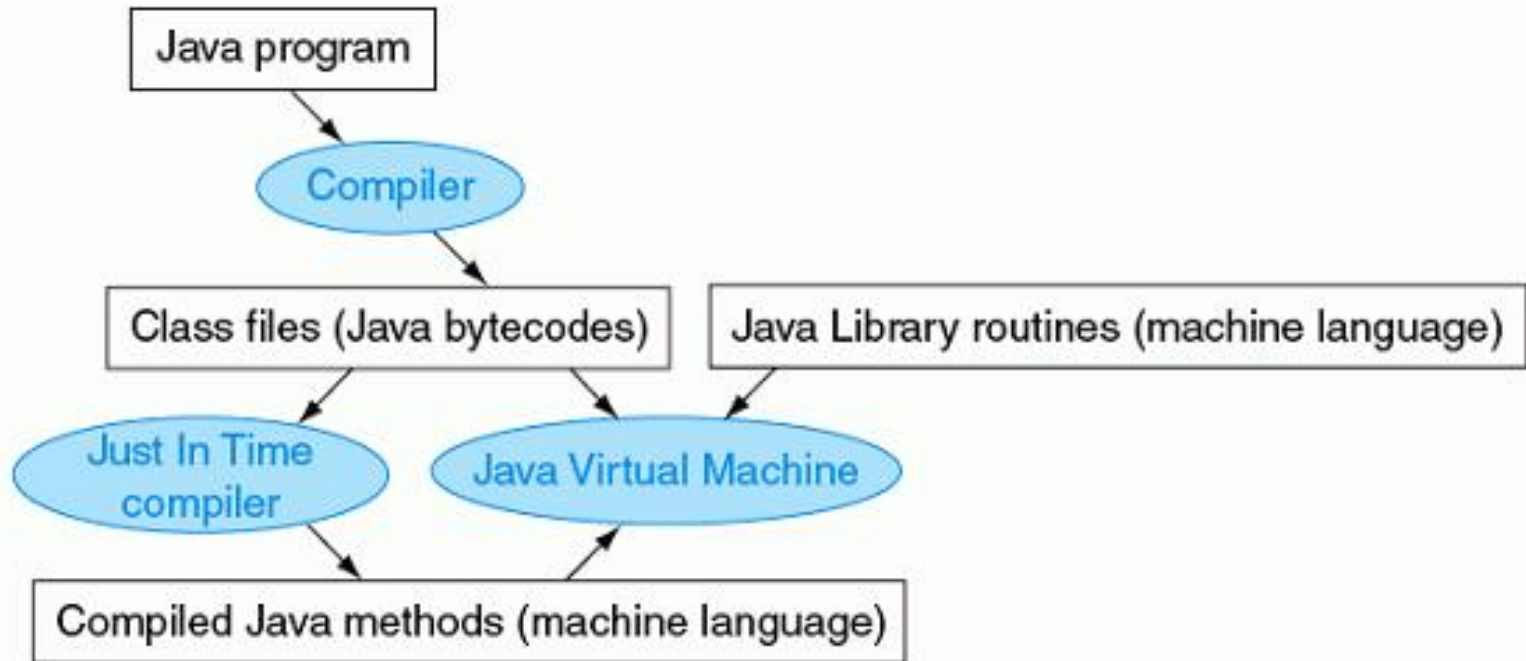
20



- Compiler và Assembler có thể được bỏ qua trong 1 số trường hợp cụ thể...
- Trong thực tế, có 1 số compiler có thể tạo file thực thi ở nhiều nền tảng kiến trúc bên dưới khác nhau, được gọi là **cross-platform compiler**
 - Compiler cho Java
 - Cygwin
 - Code::Block Studio

Quá trình thực thi file trên máy Java program

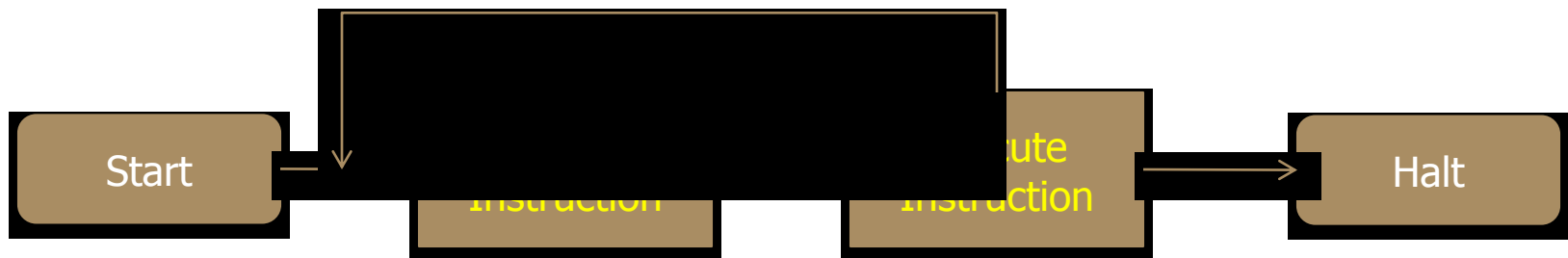
21



Hoạt động của CPU khi xử lý lệnh

22

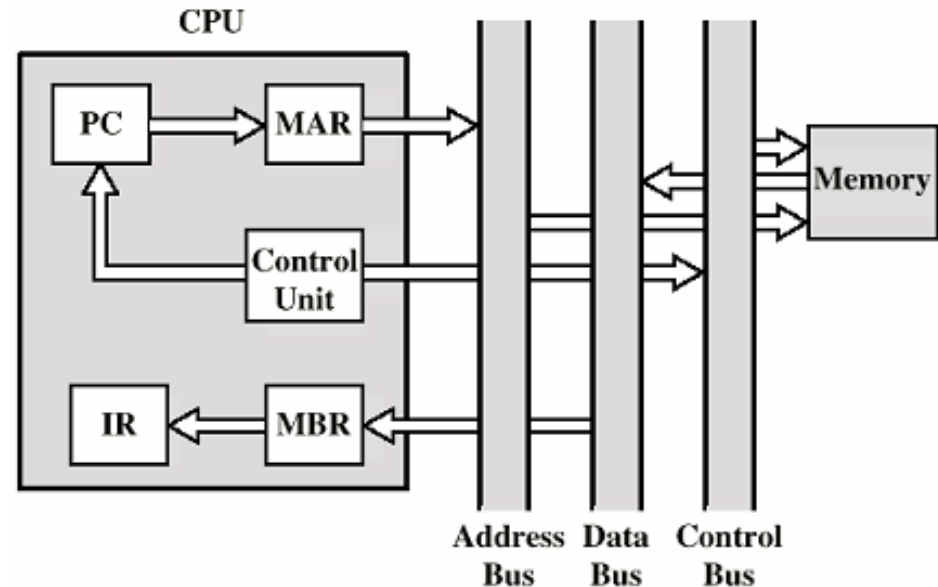
- CPU xử lý lệnh qua 2 bước, gọi là chu kỳ lệnh:
 - ▣ **Nạp lệnh (Fetch):** Di chuyển lệnh từ memory vào thanh ghi (register) trong CPU
 - ▣ **Thực thi lệnh (Excute):** Giải mã lệnh và thực thi thao tác yêu cầu



Quá trình nạp lệnh (Fetch cycle)

23

- $MAR \leftarrow PC$
- $MBR \leftarrow \text{Memory}$
- $IR \leftarrow MBR$
- $PC \leftarrow PC + 1$

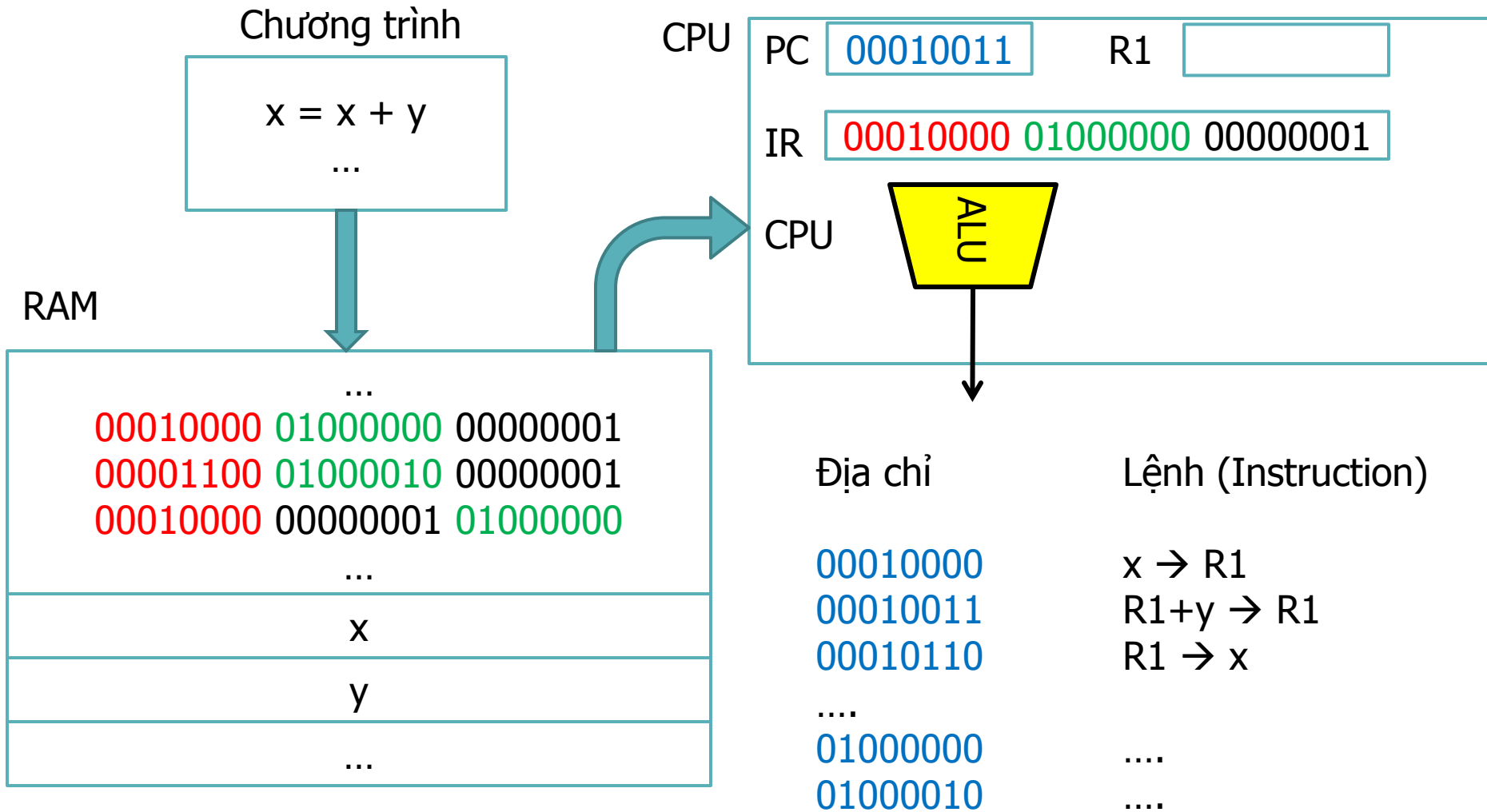


- Thanh ghi **PC (Program Counter)**
 - Lưu địa chỉ (address) của lệnh sắp được nạp
- Thanh ghi **MAR (Memory Address Register)**
 - Lưu địa chỉ (address) sẽ được output ra Address bus
- Thanh ghi **MBR (Memory Buffer Register)**
 - Lưu giá trị (value) sẽ được input / output từ Data bus
- Thanh ghi **IR (Instruction Register)**
 - Lưu mã lệnh sẽ được xử lý tiếp

- Control Unit di chuyển mã lệnh, có địa chỉ trong PC, vào thanh ghi IR
- Mặc định, giá trị thanh ghi PC sẽ tăng 1 lượng = chiều dài của lệnh vừa được nạp

Ví dụ

24



Một số câu hỏi

26

- Ngôn ngữ lập trình giống và khác ngôn ngữ tự nhiên của con người ở những điểm nào?
- Tại sao cần nhiều loại ngôn ngữ lập trình: C, C++, C#, VB, Java...?
- Một chương trình không khai báo biến nào cả thì có sử dụng bộ nhớ không?
- Chương trình được thực thi trong RAM hay CPU?
- Tại sao file .exe chỉ chạy được trên Windows mà không thể chạy trên Linux?

Homework

27

- Sách Petterson & Hennessy: Đọc chương 2 (đọc kỹ 2.12 và 2.13)
- Tài liệu tham khảo: Đọc "08_HP_AppA.pdf"

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Võ Minh Trí – vmtri@fit.hcmus.edu.vn

04 – Lập trình hợp ngữ (Phần 2)

Giới thiệu

2

- Nhiệm vụ cơ bản nhất của CPU là phải thực hiện các lệnh được yêu cầu, gọi là **instruction**
- Các CPU sẽ sử dụng các tập lệnh (instruction set) khác nhau để có thể giao tiếp với nó

Kích thước lệnh

3

- **Kích thước lệnh bị ảnh hưởng bởi:**
 - Cấu trúc đường truyền bus
 - Kích thước và tổ chức bộ nhớ
 - Tốc độ CPU
- **Giải pháp tối ưu lệnh:**
 - Dùng lệnh có kích thước ngắn, mỗi lệnh chỉ nên được thực thi trong đúng 1 chu kỳ CPU
 - Dùng bộ nhớ cache

Bộ lệnh MIPS

4

- Chúng ta sẽ làm quen với tập lệnh cho kiến trúc MIPS (PlayStation 1, 2; PSP; Windows CE, Routers...)
- Được xây dựng theo kiến trúc (RISC) với **4 nguyên tắc**:
 - Càng đơn giản, càng ổn định
 - Càng nhỏ gọn, xử lý càng nhanh
 - Tăng tốc xử lý cho những trường hợp thường xuyên xảy ra
 - Thiết kế đòi hỏi sự thỏa hiệp tốt

Cấu trúc cơ bản của 1 chương trình hợp ngữ trên MIPS

5

```
.data                # khai báo các data label (có thể hiểu là các biến)  
                    # sau chỉ thị này
```

```
label1: <kiểu lưu trữ> <giá trị khởi tạo>
```

```
label2: <kiểu lưu trữ> <giá trị khởi tạo>
```

...

```
.text                # viết các lệnh sau chỉ thị này
```

```
.globl <các text label toàn cục, có thể truy xuất từ các file khác>
```

```
.globl main          # Đây là text label toàn cục bắt buộc của program
```

...

```
main:                # điểm text label bắt đầu của program
```

...

Bộ lệnh MIPS – Thanh ghi

7

- Là đơn vị lưu trữ data duy nhất trong CPU
- Trong kiến trúc MIPS:
 - Có tổng cộng 32 thanh ghi đánh số từ \$0 → \$31
 - Càng ít càng dễ quản lý, tính toán càng nhanh
 - Có thể truy xuất thanh ghi qua tên của nó (slide sau)
 - Mỗi thanh ghi có kích thước cố định 32 bit
 - Bị giới hạn bởi khả năng tính toán của chip xử lý
 - Kích thước toán hạng trong các câu lệnh MIPS bị giới hạn ở 32 bit, nhóm 32 bit gọi là từ (word)

Thanh ghi toán hạng

8

- Như chúng ta đã biết khi lập trình, biến (variable) là khái niệm rất quan trọng khi muốn biểu diễn các toán hạng để tính toán
- Trong kiến trúc MIPS không tồn tại khái niệm biến, thay vào đó là thanh ghi toán hạng

Thanh ghi toán hạng

9

- Ngôn ngữ cấp cao (C, Java...): toán hạng = biến (variable)
 - Các biến lưu trong bộ nhớ chính
- Ngôn ngữ cấp thấp (Hợp ngữ): toán hạng chứa trong các thanh ghi
 - Thanh ghi không có kiểu dữ liệu
 - Kiểu dữ liệu thanh ghi được quyết định bởi thao tác trên thanh ghi
- So sánh:
 - **Ưu:** Thanh ghi truy xuất nhanh hơn nhiều bộ nhớ chính
 - **Khuyết:** Không như bộ nhớ chính, thanh ghi là phần cứng có số lượng giới hạn và cố định → Phải tính toán kỹ khi sử dụng

Một số thanh ghi toán hạng quan tâm

10

- **Save register:**
 - MIPS lấy ra 8 thanh ghi (\$16 - \$23) dùng để thực hiện các phép tính số học, được đặt tên tương ứng là **\$s0 - \$s7**
 - Tương ứng trong C, để chứa giá trị biến (variable)

- **Temporary register:**
 - MIPS lấy ra 8 thanh ghi (\$8 - \$15) dùng để chứa kết quả trung gian, được đặt tên tương ứng là **\$t0 - \$t7**
 - Tương ứng trong C, để chứa giá trị biến tạm (temporary variable)

Bảng danh sách thanh ghi MIPS

11

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Thanh ghi 1 (\$at) để dành cho assembler. Thanh ghi 26 – 27 (\$k0 - \$k1) để dành cho OS

Bộ lệnh MIPS – 4 thao tác chính

12

- Phần 1: Phép toán số học (Arithmetic)
- Phần 2: Di chuyển dữ liệu (Data transfer)
- Phần 3: Thao tác luận lý (Logical)
- Phần 4: Rẽ nhánh (Un/Conditional branch)

Phần 1: Phép toán số học

13

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

- **opt** (operator): Tên thao tác (toán tử, tác tử)
- **opr** (operand): Thanh ghi (toán hạng, tác tố đích)
chứa kết quả
- **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)
- **opr2** (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Ví dụ

14

- Giả sử xét câu lệnh sau:

`add a, b, c`

- ▣ Chỉ thị cho CPU thực hiện phép cộng

`a ← b + c`

- ▣ a, b, c được gọi là thanh ghi toán hạng
- ▣ Phép toán trên chỉ có thể thực hiện với đúng 3 toán hạng (không nhiều cũng không ít hơn)

Cộng, trừ số nguyên

15

□ Cộng (Add):

□ Cộng có dấu: `add $s0, $s1, $s2`

□ Cộng không dấu: `addu $s0, $s1, $s2` (u: unsigned)

□ Diễn giải: $\$s0 \leftarrow \$s1 + \$s2$

C/C++: $(a = b + c)$

□ Trừ (Subtract):

□ Trừ có dấu: `sub $s0, $s1, $s2`

□ Trừ không dấu: `subu $s0, $s1, $s2` (u: unsigned)

□ Diễn giải: $\$s0 \leftarrow \$s1 - \$s2$

C/C++: $(a = b - c)$

Nhận xét

16

- Toán hạng trong các lệnh trên phải là thanh ghi
- Trong MIPS, lệnh thao tác với số nguyên có dấu được biểu diễn dưới dạng bù 2
- Làm sao biết 1 phép toán được biên dịch từ C (ví dụ $a = b + c$) là thao tác có dấu hay không dấu? → Dựa vào trình biên dịch
- Có thể dùng 1 toán hạng vừa là nguồn vừa là đích
add \$s0, \$s0, \$s1
- Cộng, trừ với hằng số? → \$s2 sẽ đóng vai trò là hằng số
 - Cộng: addi \$s0, \$s1, 3 (addi = add immediate)
 - Trừ: addi \$s0, \$s1, -3

Ví dụ 1

17

- Chuyển thành lệnh MIPS từ lệnh C:

$$\mathbf{a = b + c + d - e}$$

- Chia nhỏ thành nhiều lệnh MIPS:

```
add  $s0, $s1, $s2      # a = b + c
```

```
add  $s0, $s0, $s3      # a = a + d
```

```
sub  $s0, $s0, $s4      # a = a - e
```

- Tại sao dùng nhiều lệnh hơn C?

→ Bị giới hạn bởi số lượng cổng mạch toán tử và thiết kế bên trong cổng mạch

- Ký tự “#” dùng để chú thích trong hợp ngữ cho MIPS

Ví dụ 2

18

- Chuyển thành lệnh MIPS từ lệnh C:

$$f = (g + h) - (i + j)$$

- Chia nhỏ thành nhiều lệnh MIPS:

```
add    $t0, $s1, $s2           # temp1 = g + h
```

```
add    $t1, $s3, $s4           # temp2 = i + j
```

```
sub    $s0, $t0, $t1           # f = temp1 - temp2
```

Lưu ý: Phép gán ?

19

- Kiến trúc MIPS không có cổng mạch dành riêng cho phép gán

→ Giải pháp: Dùng thanh ghi zero (\$0 hay \$zero) luôn mang giá trị 0

- Ví dụ:

`add $s0, $s1, $zero`

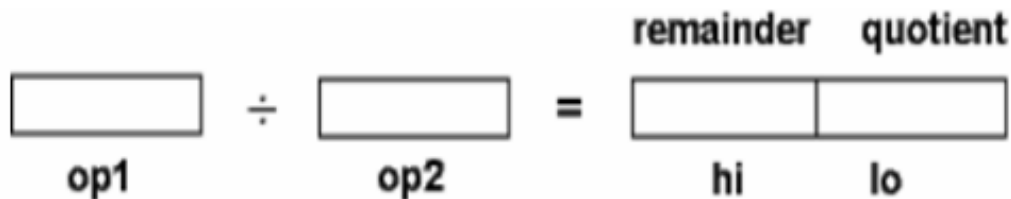
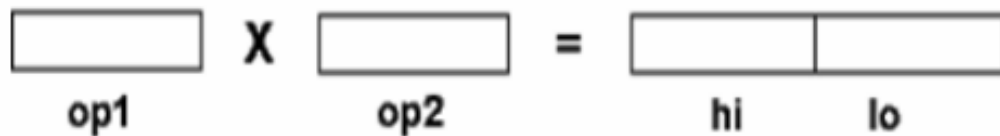
Tương đương: $\$s0 = \$s1 + 0 = \$s1$ (gán)

- Lệnh `add $zero, $zero, $s0` có hợp lệ ?

Phép nhân, chia số nguyên

20

- Thao tác nhân / chia của MIPS có kết quả chứa trong cặp 2 thanh ghi tên là \$hi và \$lo
Bit 0-31 thuộc \$lo và 32-63 thuộc \$hi



Phép nhân

21

- Cú pháp:

`mult $s0, $s1`

- Kết quả (64 bit) chứa trong 2 thanh ghi

- ▣ `$lo (32 bit)` = $((\$s0 * \$s1) \ll 32) \gg 32$

- ▣ `$hi (32 bit)` = $(\$s0 * \$s1) \gg 32$

- Câu hỏi: Làm sao truy xuất giá trị 2 thanh ghi `$lo` và `$hi`?

→ Dùng 2 cặp lệnh `mflo` (move from lo), `mfhi` (move from

hi) - `mtlo` (move to lo), `mthi` (move to high)

- ▣ `mflo $s0` ($\$s0 = \lo)

- ▣ `mfhi $s0` ($\$s0 = \hi)

Phép chia

22

- Cú pháp:

`div $s0, $s1`

- Kết quả (64 bit) chứa trong 2 thanh ghi
 - `$lo (32 bit)` = $\$s0 / \$s1$ (thương)
 - `$hi (32 bit)` = $\$s0 \% \$s1$ (số dư)

Thao tác số dấu chấm động

23

- MIPS sử dụng 32 thanh ghi dấu phẩy động để biểu diễn độ chính xác đơn của số thực. Các thanh ghi này có tên là : $\$f0 - \$f31$.
- Để biểu diễn độ chính xác kép (double precision) thì MIPS sử dụng sự ghép đôi của 2 thanh ghi có độ chính xác đơn.

Vấn đề tràn số

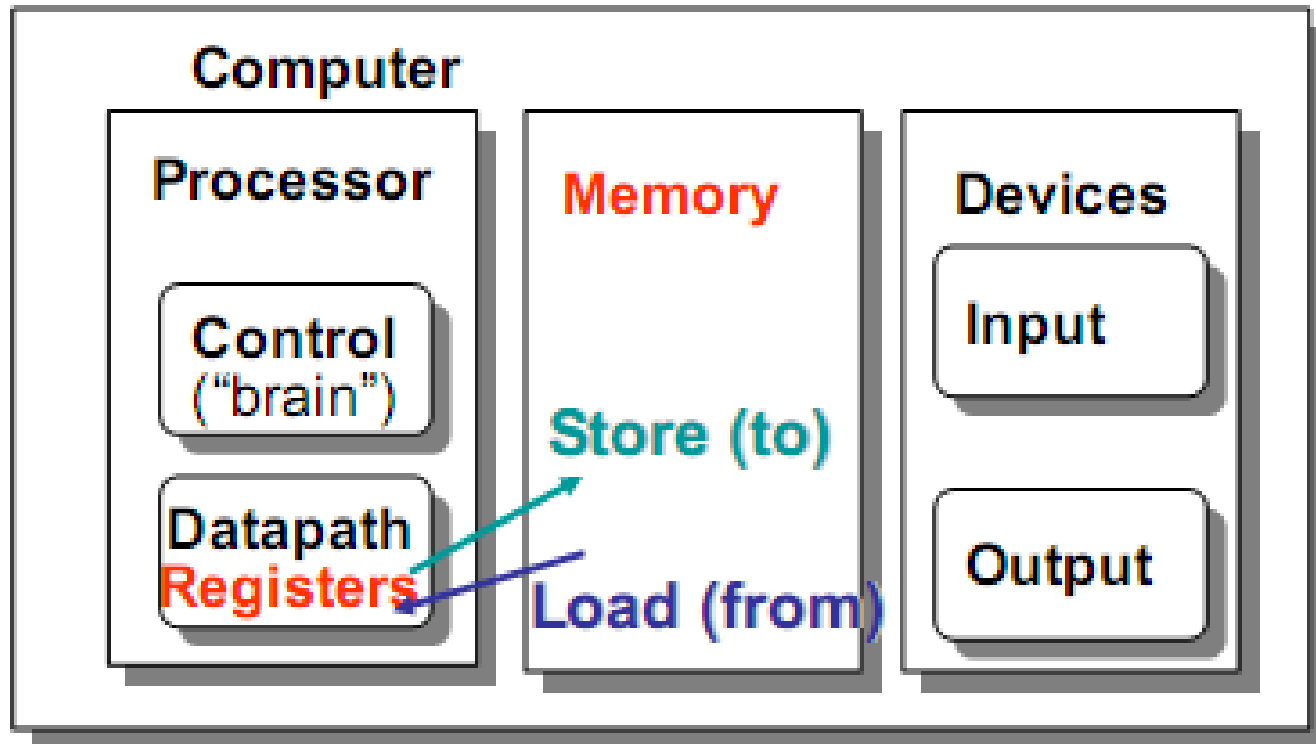
24

- Kết quả phép tính vượt qua miền giá trị cho phép → Tràn số xảy ra
- Một số ngôn ngữ có khả năng phát hiện tràn số (Ada), một số không (C)
- MIPS cung cấp 2 loại lệnh số học:
 - **add, addi, sub**: Phát hiện tràn số
 - **addu, addiu, subu**: Không phát hiện tràn số
- Trình biên dịch sẽ lựa chọn các lệnh số học tương ứng
 - Trình biên dịch C trên kiến trúc MIPS sử dụng **addu, addiu, subu**

Phần 2: Di chuyển dữ liệu

25

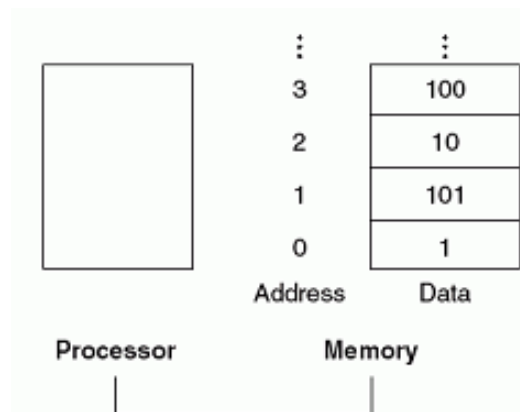
- Một số nhận xét:
 - Ngoài các biến đơn, còn có các **biến phức tạp** thể hiện nhiều kiểu cấu trúc dữ liệu khác nhau, ví dụ như **array**
 - Các cấu trúc dữ liệu phức tạp có số phần tử dữ liệu nhiều hơn số thanh ghi của CPU → làm sao lưu ??
 - Lưu phần nhiều data trong RAM, chỉ load 1 ít vào thanh ghi của CPU khi cần xử lý
 - Vấn đề lưu chuyển dữ liệu giữa thanh ghi và bộ nhớ ?
 - Nhóm lệnh **lưu chuyển dữ liệu** (data transfer)



Bộ nhớ chính

27

- Có thể được xem như là array 1 chiều rất lớn, mỗi phần tử là 1 ô nhớ có kích thước bằng nhau
- Các ô nhớ được đánh số thứ tự từ 0 trở đi
→ Gọi là **địa chỉ (address) ô nhớ**
- Để truy xuất dữ liệu trong ô nhớ cần phải cung cấp địa chỉ ô nhớ đó



Cấu trúc lệnh

28

□ Cú pháp:


opt **opr**, **opr1** (**opr2**)

- **opt** (operator): Tên thao tác (Load / Save)
- **opr** (operand): Thanh ghi lưu từ nhớ (word)
- **opr1** (operand 1): Hằng số nguyên
- **opr2** (operand 2): Thanh ghi chứa địa chỉ vùng nhớ cơ sở (địa chỉ nền)

Hai thao tác chính


29

- **lw**: Nạp 1 từ dữ liệu, từ bộ nhớ, vào 1 thanh ghi trên CPU (Load Word - lw)


lw \$t0, 12 (\$s0)

Nạp từ nhớ có địa chỉ ($\$s0 + 12$) chứa vào thanh ghi \$t0

- **sw**: Lưu 1 từ dữ liệu, từ thanh ghi trên CPU, ra bộ nhớ (Store Word – sw)


sw \$t0, 12 (\$s0)

Lưu giá trị trong thanh ghi \$t0 vào ô nhớ có địa chỉ ($\$s0 + 12$)

Lưu ý 1

30

- **\$s0** được gọi là **thanh ghi cơ sở (base register)** thường dùng để lưu địa chỉ bắt đầu của mảng / cấu trúc
- **12** gọi là **độ dời (offset)** thường dùng để truy cập các phần tử mảng hay cấu trúc

Lưu ý 2

31

- Một thanh ghi có lưu bất kỳ giá trị 32 bit nào, có thể là số nguyên (có dấu / không dấu), có thể là địa chỉ của 1 vùng nhớ trên RAM
- Ví dụ:
 - `add $t2, $t1, $t0` → \$t0, \$t1 lưu giá trị
 - `lw $t2, 4($t0)` → \$t0 lưu địa chỉ (C: con trỏ)

Lưu ý 3

32

- Số biến cần dùng của chương trình nếu nhiều hơn số thanh ghi của CPU?
- Giải pháp:
 - Thanh ghi chỉ chứa các biến đang xử lý hiện hành và các biến thường sử dụng
 - Kỹ thuật **spilling register**

Ví dụ 1

33

- Giả sử **A** là 1 array gồm 100 từ với **địa chỉ bắt đầu (địa chỉ nền – base address) chứa trong thanh ghi \$s3**. Giá trị các biến **g, h** lần lượt chứa trong các thanh ghi **\$s1** và **\$s2**
- Hãy chuyển thành mã hợp ngữ MIPS:

$$g = h + A[8]$$

- Trả lời:

```
lw    $t0, 32($s3)           # Chứa A[8] vào $t0
```

```
add   $s1, $s2, $t0
```

Ví dụ 2

34

- Hãy chuyển thành mã hợp ngữ MIPS:

$$\mathbf{A[12] = h - A[8]}$$

- Trả lời:

lw \$t0, 32(\$s3) # Chứa A[8] vào \$t0

sub \$t0, \$s2, \$t0

sw \$t0, 48(\$s3) # Kết quả vào A[12]

Nguyên tắc lưu dữ liệu trong bộ nhớ

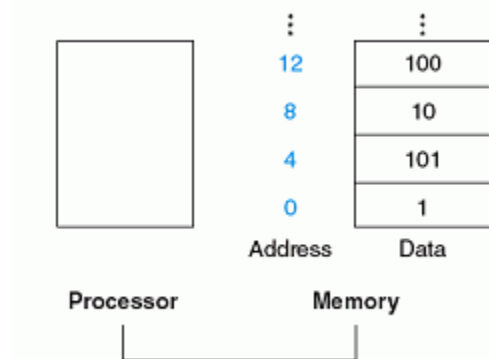
35

- MIPS thao tác và lưu trữ dữ liệu trong bộ nhớ theo 2 nguyên tắc:
 - ▣ Alignment Restriction
 - ▣ Big Endian

Alignment Restriction

36

- MIPS lưu dữ liệu trong bộ nhớ theo nguyên tắc **Alignment Restriction**
 - Các đối tượng lưu trong bộ nhớ (từ nhớ) phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng
 - Mà mỗi từ nhớ có kích thước là 32 bit = 4 byte = kích thước lưu trữ của 1 thanh ghi trong CPU
- Như vậy, từ nhớ phải bắt đầu tại địa chỉ là bội số của 4



Big Endian

37

- MIPS lưu trữ thứ tự các byte trong 1 word trong bộ nhớ theo nguyên tắc **Big Endian** (Kiến trúc x86 sử dụng Little Endian)
- Ví dụ: Lưu trữ giá trị 4 byte: **12345678h** trong bộ nhớ

Địa chỉ byte	Big Endian	Little Endian
0	12	78
1	34	56
2	56	34
3	78	12

Lưu ý

38

- Để truy xuất vào 1 từ nhớ sau 1 từ nhớ thì cần **tăng 1 lượng 4 byte** chứ không phải 1 byte
- Do đó luôn nhớ rằng các lệnh **lw** và **sw** thì **độ dời (offset) phải là bội số của 4**
- Tuy nhiên bộ nhớ các máy tính cá nhân ngày nay lại được đánh địa chỉ theo từng byte (8 bit)

Mở rộng: Load, Save 1 byte

39

- Ngoài việc hỗ trợ load, save 1 từ (lw, sw), MIPS còn hỗ trợ **load, save từng byte** (ASCII)
 - **Load byte: lb**
 - **Save byte: sb**
 - Cú pháp lệnh tương tự lw, sw
- Ví dụ:

lb \$s0, 3 (\$s1)

Lệnh này nạp giá trị byte nhớ có địa chỉ ($\$s1 + 3$) vào byte thấp của thanh ghi $\$s0$

Nguyên tắc

40

- Giả sử nạp 1 byte có giá trị **xzzz zzzz** vào thanh ghi trên CPU (x: bit dấu của byte đó)
- Giá trị thanh ghi trên CPU (32 bit) sau khi nạp có dạng:
XXXX XXXX XXXX XXXX XXXX XXXX xzzz zzzz
- Tất cả các bit từ phải sang sẽ có giá trị = bit dấu của giá trị 1 byte vừa nạp (sign-extended)
- Nếu muốn các bit còn lại từ phải sang có giá trị không theo bit dấu (=0) thì dùng lệnh:

lbu (load byte unsigned)

Mở rộng: Load, Save 2 byte (1/2 Word)

41

- MIPS còn hỗ trợ **load, save 1/2 word (2 byte)** (Unicode)
 - ▣ **Load half: lh** (nạp 2 byte nhớ vào 2 byte thấp của thanh ghi \$s0)
 - ▣ **Store half: sh**
 - ▣ Cú pháp lệnh tương tự lw, sw
- Ví dụ:

lh \$s0, 3 (\$s1)

Lệnh này nạp giá trị 2 byte nhớ có địa chỉ ($\$s1 + 3$) vào 2 byte thấp của thanh ghi \$s0

Phần 3: Thao tác luận lý

42

- Chúng ta đã xem xét các thao tác số học (+, -, *, /)
 - Dữ liệu trên thanh ghi như **1 giá trị đơn** (số nguyên có dấu / không dấu)
- Cần **thao tác trên từng bit** của dữ liệu → **Thao tác luận lý**
 - Các thao tác luận lý xem dữ liệu trong thanh ghi là dãy 32 bit riêng lẻ thay vì 1 giá trị đơn
- Có 2 loại thao tác luận lý:
 - **Phép toán luận lý**
 - **Phép dịch luận lý**

Phép toán luận lý

43

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

- **opt** (operator): Tên thao tác
- **opr** (operand): Thanh ghi (toán hạng đích)
chứa kết quả
- **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)
- **opr2** (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Phép toán luận lý

44

- MIPS hỗ trợ 2 nhóm lệnh cho các phép toán luận lý trên bit:
 - **and, or, nor**: Toán hạng nguồn thứ 2 (opr2) phải là thanh ghi
 - **andi, ori**: Toán hạng nguồn thứ 2 (opr2) là hằng số
- Lưu ý: MIPS không hỗ trợ lệnh cho các phép luận lý NOT, XOR, NAND...
- Lý do: Vì với 3 phép toán luận lý and, or, nor ta có thể tạo ra tất cả các phép luận lý khác → Tiết kiệm thiết kế cổng mạch
- Ví dụ:
$$\text{not}(A) = \text{not}(A \text{ or } 0) = A \text{ nor } 0$$

Phép dịch luận lý

45

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

- **opt** (operator): Tên thao tác
- **opr** (operand): Thanh ghi (toán hạng đích)
chứa kết quả
- **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)
- **opr2** (operand 2): Hằng số < 32 (Số bit dịch)

Phép dịch luận lý

46

- MIPS hỗ trợ 2 nhóm lệnh cho các phép dịch luận lý trên bit:
 - Dịch luận lý
 - Dịch trái (**sll** – shift left logical): Thêm vào các bit 0 bên phải
 - Dịch phải (**srl** – shift right logical): Thêm vào các bit 0 bên trái
 - Dịch số học
 - Không có dịch trái số học
 - Dịch phải (**sra** – shift right arithmetic): Thêm các bit = giá trị bit dấu bên trái

Ví dụ

47

- **sll \$s1, \$s2, 2** # dịch trái luận lý \$s2 2 bit

\$s2 = 0000 0000 0000 0000 0000 0000 0101 0101 = 85

\$s1 = 0000 0000 0000 0000 0000 0001 0101 0100 = 340 (85 * 2²)

- **srl \$s1, \$s2, 2** # dịch phải luận lý \$s2 2 bit

\$s2 = 0000 0000 0000 0000 0000 0000 0101 0101 = 85

\$s1 = 0000 0000 0000 0000 0000 0000 0001 0101 = 21 (85 / 2²)

- **sra \$s1, \$s2, 2** # dịch phải số học \$s2 2 bit

\$s2 = 1111 1111 1111 1111 1111 1111 1111 0000 = -16

\$s1 = 1111 1111 1111 1111 1111 1111 1111 1100 = -4 (-16 / 2²)

Phần 4: Rẽ nhánh

48

- Tương tự lệnh if trong C: Có 2 loại

- if (condition) clause

- if (condition)

clause1

else

clause2

- Lệnh if thứ 2 có thể diễn giải như sau:

if (condition) goto L1 // if → Làm clause1

clause2 // else → Làm clause2

goto L2 // Làm tiếp các lệnh khác

L1: clause1

L2: ...

Rẽ nhánh trong MIPS

49

- Rẽ nhánh có điều kiện
 - ▣ `beq opr1, opr2, label`
 - `beq`: Branch if (register are) equal
 - if (`opr1 == opr2`) goto `label`
 - ▣ `bne opr1, opr2, label`
 - `bne`: Branch if (register are) not equal
 - if (`opr1 != opr2`) goto `label`
- Rẽ nhánh không điều kiện
 - ▣ `j label`
 - Jump to label
 - Tương ứng trong C: `goto label`
 - Có thể viết lại thành: `beq $0, $0, label`

Ví dụ

50

- Biên dịch câu lệnh sau trong C thành lệnh hợp ngữ MIPS:

```
if (i == j)    f = g + h;
```

```
else          f = g - h;
```

- Ánh xạ biến f , g , h , i , j tương ứng vào các thanh ghi: $\$s0$, $\$s1$, $\$s2$, $\$s3$, $\$s4$
- Lệnh hợp ngữ MIPS:

```
    beq $s3, $s4, TrueCase    # branch (i == j)
```

```
    sub $s0, $s1, $s2        # f = g - h (false)
```

```
    j   Fin                  # goto "Fin" label
```

```
TrueCase:    add $s0, $s1, $s2    # f = g + h (true)
```

```
Fin:        ...
```

Xử lý vòng lặp

51

- Xét mảng `int A[]`. Giả sử ta có vòng lặp trong C:

do {

g = g + A[i];

i = i + j;

} while (i != h);

- Ta có thể viết lại:

Loop: g = g + A[i];

i = i + j;

if (i != h) goto Loop;

→ Sử dụng lệnh rẽ có điều kiện để biểu diễn vòng lặp!

Xử lý vòng lặp

52

- Ánh xạ biến vào các thanh ghi như sau:

g	h	i	j	base address of A
\$s1	\$s2	\$s3	\$s4	\$s5

- Trong ví dụ trên có thể viết lại thành lệnh MIPS như sau:

```
Loop:      sll  $t1, $s3, 2           # $t1 = i * 22
           add  $t1, $t1, $s5       # $t1 = addr A[i]
           lw   $t1, 0($t1)         # $t1 = A[i]
           add  $s1, $s1, $t1       # g = g + A[i]
           add  $s3, $s3, $s4       # i = i + j
           bne  $s3, $s2, Loop      # if (i != j) goto Label
```

Xử lý vòng lặp

53

- Tương tự cho các vòng lặp phổ biến khác trong C:
 - ▣ while
 - ▣ for
 - ▣ do...while
- Nguyên tắc chung:
 - ▣ Viết lại vòng lặp dưới dạng goto
 - ▣ Sử dụng các lệnh MIPS rẽ nhánh có điều kiện

So sánh không bằng ?

54

- **beq** và **bne** được sử dụng để **so sánh bằng** (**==** và **!=** trong C)
- Muốn so sánh lớn hơn hay nhỏ hơn?
- MIPS hỗ trợ lệnh **so sánh không bằng**:
 - **slt** **opr1**, **opr2**, **opr3**
 - **slt**: Set on Less Than
 - if (**opr2** < **opr3**)
 - opr1** = 1;
 - else
 - opr1** = 0;

So sánh không bằng

55

- Trong C, câu lệnh sau:

```
if (g < h) goto Less; # g: $s0, h: $s1
```

- Được chuyển thành lệnh MIPS như sau:

```
slt $t0, $s0, $s1 # if (g < h) then $t0 = 1
```

```
bne $t0, $0, Less # if ($t0 != 0) goto Less
```

```
# if (g < h) goto Less
```

- **Nhận xét:** Thanh ghi \$0 luôn chứa giá trị 0, nên lệnh **bne** và **bep** thường dùng để so sánh sau lệnh **slt**

Các lệnh so sánh khác?

56

- Các phép so sánh còn lại như $>$, \geq , \leq thì sao?
- MIPS **không trực tiếp hỗ trợ** cho các phép so sánh trên, tuy nhiên dựa vào các lệnh **slt**, **bne**, **beq** ta hoàn toàn có thể biểu diễn chúng!

a: \$s0, b: \$s1

57

□ a < b

```
slt  $t0, $s0, $s1    # if (a < b) then $t0 = 1
bne  $t0, $0, Label   # if (a < b) then goto Label
<do something>        # else then do something
```

□ a > b

```
slt  $t0, $s1, $s0    # if (b < a) then $t0 = 1
bne  $t0, $0, Label   # if (b < a) then goto Label
<do something>        # else then do something
```

□ a ≥ b

```
slt  $t0, $s0, $s1    # if (a < b) then $t0 = 1
beq  $t0, $0, Label   # if (a ≥ b) then goto Label
<do something>        # else then do something
```

□ a ≤ b

```
slt  $t0, $s1, $s0    # if (b < a) then $t0 = 1
beq  $t0, $0, Label   # if (b ≥ a) then goto Label
<do something>        # else then do something
```

Nhận xét

58

- So sánh $==$ → Dùng lệnh **beq**
- So sánh $!=$ → Dùng lệnh **bne**
- So sánh $<$ và $>$ → Dùng cặp lệnh (**slt** → **bne**)
- So sánh \leq và \geq → Dùng cặp lệnh (**slt** → **beq**)

So sánh với hằng số

59

- So sánh bằng: beq / bne
- So sánh không bằng: MIPS hỗ trợ sẵn lệnh **slti**
 - ▣ `slti opr, opr1, const`
 - ▣ Thường dùng cho switch...case, vòng lặp for

Ví dụ: switch...case trong C

60

- **switch (k) {**

 - case 0: f = i + j; break;**

 - case 1: f = g + h; break;**

 - case 2: f = g - h; break;**

 - }**

- Ta có thể viết lại thành các lệnh if lồng nhau:

 - if (k == 0) f = i + j;**

 - else if (k == 1) f = g + h;**

 - else if (k == 2) f = g - h;**

- Ánh xạ giá trị biến vào các thanh ghi:

f	g	h	i	j	k
\$s0	\$s1	\$s2	\$s3	\$s4	\$s5

Ví dụ: switch...case trong C

61

- Chuyển thành lệnh hợp ngữ MIPS:

```
    bne    $s5, $0, L1          # if (k != 0) then goto L1
    add    $s0, $s3, $s4       # else (k == 0) then f = i + j
    j      Exit                # end of case → Exit (break)
L1:    addi   $t0, $s5, -1      # $t0 = k - 1
    bne    $t0, $0, L2         # if (k != 1) then goto L2
    add    $s0, $s1, $s2       # else (k == 1) then f = g + h
    j      Exit                # end of case → Exit (break)
L2:    addi   $t0, $s5, -2      # $t0 = k - 2
    bne    $t0, $0, Exit       # if (k != 2) then goto Exit
    sub    $s0, $s1, $s2       # else (k == 2) then f = g - h
Exit:  ....
```


Trình con (Thủ tục)

62

- Hàm (function) trong C → (Biên dịch) → Trình con (Thủ tục) trong hợp ngữ
- Giả sử trong C, ta viết như sau:

```
void main()
```

```
{
```

```
    int a, b;
```

```
    ...
```

```
    sum(a, b);
```

```
    ...
```

```
}
```

- Hàm được chuyển thành lệnh hợp ngữ như thế nào ?
- Dữ liệu được lưu trữ ra sao ?

```
int sum(int x, int y)
```

```
{
```

```
    return (x + y);
```

```
}
```

C ... sum (a, b); ... /* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

return x + y;

}

M Địa chỉ Lệnh

I 1000 add \$a0, \$s0, \$zero # x = a

P 1004 add \$a1, \$s1, \$zero # y = b

S 1008 addi \$ra, \$zero, 1016 # lưu địa chỉ lát sau quay về vào \$ra = 1016

1012 j sum # nhảy đến nhãn sum

1016 [Làm tiếp thao tác khác...]

....

2000 sum: add \$v0, \$a0, \$a1 # thực hiện thủ tục "sum"

2024 jr \$ra # nhảy tới địa chỉ trong \$ra

Thanh ghi lưu trữ dữ liệu trong thủ tục

64

- MIPS hỗ trợ 1 số thanh ghi để lưu trữ dữ liệu cho thủ tục:
 - **Đối số input (argument input):** \$a0 \$a1 \$a2 \$a3
 - **Kết quả trả về (return ...):** \$v0 \$v1
 - **Biến cục bộ trong thủ tục:** \$s0 \$s1 ... \$s7
 - **Địa chỉ quay về (return address):** \$ra
- Nếu có nhu cầu lưu nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên?
 - Bao nhiêu thanh ghi là đủ ?
 - **Sử dụng ngăn xếp (stack)**

C ... sum (a, b); ... /* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

return x + y;

}



M Địa chỉ Lệnh

I 1000 add \$a0, \$s0, \$zero # x = a

P 1004 add \$a1, \$s1, \$zero

S 1008 addi \$ra, \$zero, 1016

1012 j sum

1016 [Làm tiếp thao tác khác...]

....

2000 sum: add \$v0, \$a0, \$a1 # thực hiện thủ tục "sum"

2024 jr \$ra # nhảy tới địa chỉ trong \$ra

• Tại sao không dùng lệnh j cho đơn giản, mà lại dùng jr ?
→ Thủ tục "sum" có thể được gọi ở nhiều chỗ khác nhau, do vậy vị trí quay về mỗi lần gọi sẽ khác nhau
→ Lệnh mới: jr

C ... sum (a, b); ... /* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

return x + y;

}



M Địa chỉ Lệnh

I 1000 add \$a0, \$s0, \$zero

P 1004 add \$a1, \$s1, \$zero

S 1008 addi \$ra, \$zero, 1016

1012 j sum

1016 [Làm tiếp thao tác khác...]

....

2000 sum: add \$v0, \$a0,

2024 jr \$ra

• Thay vì dùng 2 lệnh để lưu địa chỉ quay về vào thanh ghi \$ra và nhảy đến thủ tục "sum":

1008 addi \$ra, \$zero, 1016 # \$ra = 1016

1012 j sum # goto sum

→MIPS hỗ trợ lệnh mới: jal (jump and link) để thực hiện 2 công việc trên:

1008 jal sum # \$ra = 1012, goto sum

→ Tại sao không cần xác định tường minh địa chỉ quay về trong \$ra ?

Các lệnh nhảy mới

67

- **jr (jump register)**
 - Cú pháp: **jr register**
 - Diễn giải: Nhảy đến địa chỉ nằm trong thanh ghi register thay vì nhảy đến 1 nhãn như lệnh j (jump)

- **jal (jump and link)**
 - Cú pháp: **jal label**
 - Diễn giải: Thực hiện 2 bước:
 - **Bước 1 (link)**: Lưu địa chỉ của lệnh kế tiếp vào thanh ghi \$ra (Tại sao không phải là địa chỉ của lệnh hiện tại ?)
 - **Bước 2 (jump)**: Nhảy đến nhãn label

- Hai lệnh này được sử dụng hiệu quả trong thủ tục
 - **jal**: tự động lưu địa chỉ quay về chương trình chính vào thanh ghi \$ra và nhảy đến thủ tục con
 - **jr \$ra**: Quay lại thân chương trình chính bằng cách nhảy đến địa chỉ đã được lưu trước đó trong \$ra

Bài tập

68

- Chuyển đoạn chương trình sau thành mã hợp ngữ MIPS:

```
void main()  
{  
    int i, j, k, m;  
    ...  
    i = mult (j, k); ...  
    m = mult (i, i); ...  
}
```

```
int mult (int mcand, int mlier)  
{  
    int product = 0;  
    while (mlier > 0)  
    {  
        product = product + mcand;  
        mlier = mlier - 1;  
    }  
    return product;  
}
```

Thủ tục lồng nhau

69

- Vấn đề đặt ra khi chuyển thành mã hợp ngữ của đoạn lệnh sau:

```
int sumSquare (int x, int y)
{
    return mult (x, x) + y;
}
```
 - Thủ tục `sumSquare` sẽ gọi thủ tục `mult` trong thân hàm của nó
 - Vấn đề:
 - Địa chỉ quay về của thủ tục `sumSquare` lưu trong thanh ghi `$ra` sẽ bị ghi đè bởi địa chỉ quay về của thủ tục `mult` khi thủ tục này được gọi!
 - Như vậy cần phải lưu lại (`backup`) trong bộ nhớ chính địa chỉ quay về của thủ tục `sumSquare` (trong thanh ghi `$ra`) **trước khi gọi thủ tục `mult`**
- Sử dụng ngăn xếp (Stack)

Ngăn xếp (Stack)

70

- Là ngăn xếp gồm nhiều ô nhớ kết hợp (vùng nhớ) nằm trong bộ nhớ chính
- Cấu trúc dữ liệu lý tưởng để chứa tạm các giá trị trong thanh ghi
 - Thường chứa **địa chỉ trả về**, các **biến cục bộ của trình con**, nhất là các biến có cấu trúc (array, list...) không chứa vừa trong các thanh ghi trong CPU
- Được định vị và quản lý bởi **stack pointer**
- Có 2 tác vụ hoạt động cơ bản:
 - **push**: Đưa dữ liệu từ thanh ghi vào stack
 - **pop**: Lấy dữ liệu từ stack chép vào thanh ghi
- Trong MIPS dành sẵn 1 thanh ghi **\$sp** để lưu trữ stack pointer
- Để sử dụng Stack, cần khai báo kích vùng Stack bằng cách tăng (push) giá trị con trỏ ngăn xếp stack pointer (lưu trữ trong thanh ghi **\$sp**)
 - Lưu ý: Stack pointer tăng theo chiều **giảm địa chỉ**

C int sumSquare (int x, int y) { return mult (x, x) + y; }
/* x: \$a0, y: \$a1 */

M **sumSquare:**
I **init** → `addi $sp, $sp, -8` # khai báo kích thước stack cần dùng = 8 byte
P **push** → `sw $ra, 4 ($sp)` # cất địa chỉ quay về của thủ tục sumSquare đưa vào stack
S **push** → `sw $a1, 0 ($sp)` # cất giá trị y vào stack
`add $a1, $a0, $zero` # gán tham số thứ 2 là x (ban đầu là y) để phục vụ cho thủ tục mult sắp gọi
`jal mult` # nhảy đến thủ tục mult
pop → `lw $a1, 0 ($sp)` # sau khi thực thi xong thủ tục mult , khôi phục lại tham số thứ 2 = y
dựa trên giá trị đã lưu trước đó trong stack
`add $v0, $v0, $a1` # mult() + y
pop → `lw $ra, 4 ($sp)` # khôi phục địa chỉ quay về của thủ tục sumSquare từ stack, đưa lại vào \$ra
free → `addi $sp, $sp, 8` # khôi phục 8 byte giá trị \$sp ban đầu đã "mượn", kết thúc stack
`jr $ra` # nhảy đến đoạn lệnh ngay sau khi gọi thủ tục sumSquare trong chương trình chính, để thao tác tiếp các lệnh khác.

mult:
... # lệnh xử lý cho thủ tục mult
`jr $ra` # nhảy lại đoạn lệnh ngay sau khi gọi thủ tục mult trong thủ tục sumSquare

Tổng quát: Thao tác với stack

72

- Khởi tạo stack (init)
- Lưu trữ tạm các dữ liệu cần thiết vào stack (push)
- Gán các đối số (nếu có)
- Gọi lệnh jal để nhảy đến các thủ tục con
- Khôi phục các dữ liệu đã lưu tạm từ stack (pop)
- Khôi phục bộ nhớ, kết thúc stack (free)

Cụ thể hóa

73

□ Đầu thủ tục:

Procedure_Label:

```
addi $sp, $sp, -framesize      # khởi tạo stack, dịch chuyển stack pointer $sp lùi
```

```
sw  $ra, framesize - 4 ($sp)     # cất $ra (kích thước 4 byte) vào stack (push)
```

Lưu tạm các thanh ghi khác (nếu cần)

□ Thân thủ tục:

```
jal other_procedure             # Gọi các thủ tục khác (nếu cần)
```

□ Cuối thủ tục:

```
lw  $ra, frame_size - 4 ($sp)   # khôi phục $ra từ stack (pop)
```

```
lw  ...                         # khôi phục các thanh ghi khác (nếu cần)
```

```
addi $sp, $sp, framesize        # khôi phục $sp, giải phóng stack
```

```
jr  $ra                         # nhảy đến lệnh tiếp theo "Procedure Label"
```

trong chương trình chính

Một số nguyên tắc khi thực thi thủ tục

74

- Nhảy đến thủ tục bằng lệnh **jal** và quay về nơi trước đó đã gọi nó bằng lệnh **jr \$ra**
- 4 thanh ghi chứa đối số của thủ tục: **\$a0, \$a1, \$a2, \$a3**
- Kết quả trả về của thủ tục chứa trong thanh ghi **\$v0** (và **\$v1** nếu cần)
- Phải tuân theo **nguyên tắc sử dụng các thanh ghi** (register conventions)

Nguyên tắc sử dụng thanh ghi

75

- **\$0**: (Không thay đổi) Luôn bằng 0
- **\$s0 - \$s7**: (Khôi phục lại nếu thay đổi) Rất quan trọng, nếu thủ tục được gọi (callee) thay đổi các thanh ghi này thì nó phải khôi phục lại giá trị các thanh ghi này trước khi kết thúc
- **\$sp**: (Khôi phục lại nếu thay đổi) Thanh ghi con trỏ stack phải có giá trị không đổi trước và sau khi gọi lệnh "jal", nếu không thủ tục gọi (caller) sẽ không quay về được.
- **Tip**: Tất cả các thanh ghi này đều bắt đầu bằng ký tự s !

Nguyên tắc sử dụng thanh ghi

76

- **\$ra**: (Có thể thay đổi) Khi gọi lệnh "jal" sẽ làm thay đổi giá trị thanh ghi này. Thủ tục gọi (caller) lưu lại (backup) giá trị của thanh ghi \$ra vào stack nếu cần
- **\$v0 - \$v1**: (Có thể thay đổi) Chứa kết quả trả về của thủ tục
- **\$a0 - \$a1**: (Có thể thay đổi) Chứa đối số của thủ tục
- **\$t0 - \$t9**: (Có thể thay đổi) Đây là các thanh ghi tạm nên có thể bị thay đổi bất cứ lúc nào

Tóm tắt

77

- Nếu thủ tục R gọi thủ tục E:
 - R phải lưu vào stack các thanh ghi tạm có thể bị sử dụng trong E trước khi gọi lệnh **jal E** (goto E)
 - E phải lưu lại giá trị các thanh ghi lưu trữ (**\$s0 - \$s7**) nếu nó muốn sử dụng các thanh ghi này → trước khi kết thúc E sẽ khôi phục lại giá trị của chúng
 - **Nhớ:** Thủ tục gọi **R (caller)** và Thủ tục được gọi **E (callee)** chỉ cần lưu các thanh ghi tạm / thanh ghi lưu trữ mà nó muốn dùng, không phải tất cả các thanh ghi!

Bảng tóm tắt

78

Name	Register number	Usage	Preserved on call?
\$zero	0	the constant value 0	n.a.
\$v0-\$v1	2-3	values for results and expression evaluation	no
\$a0-\$a3	4-7	arguments	no
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

System call

79

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = integer	
read_character	12		char (in \$v0)

Hello.asm

80

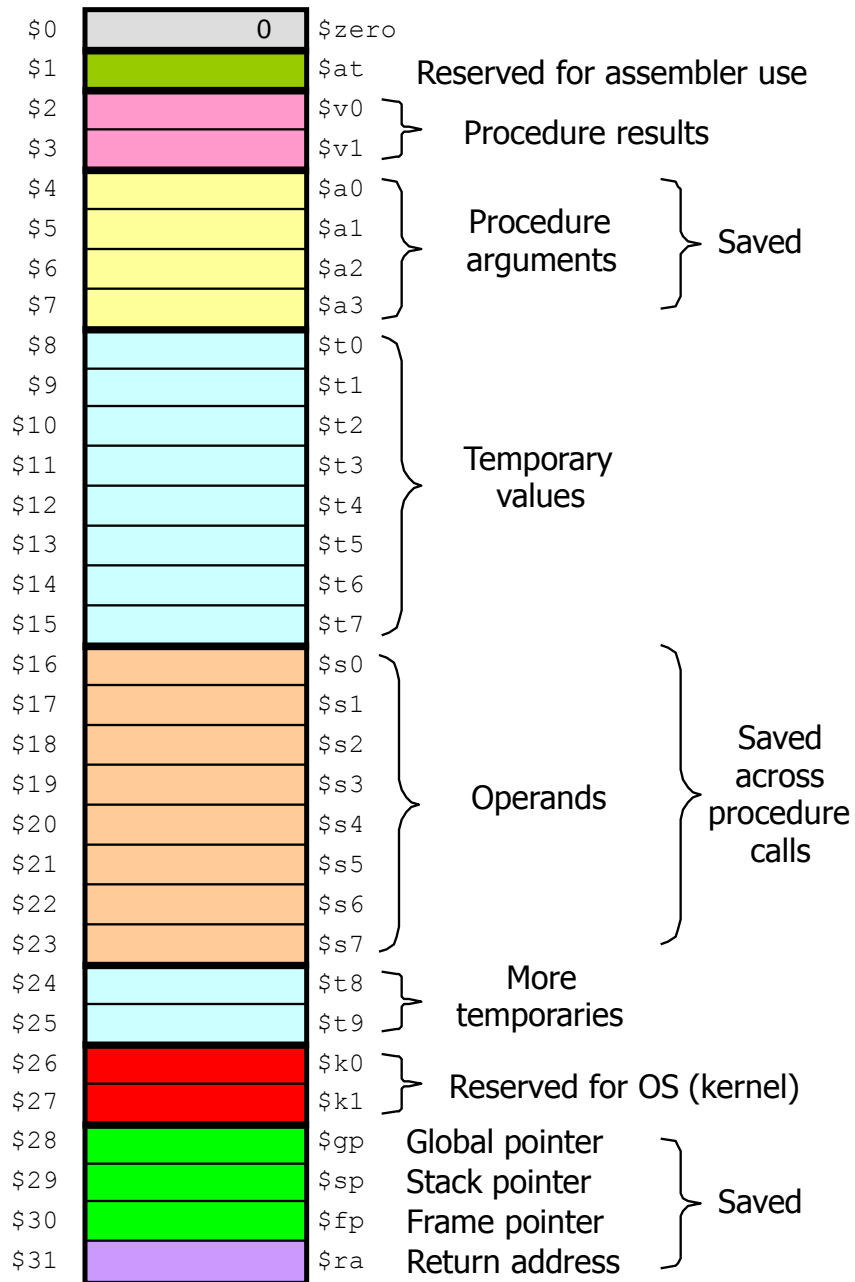
```
.data                                # data segment
str:  .asciiz "Hello asm !"

.text                                  # text segment
.globl main

main:                                  # starting point of program
    addi $v0, $0, 4                    # $v0 = 0 + 4 = 4 → print str syscall
    la $a0, str                        # $a0 = address(str)
    syscall                             # excute the system call
```

PHỤ LỤC





Phụ lục 1: 40 lệnh cơ bản MIPS

Instruction	Usage
Load upper immediate	lui rt,imm
Add	add rd,rs,rt
Subtract	sub rd,rs,rt
Set less than	slt rd,rs,rt
Add immediate	addi rt,rs,imm
Set less than immediate	slti rd,rs,imm
AND	and rd,rs,rt
OR	or rd,rs,rt
XOR	xor rd,rs,rt
NOR	nor rd,rs,rt
AND immediate	andi rt,rs,imm
OR immediate	ori rt,rs,imm
XOR immediate	xori rt,rs,imm
Load word	lw rt,imm(rs)
Store word	sw rt,imm(rs)
Jump	j L
Jump register	jr rs
Branch less than 0	bltz rs,L
Branch equal	beq rs,rt,L
Branch not equal	bne rs,rt,L

Instruction	Usage
Move from Hi	mfhi rd
Move from Lo	mflo rd
Add unsigned	addu rd,rs,rt
Subtract unsigned	subu rd,rs,rt
Multiply	mult rs,rt
Multiply unsigned	multu rs,rt
Divide	div rs,rt
Divide unsigned	divu rs,rt
Add immediate unsigned	addiu rs,rt,imm
Shift left logical	sll rd,rt,sh
Shift right logical	srl rd,rt,sh
Shift right arithmetic	sra rd,rt,sh
Shift left logical variable	sllv rd,rt,rs
Shift right logical variable	srlv rd,rt,rs
Shift right arith variable	srav rd,rt,rs
Load byte	lb rt,imm(rs)
Load byte unsigned	lbu rt,imm(rs)
Store byte	sb rt,imm(rs)
Jump and link	jal L
System call	syscall

Phụ lục 2: Pseudo Instructions

84

- “Lệnh giả”: Mặc định không được hỗ trợ bởi MIPS
- Là những lệnh cần phải biên dịch thành rất nhiều câu lệnh thật trước khi được thực hiện bởi phần cứng
→ Lệnh giả = Thủ tục
- Dùng để hỗ trợ lập trình viên thao tác nhanh chóng với những thao tác phức tạp gồm nhiều bước

Ví dụ: Tính $s1 = |s0|$

85

- Để tính được trị tuyệt đối của $s0 \rightarrow s1$, ta có lệnh giả là: `abs $s1, $s0`
- Thực sự MIPS không có lệnh này, khi chạy sẽ biên dịch lệnh này thành các lệnh thật sau:

Trị tuyệt đối của X là $-X$ nếu $X < 0$, là X nếu $X \geq 0$

abs:

```
sub    $s1, $zero, $s0
slt    $t0, $s0, $zero
bne    $t0, $zero, done
add    $s1, $s0, $zero
```

done:

```
jr     $ra
```


Một số lệnh giả phổ biến của MIPS

86

Name	instruction syntax	meaning
Move	move rd, rs	rd = rs
Load Address	la rd, rs	rd = address (rs)
Load Immediate	li rd, imm	rd = 32 bit Immediate value
Branch greater than	bgt rs, rt, Label	if($R[rs] > R[rt]$) PC=Label
Branch less than	blt rs, rt, Label	if($R[rs] < R[rt]$) PC=Label
Branch greater than or equal	bge rs, rt, Label	if($R[rs] \geq R[rt]$) PC=Label
branch less than or equal	ble rs, rt, Label	if($R[rs] \leq R[rt]$) PC=Label
branch greater than unsigned	bgtu rs, rt, Label	if($R[rs] <= R[rt]$) PC=Label
branch greater than zero	bgtz rs, Label	if($R[rs] \geq 0$) PC=Label

Phụ lục 3: Biểu diễn lệnh trong ngôn ngữ máy

87

- Chúng ta đã học 1 số nhóm lệnh hợp ngữ thao tác trên CPU tuy nhiên...
 - CPU có hiểu các lệnh hợp ngữ đã học này không?
- Tất nhiên là không vì nó chỉ hiểu được ngôn ngữ máy gồm toàn bit 0 và 1
- Dãy bit đó gọi là **lệnh máy (machine language instruction)**
 - Mỗi lệnh máy có kích thước **32 bit**, được chia thành các **nhóm bit, gọi là trường (field)**, mỗi nhóm có 1 vai trò trong lệnh máy
 - Lệnh máy có 1 cấu trúc xác định gọi là **cấu trúc lệnh (Instruction Format)**

MIPS Instruction Format

88

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, <i>imm.</i> format
J-format	op	target address					Jump instruction format

- Có 3 format lệnh trong MIPS:
 - **R-format:** Dùng trong các lệnh tính toán số học (add, sub, and, or, nor, sll, srl, sra...)
 - **I-format:** Dùng trong các lệnh thao tác với hằng số, chuyển dữ liệu với bộ nhớ, rẽ nhánh
 - **J-format:** Dùng trong các lệnh nhảy (jump – C: goto)

R-format

89

6 bits	5	5	5	5	6
opcode	rs	rt	rd	shmat	funct

- ❑ **opcode** (operation code): mã thao tác, cho biết lệnh làm gì
- ❑ **funct** (function code): kết hợp với opcode để xác định lệnh làm gì (trường hợp các lệnh có cùng mã thao tác với opcode)
- ❑ **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- ❑ **rt** (target register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 2
- ❑ **rd** (destination register): thanh ghi đích, thường chứa kết quả lệnh
- ❑ **shamt**: chứa số bit cần dịch trong các lệnh dịch, nếu không phải lệnh dịch thì trường này có giá trị 0

Nhận xét

90

- Các trường lưu địa chỉ thanh ghi **rs, rt, rd** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để biểu diễn 32 thanh ghi của MIPS

- Trường lưu số bit cần dịch **shamt** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để dịch hết 32 bit lưu trữ của 1 thanh ghi

Ví dụ R-format (1)

91

- Biểu diễn machine code của lệnh: **add \$t0, \$t1, \$t2**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	9	10	8	0	32
000000	01001	01010	01000	00000	100000

- opcode = 0
 - funct = 32
- } Xác định thao tác cộng
(tất cả các lệnh theo cấu trúc R-format đều có opcode = 0)
- rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)
 - rt = 10 (toán hạng nguồn thứ 2 là \$t2 ~ \$10)
 - rd = 8 (toán hạng đích là \$t0 ~ \$8)
 - shmat = 0 (không phải lệnh dịch)

Ví dụ R-format (2)

92

- Biểu diễn machine code của lệnh: **sll \$t2, \$s0, 4**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

- opcode = 0
 - funct = 0
- } Xác định thao tác dịch trái luận lý
(tất cả các lệnh theo cấu trúc R-format đều có opcode = 0)
- rs = 0 (không dùng trong phép dịch)
 - rt = 16 (toán hạng nguồn là \$s0 ~ \$16)
 - rd = 10 (toán hạng đích là \$t2 ~ \$10)
 - shmat = 4 (số bit dịch = 4)

Vấn đề của R-format

93

- Làm sao giải quyết trường hợp nếu câu lệnh đòi hỏi trường dành cho toán hạng phải lớn hơn 5 bit?
- Ví dụ:
 - Lệnh **addi** cộng giá trị thanh ghi với 1 hằng số, nếu giới hạn trường hằng số ở 5 bit → hằng số không thể lớn hơn $2^5 = 32$
→ Giới hạn khả năng tính toán số học!
 - Lệnh **lw**, **sw** cần biểu diễn 2 thanh ghi và 1 hằng số offset, nếu giới hạn ở 5 bit
→ Giới hạn khả năng truy xuất dữ liệu trong bộ nhớ
 - Lệnh **beq**, **bne** cần biểu diễn 2 thanh ghi và 1 hằng số chứa địa chỉ (nhãn) cần nhảy, nếu giới hạn ở 5 bit
→ Giới hạn lưu trữ chương trình trong bộ nhớ
- Giải pháp: Dùng **I-format** cho các lệnh thao tác hằng số, truy xuất dữ liệu bộ nhớ và rẽ nhánh

I-format

94

6 bits	5	5	16
opcode	rs	rt	immediate

- ❑ **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format, chỉ khác không cần thêm trường funct)
 - ❑ Đây cũng là lý do tại sao R-format có 2 trường 6 bit để xác định lệnh làm gì thay vì 1 trường 12 bit → Để nhất quán với các cấu trúc lệnh khác (I-format) trong khi kích thước mỗi trường vẫn hợp lý
- ❑ **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- ❑ **rt** (target register): thanh ghi đích, thường chứa kết quả lệnh
- ❑ **immediate**: 16 bit, có thể biểu diễn số nguyên từ -2^{15} đến $(2^{15} - 1)$
 - ❑ I-format đã có thể lưu hằng số 16 bit (thay vì 5 bit như R-format)

Ví dụ I-format

95

- Biểu diễn machine code của lệnh: **addi \$s0, \$s1, 10**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	immediate
8	17	16	10
001000	10001	10000	0000 0000 0000 0000 1010

- opcode = 8: Xác định thao tác cộng hằng số
- rs = 17 (toán hạng nguồn thứ 1 là \$s1 ~ \$17)
- rt = 16 (toán hạng đích là \$s0 ~ \$16)
- immediate = 10 (toán hạng nguồn thứ 2 = hằng số = 10)

Vấn đề I-format

96

- Trường hằng số (immediate) có kích thước 16 bit
- Nếu muốn thao tác với các hằng số 32 bit?
 - Tăng kích thước trường immediate thành 32 bit?
 - Tăng kích thước các lệnh thao tác với hằng số có cấu trúc I-format
 - Phá vỡ cấu trúc lệnh 32 bit của MIPS

Vấn đề I-format (tt)

97

- Giải pháp: MIPS cung cấp lệnh mới “lui”
 - ▣ lui register, immediate
 - ▣ Load Upper Immediate
 - ▣ Đưa hằng số 16 bit vào 2 byte cao của 1 thanh ghi
 - ▣ Giá trị 2 byte thấp của thanh ghi đó gán = 0
 - ▣ Lệnh này có cấu trúc I-format

Ví dụ

98

- Muốn cộng giá trị 32 bit **0xABABCD** vào thanh ghi **\$t0** ?

- Không thể dùng:

```
addi $t0, $t0, 0xABABCD
```

- Giải pháp dùng lệnh **lui**:

```
lui  $at, 0xABAB
```

```
ori  $at, $at, 0xCDCD
```

```
add  $t0, $0, $at
```

Vấn đề rẽ nhánh có điều kiện trong I-format

99

- Các lệnh rẽ nhánh có điều kiện có cấu trúc I-format

6 bits	5	5	16
opcode	rs	rt	immediate

- **opcode**: xác định lệnh beq hay bne
- **rs, rt**: chứa các giá trị của thanh ghi cần so sánh
- **immediate** chứa địa chỉ (nhãn) cần nhảy tới?
 - immediate chỉ có 16 bit → chỉ có thể nhảy tới địa chỉ từ 0 – 2^{16} (65535)
?
- Chương trình bị giới hạn không gian rất nhiều
- **Câu trả lời: immediate KHÔNG phải chứa địa chỉ cần nhảy tới**

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

100

- Trong MIPS, thanh ghi **PC** (Program Counter) sẽ chứa địa chỉ của lệnh **đang được thực hiện**
- **immediate**: số có dấu, chứa khoảng cách so với địa chỉ lệnh đang thực hiện nằm trong thanh ghi PC
 - $\text{immediate} + \text{PC} \rightarrow$ địa chỉ cần nhảy tới
- Cách xác định địa chỉ này gọi là **PC-Relative Addressing** (định vị theo thanh ghi PC)
 - Xem slide "Addressing Mode" (phần sau) để biết thêm về các Addressing mode trong MIPS

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

101

- Mỗi lệnh trong MIPS có kích thước 32 bit (1 word – 1 từ nhớ)
- MIPS truy xuất bộ nhớ theo nguyên tắc Alignment Restriction
- Đơn vị của **immediate**, khoảng cách so với PC, là từ nhớ (**word = 4 byte**) chứ không phải là byte
- Các lệnh rẽ nhánh có thể nhảy tới các địa chỉ có khoảng cách $\pm 2^{15}$ word tính từ địa chỉ lưu trong PC ($\pm 2^{17}$ byte)

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

102

- Cách tính địa chỉ rẽ nhánh:

- Nếu không rẽ nhánh:

$PC = PC + 4 =$ địa chỉ lệnh kế tiếp trong bộ nhớ

- Nếu thực hiện rẽ nhánh:

$PC = (PC + 4) + (\text{immediate} * 4)$

- Vì sao cộng immediate với $(PC + 4)$ thay vì PC ? → Khi rẽ nhánh bị delayed 1 lệnh kể với lệnh rẽ nhánh
- Nhận xét: immediate cho biết số lệnh cần nhảy qua để đến được nhãn

Ví dụ I-format

103

```
□ Loop:          beq    $t1, $0, End  
                  add    $t0, $t0, $t2  
                  addi   $t1, $t1, -1  
                  j      Loop
```

End: ...

opcode	rs	rt	immediate
4	9	0	3
000100	01001	00000	0000 0000 0000 0000 0011

- opcode = 4: Xác định thao tác của lệnh beq
- rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)
- rt = 0 (toán hạng nguồn thứ 2 là \$0 ~ \$0)
- immediate = 3 (nhảy qua 3 lệnh kể từ lệnh rẽ nhánh có điều kiện)

Vấn đề I-format

104

- Mỗi lệnh trong MIPS có kích thước 32 bit
- Mong muốn: Có thể nhảy đến bất kỳ lệnh nào (MIPS hỗ trợ các hàm nhảy không điều kiện như **j**)
 - Nhảy trong khoảng 2^{32} (4 GB) bộ nhớ
 - I-format bị hạn chế giới hạn vùng nhảy
 - Dùng **J-format**
- Tuy nhiên, dù format nào cũng phải **cần tối thiểu 6 bit cho opcode** để nhất quán lệnh với các format khác
 - **J-format** chỉ có thể dùng $32 - 6 = 26$ bit để biểu diễn khoảng cách nhảy

J-format

105

6 bits	26
opcode	target address

- **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format và I-format)
 - Để nhất quán với các cấu trúc lệnh khác (R-format và I-format)
- **target address**: Lưu địa chỉ đích của lệnh nhảy
 - Tương tự lệnh rẽ nhánh, địa chỉ đích của lệnh nhảy tính theo đơn vị word

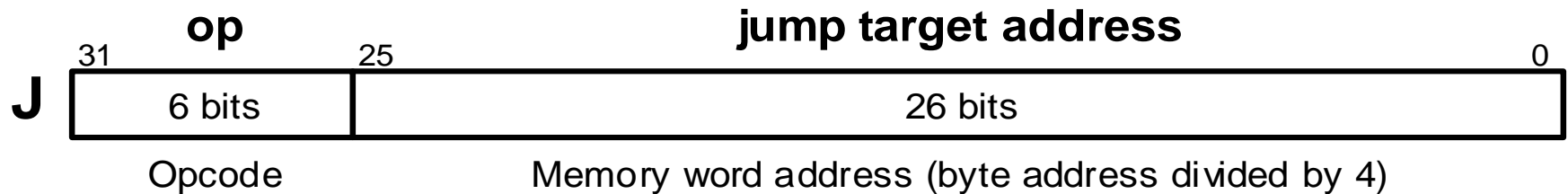
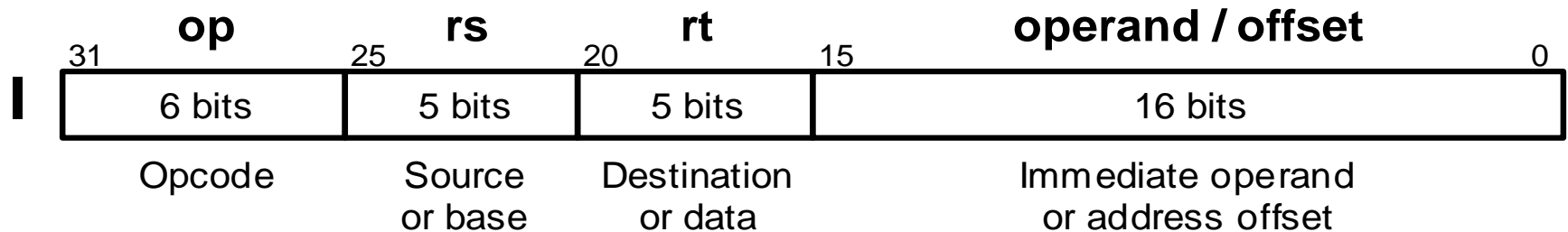
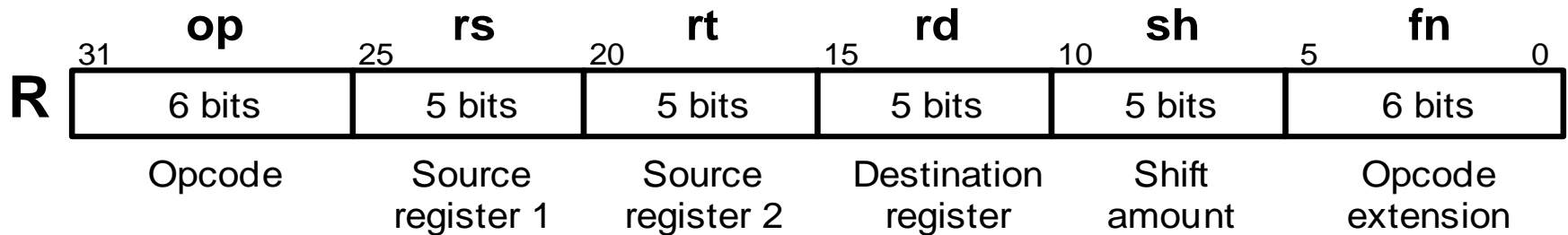
Nhận xét

106

- Trong **J-format**, các lệnh nhảy có thể nhảy tới các lệnh có địa chỉ trong khoảng 2^{26}
- Muốn nhảy tới các lệnh có địa chỉ lớn hơn từ 2^{27} đến 2^{32} ?
 - MIPS hỗ trợ lệnh **jr** (đọc trong phần thủ tục)
 - Tuy nhiên nhu cầu này không cần thiết lắm vì chương trình thường không quá lớn như vậy

Bảng tóm tắt Format

107



Phụ lục 4: Addressing mode

108

- Là phương thức định vị trí (địa chỉ hóa) các toán hạng trong kiến trúc MIPS
- Có 5 phương pháp chính:
 - **Immediate addressing** (Vd: `addi $t0, $t0, 5`)
Toán hạng = hằng số 16 bit trong câu lệnh
 - **Register addressing** (Vd: `add $t0, $t0, $t1`)
Toán hạng = nội dung thanh ghi
 - **Base addressing** (Vd: `lw $t1, 8($t0)`)
Toán hạng = nội dung ô nhớ (địa chỉ ô nhớ = nội dung thanh ghi + hằng số 16 bit trong câu lệnh)
 - **PC-relative addressing** (Vd: `beq $t0, $t1, Label`)
Toán hạng = địa chỉ đích lệnh nhảy = nội dung thanh ghi PC + hằng số 16 bit trong câu lệnh
 - **Pseudodirect addressing** (Vd: `j 2500`)
Toán hạng = địa chỉ đích lệnh nhảy = các bit cao thanh ghi PC + hằng số 26 bit trong câu lệnh

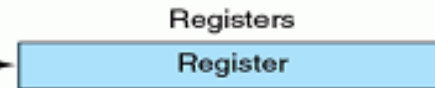
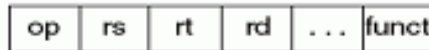
Addressing mode

109

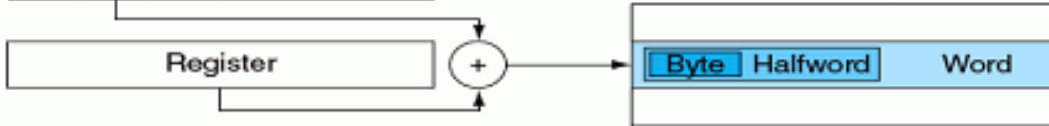
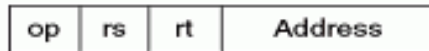
1. Immediate addressing



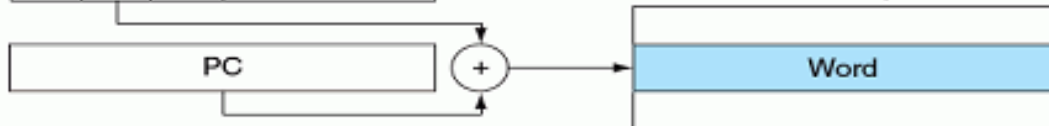
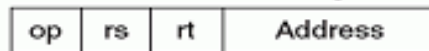
2. Register addressing



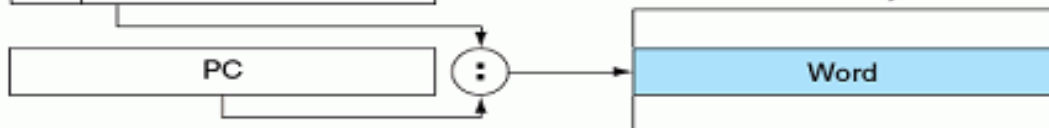
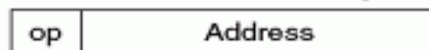
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Homework

110

- Sách Petterson & Hennessy: Đọc hết chương 2
- Tài liệu tham khảo: Đọc "08_HP_AppA.pdf"

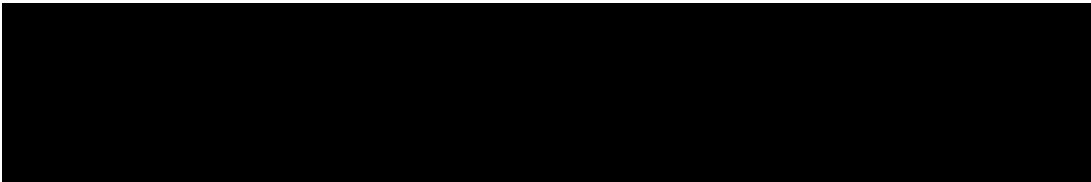
KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Vũ Minh Trí – vmtri@fit.hcmus.edu.vn

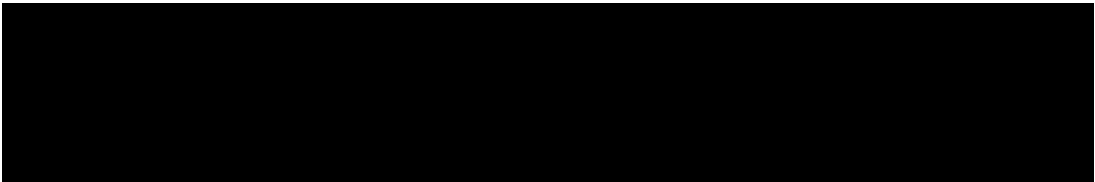
04 – Lập trình hợp ngữ (Phần 3)

THAM KHẢO

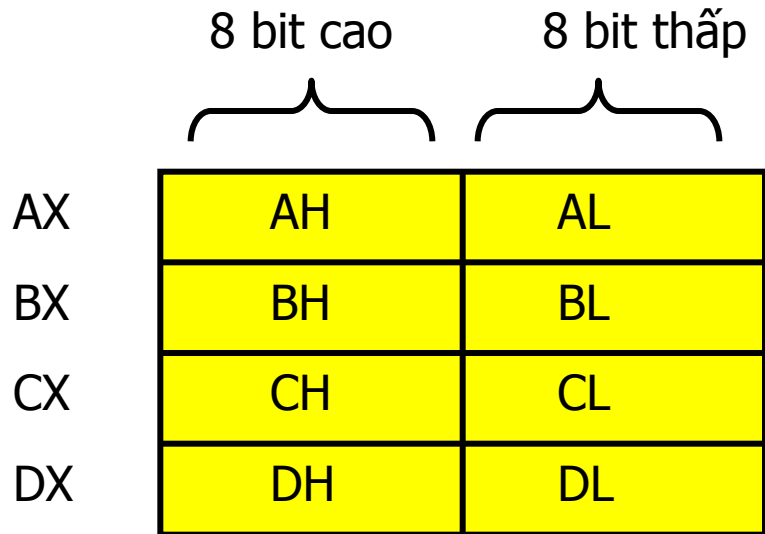
LẬP TRÌNH HỢP NGỮ CHO 8086



-
- Cấu trúc bên trong
 - Mô tả tập lệnh của 8086
 - Lập trình hợp ngữ 8086

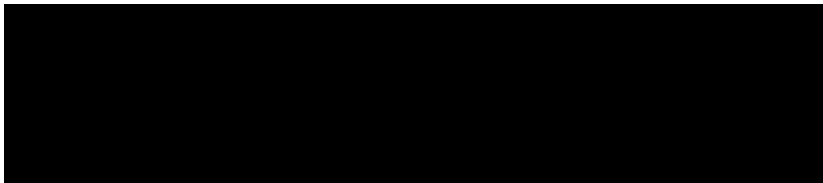


- Cấu trúc bên trong
 - Sơ đồ khối
 - Các thanh ghi đa năng
 - Các thanh ghi đoạn
 - Các thanh ghi con trỏ và chỉ số
 - Thanh ghi cờ
 - Hàng đợi lệnh
- Mô tả tập lệnh của 8086
- Lập trình hợp ngữ 8086



- 8088/8086 đến 80286 : 16 bits
- 80386 trở lên: 32 bits EAX, EBX, ECX, EDX

- Thanh ghi chứa AX (accumulator): chứa kết quả của các phép tính. Kết quả 8 bit được chứa trong AL
- Thanh ghi cơ sở BX (base): chứa địa chỉ cơ sở, ví dụ của bảng dùng trong lệnh XLAT (Translate)
- Thanh ghi đếm CX (count): dùng để chứa số lần lặp trong các lệnh lặp (Loop). CL được dùng để chứa số lần dịch hoặc quay trong các lệnh dịch và quay thanh ghi
- Thanh ghi dữ liệu DX (data): cùng AX chứa dữ liệu trong các phép tính nhân chia số 16 bit. DX còn được dùng để chứa địa chỉ cổng trong các lệnh vào ra dữ liệu trực tiếp (IN/OUT)



Tổ chức của bộ nhớ 1 Mbytes

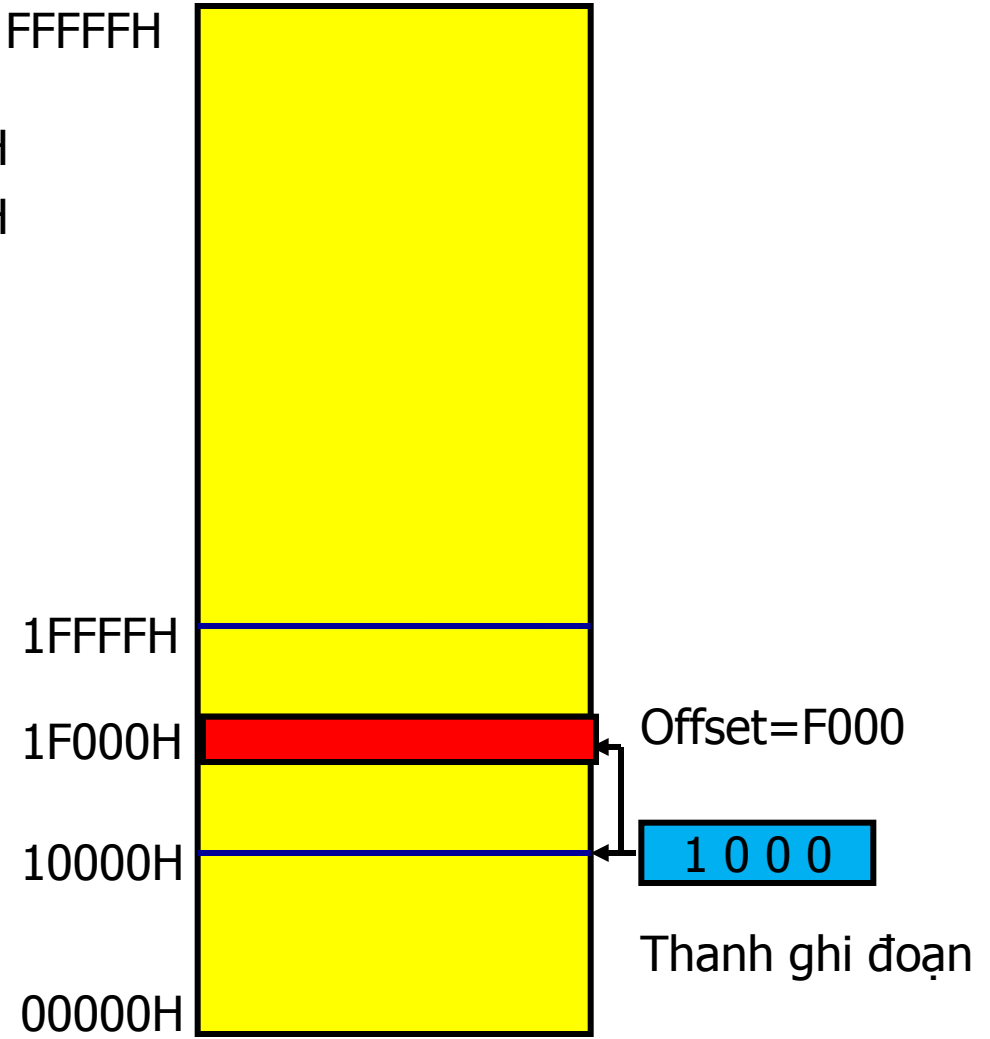
- Đoạn bộ nhớ (segment)
 - ⇒ 2^{16} bytes = 64 KB
 - ⇒ Đoạn 1: địa chỉ đầu 00000 H
 - ⇒ Đoạn 2: địa chỉ đầu 00010 H
 - ⇒ Đoạn cuối cùng: FFFF0 H

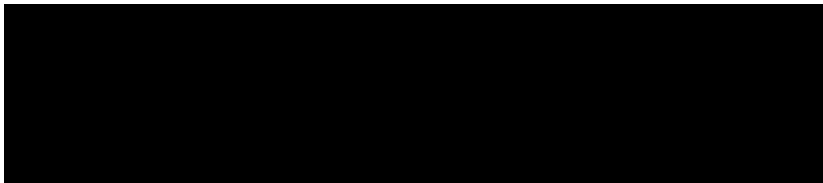
- Ô nhớ trong đoạn:
 - ⇒ địa chỉ lệch: offset
 - ⇒ Ô 1: offset: 0000
 - ⇒ Ô cuối cùng: offset: FFFF

- Địa chỉ vật lý:
 - ⇒ Segment : offset

Địa chỉ vật lý = Segment * 16 + offset

Chế độ thực (real mode)



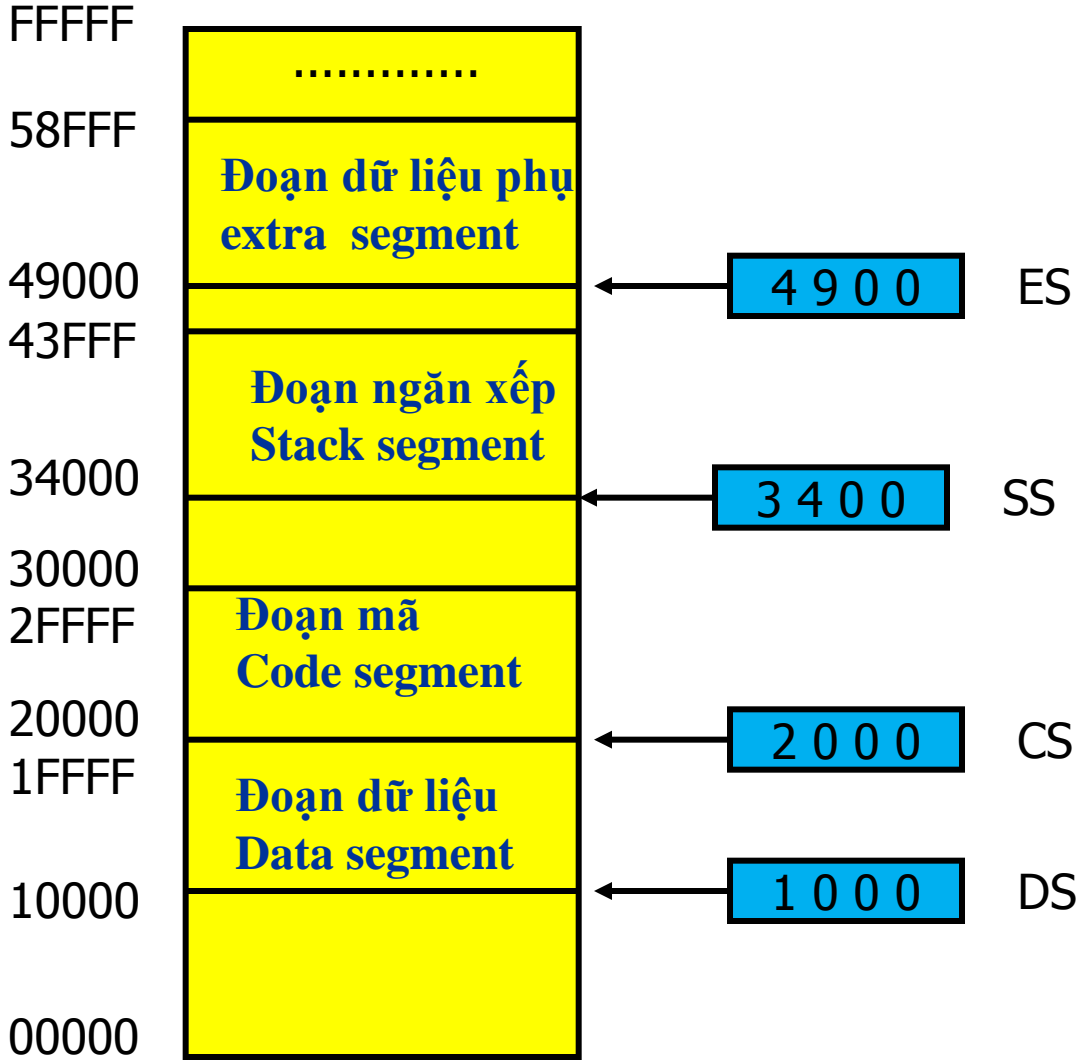


- Ví dụ: Địa chỉ vật lý 12345H

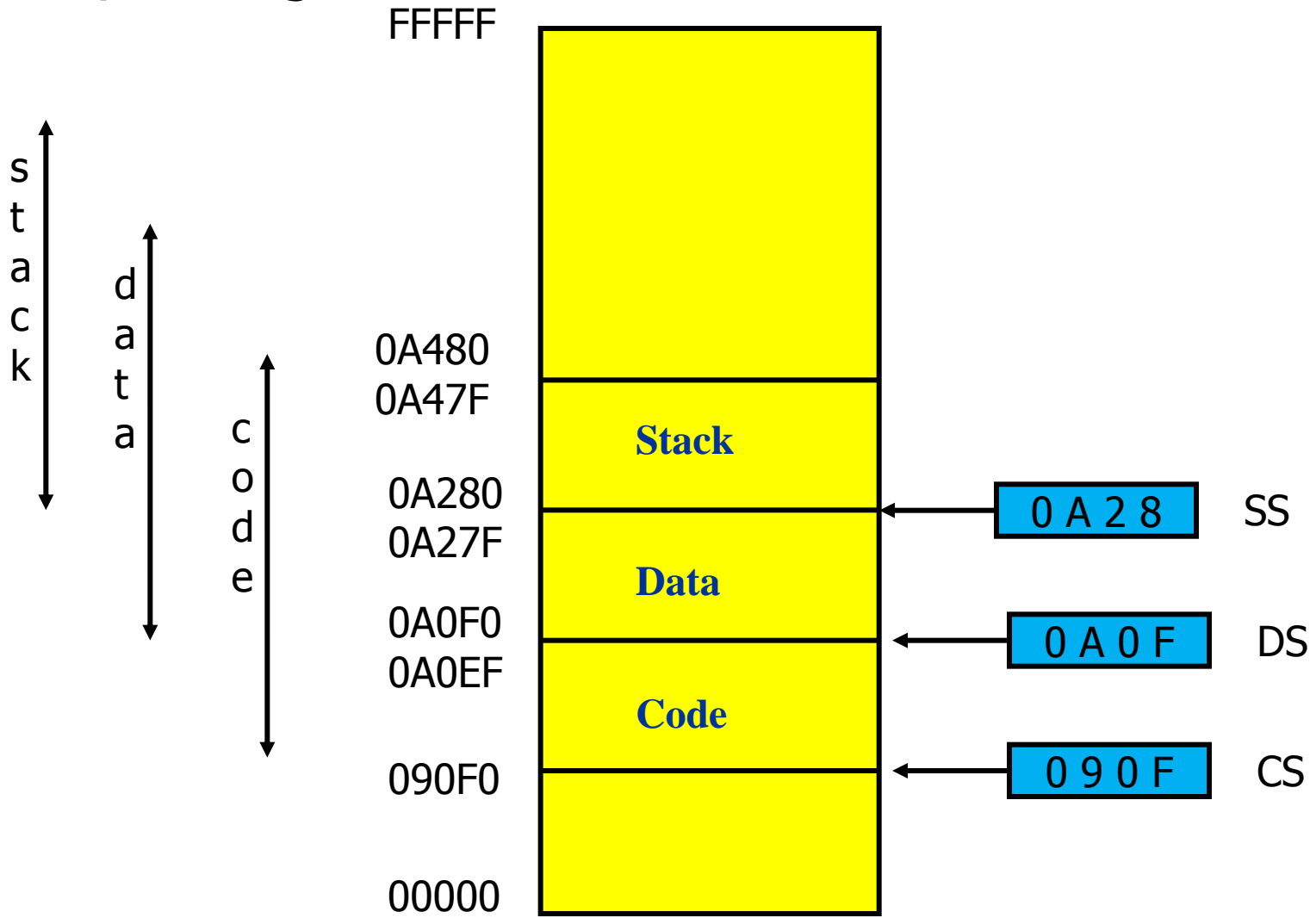
Địa chỉ đoạn	Địa chỉ lệch
1000 H	2345H
1200 H	0345H
1004 H	?
0300 H	?

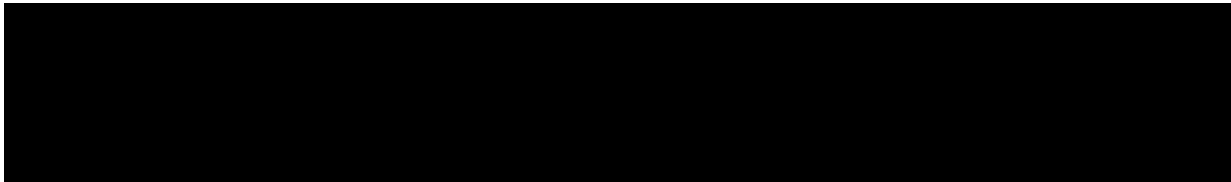
- Ví dụ: Cho địa chỉ đầu của đoạn: 49000 H, xác định địa chỉ cuối

- Các thanh ghi đoạn: chứa địa chỉ đoạn



- Các đoạn chồng nhau





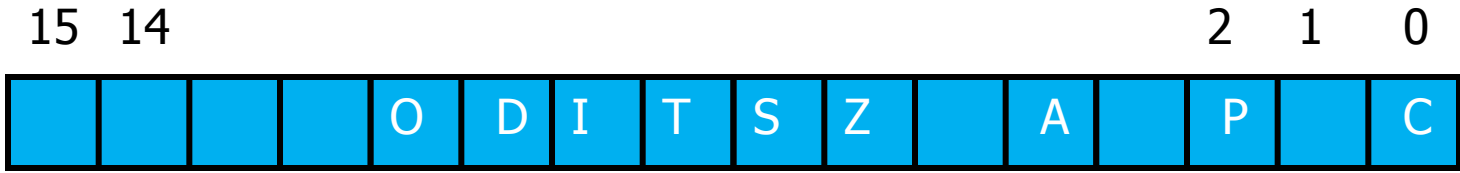
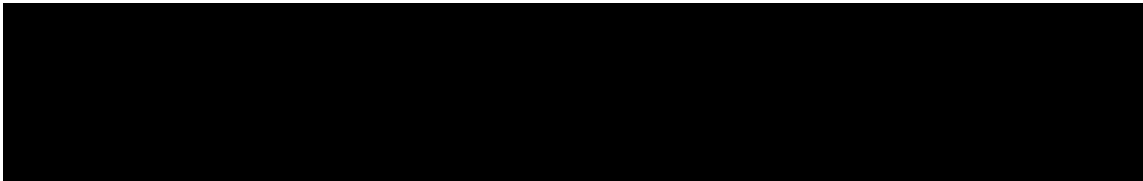
- Chứa địa chỉ lệch (offset)

- ❑ Con trỏ lệnh IP (instruction pointer): chứa địa chỉ lệnh tiếp theo trong đoạn mã lệnh CS.
⇒ CS:IP
- ❑ Con trỏ cơ sở BP (Base Pointer): chứa địa chỉ của dữ liệu trong đoạn ngăn xếp SS hoặc các đoạn khác
⇒ SS:BP
- ❑ Con trỏ ngăn xếp SP (Stack Pointer): chứa địa chỉ hiện thời của đỉnh ngăn xếp
⇒ SS:SP
- ❑ Chỉ số nguồn SI (Source Index): chứa địa chỉ dữ liệu nguồn trong đoạn dữ liệu DS trong các lệnh chuỗi
⇒ DS:SI
- ❑ Chỉ số đích (Destination Index): chứa địa chỉ dữ liệu đích trong đoạn dữ liệu DS trong các lệnh chuỗi
⇒ DS:DI
- ❑ SI và DI có thể được sử dụng như thanh ghi đa năng
- ❑ 80386 trở lên 32 bit: EIP, EBP, ESP, EDI, ESI

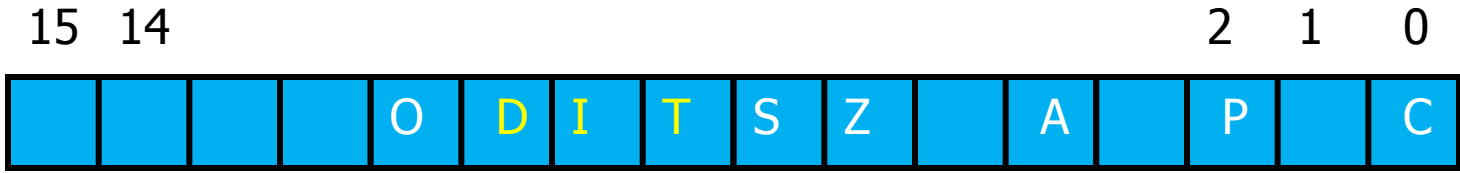
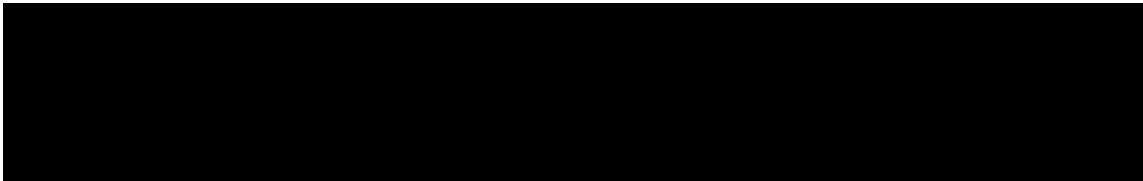


- Thanh ghi đoạn và thanh ghi lệch ngầm định

Segment	Offset	Chú thích
CS	IP	Địa chỉ lệnh
SS	SP hoặc BP	Địa chỉ ngăn xếp
DS	BX, DI, SI, số 8 bit hoặc số 16 bit	Địa chỉ dữ liệu
ES	DI	Địa chỉ chuỗi đích

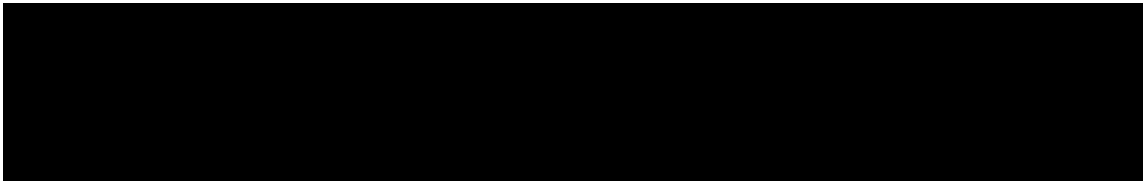


- 9 bit được sử dụng, 6 cờ trạng thái:
 - C hoặc CF (carry flag): CF=1 khi có nhớ hoặc mượn từ MSB
 - P hoặc PF (parity flag): PF=1 (0) khi tổng số bit 1 trong kết quả là chẵn (lẻ)
 - A hoặc AF (auxiliary carry flag): cờ nhớ phụ, AF=1 khi có nhớ hoặc mượn từ một số BCD thấp sang BCD cao
 - Z hoặc ZF (zero flag): ZF=1 khi kết quả bằng 0
 - S hoặc SF (Sign flag): SF=1 khi kết quả âm
 - O hoặc OF (Overflow flag): cờ tràn OF=1 khi kết quả là một số vượt ra ngoài giới hạn biểu diễn của nó trong khi thực hiện phép toán cộng trừ số có dấu



• 3 cờ điều khiển

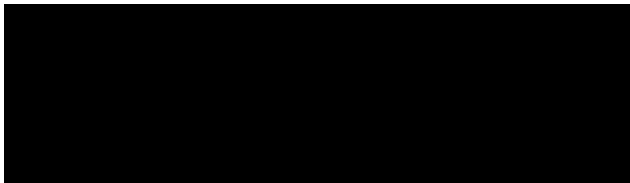
- T hoặc TF (trap flag): cờ bẫy, TF=1 khi CPU làm việc ở chế độ chạy từng lệnh
- I hoặc IF (Interrupt enable flag): cờ cho phép ngắt, IF=1 thì CPU sẽ cho phép các yêu cầu ngắt (ngắt che được) được tác động (Các lệnh: STI, CLI)
- D hoặc DF (direction flag): cờ hướng, DF=1 khi CPU làm việc với chuỗi ký tự theo thứ tự từ phải sang trái (lệnh STD, CLD)



• Ví dụ:

$$\begin{array}{r} 80h \\ + \\ 80h \\ \hline 100h \end{array}$$

- SF=0 vì msb trong kết quả =0
- PF=1 vì có 0 bit của tổng bằng 1
- ZF=1 vì kết quả thu được là 0
- CF=1 vì có nhớ từ bit msb trong phép cộng
- OF=1 vì có tràn trong phép cộng 2 số âm

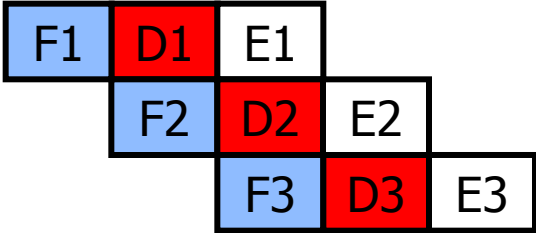


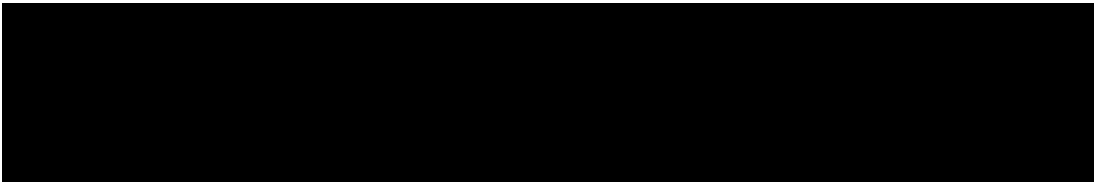
- 4 bytes đối với 8088 và 6 bytes đối với 8086
- Xử lý pipeline

Không có
pipelining

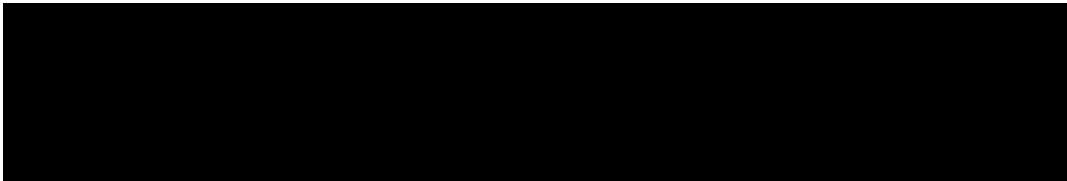


Có pipelining





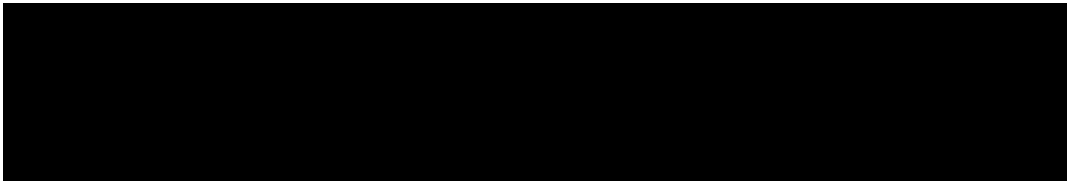
- Cấu trúc bên trong
- Mô tả tập lệnh của 8086
 - ❑ Các lệnh di chuyển dữ liệu
 - ❑ Các lệnh số học và logic
 - ❑ Các lệnh điều khiển chương trình
- Lập trình hợp ngữ với 8086



- MOV, XCHG, POP, PUSH, POPF, PUSHF, IN, OUT
- Các lệnh di chuyển chuỗi MOVS, MOVSB, MOVSW

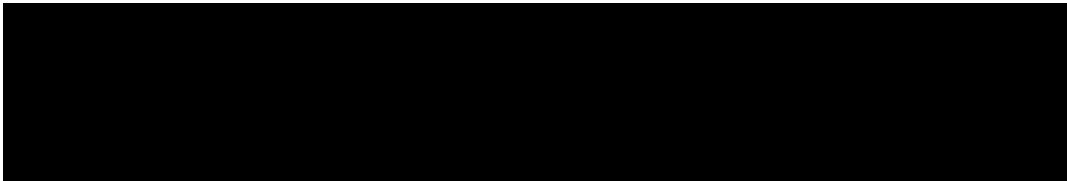
- MOV

- Dùng để chuyển giữa các thanh ghi, giữa 1 thanh ghi và 1 ô nhớ hoặc chuyển 1 số vào thanh ghi hoặc ô nhớ
- Cú pháp: MOV Đích, nguồn
- Lệnh này không tác động đến cờ
- Ví dụ:
 - ⇒ MOV AX, BX
 - ⇒ MOV AH, 'A'
 - ⇒ MOV AL, [1234H]

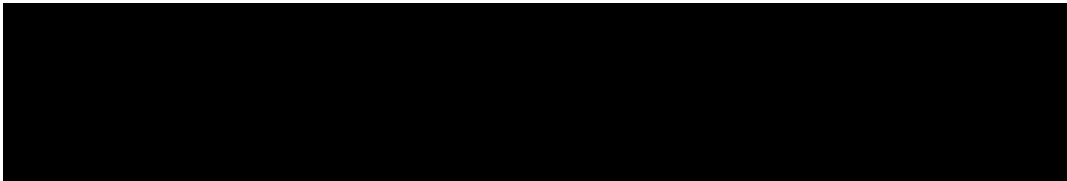


- Khả năng kết hợp toán hạng của lệnh MOV

Đích \ Nguồn	Thanh ghi đa năng	Thanh ghi đoạn	ô nhớ	Hằng số
Thanh ghi đa năng	YES	YES	YES	NO
Thanh ghi đoạn	YES	NO	YES	NO
Ô nhớ	YES	YES	NO	NO
Hằng số	YES	NO	YES	NO



- **Lệnh XCHG**
 - ❑ Dùng để hoán chuyển nội dung giữa hai thanh ghi, giữa 1 thanh ghi và 1 ô nhớ
 - ❑ Cú pháp: XCHG Đích, nguồn
 - ❑ Giới hạn: toán hạng không được là thanh ghi đoạn
 - ❑ Lệnh này không tác động đến cờ
 - ❑ Ví dụ:
 - ⇒ XCHG AX, BX
 - ⇒ XCHG AX, [BX]

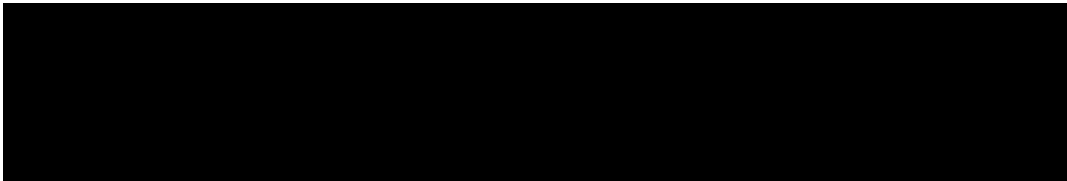


- **Lệnh PUSH**

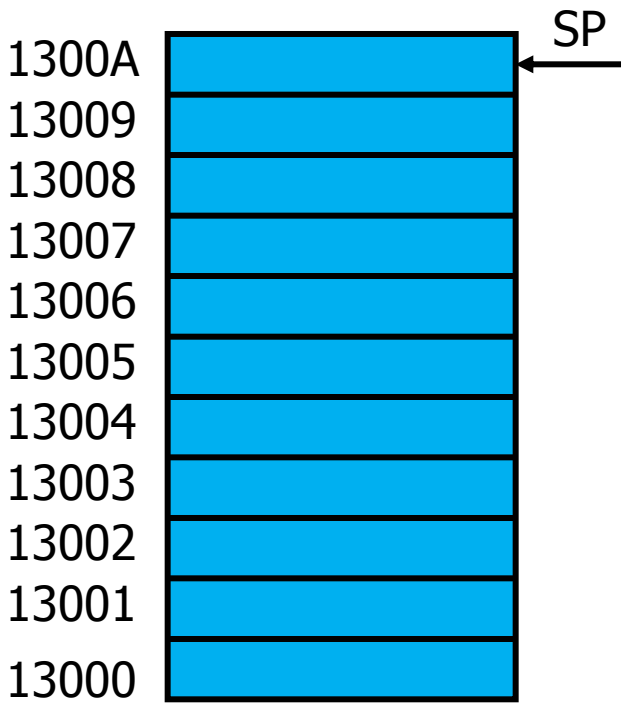
- Dùng để cất 1 từ từ thanh ghi hoặc ô nhớ vào đỉnh ngăn xếp
- Cú pháp: PUSH Nguồn
- Mô tả: $SP=SP-2$, Nguồn $\Rightarrow \{SP\}$
- Giới hạn: thanh ghi 16 bit hoặc là 1 từ nhớ
- Lệnh này không tác động đến cờ
- Ví dụ:
 - \Rightarrow PUSH BX
 - \Rightarrow PUSH PTR[BX]

- **Lệnh PUSHF**

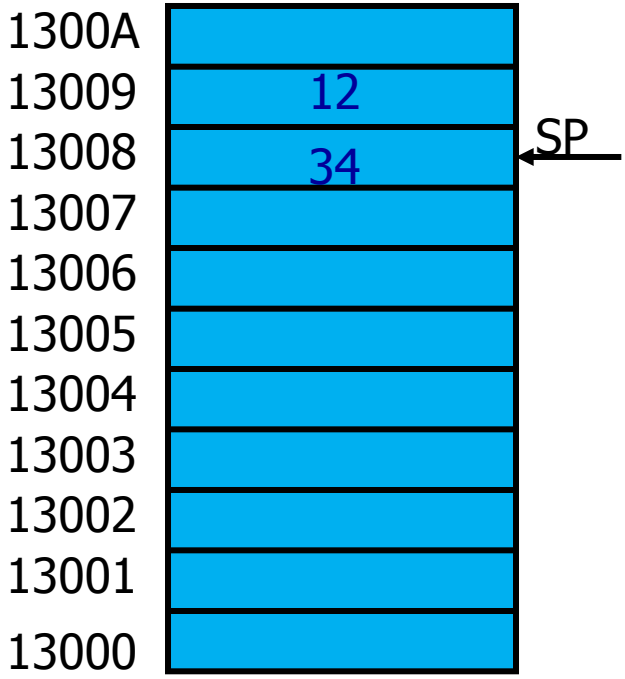
- Cất nội dung của thanh ghi cờ vào ngăn xếp



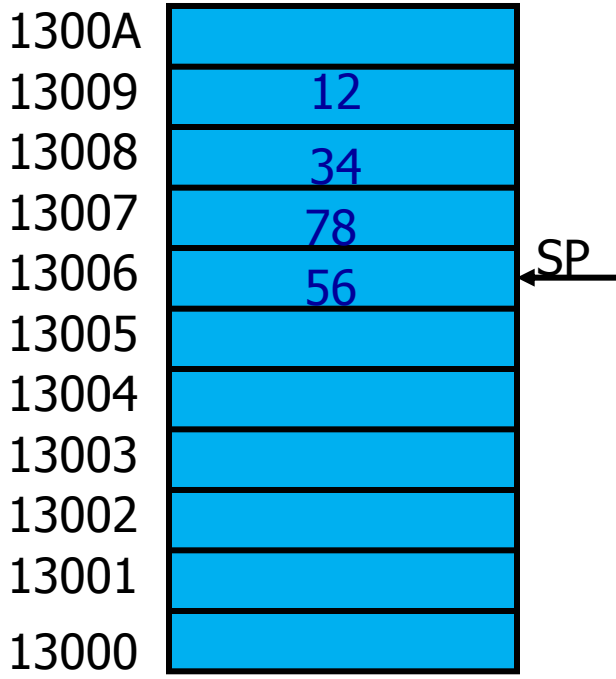
• Ví dụ về lệnh PUSH



PUSH AX



PUSH BX



SS 1300

SP 000A

AX 1234

SS 1300

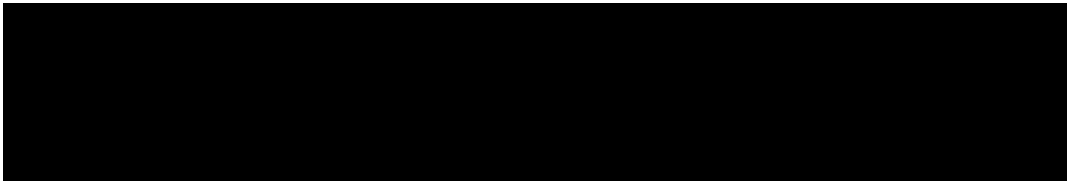
SP 0008

AX 1234

SS 1300

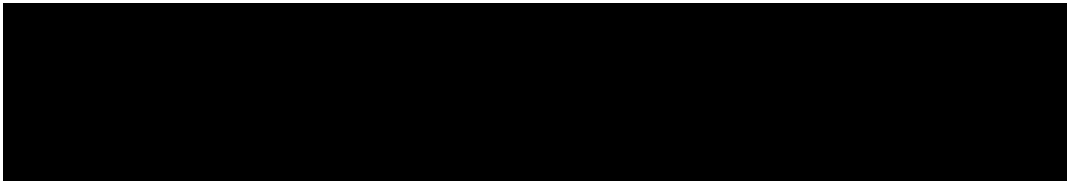
SP 0006

BX 7856



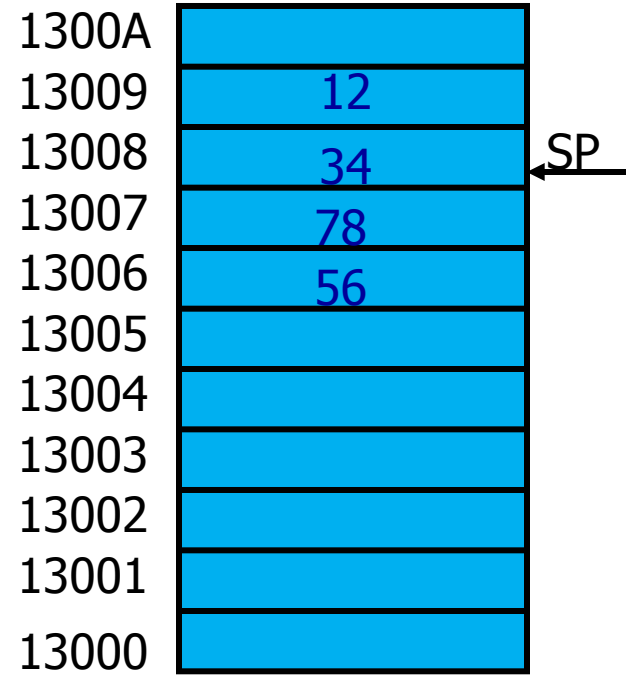
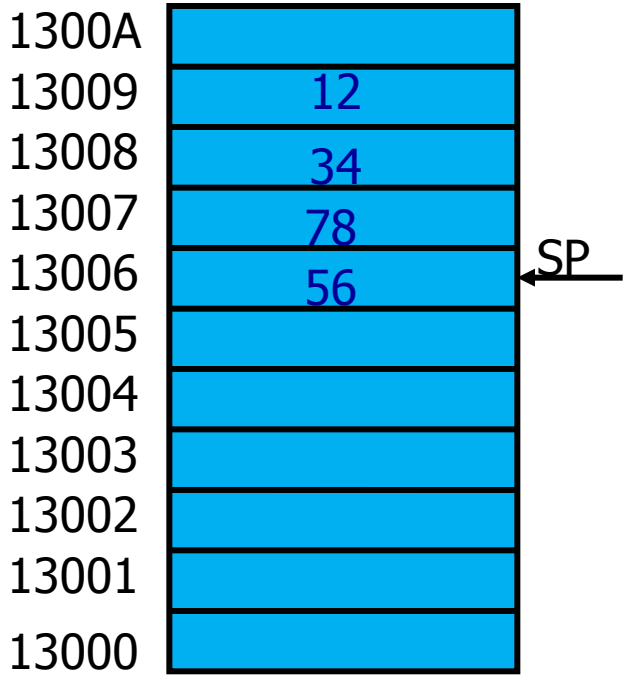
- **Lệnh POP**
 - ❑ Dùng để lấy lại 1 từ vào thanh ghi hoặc ô nhớ từ đỉnh ngăn xếp
 - ❑ Cú pháp: POP Đích
 - ❑ Mô tả: {SP} => Đích, SP=SP+2
 - ❑ Giới hạn: thanh ghi 16 bit (trừ CS) hoặc là 1 từ nhớ
 - ❑ Lệnh này không tác động đến cờ
 - ❑ Ví dụ:
 - ⇒ POP BX
 - ⇒ POP PTR[BX]

- **Lệnh POPF**
 - ❑ Lấy 1 từ từ đỉnh ngăn xếp rồi đưa vào thanh ghi cờ



- Ví dụ lệnh POP

POP DX



SS 1 3 0 0

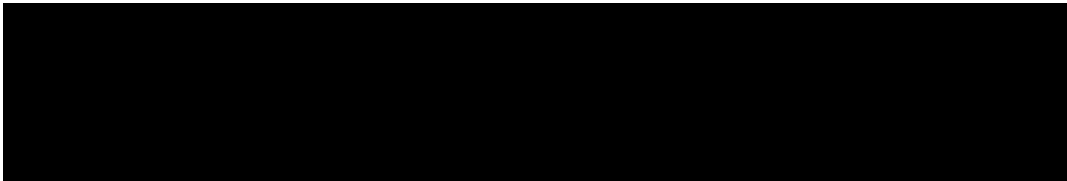
SP 0 0 0 6

DX 3 2 5 4

SS 1 3 0 0

SP 0 0 0 8

DX 7 8 5 6

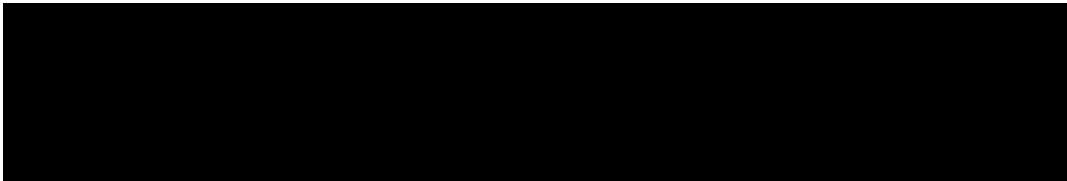


- **Lệnh IN**

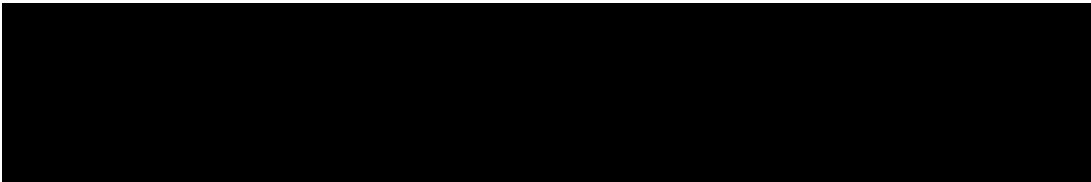
- ❑ Dùng để đọc 1 byte hoặc 2 byte dữ liệu từ cổng vào thanh ghi AL hoặc AX
- ❑ Cú pháp: IN Acc, Port
- ❑ Lệnh này không tác động đến cờ
- ❑ Ví dụ:
 - ⇒ IN AX, 00H
 - ⇒ IN AL, F0H
 - ⇒ IN AX, DX

- **Lệnh OUT**

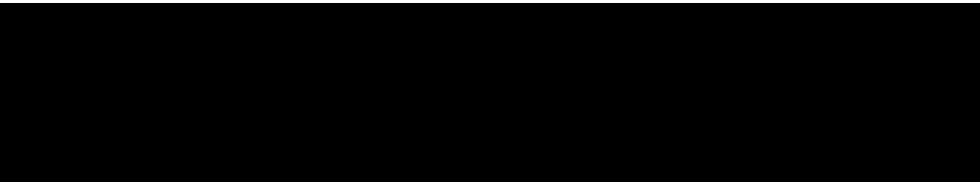
- ❑ Dùng để đưa 1 byte hoặc 2 byte dữ liệu từ thanh ghi AL hoặc AX ra cổng
- ❑ Cú pháp: OUT Port, Acc
- ❑ Lệnh này không tác động đến cờ
- ❑ Ví dụ:
 - ⇒ OUT 00H, AX
 - ⇒ OUT F0H, AL
 - ⇒ OUT DX, AX



- Các lệnh di chuyển chuỗi MOVSB, MOVSW
 - ❑ Dùng để chuyển một phần tử của chuỗi này sang một chuỗi khác
 - ❑ Cú pháp: MOVSB chuỗi đích, chuỗi nguồn
MOVSW
 - ❑ Thực hiện:
 - ⇒ DS:SI là địa chỉ của phần tử trong chuỗi nguồn
 - ⇒ ES:DI là địa chỉ của phần tử trong chuỗi đích
 - ⇒ Sau mỗi lần chuyển $SI=SI \pm 1$, $DI=DI \pm 1$ hoặc $SI=SI \pm 2$, $DI=DI \pm 2$ tùy thuộc vào cờ hướng DF là 0/1
 - ❑ Lệnh này không tác động đến cờ
 - ❑ Ví dụ:
 - ⇒ MOVSB byte1, byte2

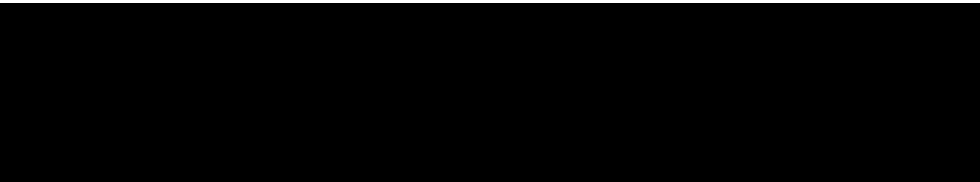


- Cấu trúc bên trong
- Mô tả tập lệnh của 8086
 - Các lệnh di chuyển dữ liệu
 - Các lệnh số học và logic
 - Các lệnh điều khiển chương trình
- Lập trình hợp ngữ với 8086



- ADD, ADC, SUB, MUL, IMUL, DIV, IDIV, INC, DEC
- AND, OR, NOT, NEG, XOR
- Lệnh quay và dịch: RCL, RCR, SAL, SAR, SHL, SHR
- Lệnh so sánh: CMP, CMPS

- Lệnh ADD
 - ❑ Lệnh cộng hai toán hạng
 - ❑ Cú pháp: ADD Đích, nguồn
 - ❑ Thực hiện: Đích=Đích + nguồn
 - ❑ Giới hạn: toán hạng không được là 2 ô nhớ và thanh ghi đoạn
 - ❑ Lệnh này thay đổi cờ: AF, CF, OF, PF, SF, ZF
 - ❑ Ví dụ:
 - ⇒ ADD AX, BX
 - ⇒ ADD AX, 40H

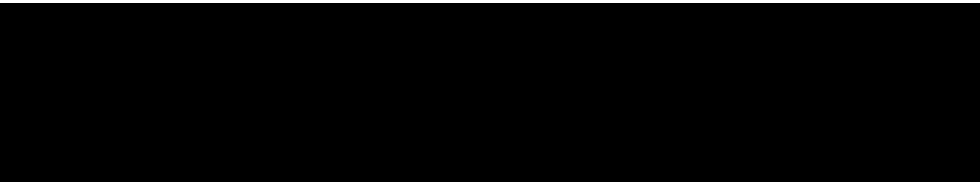


- **Lệnh ADC**

- Lệnh cộng có nhớ hai toán hạng
- Cú pháp: ADC Đích, nguồn
- Thực hiện: $\text{Đích} = \text{Đích} + \text{nguồn} + \text{CF}$
- Giới hạn: toán hạng không được là 2 ô nhớ và thanh ghi đoạn
- Lệnh này thay đổi cờ: AF, CF, OF, PF, SF, ZF
- Ví dụ:
 - ⇒ ADC AL, 30H

- **Lệnh SUB**

- Lệnh trừ
- Cú pháp: SUB Đích, nguồn
- Thực hiện: $\text{Đích} = \text{Đích} - \text{nguồn}$
- Giới hạn: toán hạng không được là 2 ô nhớ và thanh ghi đoạn
- Lệnh này thay đổi cờ: AF, CF, OF, PF, SF, ZF
- Ví dụ:
 - ⇒ SUB AL, 30H

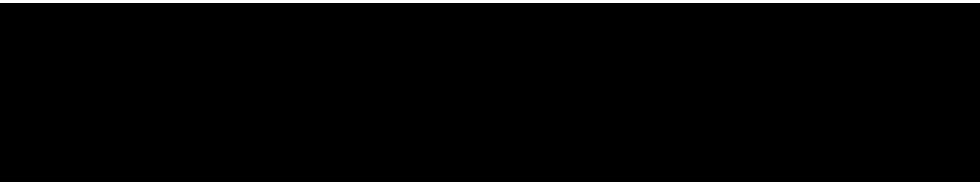


- **Lệnh MUL**

- Lệnh nhân số không dấu
- Cú pháp: MUL nguồn
- Thực hiện:
 - ⇒ $AX = AL * \text{nguồn}_{8\text{bit}}$
 - ⇒ $DXAX = AX * \text{nguồn}_{16\text{bit}}$
- Lệnh này thay đổi cờ: CF, OF
- Ví dụ:
 - ⇒ MUL BL

- **Lệnh IMUL**

- nhân số có dấu



- **Lệnh DIV**

- Lệnh chia 2 số không dấu

- Cú pháp: DIV nguồn

- Thực hiện:

- ⇒ AL = thương (AX / nguồn8bit) ; AH=dư (AX / nguồn8bit)

- ⇒ AX = thương (DXAX / nguồn16bit) ; DX=dư (DXAX / nguồn16bit)

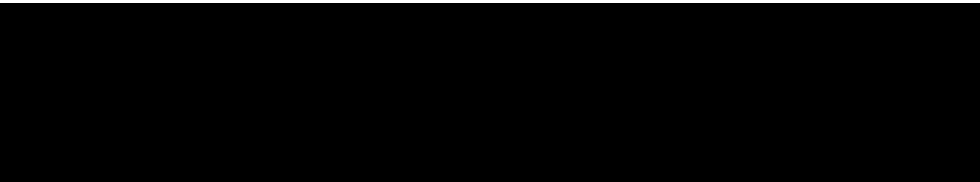
- Lệnh này không thay đổi cờ

- Ví dụ:

- ⇒ DIV BL

- **Lệnh IDIV**

- chia 2 số có dấu

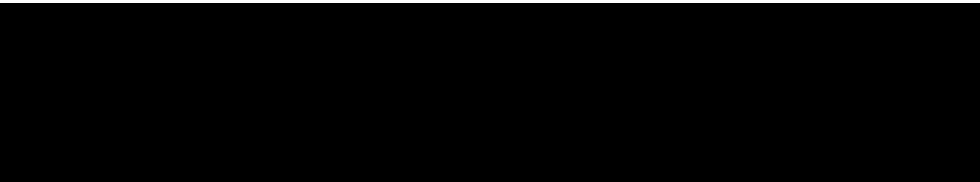


- **Lệnh INC**

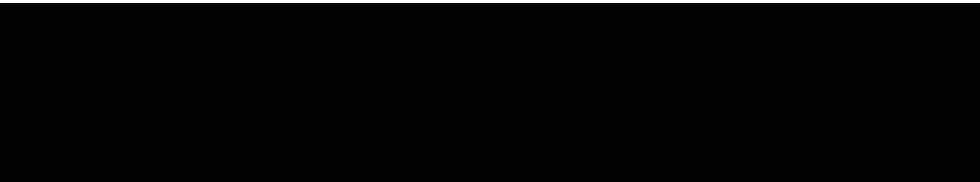
- Lệnh cộng 1 vào toán hạng là thanh ghi hoặc ô nhớ
- Cú pháp: INC Đích
- Thực hiện: Đích=Đích + 1
- Lệnh này thay đổi cờ: AF, OF, PF, SF, ZF
- Ví dụ:
 - ⇒ INC AX

- **Lệnh DEC**

- Lệnh trừ 1 từ nội dung một thanh ghi hoặc ô nhớ
- Cú pháp: DEC Đích
- Thực hiện: Đích=Đích - 1
- Lệnh này thay đổi cờ: AF, OF, PF, SF, ZF
- Ví dụ:
 - ⇒ DEC [BX]



- **Lệnh AND**
 - ❑ Lệnh AND logic 2 toán hạng
 - ❑ Cú pháp: AND Đích, nguồn
 - ❑ Thực hiện: Đích=Đích And nguồn
 - ❑ Giới hạn: toán hạng không được là 2 ô nhớ hoặc thanh ghi đoạn
 - ❑ Lệnh này thay đổi cờ: PF, SF, ZF và xoá cờ CF, OF
 - ❑ Ví dụ:
 - ⇒ AND BL, 0FH
- **Lệnh XOR, OR: tương tự như lệnh AND**
- **Lệnh NOT: đảo từng bit của toán hạng**
- **Lệnh NEG: xác định số bù 2 của toán hạng**



- **Lệnh CMP**

- ❑ Lệnh so sánh 2 byte hoặc 2 từ

- ❑ Cú pháp: CMP Đích, nguồn

- ❑ Thực hiện:

- ⇒ Đích = nguồn : CF=0 ZF=1

- ⇒ Đích > nguồn : CF=0 ZF=0

- ⇒ Đích < nguồn : CF=1 ZF=0

- ❑ Giới hạn: toán hạng phải cùng độ dài và không được là 2 ô nhớ

- **Lệnh CMPS**

- ❑ Dùng để so sánh từng phần tử của 2 chuỗi có các phần tử cùng loại

- ❑ Cú pháp: CMPS chuỗi đích, chuỗi nguồn

- CMPSB

- CMPSW

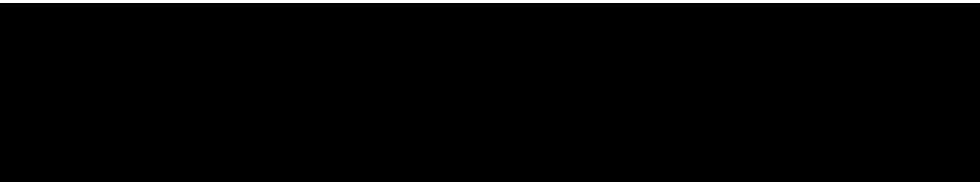
- ❑ Thực hiện:

- ⇒ DS:SI là địa chỉ của phần tử trong chuỗi nguồn

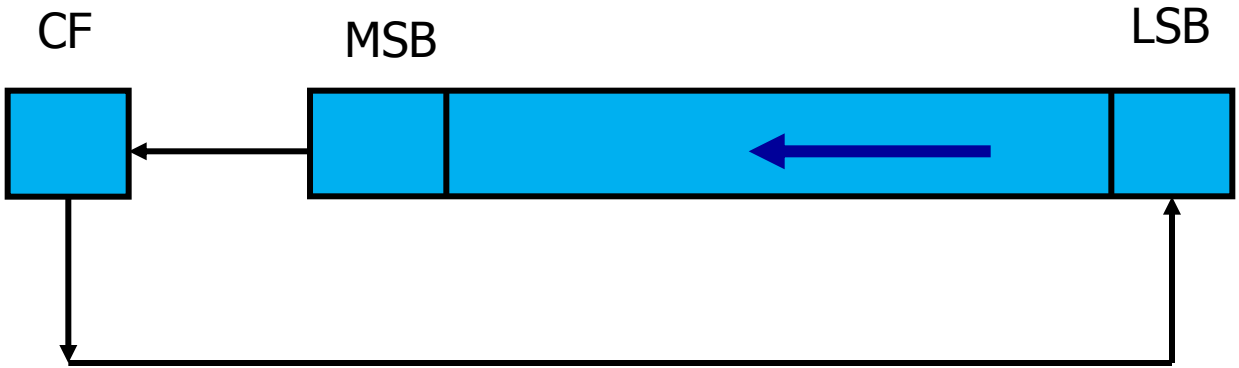
- ⇒ ES:DI là địa chỉ của phần tử trong chuỗi đích

- ⇒ Sau mỗi lần so sánh SI=SI +/- 1, DI=DI +/- 1 hoặc SI=SI +/- 2, DI=DI +/- 2 tùy thuộc vào cờ hướng DF là 0/1

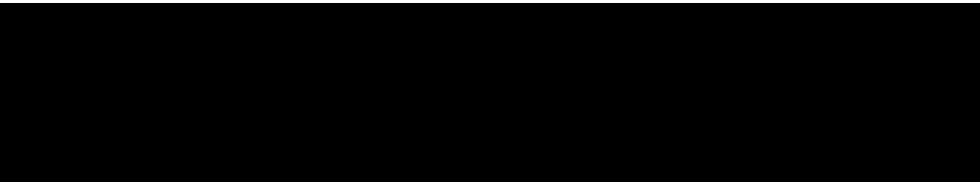
- ❑ Cập nhật cờ AF, CF, OF, PF, SF, ZF



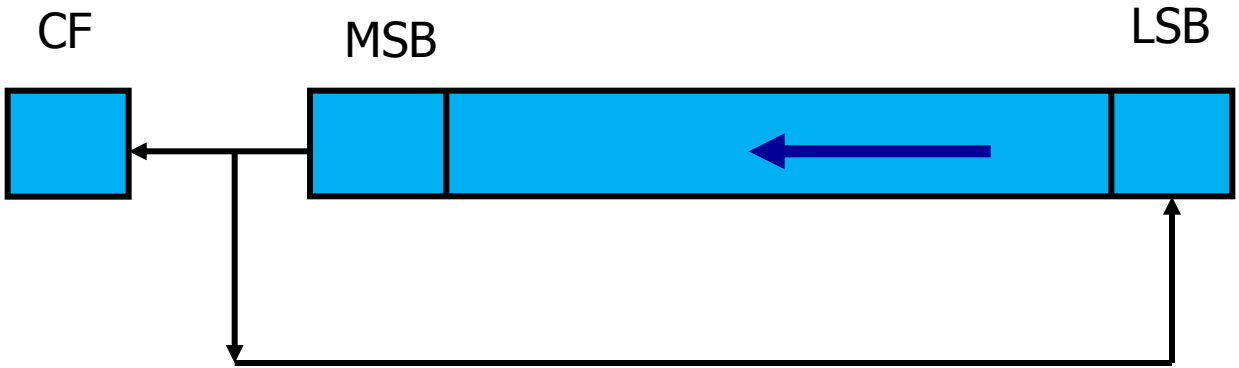
- **Lệnh RCL**
 - ❑ Lệnh quay trái thông qua cờ nhớ
 - ❑ Cú pháp: RCL Đích, CL (với số lần quay lớn hơn 1)
RCL Đích, 1
RCL Đích, Số lần quay (80286 trở lên)
 - ❑ Thực hiện: quay trái đích CL lần
 - ❑ Đích là thanh ghi (trừ thanh ghi đoạn) hoặc ô nhớ
 - ❑ Lệnh này thay đổi cờ: CF, OF



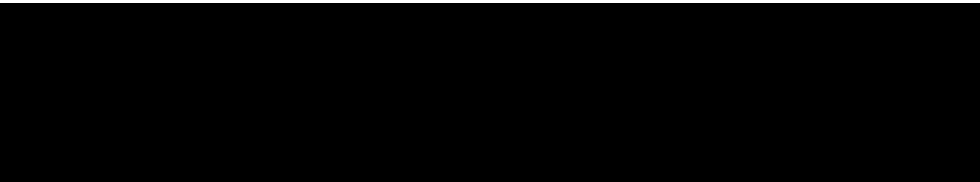
- **Lệnh RCR**
 - ❑ Lệnh quay phải thông qua cờ nhớ



- **Lệnh ROL**
 - ❑ Lệnh quay trái
 - ❑ Cú pháp: ROL Đích, CL (với số lần quay lớn hơn 1)
ROL Đích, 1
 - ROL Đích, Số lần quay (80286 trở lên)
 - ❑ Thực hiện: quay trái đích CL lần
 - ❑ Đích là thanh ghi (trừ thanh ghi đoạn) hoặc ô nhớ
 - ❑ Lệnh này thay đổi cờ: CF, OF



- **Lệnh ROR**
 - ❑ Lệnh quay phải



- **Lệnh SAL**

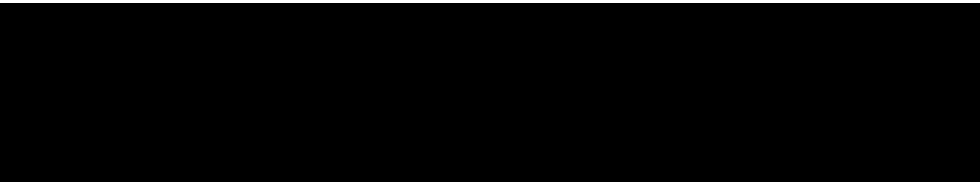
- Lệnh dịch trái số học
- Cú pháp: SAL Đích, CL (với số lần dịch lớn hơn 1)
SAL Đích, 1
SAL Đích, số lần dịch (80286 trở lên)

- Thực hiện: dịch trái đích CL bit tương đương với $\text{Đích} = \text{Đích} * 2^{CL}$
- Lệnh này thay đổi cờ SF, ZF, PF



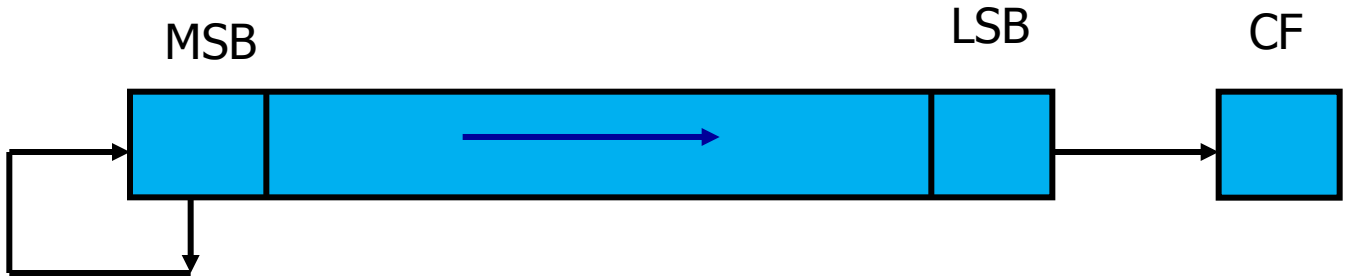
- **Lệnh SHL**

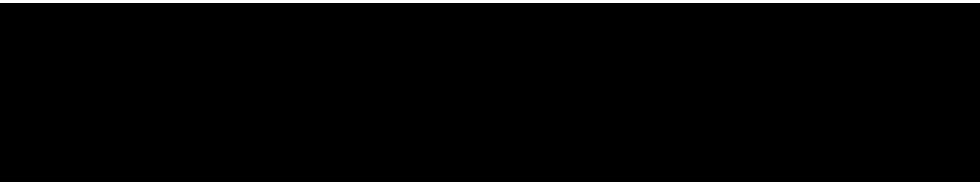
- Lệnh dịch trái logic tương tự như SAL



- **Lệnh SAR**

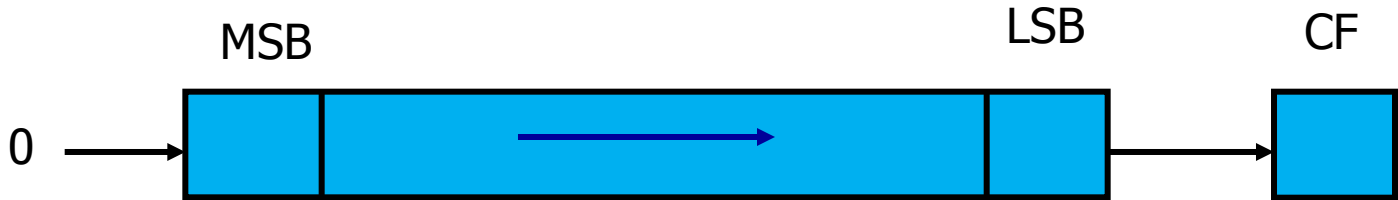
- ❑ Lệnh dịch phải số học
- ❑ Cú pháp: SAR Đích, CL (với số lần dịch lớn hơn 1)
SAR Đích, 1
hoặc SAR Đích, số lần dịch (80286 trở lên)
- ❑ Thực hiện: dịch phải đích CL bit
- ❑ Lệnh này thay đổi cờ SF, ZF, PF, CF mang giá trị của MSB



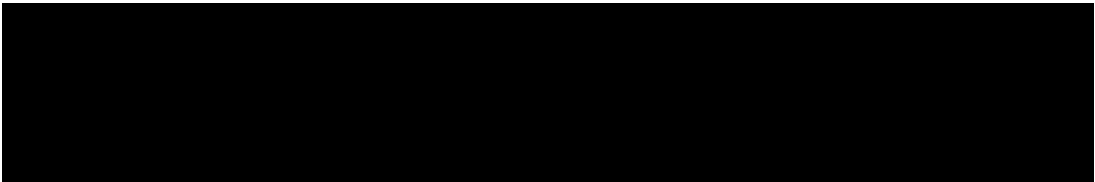


- **Lệnh SHR**

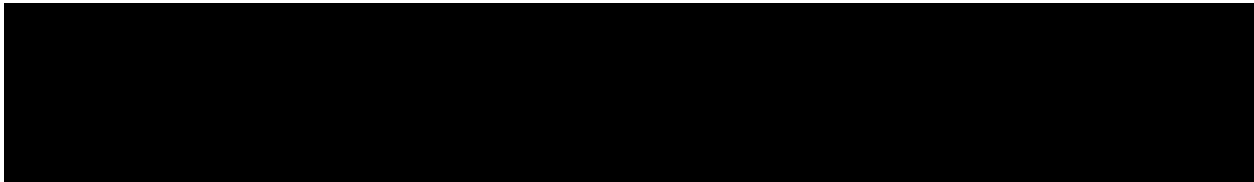
- ❑ Lệnh dịch phải logic
- ❑ Cú pháp: SHR Đích, CL (với số lần dịch lớn hơn 1)
SHR Đích, 1
hoặc SHR Đích, số lần dịch (80286 trở lên)
- ❑ Thực hiện: dịch phải đích CL bit
- ❑ Lệnh này thay đổi cờ SF, ZF, PF, CF mang giá trị của LSB



Chú ý:
Trong các lệnh dịch và quay, toán hạng không được là thanh ghi đoạn



- Cấu trúc bên trong
- Mô tả tập lệnh của 8086
 - Các lệnh di chuyển dữ liệu
 - Các lệnh số học và logic
 - Các lệnh điều khiển chương trình**
 - ⇒ Lệnh nhảy không điều kiện: JMP
 - ⇒ Lệnh nhảy có điều kiện JE, JG, JGE, JL, JLE...
 - ⇒ Lệnh lặp LOOP
 - ⇒ Lệnh gọi chương trình con CALL
 - ⇒ Lệnh gọi chương trình con phục vụ ngắt INT và IRET
- Lập trình hợp ngữ với 8086



- Dùng để nhảy tới một địa chỉ trong bộ nhớ
- 3 loại: nhảy ngắn, gần và xa

☐ Lệnh nhảy ngắn (short jump)

⇒ Độ dài lệnh 2 bytes:



⇒ Phạm vi nhảy: -128 đến 127 bytes so với lệnh tiếp theo lệnh JMP

⇒ Thực hiện: $IP = IP + \text{độ lệch}$

⇒ Ví dụ:

```
XOR BX, BX
Nhan: MOV AX, 1
      ADD AX, BX
      JMP SHORT Nhan
```

□ Lệnh nhảy gần (near jump)

⇒ Phạm vi nhảy: ± 32 Kbytes so với lệnh tiếp theo lệnh JMP

⇒ Ví dụ:

XOR BX, BX

Nhan: MOV AX, 1

ADD AX, BX

JMP NEAR Nhan

XOR CX, CX

MOV AX, 1

ADD AX, BX

JMP NEAR PTR BX

XOR CX, CX

MOV AX, 1

ADD AX, BX

JMP WORD PTR [BX]

Thực hiện: $IP = IP + \text{độ lệch}$

$IP = BX$

$IP = [BX+1] [BX]$

E 9

Độ lệchLo

Độ lệchHi

Nhảy gián tiếp

□ Lệnh nhảy xa (far jump)

⇒ Độ dài lệnh 5 bytes đối với nhảy tới nhãn:



⇒ Phạm vi nhảy: nhảy trong 1 đoạn mã hoặc nhảy sang đoạn mã khác

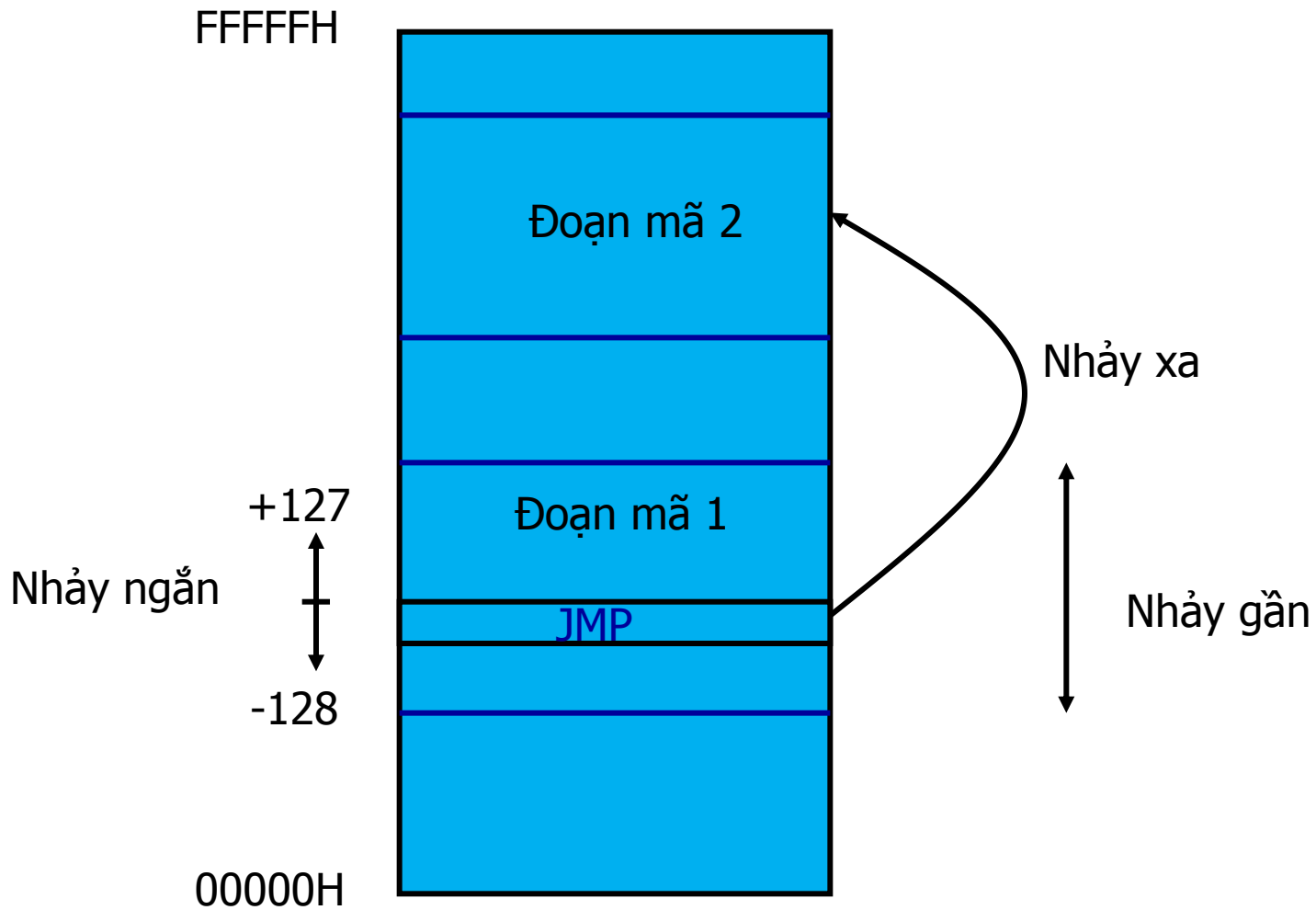
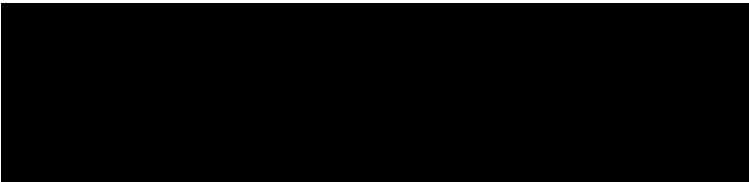
⇒ Ví dụ:

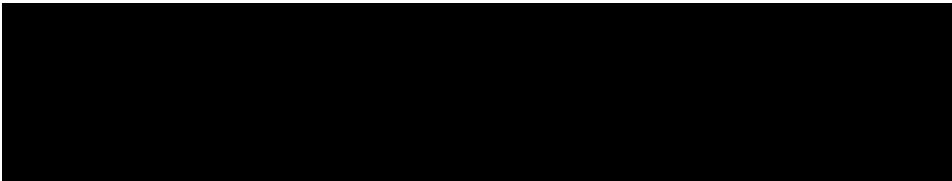
```
EXTRN  Nhan: FAR
Next: MOV AX, 1
      ADD AX, BX
      JMP FAR PTR Next
      .....
      JMP FAR Nhan
```

```
XOR CX, CX
MOV AX, 1
ADD AX, BX
JMP DWORD PTR [BX]
```

Thực hiện: IP=IP của nhãn
CS=CS của nhãn

IP = [BX+1][BX]
CS = [BX+3][BX+2]

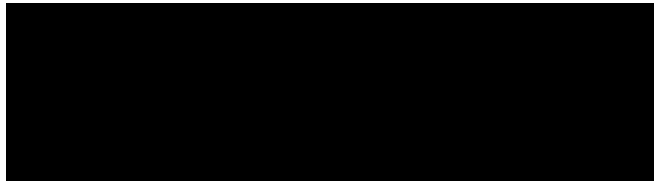




- JE or JZ, JNE or JNZ, JG, JGE, JL, JLE (dùng cho số có dấu) và JA, JB, JAE, JBE (dùng cho số không dấu) ...
- Nhảy được thực hiện phụ thuộc vào các cờ
- Là các lệnh nhảy ngắn
- Ví dụ:

```
Nhan1: XOR BX, BX
Nhan2: MOV AX, 1
        CMP AL, 10H
        JNE Nhan1
        JE  Nhan2
```

Thực hiện: $IP = IP + \text{độ dịch}$



- LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ
- Là lệnh phối hợp giữa DEC CX và JNZ

```
XOR AL, AL
MOV CX, 16
Lap: INC AL
LOOP Lap
```

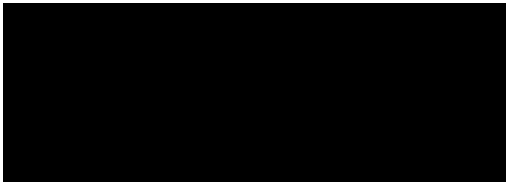
Lặp đến khi CX=0

```
XOR AL, AL
MOV CX, 16
Lap: INC AL
      CMP AL, 10
LOOPE Lap
```

Lặp đến khi CX=0
hoặc AL<>10

```
XOR AL, AL
MOV CX, 16
Lap: INC AL
      CMP AL, 10
LOOPNE Lap
```

Lặp đến khi CX=0
hoặc AL=10



- Dùng để gọi chương trình con
- Có 2 loại: CALL gần và CALL xa
 - CALL gần (near call): tương tự như nhảy gần
 - ⇒ Gọi chương trình con ở trong cùng một đoạn mã

```
Tong PROC NEAR
    ADD AX, BX
    ADD AX, CX
    RET
Tong ENDP
...
CALL Tong
```

Cất IP vào ngăn xếp
IP=IP + dịch chuyển
RET: lấy IP từ ngăn xếp

```
Tong PROC NEAR
    ADD AX, BX
    ADD AX, CX
    RET
Tong ENDP
...
MOV BX, OFFSET Tong
CALL BX
```

Cất IP vào ngăn xếp
IP= BX
RET: lấy IP từ ngăn xếp

```
CALL WORD PTR [BX]
```

Cất IP vào ngăn xếp
IP= [BX+1] [BX]
RET: lấy IP từ ngăn xếp

- CALL xa (far call): tương tự như nhảy xa
⇒ Gọi chương trình con ở ngoài đoạn mã

Tong PROC FAR

ADD AX, BX

ADD AX, CX

RET

Tong ENDP

...

CALL Tong

CALL DWORD PTR [BX]

Cất CS vào ngăn xếp

Cất IP vào ngăn xếp

IP = [BX+1][BX]

CS = [BX+3][BX+2]

RET: lấy IP từ ngăn xếp

lấy CS từ ngăn xếp

Cất CS vào ngăn xếp

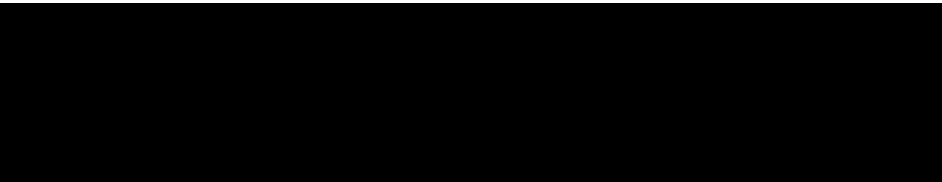
Cất IP vào ngăn xếp

IP = IP của Tong

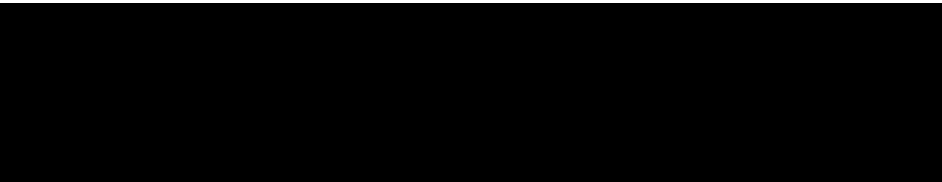
CS = CS của Tong

RET: lấy IP từ ngăn xếp

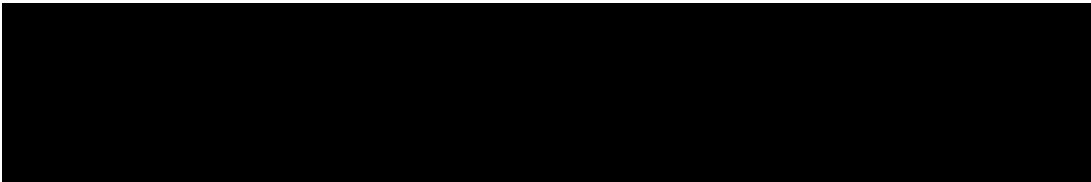
lấy CS từ ngăn xếp



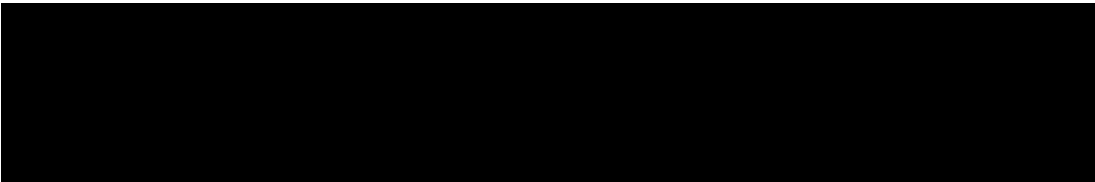
- INT gọi chương trình con phục vụ ngắt (CTCPVN)
- Bảng vector ngắt: 1 Kbytes 00000H đến 003FF H
 - ❑ 256 vector ngắt
 - ❑ 1 vector 4 bytes, chứa IP và CS của CTCPVN
 - ❑ 32 vector đầu dành riêng cho Intel
 - ❑ 224 vector sau dành cho người dùng
- Cú pháp: INT Number
- Ví dụ: INT 21H gọi CTCPVN của DOS



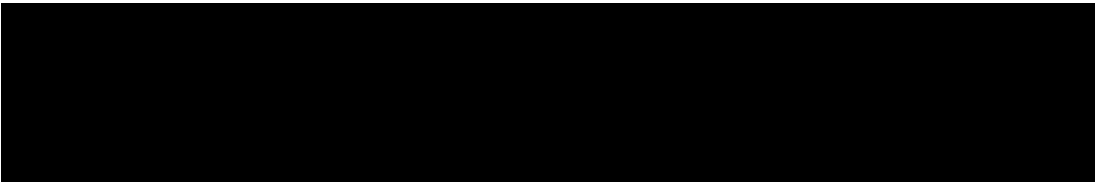
- Thực hiện INT:
 - Cắt thanh ghi cờ vào ngăn xếp
 - IF=0 (cấm các ngắt khác tác động), TF=0 (chạy suốt)
 - Cắt CS vào ngăn xếp
 - Cắt IP vào ngăn xếp
 - $IP=[N*4]$, $CS=[N*4+2]$
- Gặp IRET:
 - Lấy IP từ ngăn xếp
 - Lấy CS từ ngăn xếp
 - Lấy thanh ghi cờ từ ngăn xếp



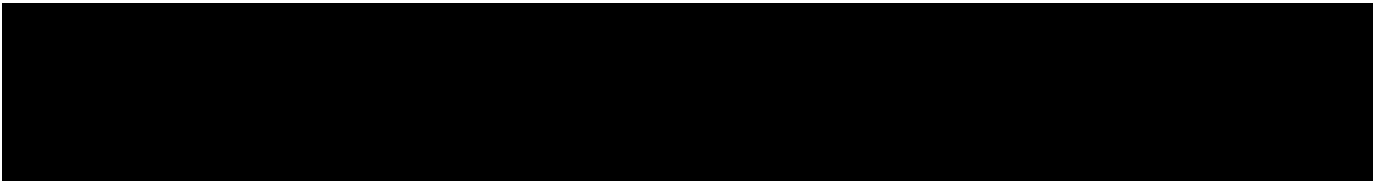
-
- Cấu trúc bên trong
 - Mô tả tập lệnh của 8086
 - Lập trình hợp ngữ 8086



-
- Giới thiệu khung của chương trình hợp ngữ
 - Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
 - Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
 - Một số chương trình cụ thể



- Giới thiệu khung của chương trình hợp ngữ
 - ❑ Cú pháp của chương trình hợp ngữ
 - ❑ Dữ liệu cho chương trình
 - ❑ Biến và hằng
 - ❑ Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể



1. **.Model Small** → khai báo kiểu kích thước bộ nhớ

2. **.Stack 100** → khai báo đoạn ngăn xếp

3. **.Data** → khai báo đoạn dữ liệu

4. Tbao DB 'Chuoi da sap xep:', 10, 13

5. MGB DB 'a', 'Y', 'G', 'T', 'y', 'Z', 'U', 'B', 'D', 'E',

6. DB '\$'

7. **.Code** → khai báo đoạn mã lệnh

8. **MAIN Proc** → bắt đầu chương trình chính

9. MOV AX, @Data ;khởi đầu DS

10. MOV DS, AX

11. MOV BX, 10 ;BX: so phan tu cua mang

12. LEA DX, MGB ;DX chỉ vào đầu mang byte

13. DEC BX ;so vong so sanh phai lam

14. LAP: MOV SI, DX ; SI chỉ vào đầu mang

15. MOV CX, BX ; CX số lần số của vòng số

16. MOV DI, SI ;gia su ptu dau la max

17. MOV AL, [DI] ;AL chưa phân tử max

18. TIMMAX:

19. INC SI ;chỉ vào phân tử bên cạnh

20. CMP [SI], AL ; phân tử mới > max?

21. JNG TIEP ;không, tìm max

22. MOV DI, SI ; dung, DI chỉ vào max

23. MOV AL, [DI] ;AL chưa phân tử max

24. TIEP: LOOP TIMMAX ;tìm max của một vòng số

25. CALL DOICHO ;đổi cho max với số mới

26. DEC BX ;so vong so con lai

27. JNZ LAP ;lam tien vong so moi

28. MOV AH, 9 ; hien thi chuoi da sap xep

29. MOV DX, Tbao

30. INT 21H

31. MOV AH, 4CH ;ve DOS

32. INT 21H

33. **MAIN Endp** → kết thúc chương trình chính

34. **DOICHO Proc** → bắt đầu chương trình con

35. PUSH AX

36. MOV AL, [SI]

37. XCHG AL, [DI]

38. MOV [SI], AL

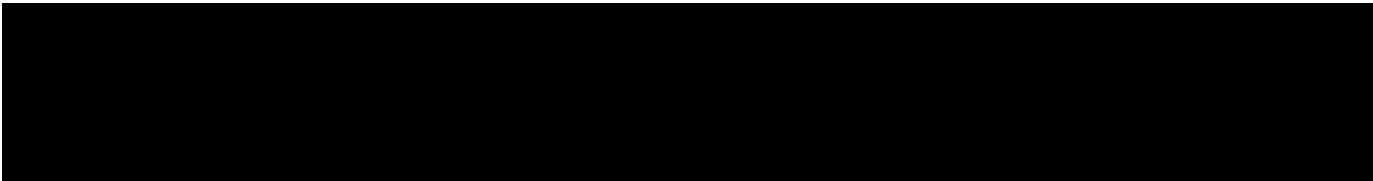
39. POP AX

40. RET

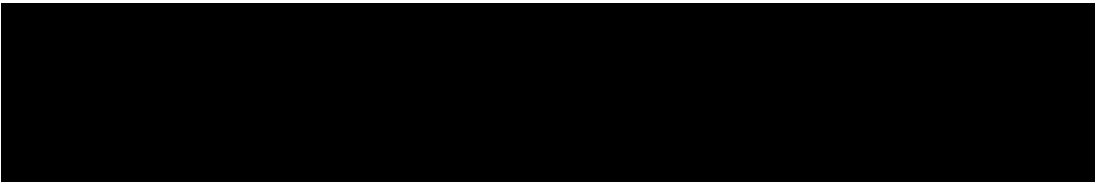
41. **DOICHO Endp** → kết thúc đoạn mã

42. **END MAIN**

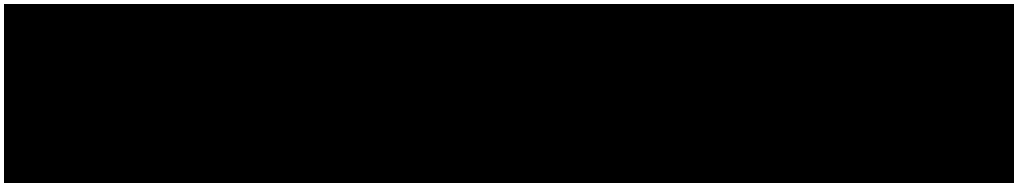
chú thích bắt đầu bằng dấu ;



- Tên Mã lệnh Các toán hạng ; chú giải
- Chương trình dịch không phân biệt chữ hoa, chữ thường
- Trường tên:
 - chứa các nhãn, tên biến, tên thủ tục
 - độ dài: 1 đến 31 ký tự
 - tên không được có dấu cách, không bắt đầu bằng số
 - được dùng các ký tự đặc biệt: ? . @ _ \$ %
 - dấu . phải được đặt ở vị trí đầu tiên nếu sử dụng

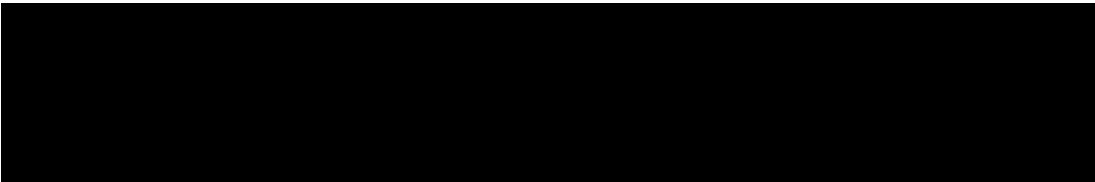


- Giới thiệu khung của chương trình hợp ngữ
 - Cú pháp của chương trình hợp ngữ
 - Dữ liệu cho chương trình**
 - Biến và hằng
 - Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

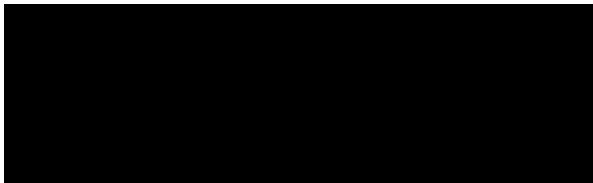


- Dữ liệu:

- các số hệ số 2: 0011B
- hệ số 10: 1234
- hệ số 16: 1EF1H, 0ABBAAH
- Ký tự, chuỗi ký tự: 'A', 'abcd'



- Giới thiệu khung của chương trình hợp ngữ
 - Cú pháp của chương trình hợp ngữ
 - Dữ liệu cho chương trình
 - Biến và hằng**
 - Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể



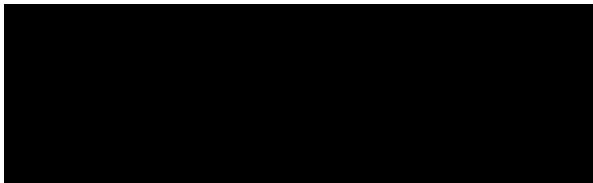
- DB (define byte): định nghĩa biến kiểu byte
- DW (define word): định nghĩa biến kiểu từ
- DD (define double word): định nghĩa biến kiểu từ kép

- Biến byte:

- ❑ Tên DB gia_trị_khởi_đầu

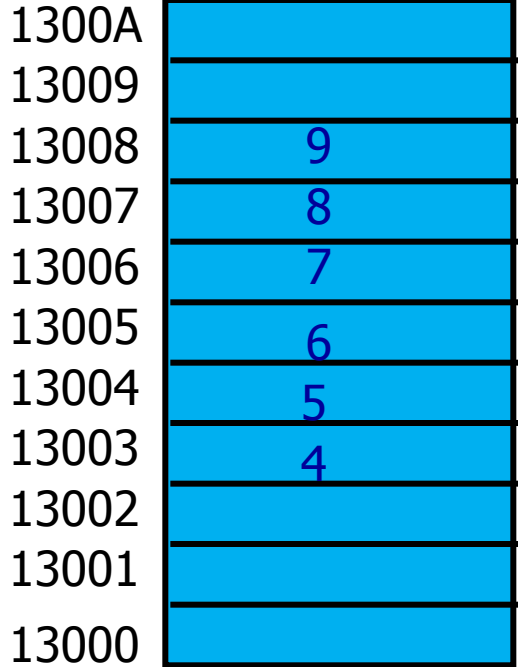
- ❑ Ví dụ:

⇒ B1	DB	4	MOV AL, B1
⇒ B1	DB	?	LEA BX, B1
⇒ C1	DB	'\$'	MOV AL, [BX]
⇒ C1	DB	34	



• Biến từ:

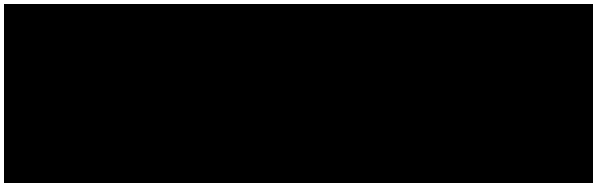
- Tên DW gia_trị_khởi đầu
- Ví dụ:
 - ⇒ W1 DW 4
 - ⇒ W2 DW ?



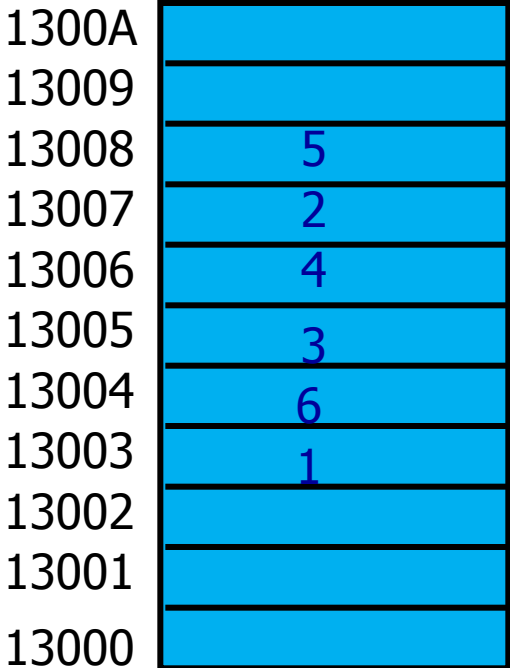
M1

• Biến mảng:

- M1 DB 4, 5, 6, 7, 8, 9
- M2 DB 100 DUP(0)
- M3 DB 100 DUP(?)
- M4 DB 4, 3, 2, 2 DUP (1, 2 DUP(5), 6)
- M4 DB 4, 3, 2, 1, 5, 5, 6, 1, 5, 5, 6



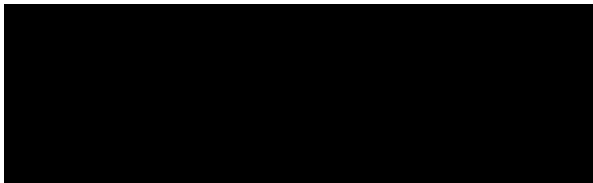
• Biến mảng 2 chiều:

$$\begin{pmatrix} 1 & 6 & 3 \\ 4 & 2 & 5 \end{pmatrix}$$


```
□ M1 DB 1, 6, 3
      DB 4, 2, 5
```

```
□ M2 DB 1, 4
      DB 6, 2
      DB 3, 5
```

```
MOV AL, M1 ; copy 1 vao AL
MOV AH, M1[2]
MOV BX, 1
MOV SI, 1
MOV CL, M1[BX+SI]
MOV AX, Word Ptr M1[BX+SI+2]
MOV DL, M1[BX][SI]
```



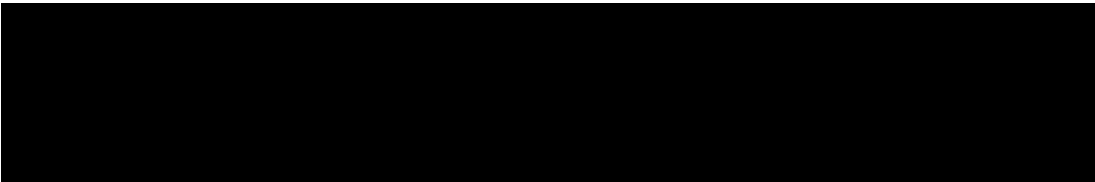
- Biến kiểu xâu ký tự

- STR1 DB 'string'
- STR2 DB 73h, 74h, 72h, 69h, 6Eh, 67h
- STR3 DB 73h, 74h, 'r', 'i', 6Eh, 67h

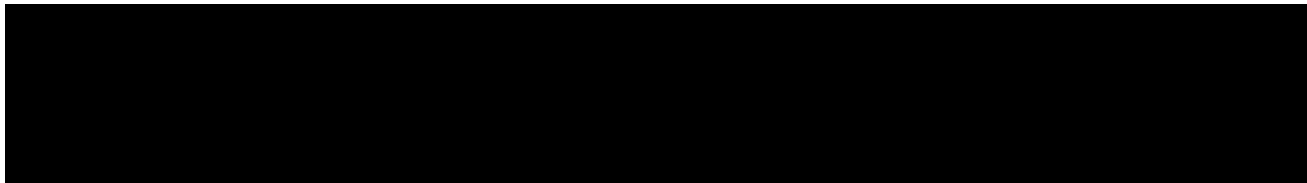
- Hằng có tên

- Có thể khai báo hằng ở trong chương trình
- Thường được khai báo ở đoạn dữ liệu
- Ví dụ:
 - ⇒ CR EQU 0Dh ;CR là carriage return
 - ⇒ LF EQU 0Ah ; LF là line feed
 - ⇒ CHAO EQU 'Hello'

 - ⇒ MSG DB CHAO, '\$'

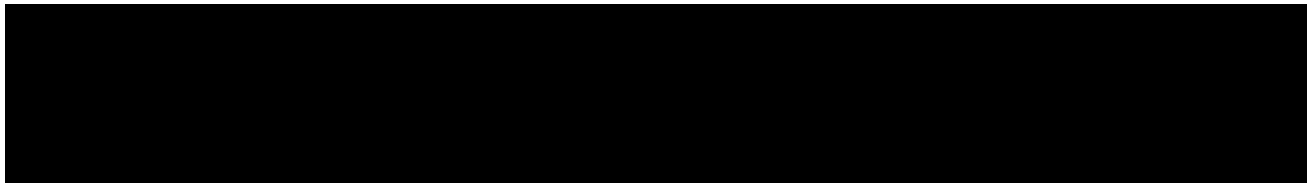


- Giới thiệu khung của chương trình hợp ngữ
 - Cú pháp của chương trình hợp ngữ
 - Dữ liệu cho chương trình
 - Biến và hằng
 - Khung của một chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể

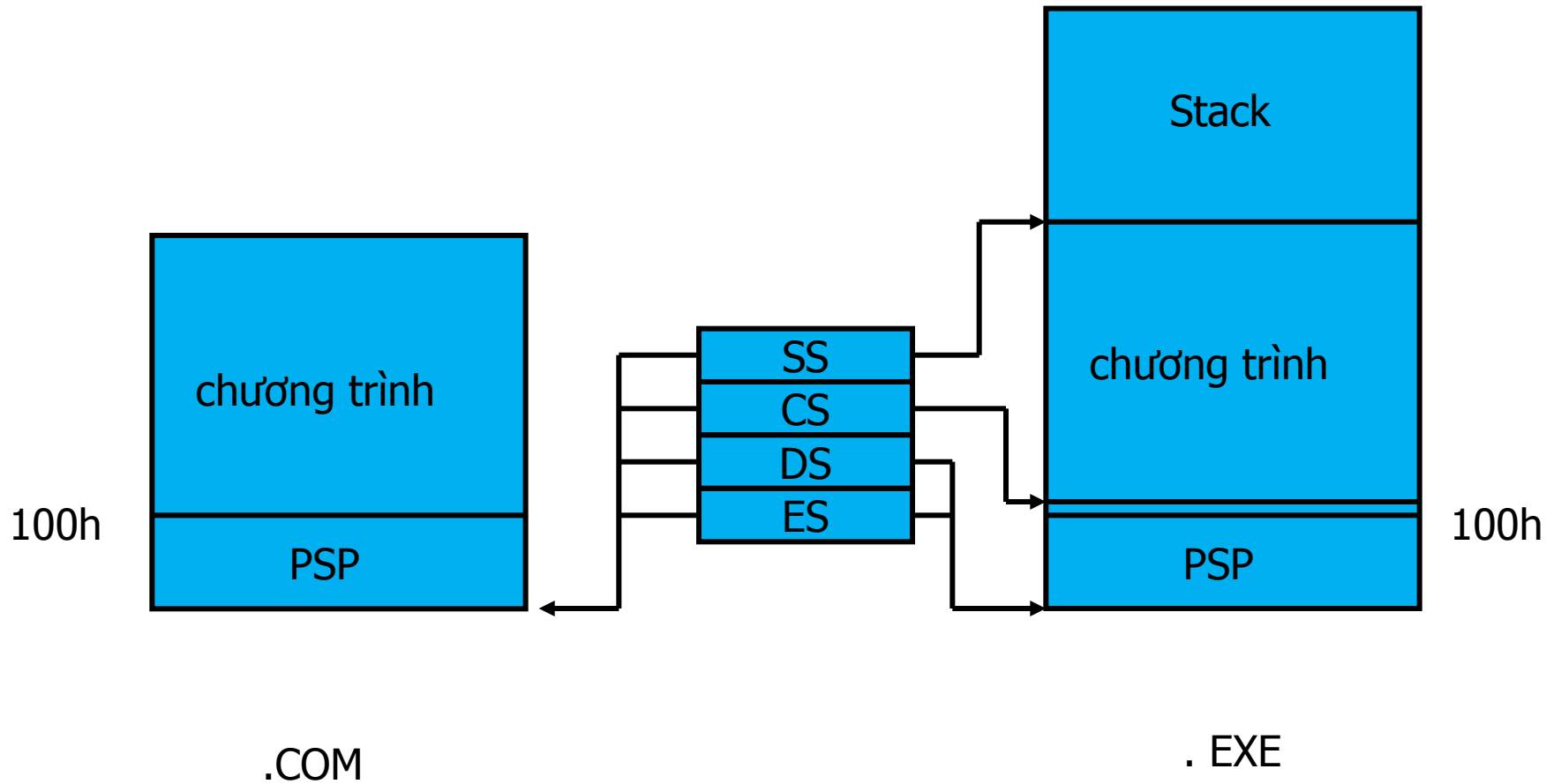


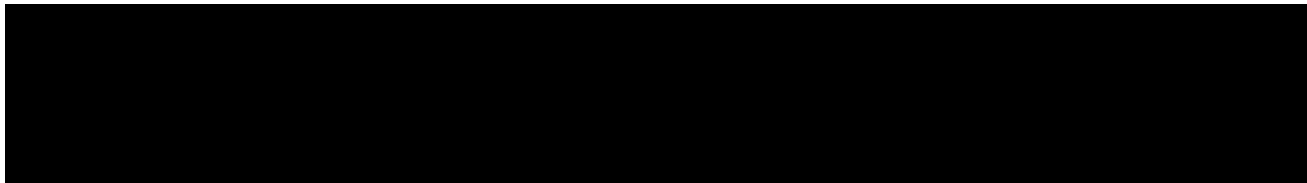
- Khai báo quy mô sử dụng bộ nhớ
 - ❑ .MODEL Kiểu kích thước bộ nhớ
 - ❑ Ví dụ: .Model Small

Kiểu	Mô tả
Tiny (hẹp)	mã lệnh và dữ liệu gói gọn trong một đoạn
Small (nhỏ)	mã lệnh nằm trong 1 đoạn, dữ liệu 1 đoạn
Medium (tB)	mã lệnh nằm trong nhiều đoạn, dữ liệu 1 đoạn
Compact (gọn)	mã lệnh nằm trong 1 đoạn, dữ liệu trong nhiều đoạn
Large (lớn)	mã lệnh nằm trong nhiều đoạn, dữ liệu trong nhiều đoạn, không có mảng nào lớn hơn 64 K
Huge (đồ sộ)	mã lệnh nằm trong nhiều đoạn, dữ liệu trong nhiều đoạn, các mảng có thể lớn hơn 64 K



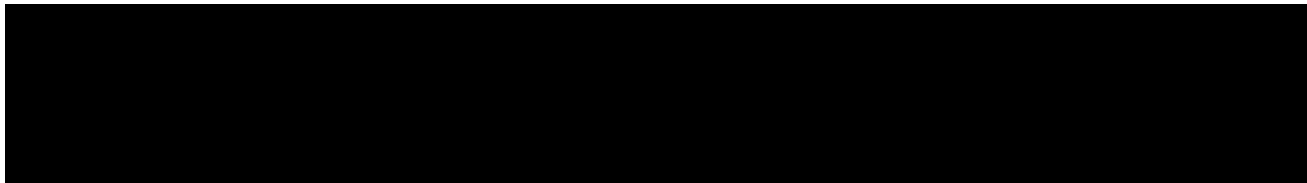
- Khai báo đoạn ngăn xếp
 - ❑ .Stack kích thước (bytes)
 - ❑ Ví dụ:
 - ⇒ .Stack 100 ; khai báo stack có kích thước 100 bytes
 - ❑ Giá trị ngầm định 1KB
- Khai báo đoạn dữ liệu:
 - ❑ .Data
 - ❑ Khai báo các biến và hằng
- Khai báo đoạn mã
 - ❑ .Code





- Khung của chương trình hợp ngữ để dịch ra file .EXE

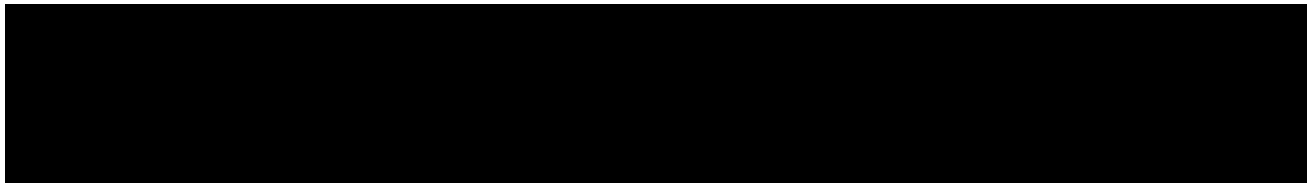
```
.Model Small  
.Stack 100  
.Data  
; các định nghĩa cho biến và hằng  
.Code  
MAIN Proc  
; khởi đầu cho DS  
MOV AX, @data  
MOV DS, AX  
; các lệnh của chương trình  
; trở về DOS dùng hàm 4CH của INT 21H  
MOV AH, 4CH  
INT 21H  
MAIN Endp  
; các chương trình con nếu có  
END MAIN
```



- Chương trình Hello.EXE

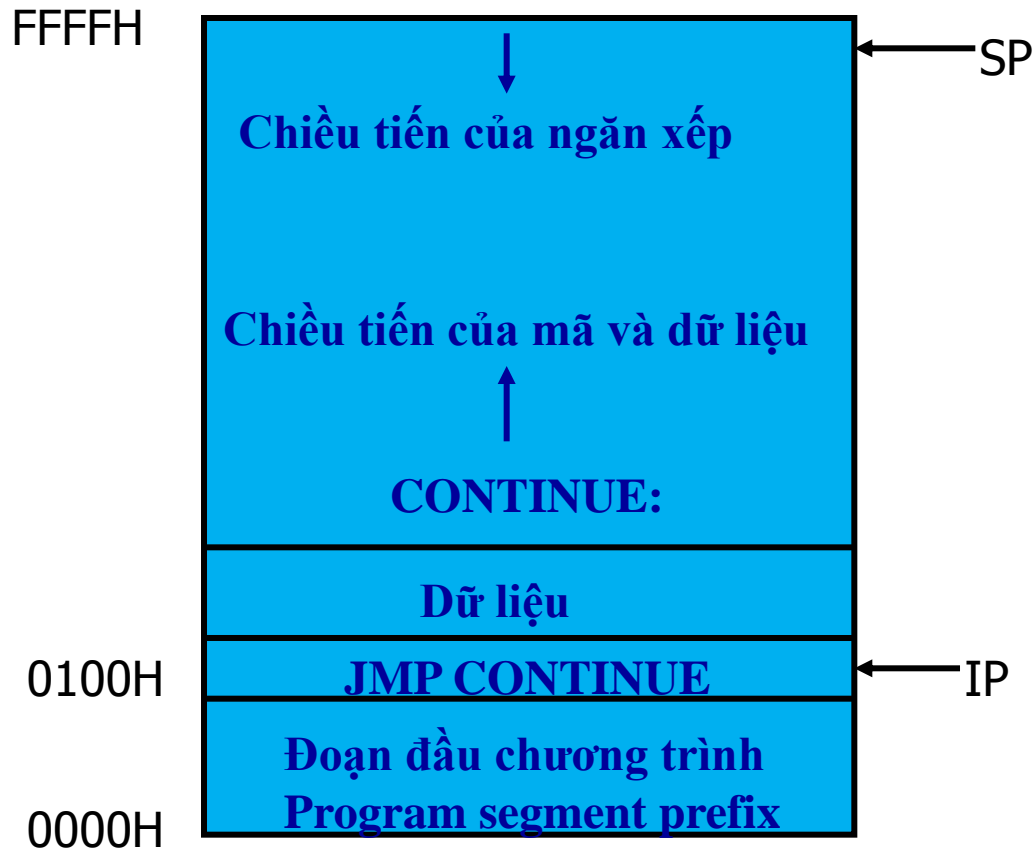
```
.Model      Small
.Stack      100
.Data
            CRLF      DB      13,10,'$'
            MSG       DB      'Hello! $'

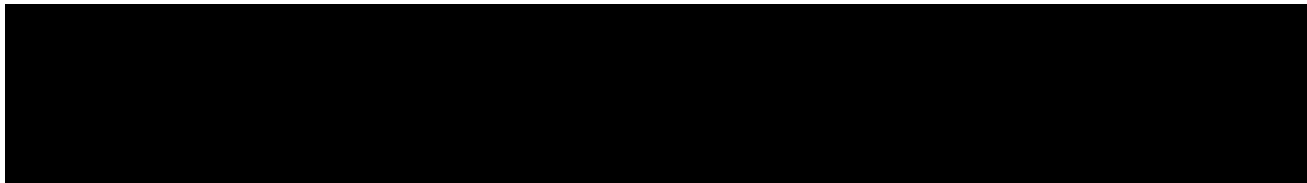
.Code
MAIN Proc
;khởi đầu cho DS
MOV        AX, @data
MOV        DS, AX
;về đầu dòng mới dùng hàm 9 của INT 21H
MOV        AH,9
LEA        DX, CRLF
INT        21H
;Hiển thị lời chào dùng hàm 9 của INT 21H
MOV        AH,9
LEA        DX, MSG
INT        21H
;về đầu dòng mới dùng hàm 9 của INT 21H
MOV        AH,9
LEA        DX, CRLF
INT        21H
;trở về DOS dùng hàm 4CH của INT 21H
MOV        AH, 4CH
INT        21H
MAIN Endp
END MAIN
```



- Khung của chương trình hợp ngữ để dịch ra file .COM

```
.Model Tiny  
.Code  
ORG 100h  
START: JMP CONTINUE  
          ;các định nghĩa cho biến và hằng  
CONTINUE:  
MAIN Proc  
          ;các lệnh của chương trình  
INT 20H ;trở về DOS  
MAIN Endp  
          ;các chương trình con nếu có  
END START
```



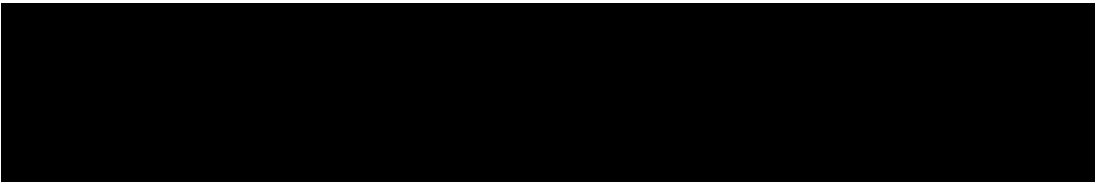


- Chương trình Hello.COM

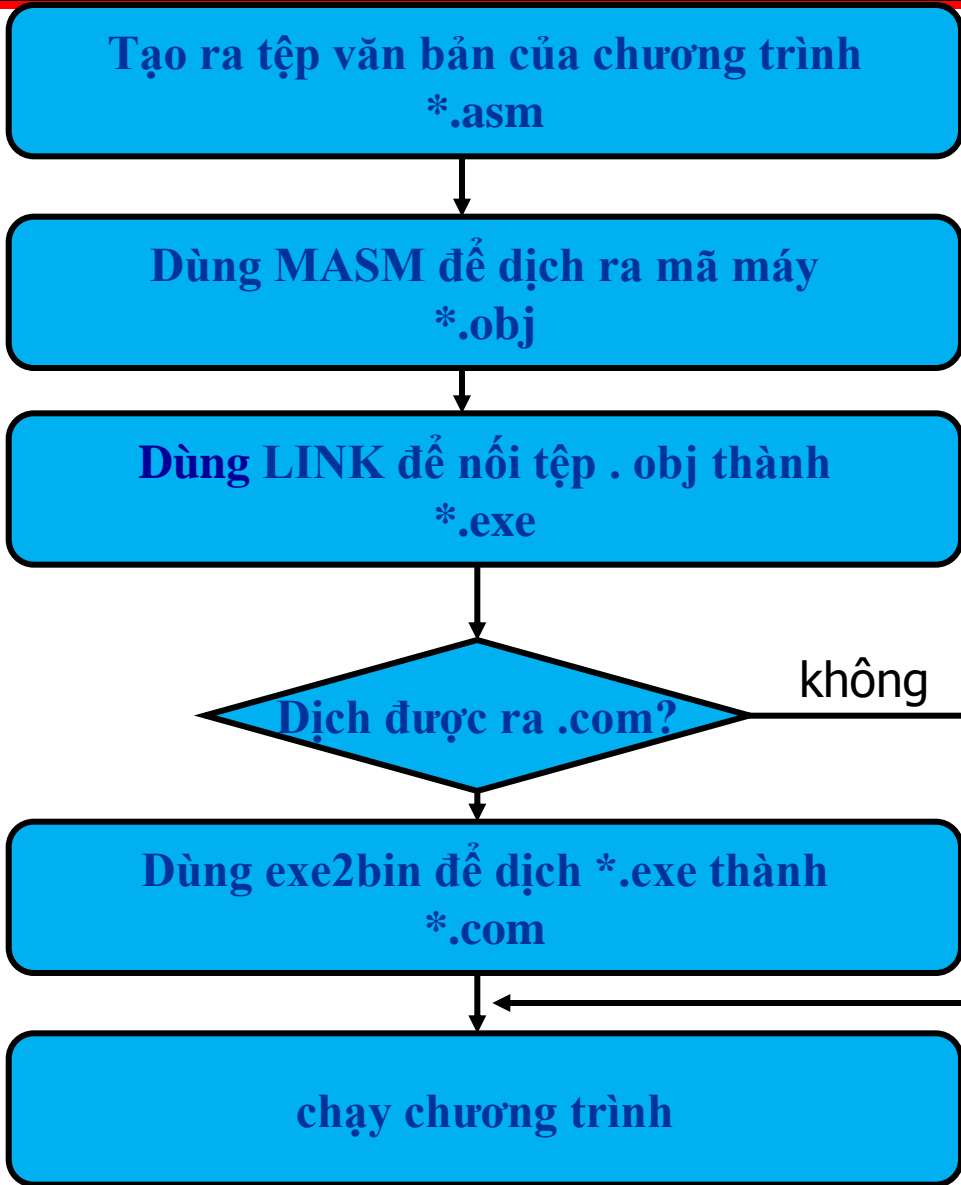
```
.Model    Tiny
.Code

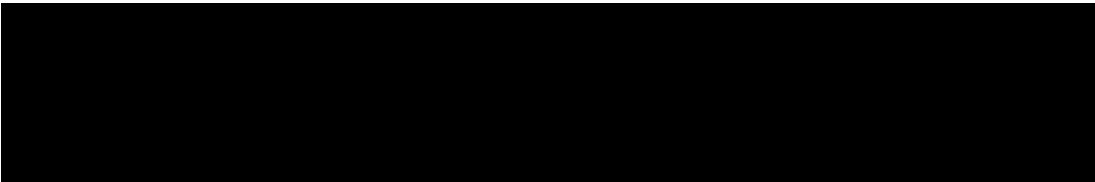
                ORG    100H
START: JMP CONTINUE
                CRLF    DB    13,10,'$'
                MSG     DB    'Hello! $'

CONTINUE:
MAIN    Proc
        ;về đầu dòng mới dùng hàm 9 của INT 21H
        MOV     AH,9
        LEA    DX, CRLF
        INT     21H
        ;Hiển thị lời chào dùng hàm 9 của INT 21H
        MOV     AH,9
        LEA    DX, MSG
        INT     21H
        ;về đầu dòng mới dùng hàm 9 của INT 21H
        MOV     AH,9
        LEA    DX, CRLF
        INT     21H
        ;trở về DOS
        INT     20H
MAIN    Endp
END START
```

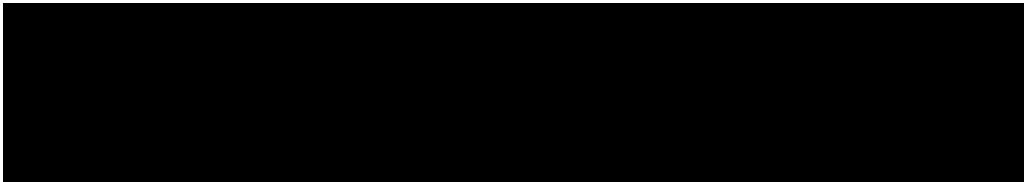



- Giới thiệu khung của chương trình hợp ngữ
- **Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC**
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể





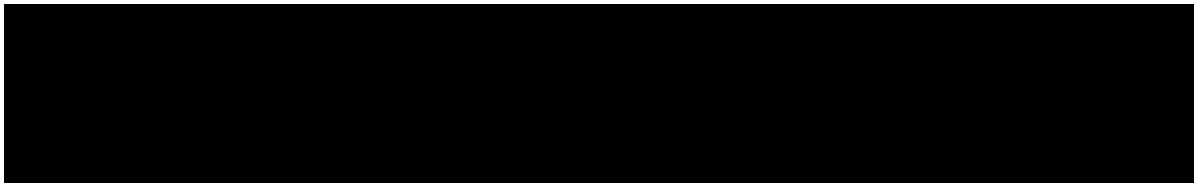
- Giới thiệu khung của chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
 - Cấu trúc lựa chọn
 - Cấu trúc lặp
- Một số chương trình cụ thể



- If (điều_kiện) then (công_việc)
- Ví dụ: Gán cho BX giá trị tuyệt đối của AX

```
; If AX<0
CMP      AX, 0          ; AX<0 ?
JNL End_if             ; không, thoát ra

; then
NEGAX    ; đúng, đảo dấu
End_if: MOV    BX, AX   ;gán
```

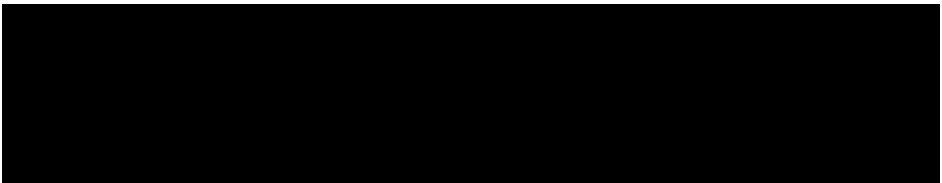


- If (điều_kiện) then (công_việc1)
else (công_việc2)
- Ví dụ: if AX<BX then CX=0 else CX=1

```
; if AX<BX
CMP      AX, BX      ; AX<BX ?
JL       Then_       ; đúng, CX=0

;else
MOV      CX, 1       ; sai, CX=1
JMP      End_if

Then_:   MOV      CX, 0;
End_if:
```



- case Biểu thức

Giá trị 1: công việc 1

Giá trị 2: công việc 2

...

Giá trị N: công việc N

End Case

- Ví dụ:

Nếu $AX < 0$ thì $CX = -1$

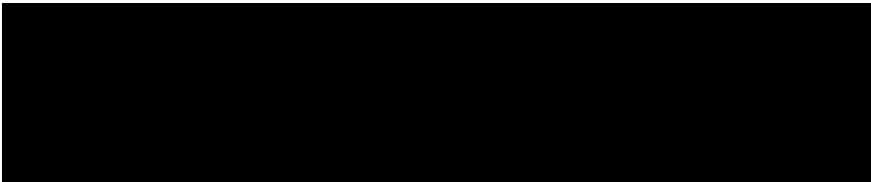
Nếu $AX = 0$ thì $CX = 0$

Nếu $AX > 0$ thì $CX = 1$

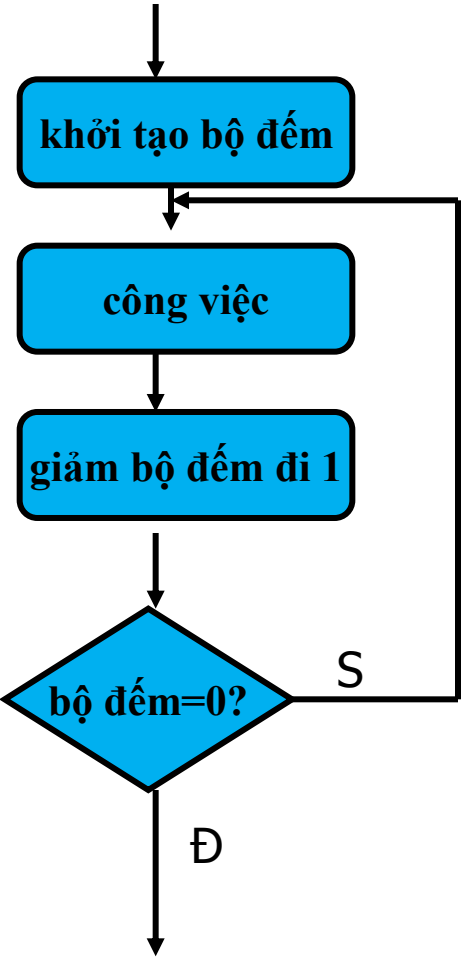
```
CMP      AX, 0      ;
JL       AM        ; AX<0
JE       Khong     ; AX=0
JG       DUONG     ; AX>0
AM:      MOV       CX, -1
        JMP       End_case

Khong:   MOV       CX, 0
        JMP       End_case

DUONG:   MOV       CX, 1
End_case:
```

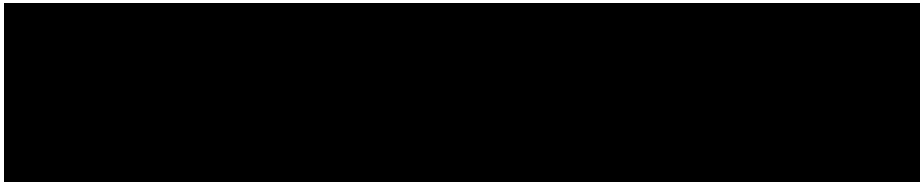


- for (số lần lặp) do (công việc)

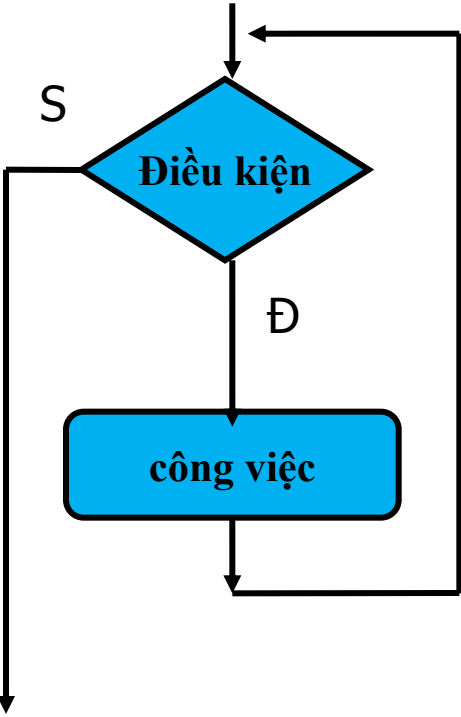


ví dụ: Hiển thị một dòng ký tự \$ trên màn hình

```
MOV CX, 80           ;số lần lặp
MOV AH,2             ;hàm hiển thị
MOV DL,'$'           ;DL chứa ký tự cần hiển thị
HIEN: INT 21H         ; Hiển thị
      LOOP HIEN
End_for
```

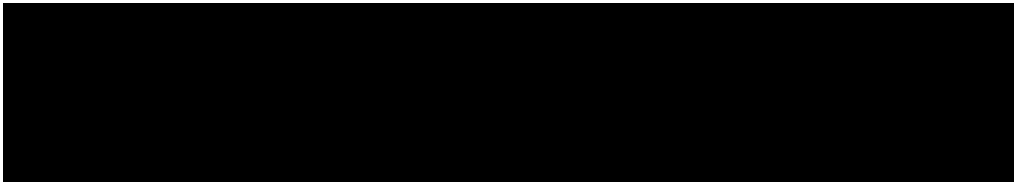


- while (điều kiện) do (công việc)

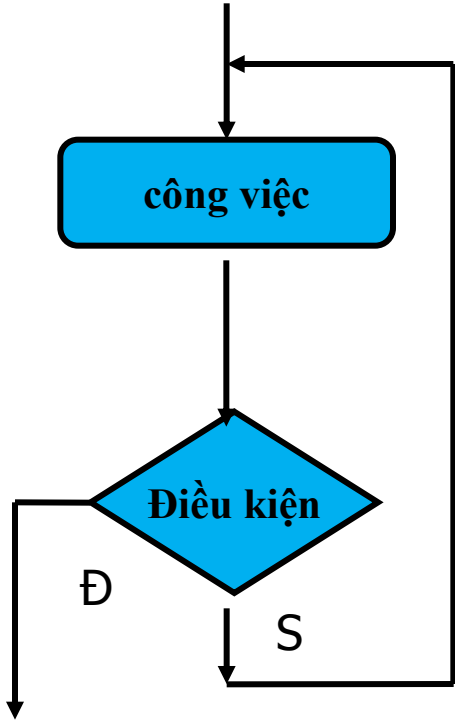


ví dụ: đếm số ký tự đọc được từ bàn phím,
khi gặp ký tự CR thì thôi

```
XOR CX, CX           ;CX=0
MOV AH,1           ;hàm đọc ký tự từ bàn phím
TIEP:
INT 21H            ; đọc một ký tự vào AL
CMP AL, 13         ; đọc CR?
JE End_while      ; đúng, thoát
INC CX             ; sai, thêm 1 ký tự vào tổng
JMP TIEP          ; đọc tiếp
End_while:
```

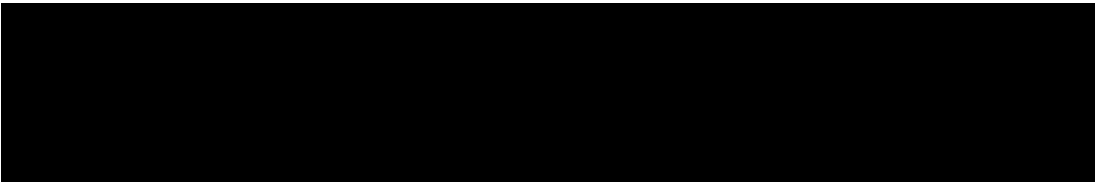



- Repeat (công việc) until (điều kiện)

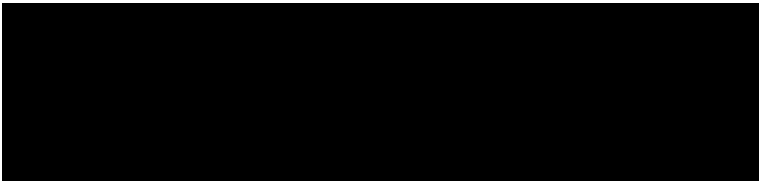


ví dụ: đọc từ bàn phím cho tới khi gặp ký tự CR thì thôi

```
MOV AH,1           ;hàm đọc ký tự từ bàn phím
TIEP:
INT 21H           ; đọc một ký tự vào AL
CMP AL, 13        ; đọc CR?
JNE TIEP          ; chưa, đọc tiếp
End_:
```



- Giới thiệu khung của chương trình hợp ngữ
- Cách tạo và chạy một chương trình hợp ngữ trên máy IBM PC
- Các cấu trúc lập trình cơ bản thực hiện bằng hợp ngữ
- Một số chương trình cụ thể



- 2 cách:

- ❑ Dùng lệnh IN, OUT để trao đổi với các thiết bị ngoại vi

- ⇒ phức tạp vì phải biết địa chỉ cổng ghép nối thiết bị

- ⇒ Các hệ thống khác nhau có địa chỉ khác nhau

- ❑ Dùng các chương trình con phục vụ ngắt của DOS và BIOS

- ⇒ đơn giản, dễ sử dụng

- ⇒ không phụ thuộc vào hệ thống

- Ngắt 21h của DOS:

- ❑ Hàm 1: đọc 1 ký tự từ bàn phím

- ⇒ Vào: AH=1

- ⇒ Ra: AL=mã ASCII của ký tự, AL=0 khi ký tự là phím chức năng

- ❑ Hàm 2: hiện 1 ký tự lên màn hình

- ⇒ Vào: AH=2

- DL=mã ASCII của ký tự cần hiển thị

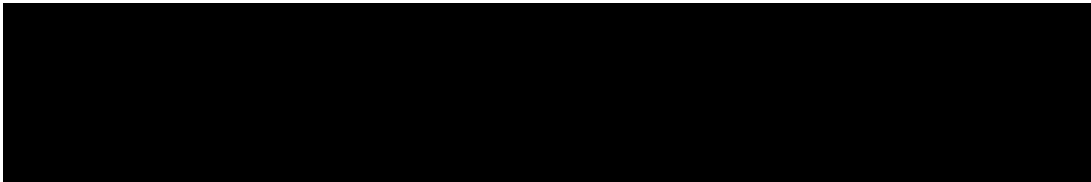
- ❑ Hàm 9: hiện chuỗi ký tự với \$ ở cuối lên màn hình

- ⇒ Vào: AH=9

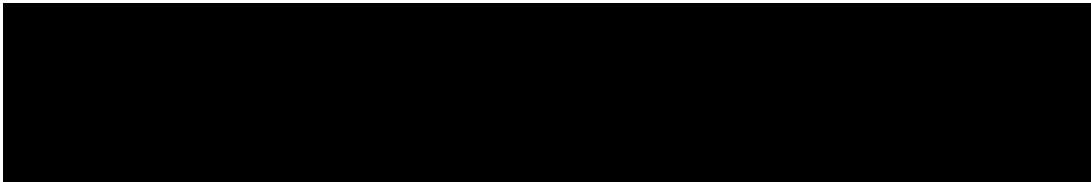
- DX=địa chỉ lệch của chuỗi ký tự cần hiển thị

- ❑ Hàm 4CH: kết thúc chương trình loại .exe

- ⇒ Vào: AH=4CH



- Ví dụ 1: Lập chương trình yêu cầu người sử dụng gõ vào một chữ cái thường và hiển thị dạng chữ hoa và mã ASCII dưới dạng nhị phân của chữ cái đó lên màn hình
 - Ví dụ:
 - ⇒ Hay nhập vào một chữ cái thường: a
 - ⇒ Mã ASCII dưới dạng nhị phân của a là: 11000001
 - ⇒ Dạng chữ hoa của a là: A
- Ví dụ 2: Đọc từ bàn phím một số hệ hai (dài nhất là 16 bit), kết quả đọc được để tại thanh ghi BX. Sau đó hiển thị nội dung thanh ghi BX ra màn hình.
- Ví dụ 3: Nhập một dãy số 8 bit ở dạng thập phân, các số cách nhau bằng 1 dấu cách và kết thúc bằng phím Enter. Sắp xếp dãy số theo thứ tự tăng dần và in dãy số đã sắp xếp ra màn hình.

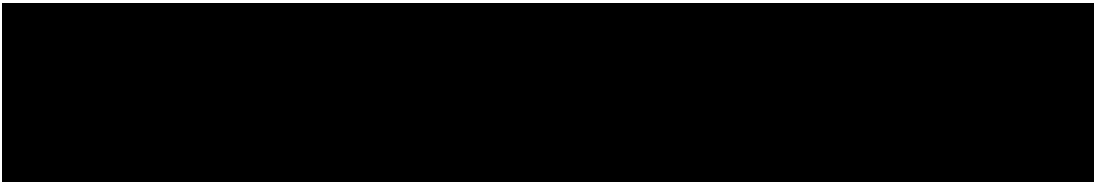


- Ví dụ 4: Viết chương trình cho phép nhập vào kích thước $M*N$ và các phần tử của một mảng 2 chiều gồm các số thập phân 8 bit.
 - ⇒ Tìm số lớn nhất và nhỏ nhất của mảng, in ra màn hình
 - ⇒ Tính tổng các phần tử của mảng và in ra màn hình
 - ⇒ Chuyển thành mảng $N*M$ và in mảng mới ra màn hình

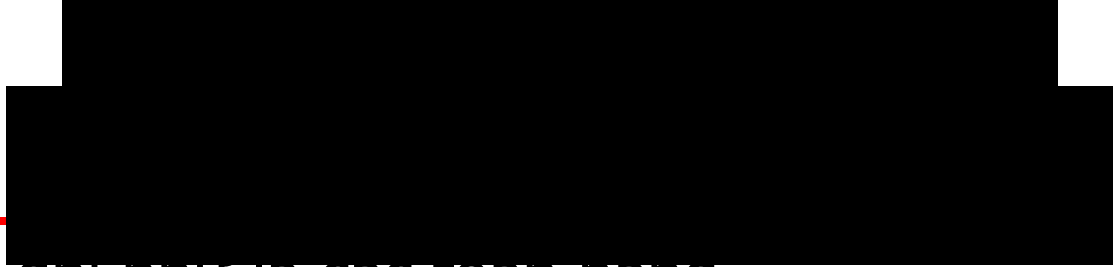
Hãy nhập giá trị M=
Hãy nhập giá trị N=
Nhập phần tử [1,1]=
Nhập phần tử [1,2]
.....
Số lớn nhất là phần tử [3,4]=15
Số nhỏ nhất là phần tử [1,2]=2
Tổng=256
...

PHỤ LỤC





- Các chế độ địa chỉ của 8086
 - Chế độ địa chỉ thanh ghi
 - Chế độ địa chỉ tức thì
 - Chế độ địa chỉ trực tiếp
 - Chế độ địa chỉ gián tiếp qua thanh ghi
 - Chế độ địa chỉ tương đối cơ sở
 - Chế độ địa chỉ tương đối chỉ số
 - Chế độ địa chỉ tương đối chỉ số cơ sở
- Cách mã hoá lệnh của 8086

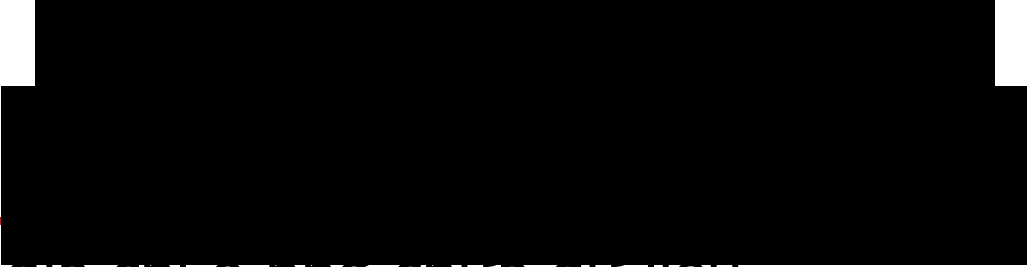


- Dùng các thanh ghi như là các toán hạng
- Tốc độ thực hiện lệnh cao

- Ví dụ:
 - MOV BX, DX ; Copy nội dung DX vào BX
 - MOV AL, BL ; Copy nội dung BL vào AL
 - MOV AL, BX ; không hợp lệ vì các thanh ghi có kích thước khác nhau
 - MOV ES, DS ; không hợp lệ (segment to segment)
 - MOV CS, AX ; không hợp lệ vì CS không được dùng làm thanh ghi đích

 - ADD AL, DL ; Cộng nội dung AL và DL rồi đưa vào AL





- Một toán hạng là địa chỉ ô nhớ chứa dữ liệu
- Toán hạng kia chỉ có thể là thanh ghi

- Ví dụ:
 - ❑ `MOV AL, [1234H]` ; Copy nội dung ô nhớ có địa chỉ DS:1234 vào AL
 - ❑ `MOV [4320H], CX` ; Copy nội dung của CX vào 2 ô nhớ liên tiếp DS: 4320 và DS: 4321

- Một toán hạng là thanh ghi chứa địa chỉ của 1 ô nhớ dữ liệu
- Toán hạng kia chỉ có thể là thanh ghi

- Ví dụ:
 - ❑ `MOV AL, [BX]` ; Copy nội dung ô nhớ có địa chỉ DS:BX vào AL
 - ❑ `MOV [SI], CL` ; Copy nội dung của CL vào ô nhớ có địa chỉ DS:SI
 - ❑ `MOV [DI], AX` ; copy nội dung của AX vào 2 ô nhớ liên tiếp DS: DI và DS: (DI +1)



- Một toán hạng là thanh ghi chỉ số SI, DI và các hằng số biểu diễn giá trị dịch chuyển
- Toán hạng kia chỉ có thể là thanh ghi

- Ví dụ:
 - ❑ `MOV AX, [SI]+10` ; Copy nội dung 2 ô nhớ liên tiếp có địa chỉ DS:SI+10 và DS:SI+11 vào AX
 - ❑ `MOV AX, [SI+10]` ; Cách viết khác của lệnh trên
 - ❑ `MOV AL, [DI]+5` ; copy nội dung của ô nhớ DS:DI+5 vào thanh ghi AL



- Ví dụ:

- ❑ `MOV AX, [BX] [SI]+8` ; Copy nội dung 2 ô nhớ liên tiếp có địa chỉ `DS:BX+SI+8` và `DS:BX+SI+9` vào `AX`
- ❑ `MOV AX, [BX+SI+8]` ; Cách viết khác của lệnh trên
- ❑ `MOV CL, [BP+DI+5]` ; copy nội dung của ô nhớ `SS:BP+DI+5` vào thanh ghi `CL`

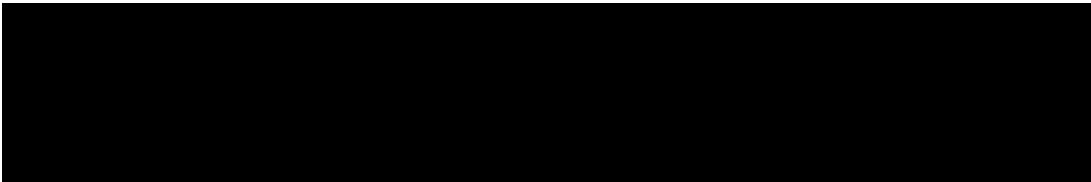
Chế độ địa chỉ	Toán hạng	Thanh ghi đoạn ngầm định
Thanh ghi	Thanh ghi	
Tức thì	Dữ liệu	
Trực tiếp	[offset]	DS
Gián tiếp qua thanh ghi	[BX]	DS
	[SI]	DS
	[DI]	DS
Tương đối cơ sở	[BX] + dịch chuyển	DS
	[BP] + dịch chuyển	SS
Tương đối chỉ số	[DI] + dịch chuyển	DS
	[SI] + dịch chuyển	DS
Tương đối chỉ số cơ sở	[BX] + [DI] + dịch chuyển	DS
	[BX] + [SI] + dịch chuyển	DS
	[BP] + [DI] + dịch chuyển	SS
	[BP] + [SI] + dịch chuyển	SS



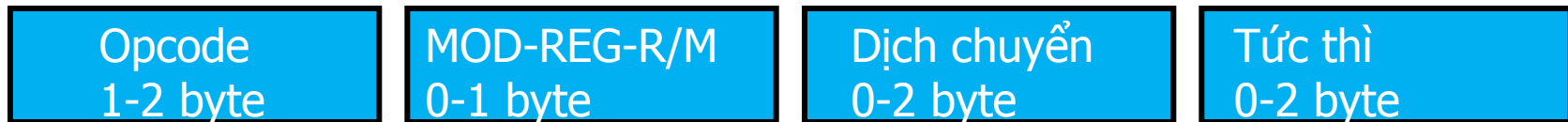
- Ví dụ:

- MOV AL, [BX]; Copy nội dung ô nhớ có địa chỉ DS:BX vào AL

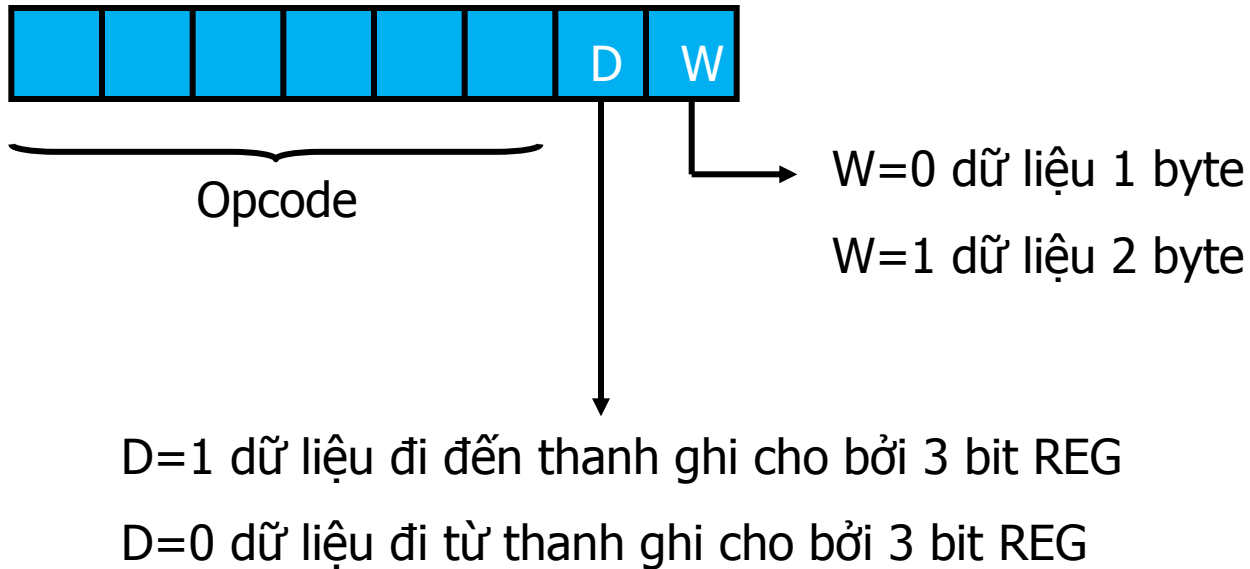
- MOV AL, ES:[BX] ; Copy nội dung ô nhớ có địa chỉ ES:BX vào AL

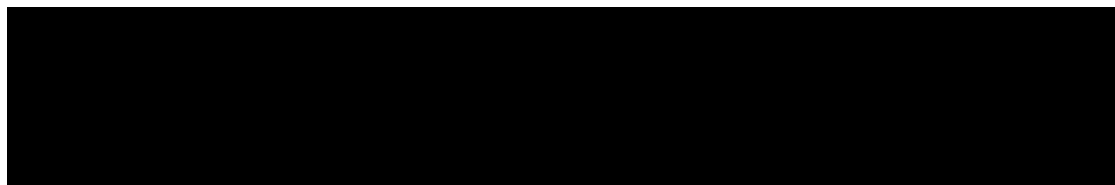


- Các chế độ địa chỉ của 8086
- Cách mã hoá lệnh của 8086



- Một lệnh có độ dài từ 1 đến 6 byte





MOD

REG

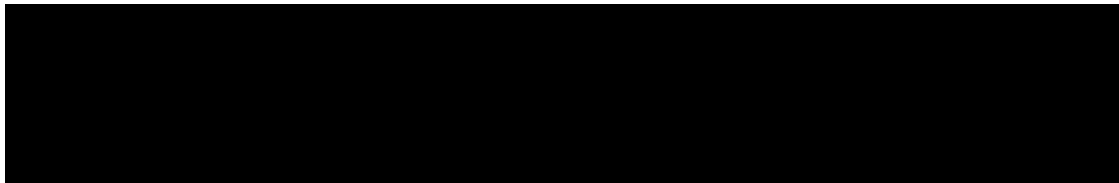
R/M

MOD <> 11

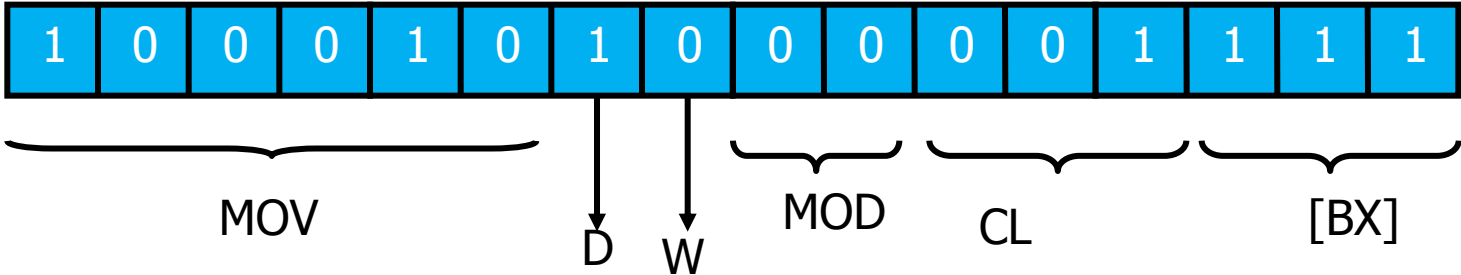
00	không có dịch chuyển
01	dịch chuyển 8 bit
10	dịch chuyển 16 bit
11	R/M là thanh ghi

Thanh ghi		Mã
W=1	W=0	
AX	AL	000
BX	BL	011
CX	CL	001
DX	DL	010
SP	AH	100
DI	BH	111
BP	CH	101
SI	DH	110

Mã	Chế độ địa chỉ
000	DS:[BX+SI]
001	DS:[BX+DI]
010	SS:[BP+SI]
011	SS:[BP+DI]
100	DS:[SI]
101	DS:[DI]
110	SS:[BP]
111	DS:[BX]



- Ví dụ: chuyển lệnh MOV CL, [BX] sang mã máy
 - ❑ opcode MOV: 100010
 - ❑ Dữ liệu là 1 byte: W=0
 - ❑ Chuyển tới thanh ghi: D=1
 - ❑ Không có dịch chuyển: MOD=00
 - ❑ [BX] nên R/M=111
 - ❑ CL nên REG=001



Ví dụ 2: chuyển lệnh MOV [SI+F3H], CL sang mã máy

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Võ Minh Trí – vmtri@fit.hcmus.edu.vn

06 – Mạch Logic

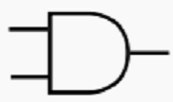
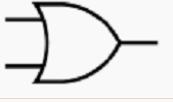

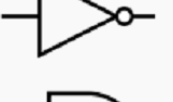



Mạch số

2

- Là thiết bị điện tử hoạt động với **2 mức điện áp**:
 - **Cao**: thể hiện bằng giá trị luận lý (quy ước) là **1**
 - **Thấp**: thể hiện bằng giá trị luận lý (quy ước) là **0**
- Được xây dựng từ những thành phần cơ bản là **cổng luận lý (logic gate)**
 - Cổng luận lý là thiết bị điện tử gồm 1 / nhiều tín hiệu đầu vào (input) - 1 tín hiệu đầu ra (output)
 - $output = F(input_1, input_2, \dots, input_n)$
 - Tùy thuộc vào cách xử lý của hàm F sẽ tạo ra nhiều loại cổng luận lý
- Hiện nay linh kiện cơ bản để tạo ra mạch số là **transistor**

Cổng luận lý (Logic gate)

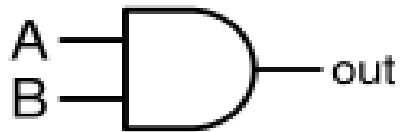
3

Tên cổng	Hình vẽ đại diện	Hàm đại số Bun
AND		$x.y$ hay xy
OR		$x + y$
XOR		$x \oplus y$
NOT		x' hay \bar{x}
NAND		$(x . y)'$ hay $\overline{x.y}$
NOR		$(x + y)'$ hay $\overline{x + y}$
NXOR		$(x \oplus y)'$ hay $\overline{x \oplus y}$

Bảng chân trị

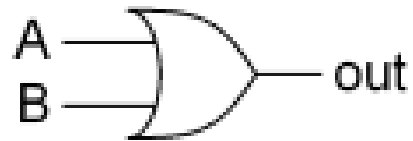
4

AND



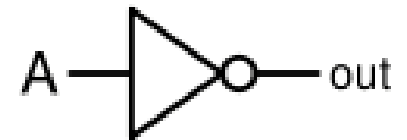
A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

OR



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

NOT



A	out
0	1
1	0

Bảng chân trị

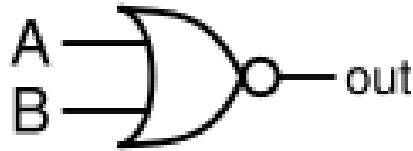
5

NAND



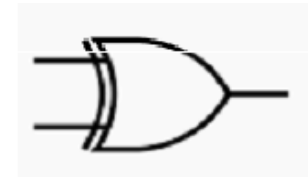
A	B	out
0	0	1
0	1	1
1	0	1
1	1	0

NOR



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

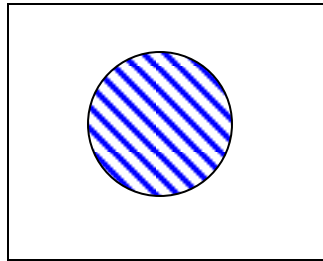
XOR



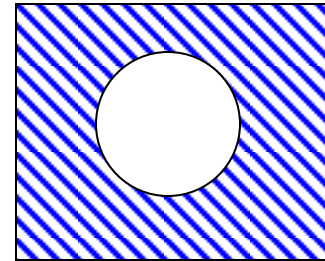
A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

Lược đồ Venn

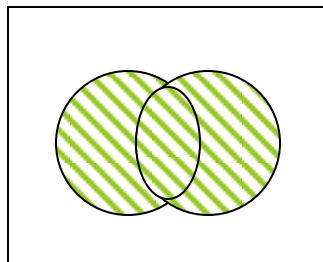
6



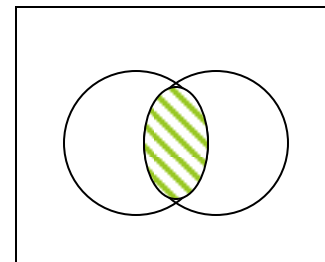
A



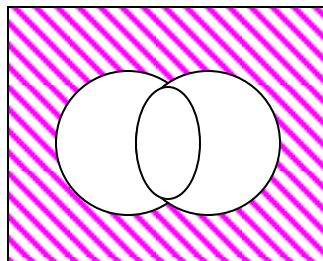
\overline{A}



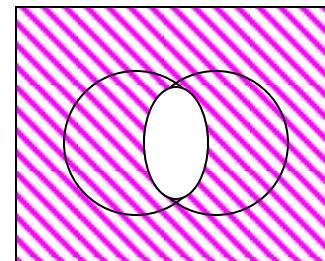
$A+B$



$A.B$



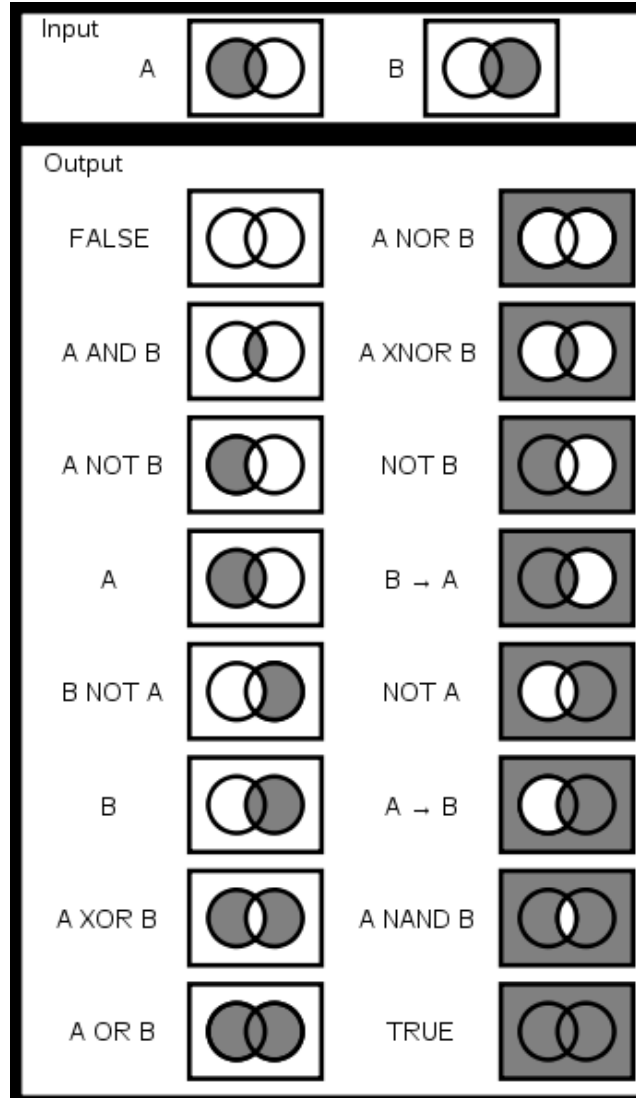
$\overline{A.B}$



$\overline{A+B}$

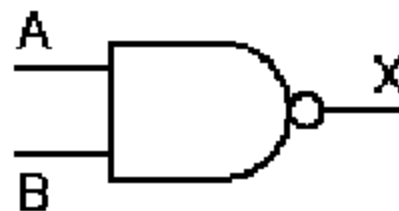
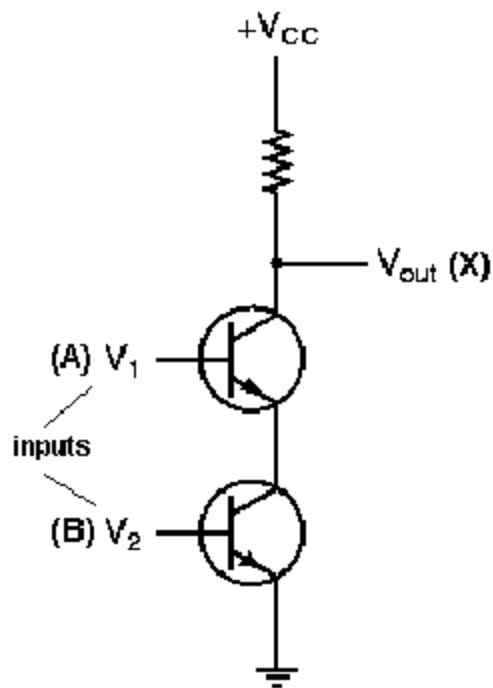
Lược đồ Venn

7



Ví dụ công luận lý

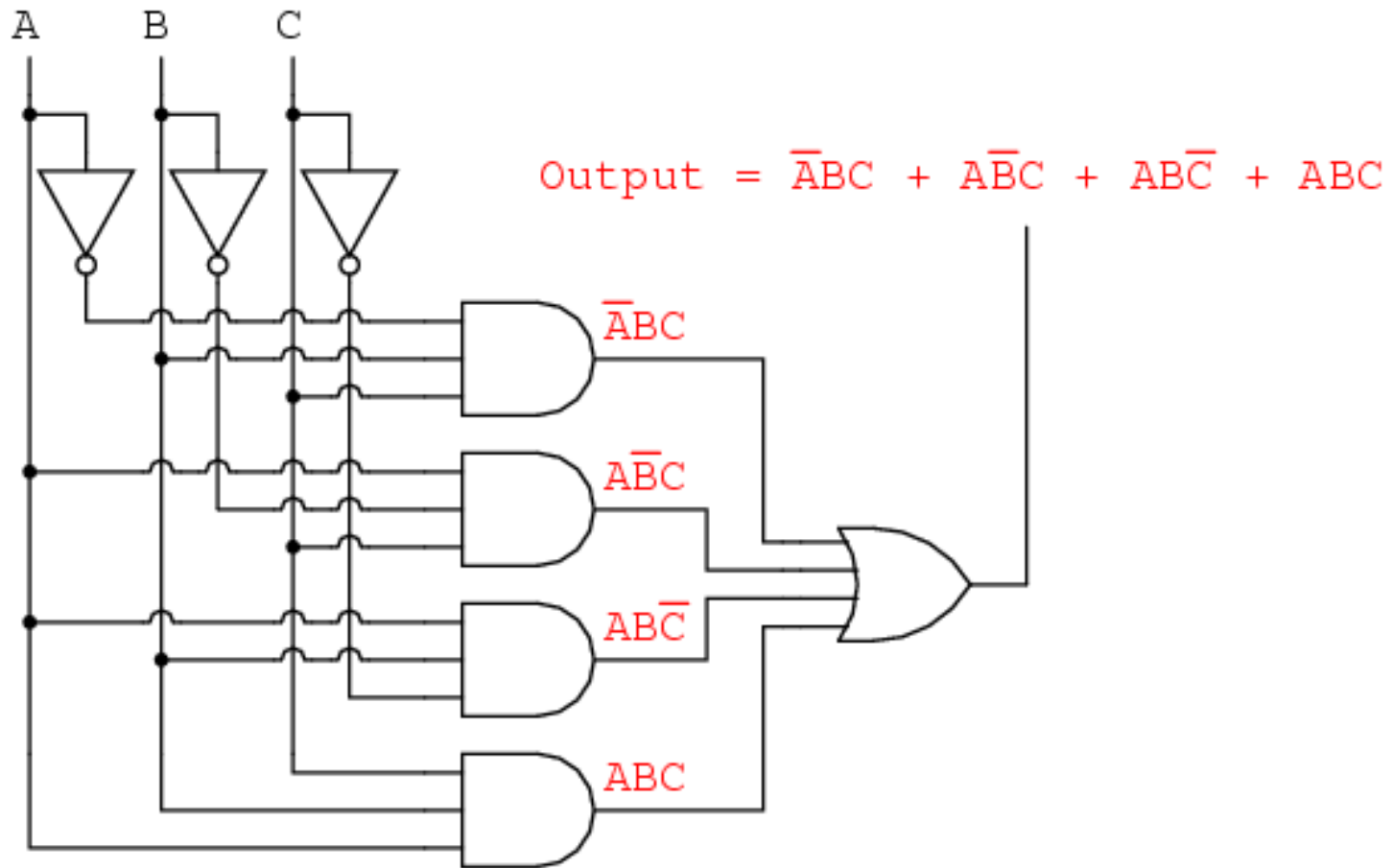
8



A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Ví dụ mạch số

9



Một số đẳng thức cơ bản

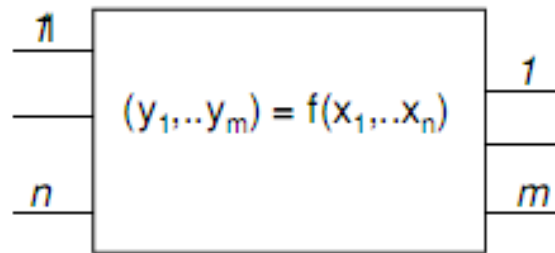
10

$x + 0 = x$	$x \cdot 0 = 0$
$x + 1 = 1$	$x \cdot 1 = x$
$x + x = x$	$x \cdot x = x$
$x + x' = 1$	$x \cdot x' = 0$
$x + y = y + x$	$xy = yx$
$x + (y + z) = (x + y) + z$	$x(yz) = (xy)z$
$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
$(x + y)' = x'.y'$ (De Morgan)	$(xy)' = x' + y'$ (De Morgan)
$(x')' = x$	

Mạch tổ hợp (tích hợp)

11

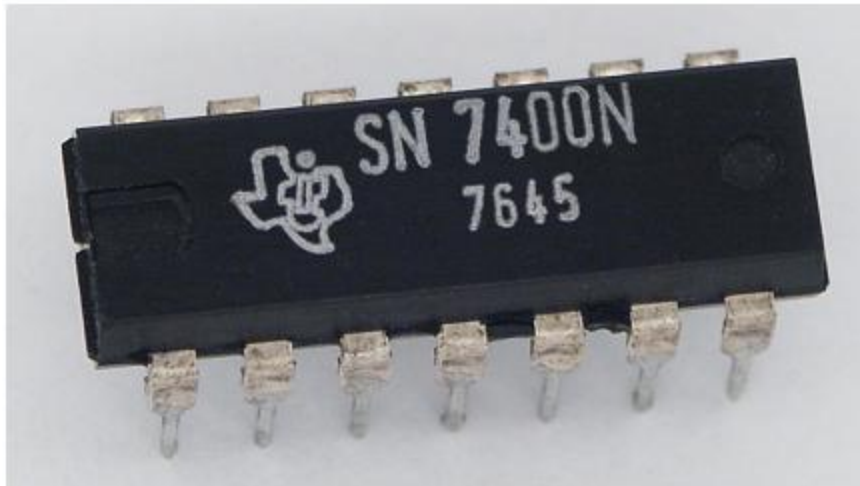
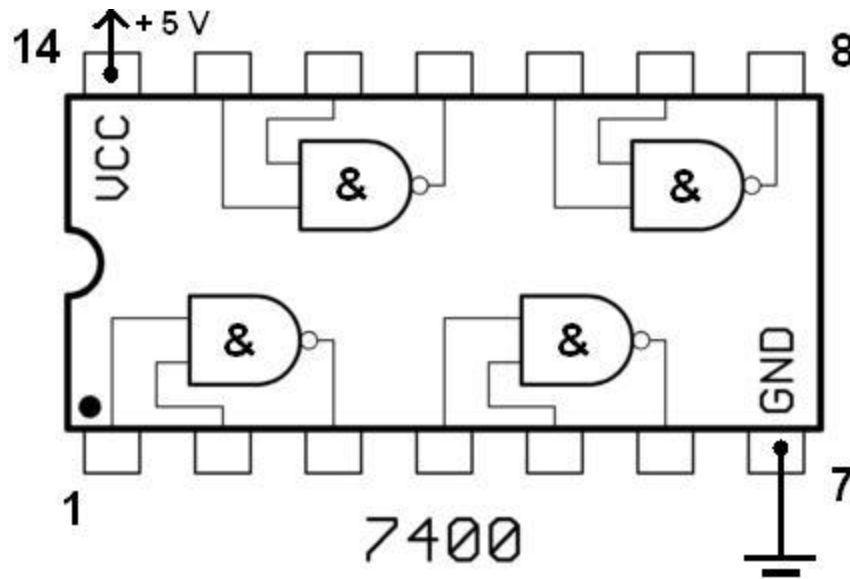
- Gồm n ngõ vào (input); m ngõ ra (output)
 - ▣ Mỗi ngõ ra là 1 hàm luận lý của các ngõ vào



- Mạch tổ hợp không mang tính ghi nhớ: Ngõ ra chỉ phụ thuộc vào Ngõ vào hiện tại, không xét những giá trị trong quá khứ

Ví dụ mạch tổ hợp

12



- The 7400 chip, containing four NAND gate
- The two additional pins supply power (+5 V) and connect the ground.

Mô tả mạch tổ hợp

14

- Bảng ngôn ngữ
- Bảng bảng chân trị
 - n input – m output
 - 2^n hàng – $(n + m)$ cột
- Bảng công thức (hàm luận lý)
- Bảng sơ đồ

Thiết kế

15

□ Thường trải qua 3 bước:

□ Lập bảng chân trị

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

□ Viết hàm luận lý

$$F = (AB)'$$

□ Vẽ sơ đồ mạch và thử nghiệm



SOP – Sum of Products

16

- Giả sử đã có bảng chân trị cho mạch n đầu vào x_1, \dots, x_n và 1 đầu ra f
- Ta dễ dàng thiết lập công thức (hàm) logic theo thuật toán sau:
 - ▣ Ứng với mỗi hàng của bảng chân trị có đầu ra = 1 ta tạo thành 1 tích có dạng $u_1 \cdot u_2 \dots u_n$ với:

$$u_i = \begin{cases} x_i & \text{nếu } x_i = 1 \\ (x_i)' & \text{nếu } x_i = 0 \end{cases}$$

- ▣ Cộng các tích tìm được lại thành tổng \rightarrow công thức của f

Ví dụ

17

x_1	x_2	x_3	f	
0	0	0	0	
0	0	1	1	$\rightarrow \bar{x}_1 \cdot \bar{x}_2 \cdot x_3$
0	1	0	1	$\rightarrow \bar{x}_1 \cdot x_2 \cdot \bar{x}_3$
0	1	1	0	
1	0	0	0	
1	0	1	1	$\rightarrow x_1 \cdot \bar{x}_2 \cdot x_3$
1	1	0	0	
1	1	1	0	

$$f = \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3$$

POS – Product of Sum

18

- Trường hợp số hàng có giá trị **đầu ra = 1**
nhều hơn = 0, ta có thể đặt $g = (f)'$
- Viết công thức dạng **SOP** cho g
- Lấy $f = (g)' = (f'')'$ để có công thức dạng POS
(Tích các tổng) của f

Ví dụ

19

x	y	z	f	g
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

$$g = \bar{x}.y.\bar{z} + x.\bar{y}.\bar{z}$$

$$f = \bar{g} = (x + \bar{y} + z)(\bar{x} + y + z)$$

Đơn giản hoá hàm logic

20

- Sau khi viết được hàm logic, ta có thể vẽ sơ đồ của mạch tổ hợp từ những công luận lý cơ bản
 - Ví dụ: $f = xy + xz$
- Tuy nhiên ta có thể viết lại hàm logic sao cho sơ đồ mạch sử dụng **ít cổng hơn**
 - Ví dụ: $f = xy + xz = x(y + z)$
- Cách đơn giản hoá hàm tổng quát? Một số cách phổ biến:
 - Dùng đại số Bun (Xem lại bảng 1 số đẳng thức cơ bản để áp dụng)
 - Dùng bản đồ Karnaugh (Cac-nô)

Đại số Bun

21

- Dùng các phép biến đổi đại số Bun để lược giản hàm logic
- Khuyết điểm:
 - Không có cách làm tổng quát cho mọi bài toán
 - Không chắc kết quả cuối cùng đã tối giản chưa
- Ví dụ: Đơn giản hoá các hàm sau
 - $F(x,y,z) = xyz + x'yz + xy'z + xyz'$

Bản đồ Karnaugh

22

- Mỗi tổ hợp biến trong bảng chân trị gọi là bộ trị (tạm hiểu là 1 dòng)
- Biểu diễn hàm có n biến thì sẽ cho ra tương ứng 2^n bộ trị, với vị trí các bộ trị được đánh số từ 0
- Thông tin trong bảng chân trị có thể **cô đọng** bằng cách:
 - Liệt kê vị trí các bộ trị (**minterm**) với giá trị đầu ra = **1** (**SOP**)
 - Liệt kê vị trí các bộ trị (**maxterm**) với giá trị đầu ra = **0** (**POS**)

Ví dụ

23

- $F(x,y,z) = m_1 + m_4 + m_5 + m_6 + m_7 = \Sigma(1,4,5,6,7)$
- $F(x,y,z) = M_0M_2M_3 = \Pi(0,2,3)$

Vị trí	x	y	z	minterm	maxterm	F
0	0	0	0	$m_0 = x'y'z'$	$M_0 = x + y + z$	0
1	0	0	1	$m_1 = x'y'z$	$M_1 = x + y + z'$	1
2	0	1	0	$m_2 = x'yz'$	$M_2 = x + y' + z$	0
3	0	1	1	$m_3 = x'yz$	$M_3 = x + y' + z'$	0
4	1	0	0	$m_4 = xy'z'$	$M_4 = x' + y + z$	1
5	1	0	1	$m_5 = xy'z$	$M_5 = x' + y + z'$	1
6	1	1	0	$m_6 = xyz'$	$M_6 = x' + y' + z$	1
7	1	1	1	$m_7 = xyz$	$M_7 = x' + y' + z'$	1

Các dạng bản đồ Karnaugh cơ bản

24

B

	B	0	1
A	0	0	1
A	1	2	3

B

	BC	00	01	11	10
A	0	0	1	3	2
A	1	4	5	7	6

C

B

	CD	00	01	11	10
AB	00	0	1	3	2
	01	4	5	7	6
A	11	12	13	15	14
	10	8	9	11	10

C

D

Ví dụ

25

□ $F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$

		BC			
		00	01	B	
A	0	0	1	0	0
	1	1	1	1	1

Diagram 1: Karnaugh map for $F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$. The map shows the function's output for all combinations of A, B, and C. The output is 1 for minterms 1, 4, 5, 6, and 7, and 0 for minterms 0 and 3. The map is annotated with a blue bracket labeled 'B' over the columns 11 and 10, and a yellow bracket labeled 'C' under the columns 01, 11, and 10. A red bracket labeled 'A' is on the left side of the rows 0 and 1.

==

		BC			
		00	01	B	
A	0		1		
	1	1	1	1	1

Diagram 2: Karnaugh map for $F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$. The map shows the function's output for all combinations of A, B, and C. The output is 1 for minterms 1, 4, 5, 6, and 7, and 0 for minterms 0 and 3. The map is annotated with a blue bracket labeled 'B' over the columns 11 and 10, and a yellow bracket labeled 'C' under the columns 01, 11, and 10. A red bracket labeled 'A' is on the left side of the rows 0 and 1.

Nhận xét

26

- Bộ trị giữa **2 ô liền kề** trong bản đồ chỉ **khác nhau 1 biến**
 - Biến đó bù 1 ô, không bù ở ô kế hoặc ngược lại
- Các ô đầu / cuối của các dòng / cột là các ô liền kề
- 4 ô nằm ở 4 góc bản đồ cũng coi là ô liền kề

Đơn giản hàm theo dạng SOP

27

- Hàm logic F biểu diễn bảng chân trị được đưa vào bản đồ bằng các trị 1 tương ứng
- Các ô liền kề có giá trị 1 được gom thành nhóm sao cho mỗi nhóm sau khi gom có tổng số ô là lũy thừa của 2 (2, 4, 8,...)
- Các nhóm có thể dùng chung ô có giá trị 1 để tạo thành nhóm lớn hơn. Cố gắng tạo những nhóm lớn nhất có thể
- Nhóm 2/4/8 ô sẽ đơn giản bớt 1/2/3 biến trong số hạng
- Mỗi nhóm biểu diễn 1 số hạng nhân (Product), Cộng (Sum – OR)
các số hạng này ta sẽ được biểu thức tối giản của hàm logic F

Ví dụ 1

28

□ $F(A, B, C) = \Sigma(3, 4, 6, 7)$

		BC			
		00	01	11	10
A	0			1	
	1	1		1	1

Diagram illustrating the truth table for the function $F(A, B, C) = \Sigma(3, 4, 6, 7)$. The variables are A, B, and C. The output is 1 for the minterms (3, 4, 6, 7). The diagram shows the truth table with the output values 1 highlighted in red. Brackets indicate the groups for variables B and C.

		BC			
		00	01	11	10
A	0			1	
	1	1		1	1

Diagram illustrating the truth table for the function $F(A, B, C) = \Sigma(3, 4, 6, 7)$. The variables are A, B, and C. The output is 1 for the minterms (3, 4, 6, 7). The diagram shows the truth table with the output values 1 highlighted in green and yellow. Brackets indicate the groups for variables B and C.

$$F(A, B, C) = BC + AC'$$

Ví dụ 2

29

□ $F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$

		BC			
		00	01	11	10
A	0	1			1
	1	1	1		1

Diagram illustrating the truth table for the function $F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$. The variables are A, B, and C. The output is 1 for the minterms (0, 2, 4, 5, 6). The truth table is shown with the output values highlighted in red. Brackets indicate the grouping of variables: B for the columns (11, 10) and C for the rows (01, 11).

		BC			
		00	01	11	10
A	0	1			1
	1	1	1		1

Diagram illustrating the truth table for the function $F(A, B, C) = \Sigma(0, 2, 4, 5, 6)$. The variables are A, B, and C. The output is 1 for the minterms (0, 2, 4, 5, 6). The truth table is shown with the output values highlighted in red. Brackets indicate the grouping of variables: B for the columns (11, 10) and C for the rows (01, 11). The prime implicants are highlighted: a yellow box covers the cells (A=0, B=0) and (A=1, B=0), representing the term AB' ; a green box covers the cells (A=1, B=0) and (A=1, B=1), representing the term C' .

$$F(A, B, C) = C' + AB'$$

Ví dụ

30

□ $F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$

		B			
		00	01	11	10
A	00	1	1		1
	01				1
	11				
	10	1	1		1

Diagram illustrating a 4x4 Karnaugh map for the function $F(A, B, C, D) = \Sigma(0, 1, 2, 6, 8, 9, 10)$. The map is labeled with variables A, B, C, and D. The rows are labeled AB (00, 01, 11, 10) and the columns are labeled CD (00, 01, 11, 10). The 1s are located at the following cells: (00, 00), (00, 01), (00, 10), (01, 10), (10, 00), (10, 01), and (10, 10). Brackets indicate groupings: a horizontal bracket labeled 'D' under the first two columns (CD=00, 01), a vertical bracket labeled 'C' on the right side (CD=10), and a vertical bracket labeled 'A' on the left side (AB=10).

		B			
		00	01	11	10
A	00	1	1		1
	01				1
	11				
	10	1	1		1

Diagram illustrating the same 4x4 Karnaugh map as above, but with the 1s grouped into three prime implicants: a yellow box around the 1s at (00, 00), (00, 01), (10, 00), and (10, 01); a green box around the 1s at (00, 10) and (01, 10); and an orange box around the 1s at (10, 00) and (10, 10). The labels A, B, C, and D are also present.

$$F(A, B, C) = B'D' + B'C' + A'CD'$$

Đơn giản hàm theo dạng POS

31

- Đôi khi biểu diễn dạng tổng các tích (SOP) sẽ khó làm khi **số bộ trị có đầu ra = 1 < số bộ trị có đầu ra = 0**
 - Dùng phương pháp tích các tổng (POS)
- Hoàn toàn giống phương pháp đơn giản hàm theo dạng SOP, chỉ khác ta **nhóm các ô liền kề = 0** thay vì 1
 - **Tìm được F'**
 - **$F = (F')'$**

Ví dụ 3

32

□ $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$

		B			
		00	01	11	10
A	00	1	1	0	1
	01	0	1	0	0
	11	0	0	0	0
	10	1	1	0	1

Labels: A (rows), B (columns), C (rows 01, 11), D (columns 00, 01)

		B			
		00	01	11	10
A	00	1	1	0	1
	01	0	1	0	0
	11	0	0	0	0
	10	1	1	0	1

Labels: A (rows), B (columns), C (rows 01, 11), D (columns 00, 01)

Groupings: Yellow vertical bar at column B=11, orange vertical bars at columns B=00 and B=10, green horizontal bar at row A=11.

$$F'(A, B, C) = AB + CD + BD'$$

$$F = (F')' = (A' + B')(C' + D')(B' + D)$$

Điều kiện không cần / tùy chọn

33

- Trong 1 số trường hợp ta **không cần quan tâm** đến giá trị ngõ ra của 1 số bộ trị nào đó (**1 hay 0 đều được**)
- Trong bản đồ ta sẽ ghi tương ứng những ô đó là x (gọi là giá trị tùy chọn /không cần)
- **x có thể dùng để gom nhóm với các ô liền kề nhằm đơn giản hàm**
- Lưu ý: Không được gom nhóm bao gồm toàn những ô có giá trị x

Ví dụ

34

- $F(A, B, C) = \Sigma(0, 2, 6)$
- $d(A, B, C) = \Sigma(1, 3, 5)$

Vị trí	A	B	C	F
0	0	0	0	1
1	0	0	1	x
2	0	1	0	1
3	0	1	1	x
4	1	0	0	0
5	1	0	1	x
6	1	1	0	1
7	1	1	1	0

Ví dụ

35

- $F(A, B, C) = \Sigma(0, 2, 6)$
- $d(A, B, C) = \Sigma(1, 3, 5)$

		BC			
		00	01	11	10
A	0	1	x	x	1
	1		x		1

Diagram illustrating the truth table for the function $F(A, B, C)$. The variables are A, B, and C. The table shows the output values (1, x, 0) for each combination of inputs. The output is 1 for (A, B, C) = (0, 0, 0), (0, 1, 1), (1, 0, 0), and (1, 0, 1). The output is x for (A, B, C) = (0, 1, 0) and (1, 1, 0). The output is 0 for (A, B, C) = (1, 1, 1).

		BC			
		00	01	11	10
A	0	1	x	x	1
	1		x		1

Diagram illustrating the truth table for the function $F(A, B, C)$ with highlighted cells. The variables are A, B, and C. The table shows the output values (1, x, 0) for each combination of inputs. The output is 1 for (A, B, C) = (0, 0, 0), (0, 1, 1), (1, 0, 0), and (1, 0, 1). The output is x for (A, B, C) = (0, 1, 0) and (1, 1, 0). The output is 0 for (A, B, C) = (1, 1, 1). The cells containing 1 are highlighted in yellow, and the cell containing x at (A=1, BC=10) is highlighted in orange.

$$F(A, B, C) = A' + BC'$$

Bài tập thiết kế mạch tổ hợp

36

- Yêu cầu: Thiết kế mạch tổ hợp 3 ngõ vào, 1 ngõ ra, sao cho giá trị logic ở ngõ ra là giá trị nào chiếm đa số trong các ngõ vào

Bước 1: Lập bảng chân trị

37

- Gọi các ngõ vào là x, y, z - ngõ ra là f

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$f(x, y, z) = \Sigma(3, 5, 6, 7)$$

Bước 2: Viết hàm logic

38

□ $f(x, y, z) = \Sigma(3, 5, 6, 7)$

		yz			
		00	01	11	10
x	0			1	
	1		1	1	1

Brackets in the original image indicate groupings: a blue bracket above the '11' and '10' columns is labeled 'y'; a yellow bracket below the '01', '11', and '10' columns is labeled 'z'; a red bracket to the left of the '1' row is labeled 'x'.

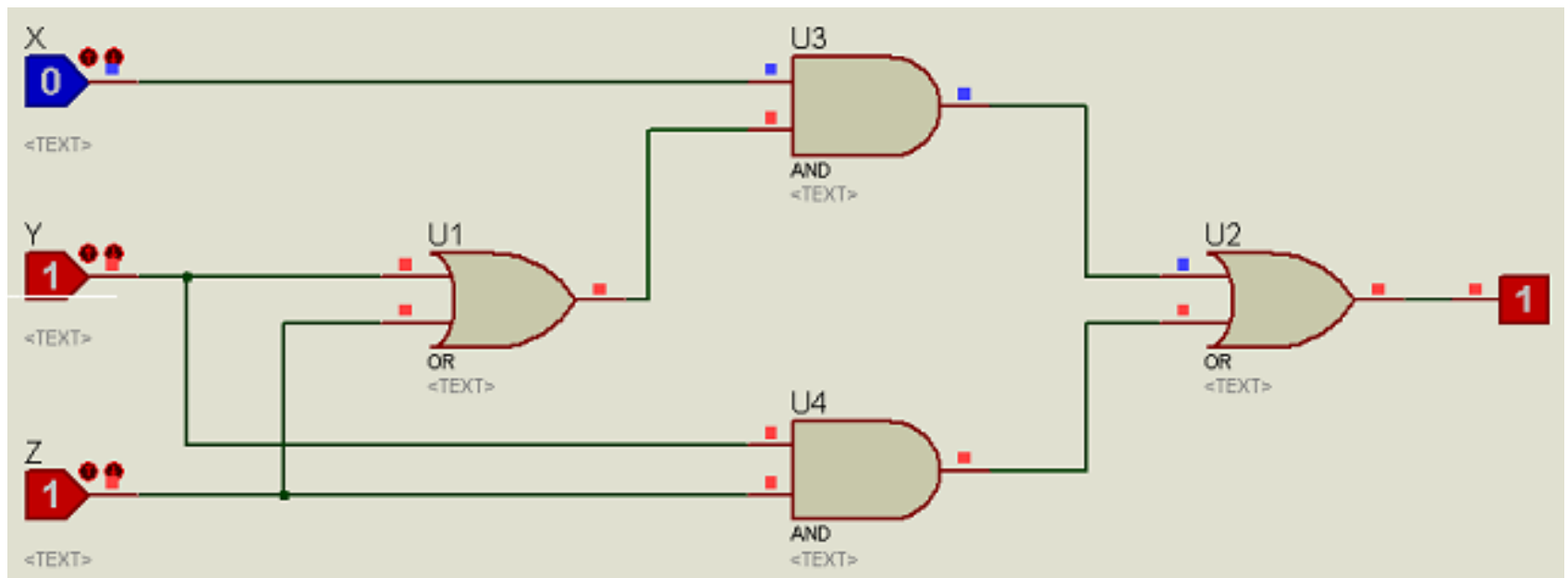
		yz			
		00	01	11	10
x	0			1	
	1		1	1	1

Brackets in the original image indicate groupings: a blue bracket above the '11' and '10' columns is labeled 'y'; a yellow bracket below the '01', '11', and '10' columns is labeled 'z'; a red bracket to the left of the '1' row is labeled 'x'. The '1' cells are highlighted with colored boxes: yellow for (0,11), green for (1,01), and orange for (1,10).

$$f(x, y, z) = xz + xy + yz = x.(y+z) + yz$$

Bước 3: Vẽ sơ đồ mạch và Thử nghiệm

39



Phần 2: Một số mạch tổ hợp cơ bản

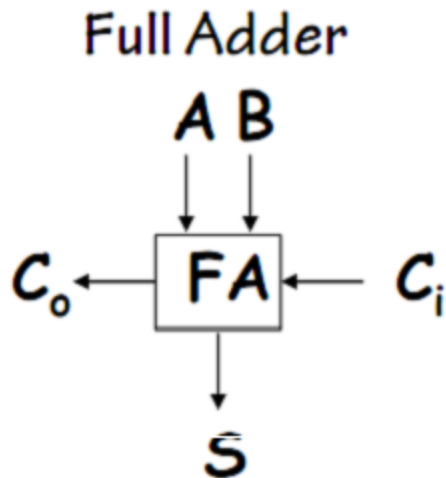
40

- Mạch toàn cộng (Full adder)
- Mạch giải mã (Decoder)
- Mạch mã hoá (Encoder)

Mạch toàn cộng (Full adder - FA)

41

- Mạch tổ hợp thực hiện phép **cộng số học 3 bit**
- Gồm **3 ngõ vào** (A, B: bit cần cộng – C_i : bit nhớ) và **2 ngõ ra** (kết quả có thể từ 0 đến 3 với giá trị 2 và 3 cần 2 bit biểu diễn – S: ngõ tổng, C_0 : ngõ nhớ)



A	B	C_i	S	C_0
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

$$S = F(A, B, C_i) = \Sigma(1, 2, 4, 7)$$

$$C_0 = F(A, B, C_i) = \Sigma(3, 5, 6, 7)$$

Bước 2: Viết hàm logic

42

		A			
		00	01	11	10
Ci	0		1		1
	1	1		1	

Brackets in the original image group columns 01 and 10 under 'A', and columns 00 and 11 under 'B'. The '1' in the cell (Ci=0, AB=10) is highlighted in blue.

$$S = F(A, B, Ci) = \Sigma(1, 2, 4, 7)$$

$$S = A'BCi' + AB'Ci' + A'B'Ci + ABCi$$

$$S = A \oplus B \oplus Ci$$

(Lưu ý: $x \oplus y = x'y + xy'$)

		A			
		00	01	11	10
Ci	0			1	
	1		1	1	1

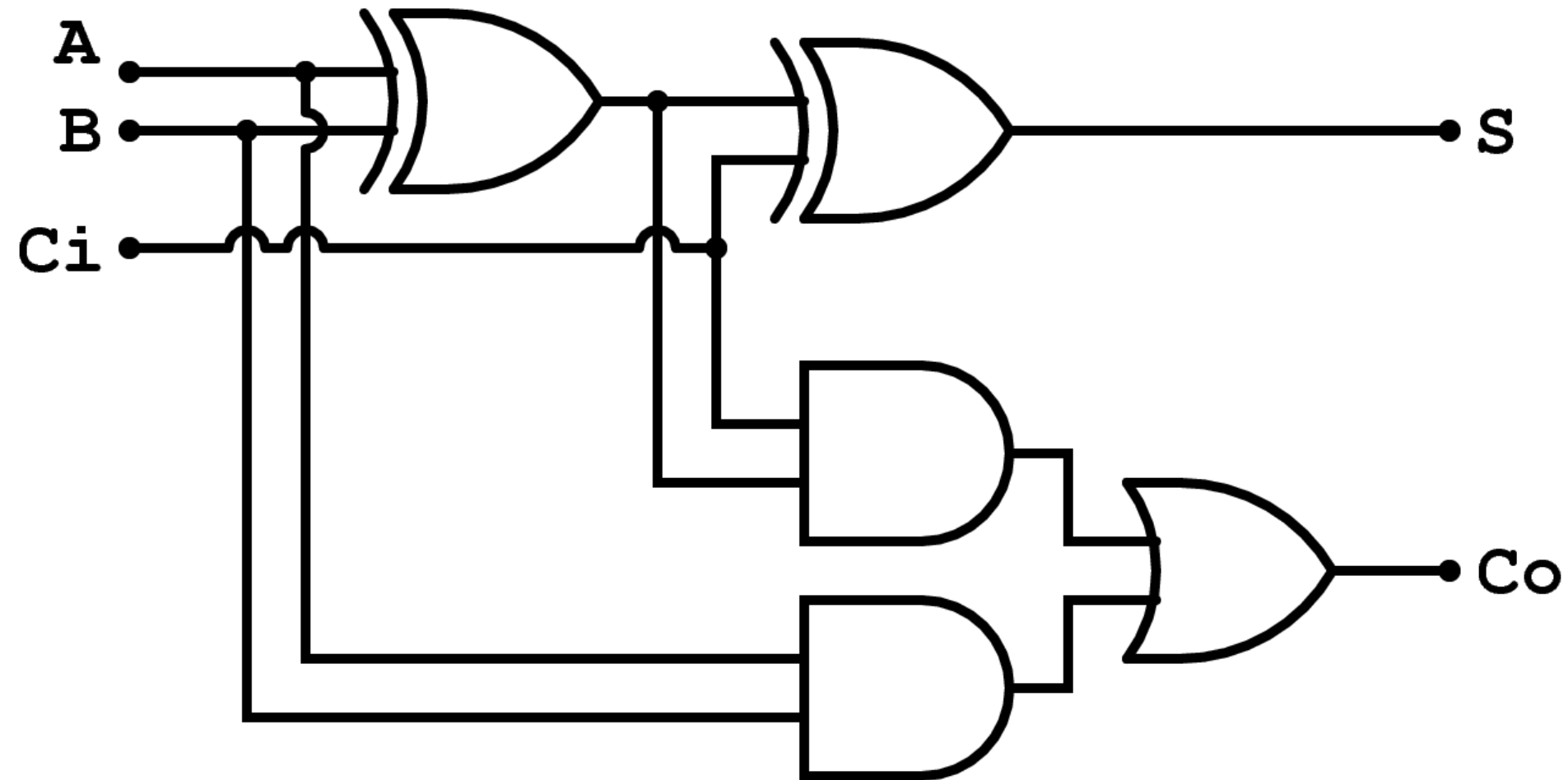
Brackets in the original image group columns 11 and 10 under 'A', and columns 01 and 10 under 'B'. The '1' in the cell (Ci=0, AB=11) is highlighted in yellow, and the '1' in the cell (Ci=1, AB=10) is highlighted in orange.

$$C_0 = F(A, B, Ci) = \Sigma(3, 5, 6, 7)$$

$$C_0 = AB + BCi + ACi$$

Sơ đồ mạch Full adder

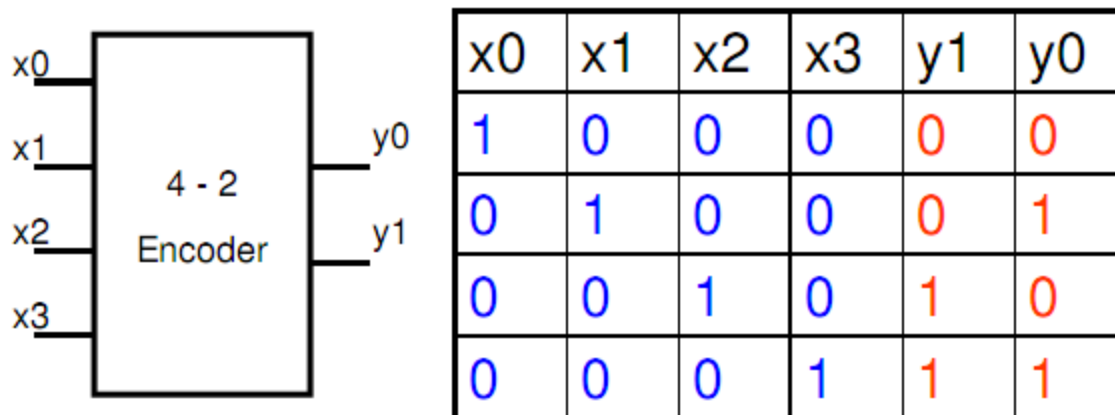
43



Mạch mã hoá nhị phân (Binary Encoder)

44

- Có 2^n (hoặc ít hơn) ngõ vào, n ngõ ra
- Quy định chỉ có **duy nhất một ngõ vào mang giá trị = 1** tại một thời điểm
- Nếu ngõ vào = 1 đó là ngõ thứ k thì các ngõ ra tạo thành số nhị phân có giá trị = k



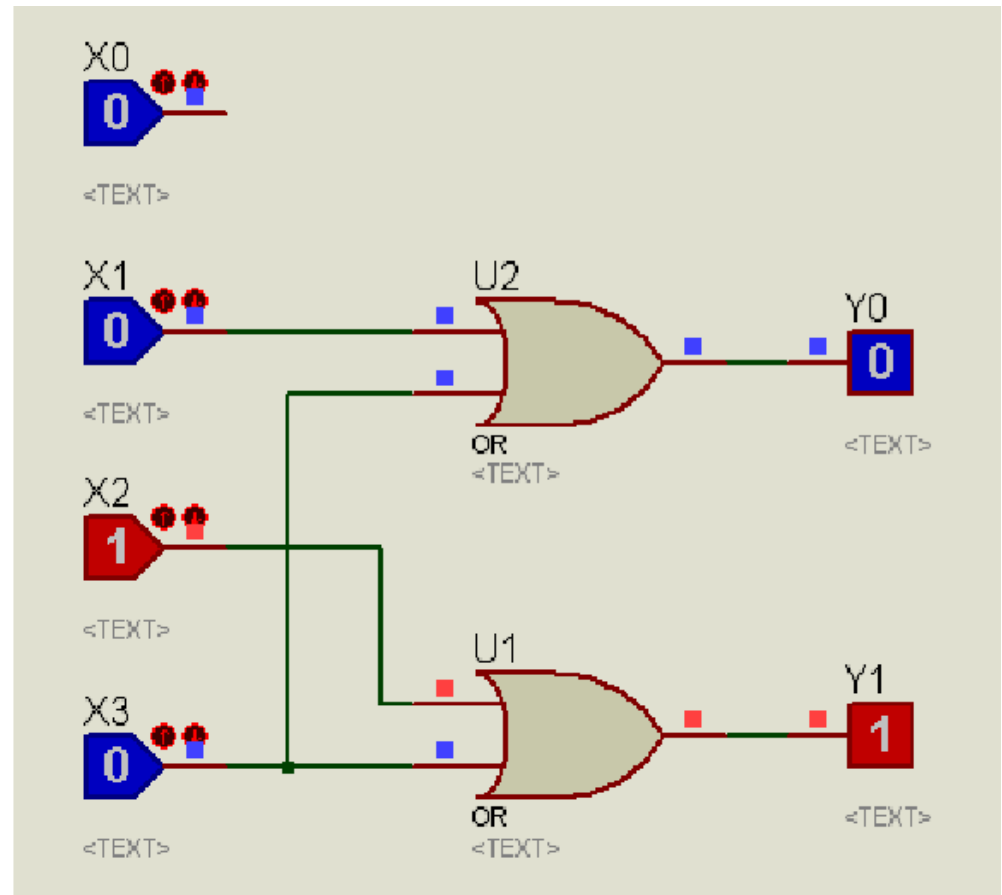
Sơ đồ mạch 4-2 Binary Encoder

45

- Ngõ vào: X_0, X_1, X_2, X_3
- Ngõ ra: Y_0, Y_1

$$Y_0 = X_1 + X_3$$

$$Y_1 = X_2 + X_3$$



Mạch mã hoá theo thứ tự (Priority Encoder)

46

- Các ngõ vào được xem như có độ ưu tiên
- Giá trị ngõ ra phụ thuộc vào các ngõ vào có độ ưu tiên cao nhất
- Ví dụ: Độ ưu tiên ngõ vào $x_3 > x_2 > x_1 > x_0$

x0	x1	x2	x3	y2	y1	y0
0	0	0	0	0	0	0
1	0	0	0	0	0	1
x	1	0	0	0	1	0
x	x	1	0	0	1	1
x	x	x	1	1	0	0

$$y_0 = (x_2 + x_0x_1').x_3$$

$$y_1 = (x_2 + x_1).x_3'$$

$$y_2 = x_3$$

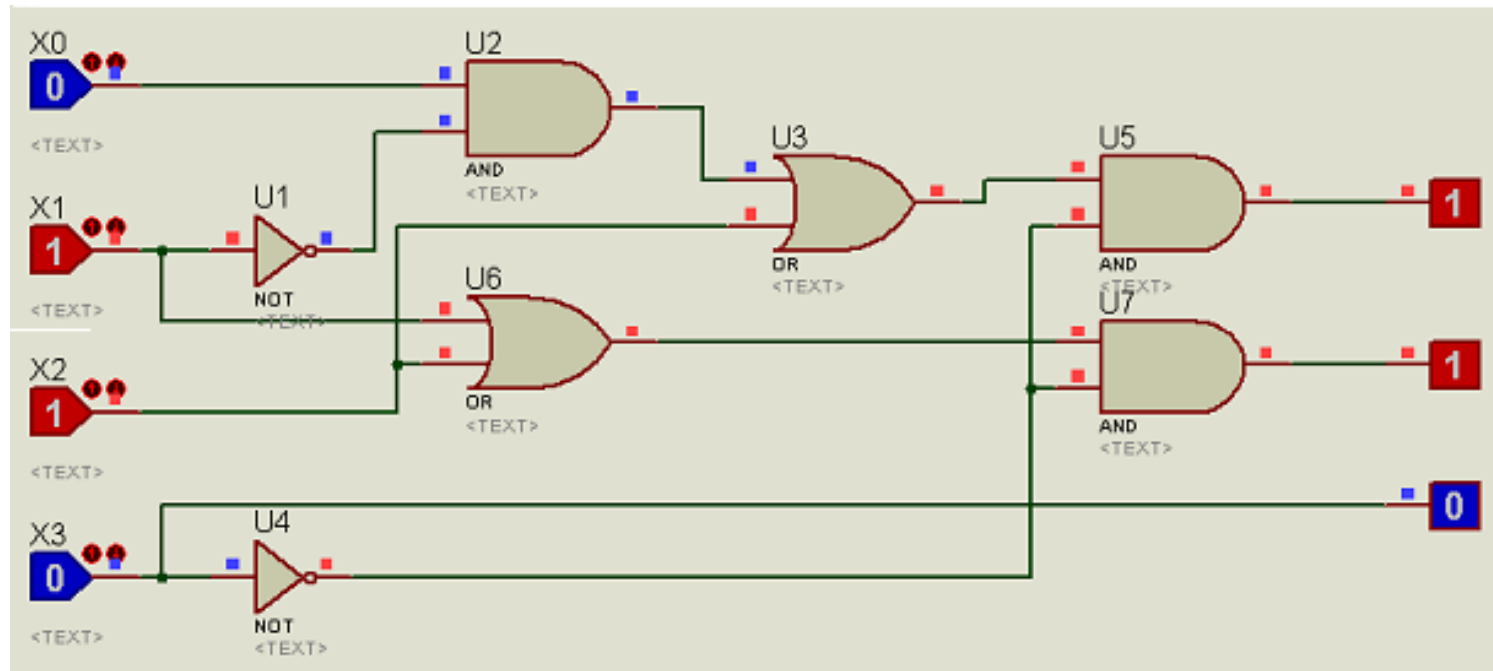
Sơ đồ mạch 4-3 Priority Encoder

47

$$y_0 = (x_2 + x_0x_1').x_3$$

$$y_1 = (x_2 + x_1).x_3'$$

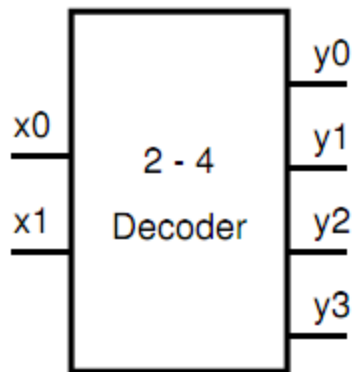
$$y_2 = x_3$$



Mạch giải mã (Decoder)

48

- Có n ngõ vào, 2^n (hoặc ít hơn) ngõ ra
- Quy định chỉ có **duy nhất một ngõ ra mang giá trị = 1** tại một thời điểm
- Nếu các ngõ vào tạo thành số nhị phân có giá trị = k thì ngõ ra = 1 đó là ngõ thứ k



x_1	x_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$y_0 = \overline{x_1} \cdot \overline{x_0}$$

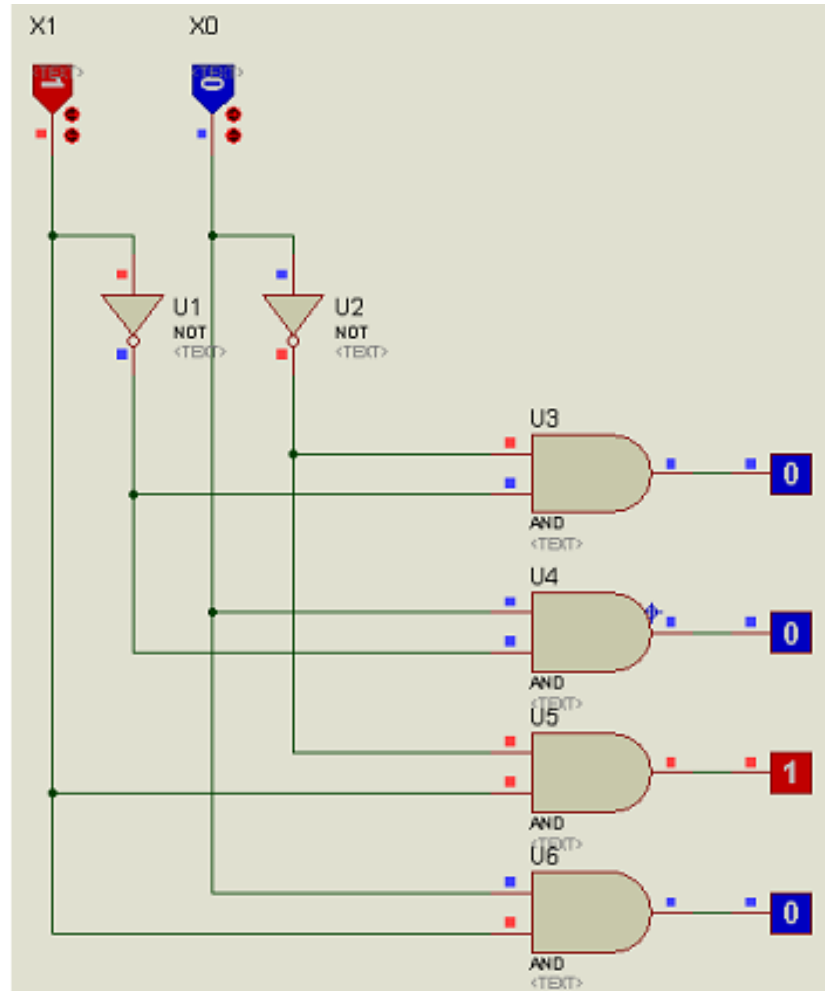
$$y_1 = \overline{x_1} \cdot x_0$$

$$y_2 = x_1 \cdot \overline{x_0}$$

$$y_3 = x_1 \cdot x_0$$

Sơ đồ mạch 2-4 Decoder

49



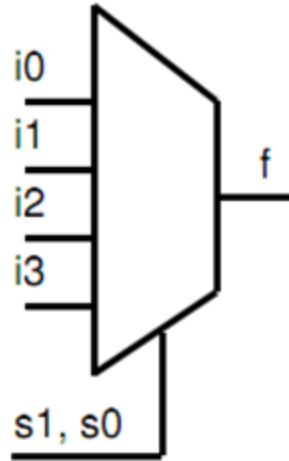
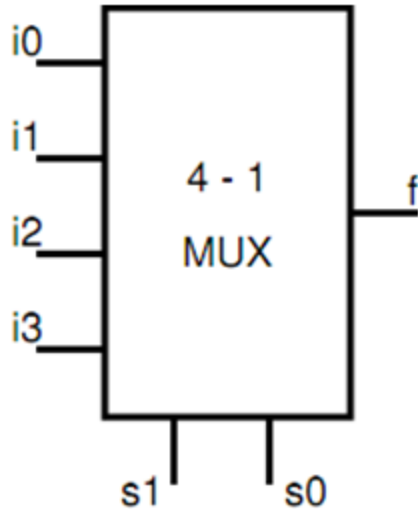
Mạch dồn (Multiplexer - MUX)

50

- Còn gọi là mạch chọn dữ liệu
- Chọn n ngõ trong 2^n ngõ vào để quyết định giá trị của duy nhất 1 ngõ ra
- Mạch dồn $2^n - 1$ có 2^n ngõ nhập, 1 ngõ xuất và n ngõ nhập chọn

Ví dụ: Mạch 4-1 MUX

51

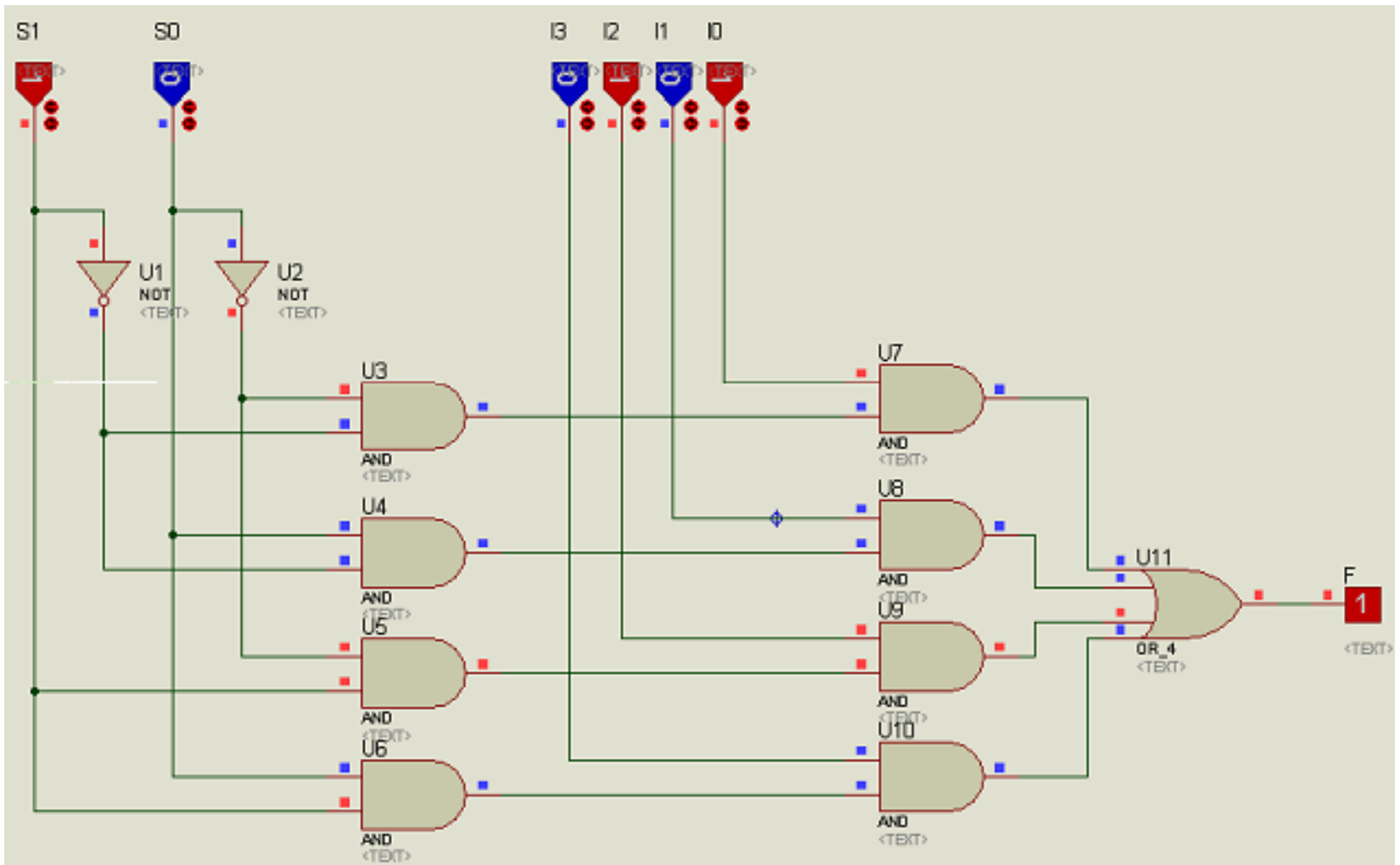


s_1	s_0	f
0	0	i_0
0	1	i_1
1	0	i_2
1	1	i_3

tín hiệu s_1, s_0 dùng để lựa chọn xem tín hiệu nào trong các ngõ vào i_0, i_1, i_2, i_3 được chuyển đến ngõ ra f

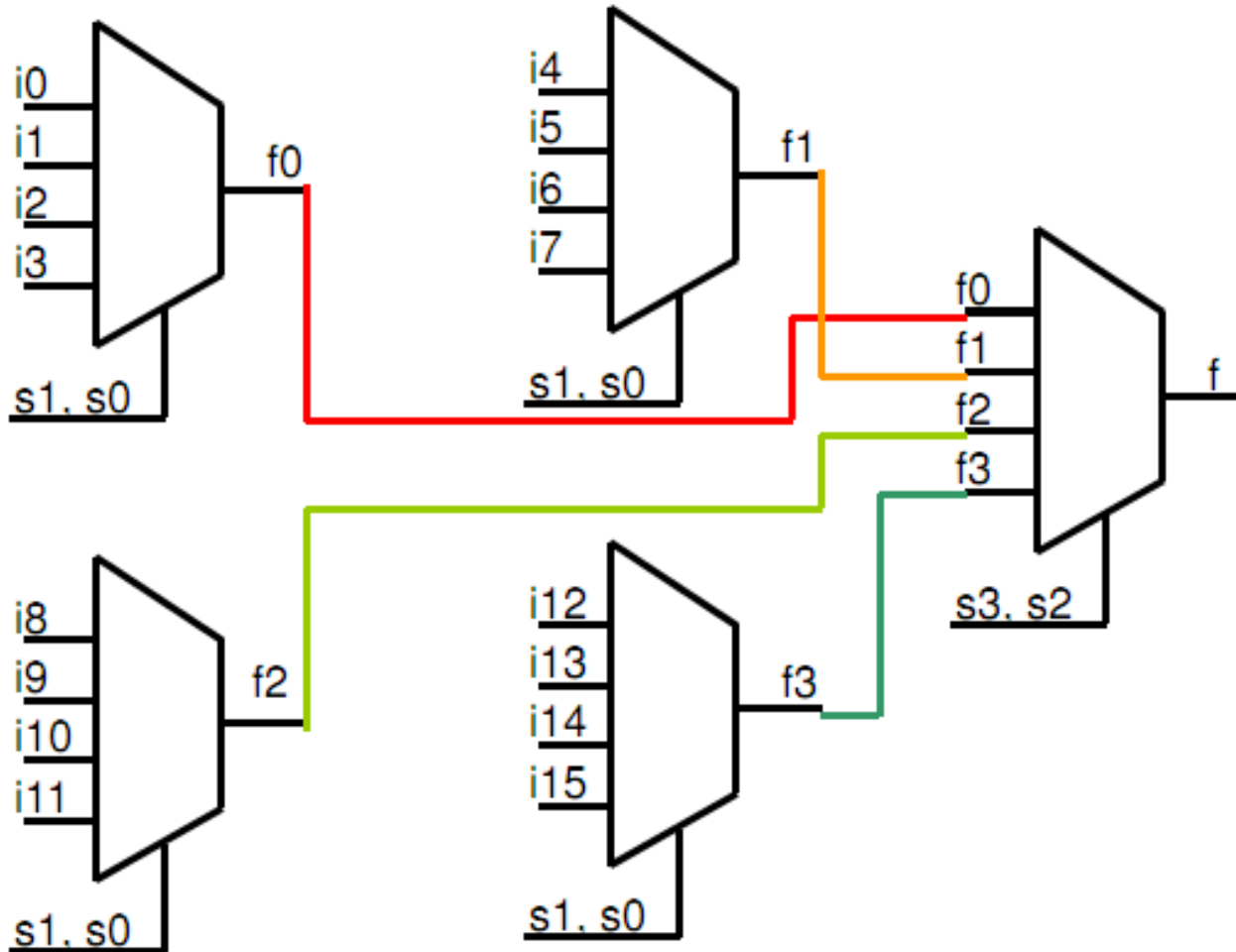
Sơ đồ mạch 4-1 MUX

52



16-1 MUX

53



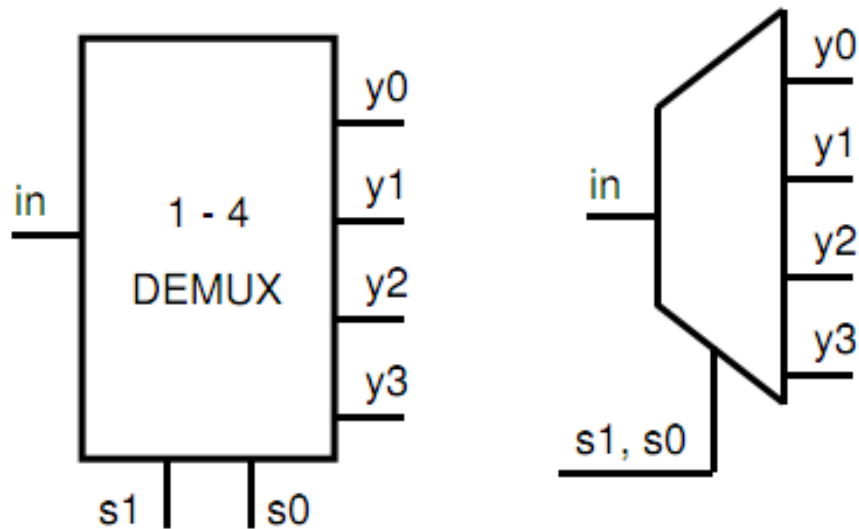
Mạch tách Demultiplexer (DEMUX)

54

- Chọn n ngõ trong 2^n ngõ vào để quyết định giá trị của duy nhất 1 ngõ ra
- Mạch DEMUX $1-2^n$ có 1 ngõ nhập, 2^n ngõ xuất và n ngõ nhập chọn

Ví dụ: Mạch 1-4 DEMUX

55

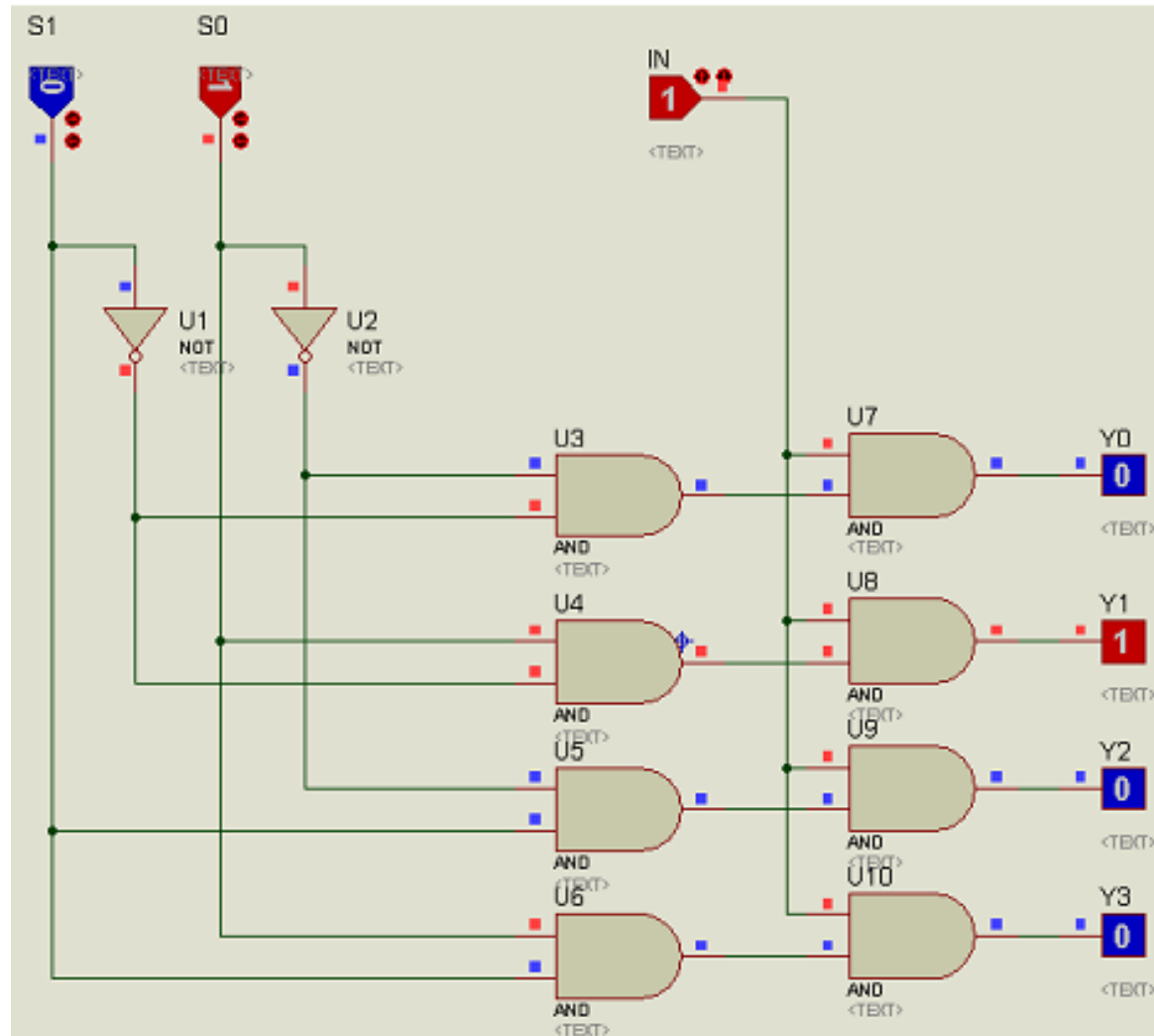


tín hiệu **s1**, **s0** dùng để lựa chọn xem tín hiệu vào **in** sẽ được chuyển đến ngõ nào trong các ngõ ra **y0,y1,y2,y3**

s1	s0	y0	y1	y2	y3
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

Sơ đồ mạch 1-4 DEMUX

56

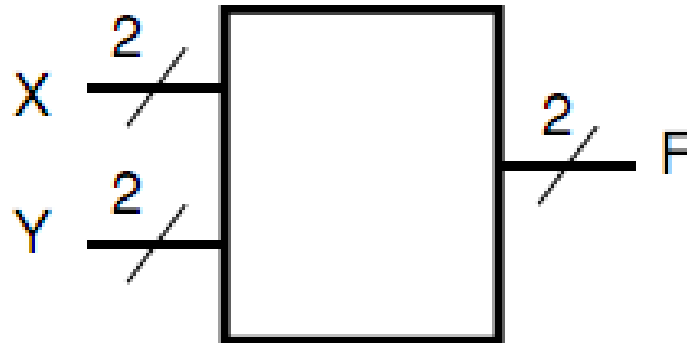


Bài tập: Thiết kế mạch ALU

57

- $F = (5X + 2Y) \% 4$
- Input: X (2 bit), Y (2 bit)
- Output: F (2 bit)

→ Có 4 ngõ vào, 2 ngõ ra (mỗi ngõ có 1 tín hiệu biểu diễn cho 1 bit)



Bước 1: Lập bảng chân trị

58

X	Y	F
0 (00)	0 (00)	0 (00)
0 (00)	1 (01)	2 (10)
0 (00)	2 (10)	0 (00)
0 (00)	3 (11)	2 (10)
1 (01)	0 (00)	1 (01)
1 (01)	1 (01)	3 (11)
1 (01)	2 (10)	1 (01)
1 (01)	3 (11)	3 (11)
2 (10)	0 (00)	2 (10)
2 (10)	1 (01)	0 (00)
2 (10)	2 (10)	2 (10)
2 (10)	3 (11)	0 (00)
3 (11)	0 (00)	3 (11)
3 (11)	1 (01)	1 (01)
3 (11)	2 (10)	3 (11)
3 (11)	3 (11)	1 (01)

Bước 2: Xác định hàm

59

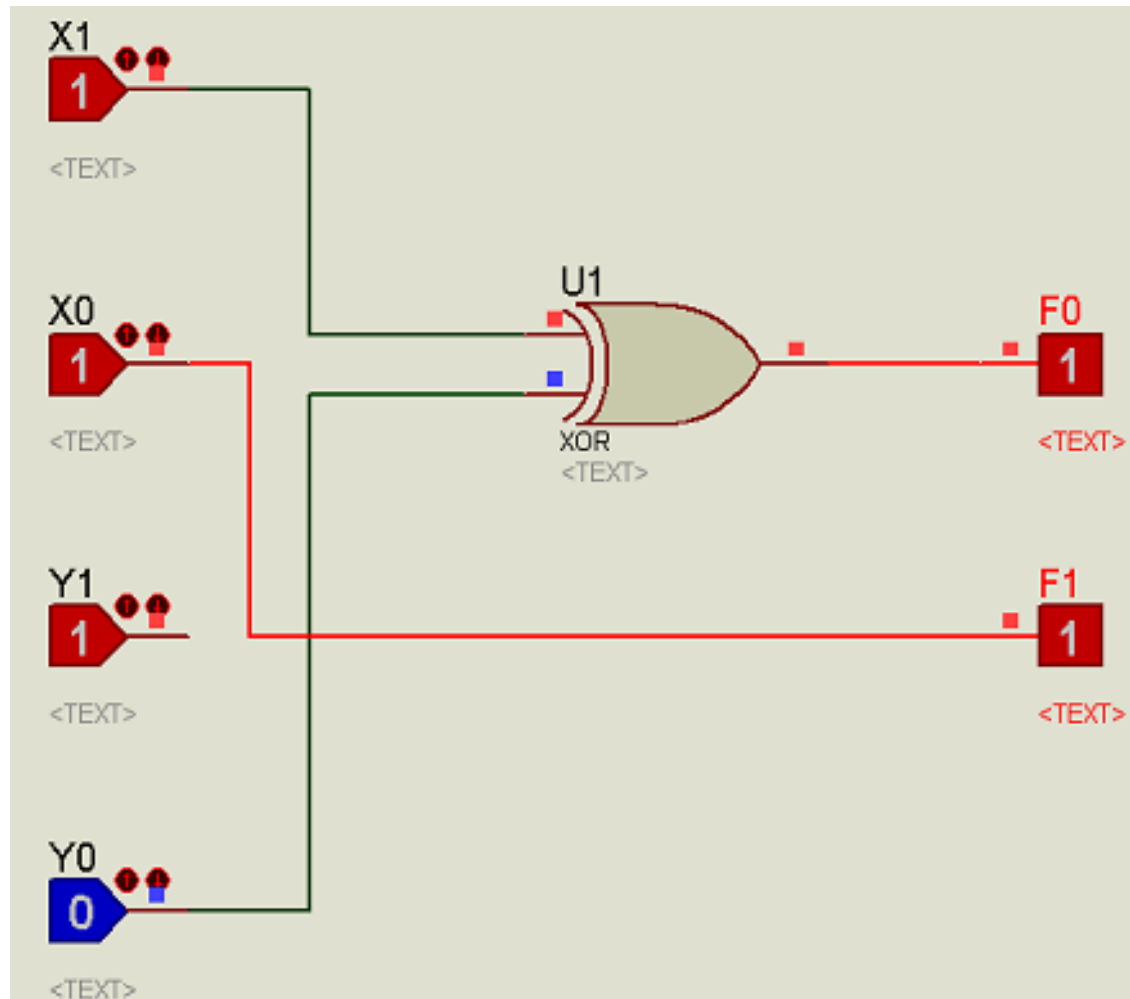
		Y1 Y0			
		00	01	11	10
X1 X0	00	0	1	1	0
	01	0	1	1	0
	11	1	0	0	1
	10	1	0	0	1

$$F1 = X1.Y0' + X1'.Y0$$

$$F0 = X0$$

Bước 3: Vẽ mạch

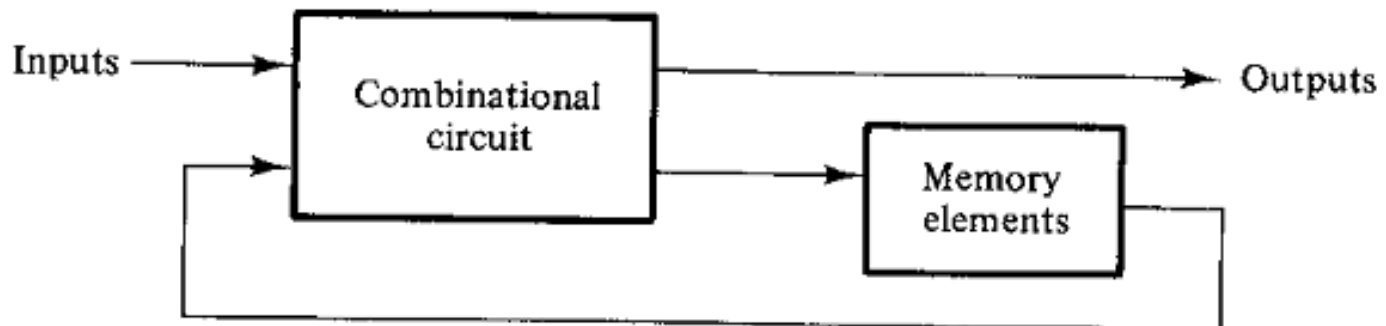
60



Phần 3: Mạch tuần tự

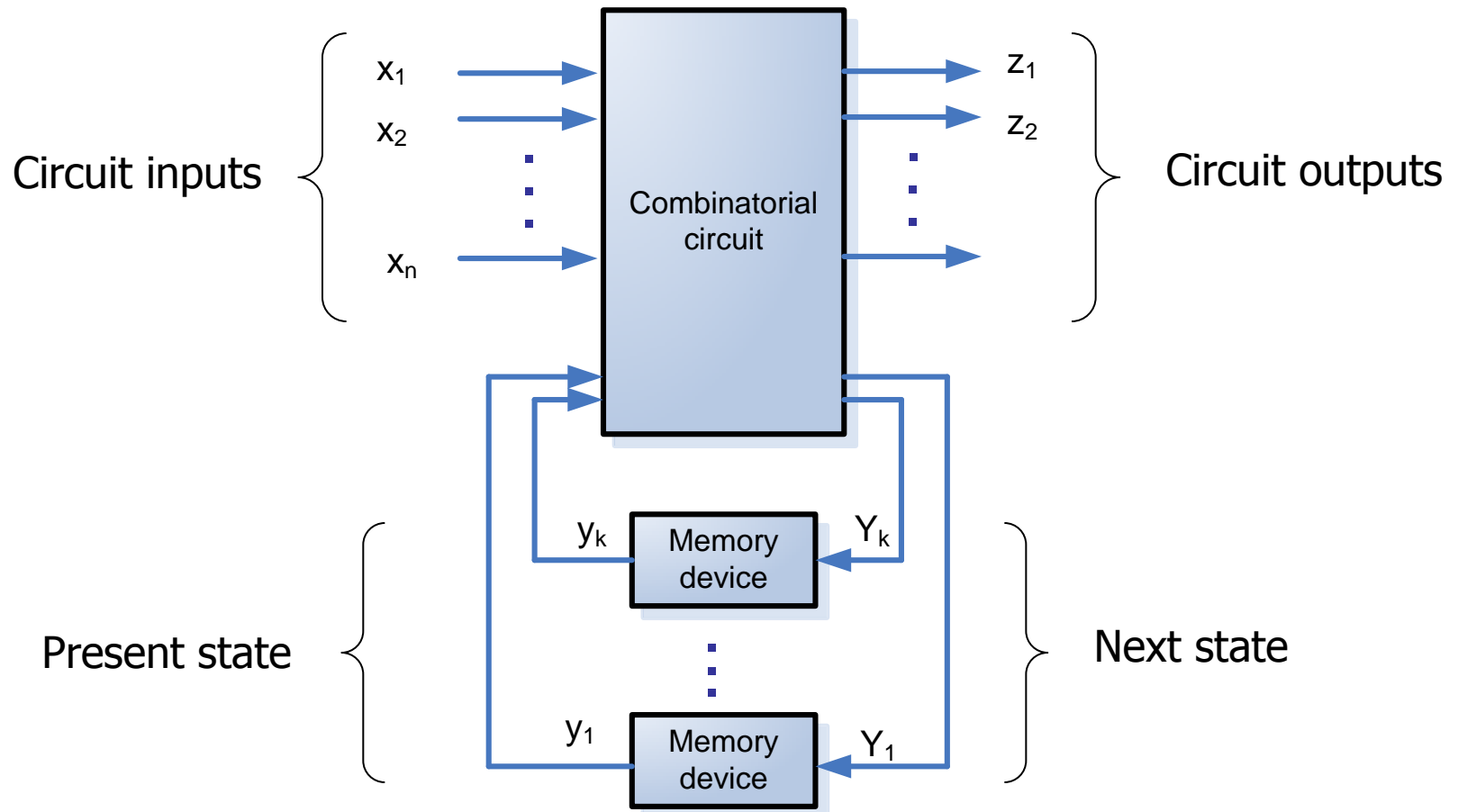
61

- Khác với mạch tổ hợp, ở mạch tuần tự thì ngõ ra không chỉ phụ thuộc vào giá trị hiện thời của ngõ vào, mà còn phụ thuộc giá trị quá khứ
- Mạch tuần tự có khả năng “ghi nhớ các trạng thái trong quá khứ”



Mạch tuần tự

62



Mạch lật

63

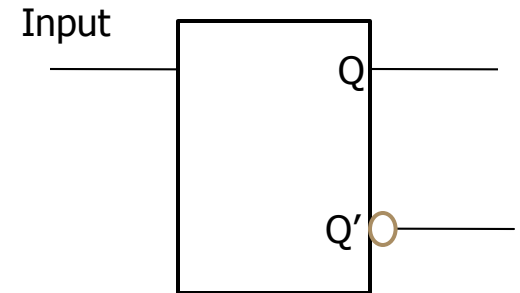
- Là 1 thành phần cấu thành mạch tuần tự
- Có chức năng lưu trữ 1 bit nhị phân
- Có nhiều loại mạch lật, sự khác nhau ở chỗ số ngõ vào và cách thức các ngõ vào tác động đến trạng thái bit nhị phân

Phân loại mạch lật

64

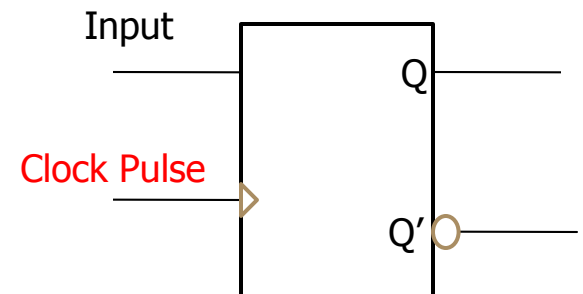
□ Latch

- Ngõ ra thay đổi trạng thái khi ngõ vào thay đổi giá trị
- Độ trễ mạch (delayed gate) giá trị mới của ngõ ra được xác định bằng độ trễ giữa ngõ vào và ngõ ra
- Được sử dụng như 1 thành phần nhớ của mạch tuần tự **bất đồng bộ**



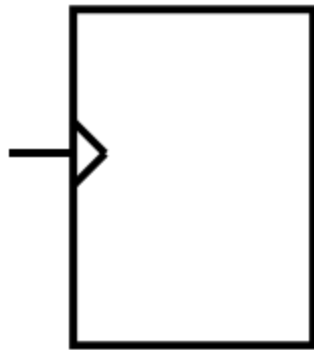
□ Flip-Flop

- Bên cạnh những ngõ vào thông thường thì luôn có 1 ngõ vào kích hoạt (trigger input), gọi là clock
- Trạng thái của ngõ ra chỉ có thể thay đổi khi ngõ vào kích hoạt (clock) thay đổi xung đồng hồ (clock pulse) của nó (0 → 1 hoặc 1 → 0)
- Được sử dụng như 1 thành phần nhớ của mạch tuần tự **đồng bộ**



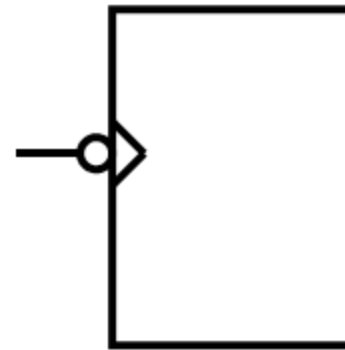
Tín hiệu lễ xung đồng hồ - Clock edge

65



Clk

Chuyển tiếp lễ dương ($0 \rightarrow 1$)



Clk

Chuyển tiếp lễ âm ($1 \rightarrow 0$)



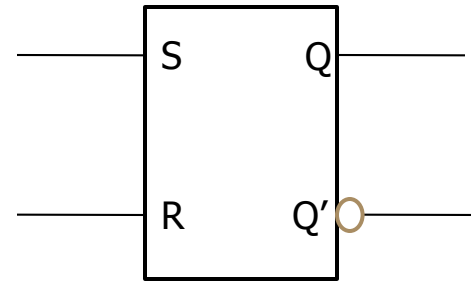
Positive edge

Negative edge

RS Latch (SR Latch)

66

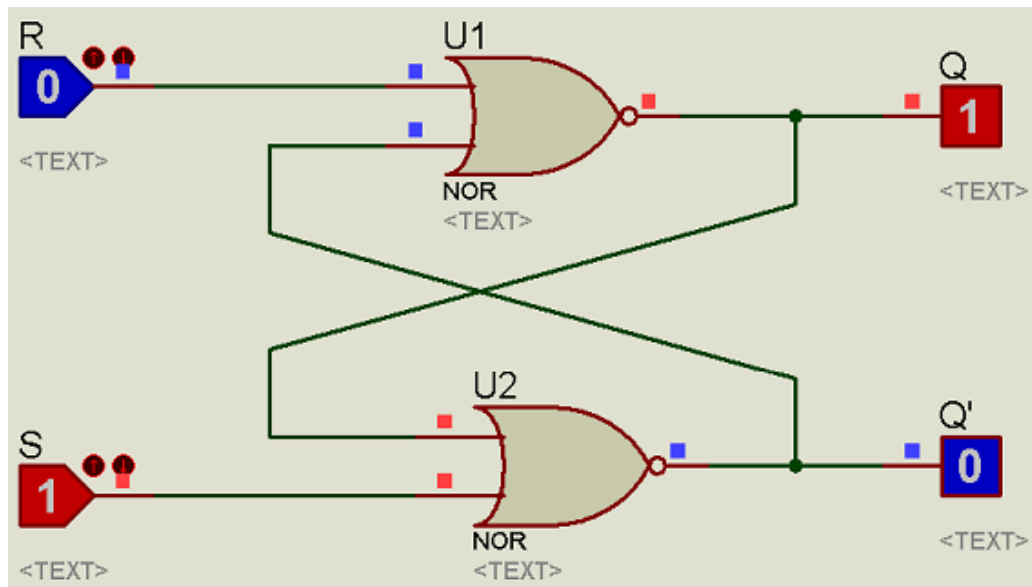
- Có 2 ngõ vào:
 - S (Set): Đặt
 - R (Reset): Khởi động
- Có 2 ngõ ra Q và Q' (tín hiệu đảo của Q)
- Trạng thái ngõ ra $Q_{\text{next}} = Q(t+1)$ phụ thuộc vào trạng thái ngõ vào S, R và tình trạng hiện tại của mạch $Q_{\text{current}} = Q(t)$



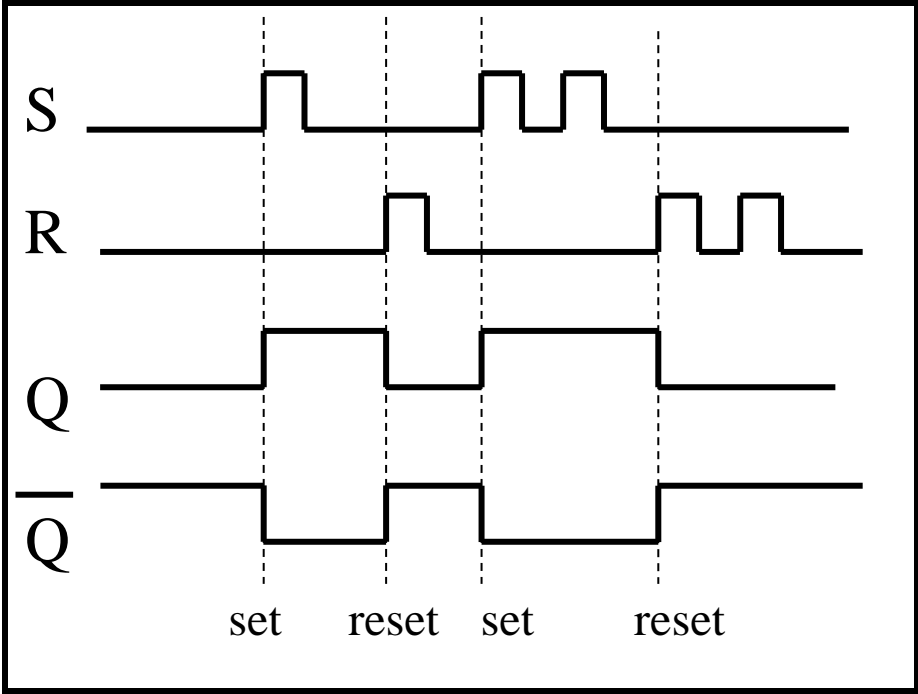
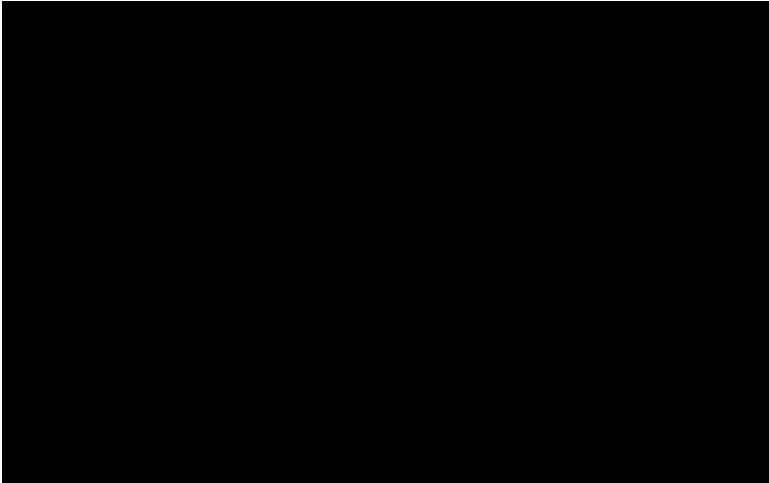
RS Latch

67

S	R	$Q = Q(t+1)$	Q'	Ý nghĩa
0	0	$Q(t)$	$(Q(t))'$	Không đổi
0	1	0	1	= 0
1	0	1	0	= 1
1	1	undefined	undefined	Không xác định



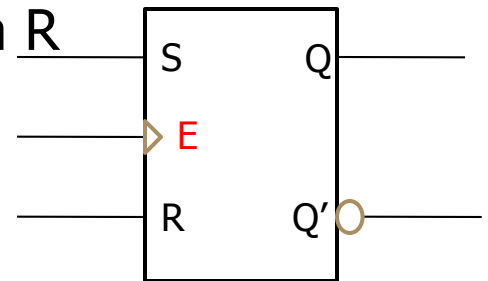
Timing chart



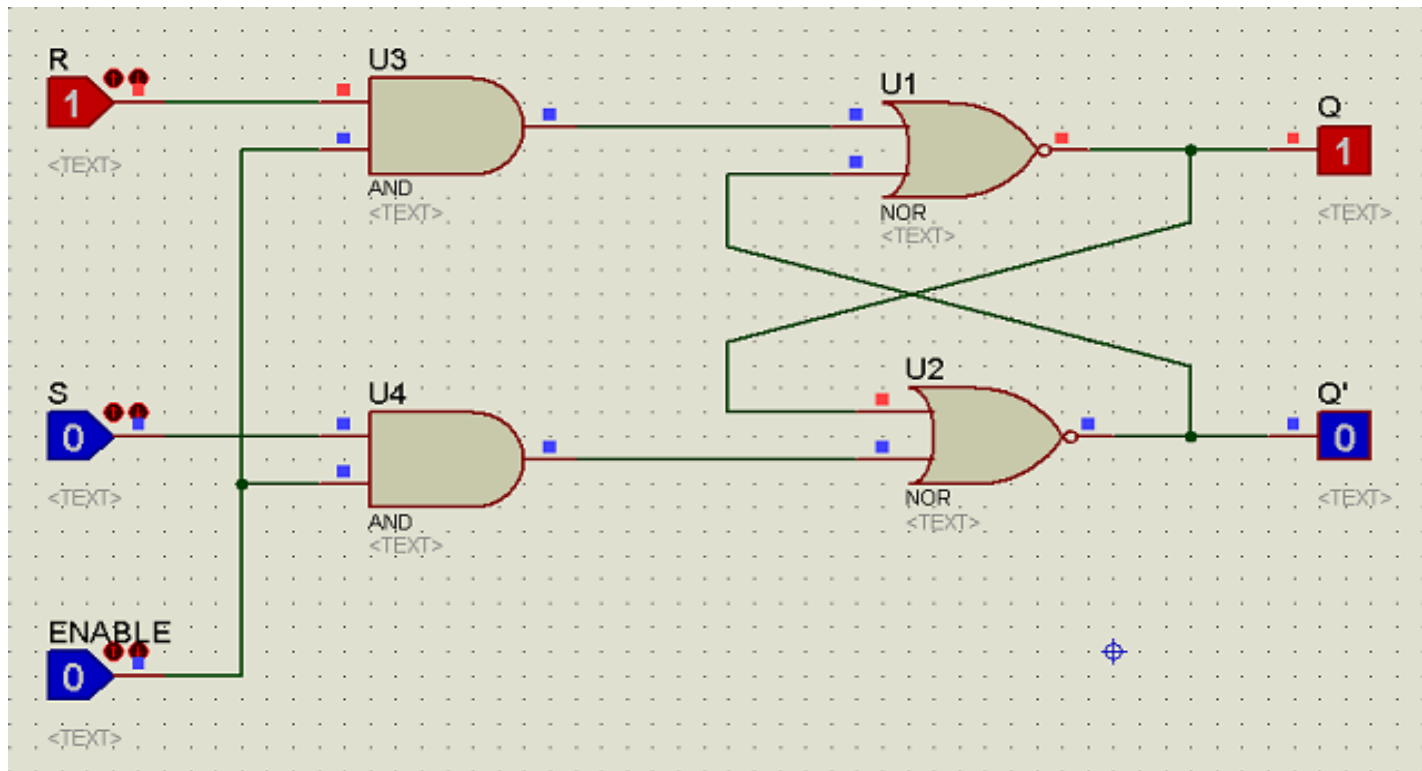
RS Flip-Flop

69

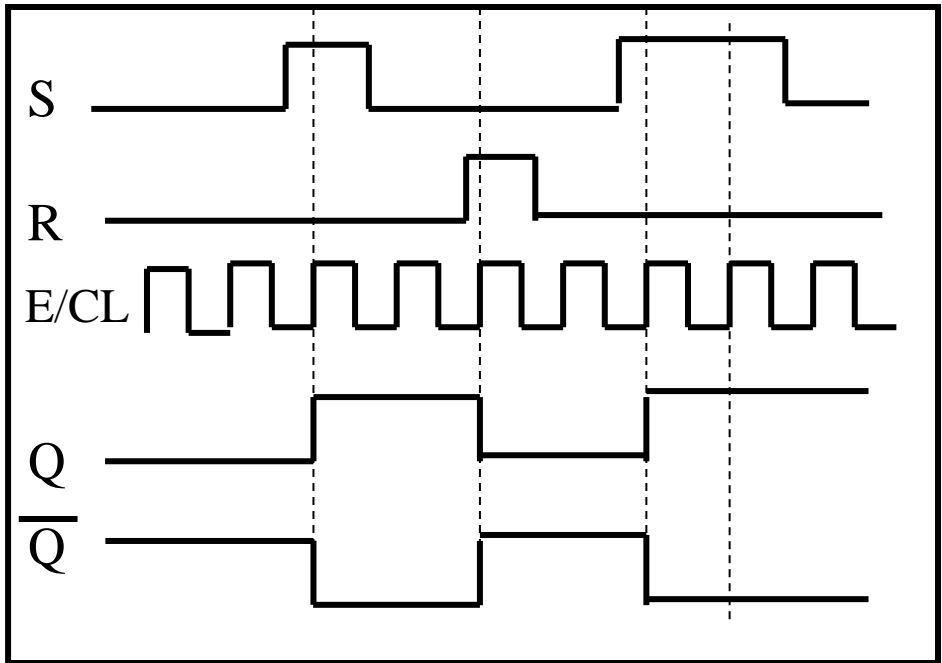
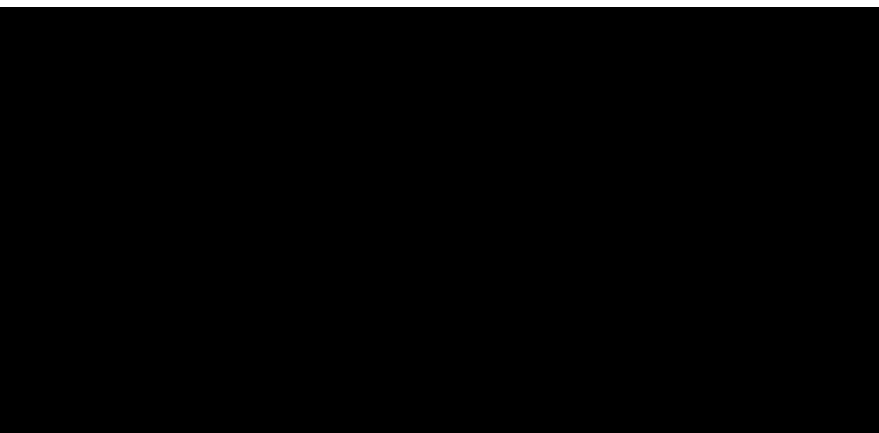
- Dùng thêm 1 tín hiệu ngõ vào kích hoạt “Enabled” (thường là tín hiệu xung đồng hồ Clock - C) để điều khiển mạch
 - **Enabled = 1 (Positive Clock Edge):** mạch hoạt động như mạch lật RS Latch
 - **Enabled = 0 (Negative Clock Edge):** mạch bị vô hiệu hoá,
→ Q giữ nguyên giá trị → $Q(t+1) = Q(t)$
- Chỉ khi tín hiệu Enabled đổi từ 0 sang 1 (positive edge triggered), ngõ ra mới có thể bị ảnh hưởng, nếu không thì không thể thay đổi bất chấp giá trị của S và R



E	S	R	$Q = Q(t+1)$	Q'	Ý nghĩa
0	x	x	$Q(t)$	$(Q(t))'$	Không đổi
1	0	0	$Q(t)$	$(Q(t))'$	Không đổi
1	0	1	0	1	= 0
1	1	0	1	0	= 1
1	1	1	undefined	undefined	Không xác định



Timing chart

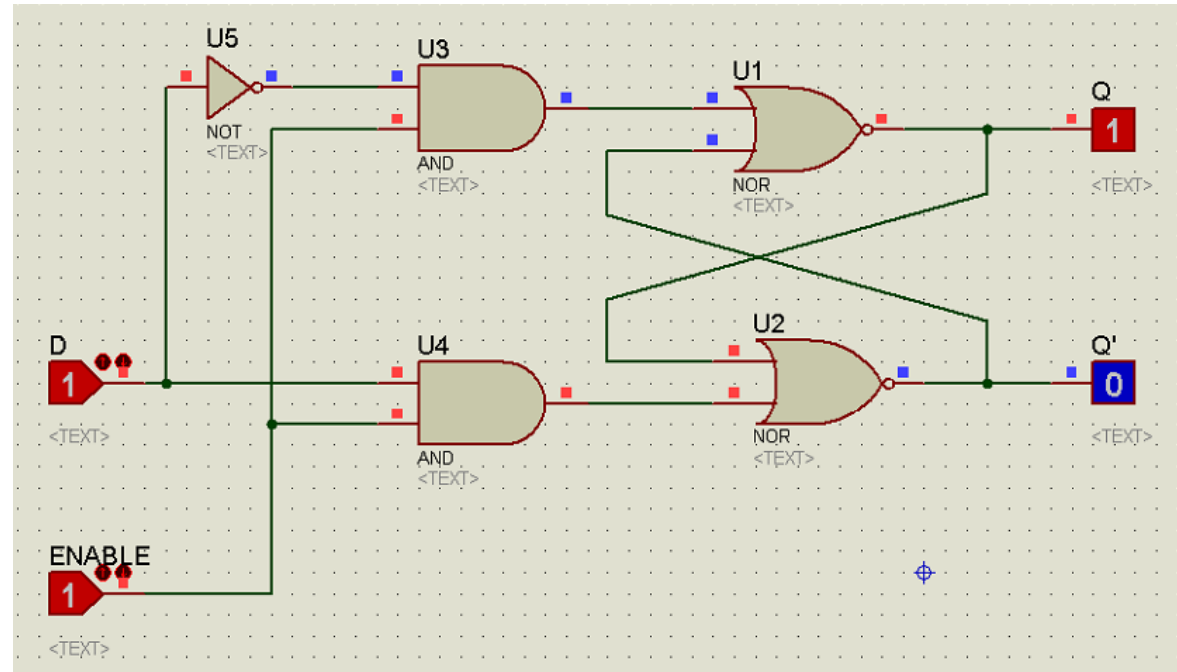
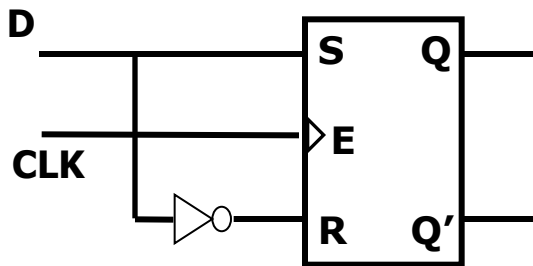
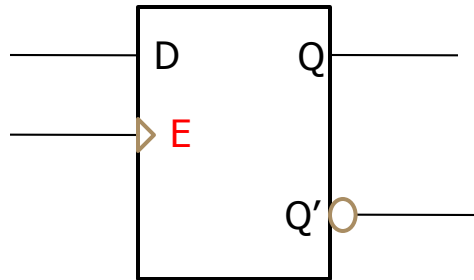


D (Data) Flip-Flop

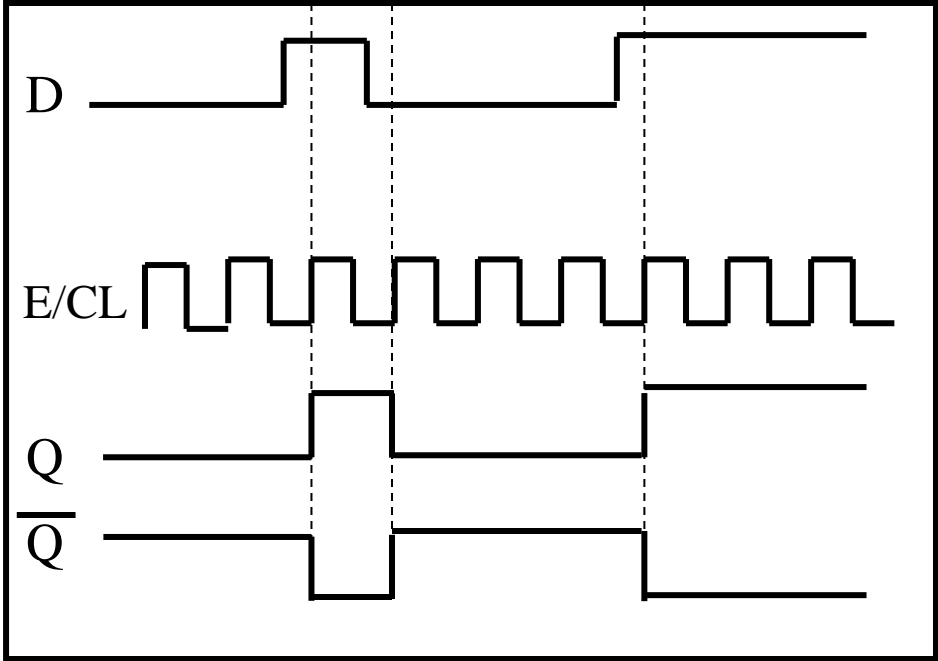
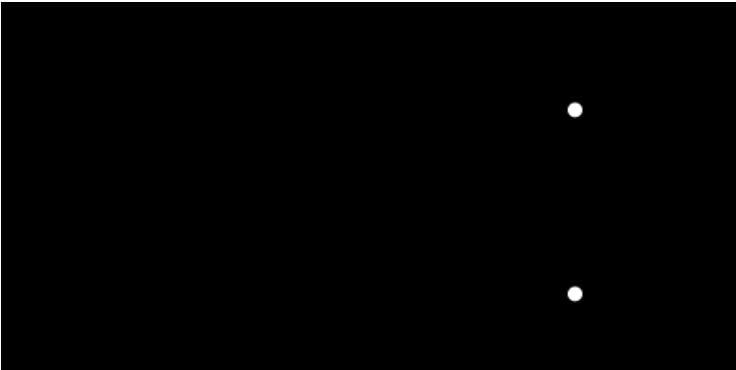
72

- Để tránh trường hợp $R = S = 1$ trong RS Flip-Flop, trong mạch lật D Flip-Flop ta chỉ dùng **1 ngõ vào D** nhưng tách ra 2 tín hiệu, 1 trong 2 tín hiệu sẽ đi qua cổng NOT để tạo tín hiệu đảo của D
- Không bao giờ xảy ra trường hợp 2 tín hiệu vào mạch đều bằng 1
- Nhưng bên cạnh đó cũng không bao giờ xảy ra 2 tín hiệu vào mạch đều bằng 0 ☹️
- Ta không thể giữ nguyên trạng thái tín hiệu ngõ ra $Q(t + 1) = Q(t)$
- ***Để khắc phục ta sẽ dùng tín hiệu xung đồng hồ để vô hiệu hoá mạch khi cần, lúc đó trạng thái tín hiệu ngõ ra sẽ không đổi***

E	D	$Q = Q(t+1)$	Q'	Ý nghĩa
0	x	$Q(t)$	$(Q(t))'$	Không đổi
1	0	0	1	= 0
1	1	1	0	= 1



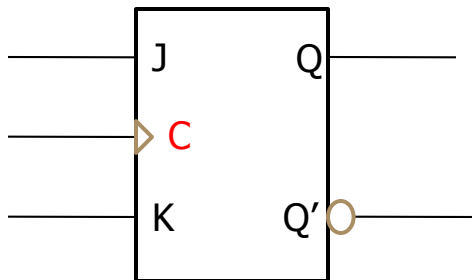
Timing chart



JK Flip-Flop

75

- Là 1 cải tiến của mạch RS Flip-Flop đối với trường hợp $R = S = 1$
- Nguyên tắc:
 - $J = S$
 - $K = R$
 - Nếu $J = K = 1$ thì khi đó với 1 chuyển tiếp của tín hiệu xung đồng hồ sẽ chuyển tín hiệu ngõ ra Q sang trạng thái bù Q'



J	K	$Q = Q(t+1)$	Q'	Ý nghĩa
0	0	$Q(t)$	$(Q(t))'$	Không đổi
0	1	0	1	= 0
1	0	1	0	= 1
1	1	$Q'(t)$	$Q(t)$	Đảo bit

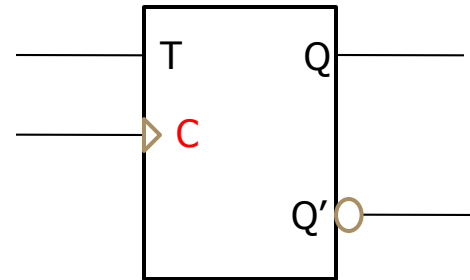
Mạch lật T

76

- Xuất phát từ mạch JK Flip-Flop với sự kết hợp 2 ngõ vào J, K thành duy nhất 1 ngõ vào T ($T = J = K$)

- $T = 0: Q(t + 1) = Q(t)$

- $T = 1: Q(t + 1) = (Q(t))'$

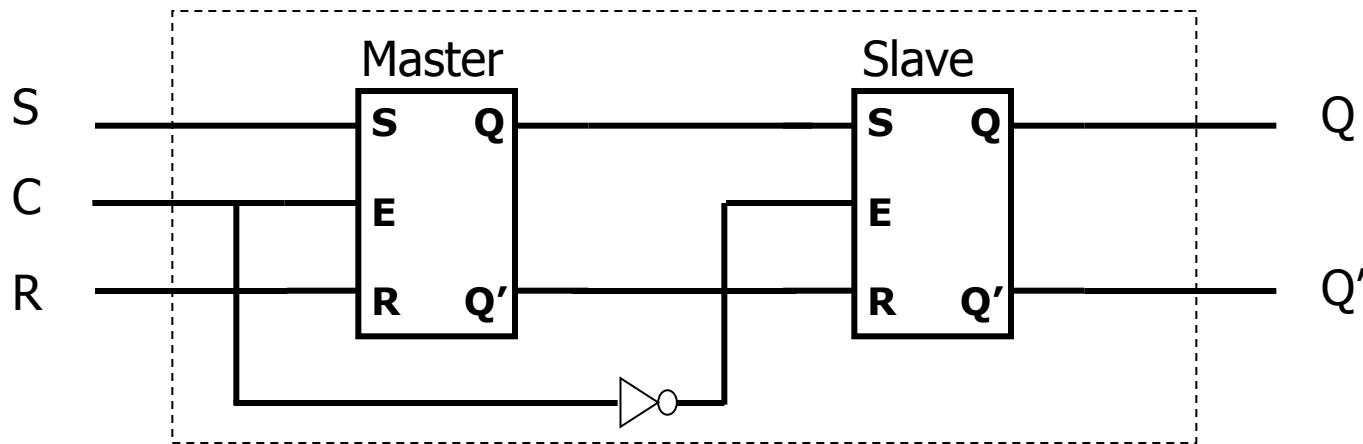


T	$Q = Q(t+1)$	Q'	Ý nghĩa
0	$Q(t)$	$(Q(t))'$	Không đổi
1	$(Q(t))'$	$Q(t)$	Đảo bit

Master-Slave Flip-Flop

77

- Bao gồm 2 bản mạch flip-flop tuần tự nối với nhau (master – slave)
- Tín hiệu ngõ ra Q phụ thuộc vào giá trị của những ngõ vào tại những chuyển tiếp lề âm / dương của xung đồng hồ (clock edge)
- Master flip-flop (trước) thay đổi \rightarrow Slave flip-flop (sau) thay đổi

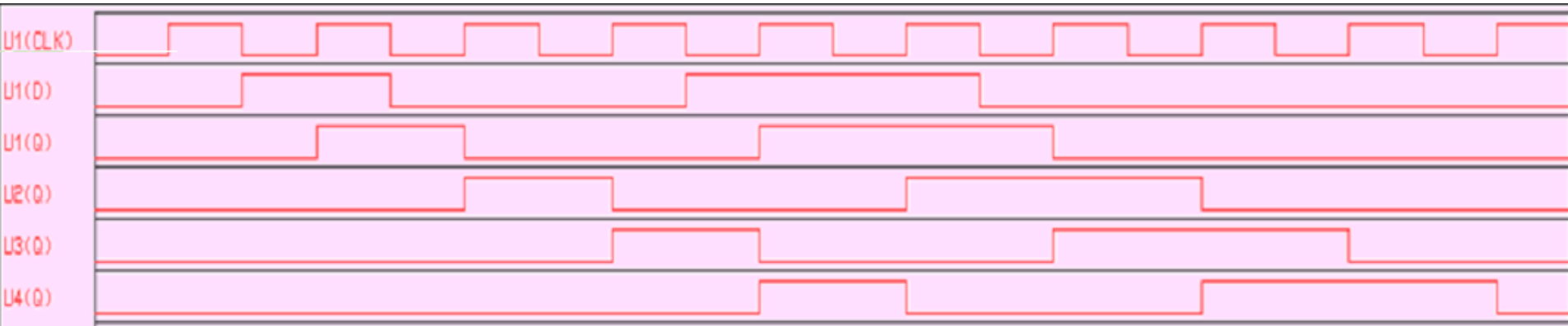
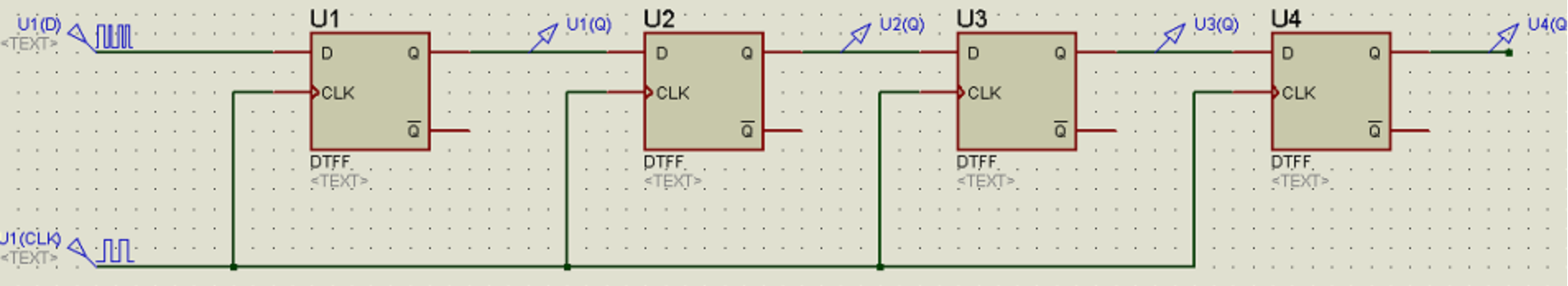


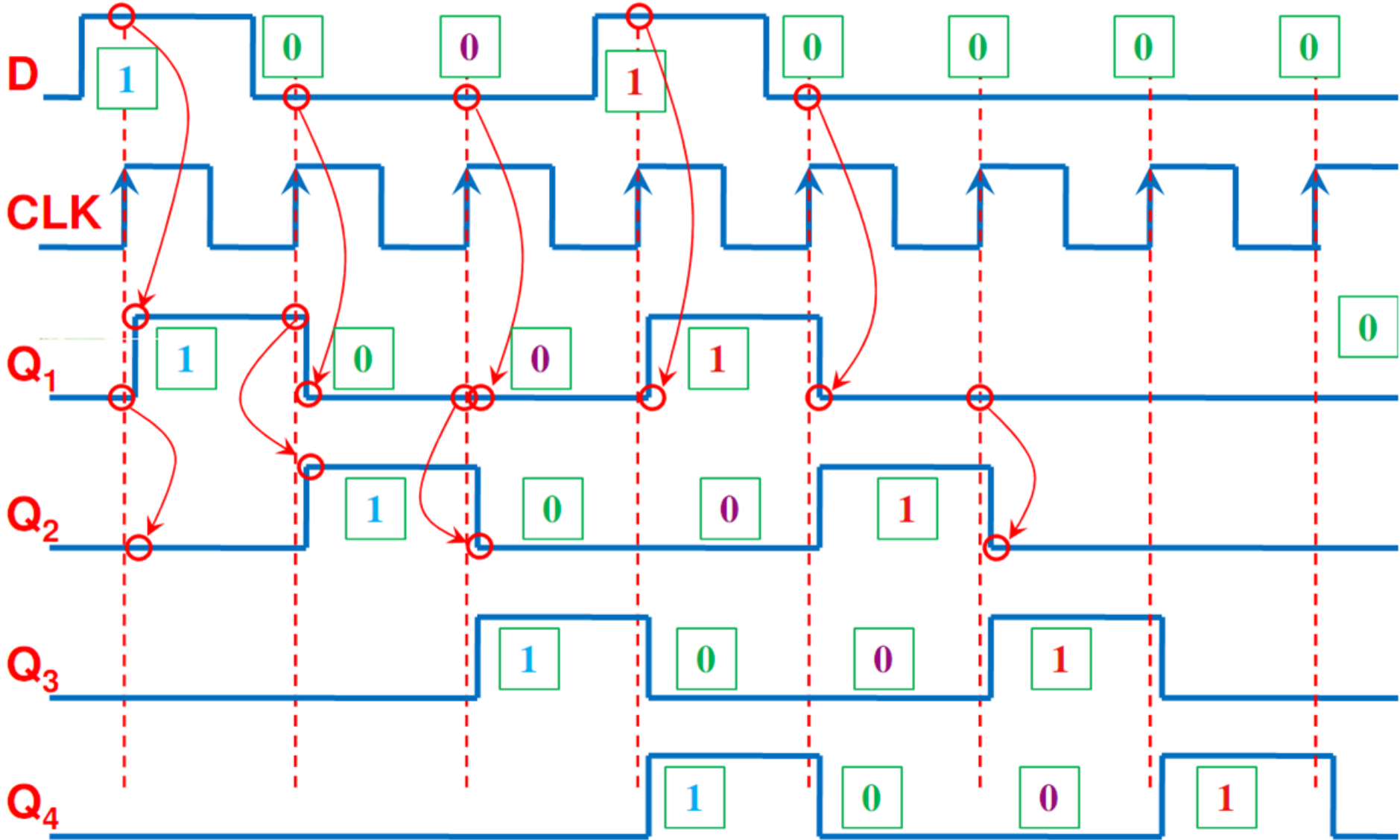
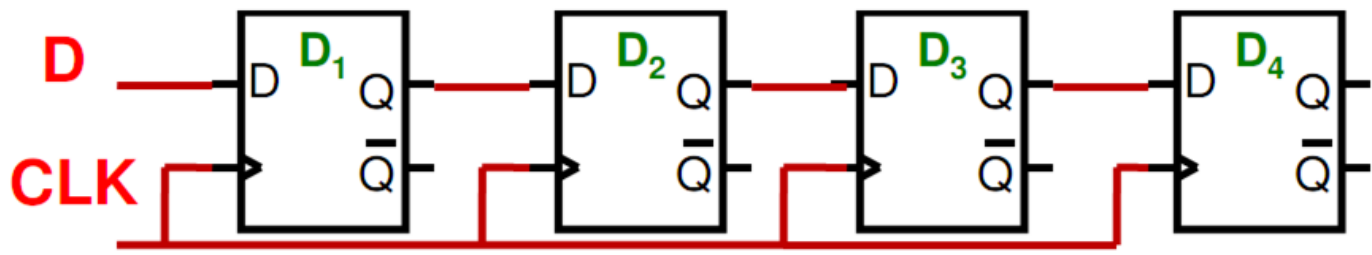
Master works when $C=1$
Slave works when $C=0$

Thanh ghi dịch (Shift Register)

78

- Thanh ghi dịch 4 bit
- $U1(D) = 01001100\dots \rightarrow U4(Q) = 00000100\dots$

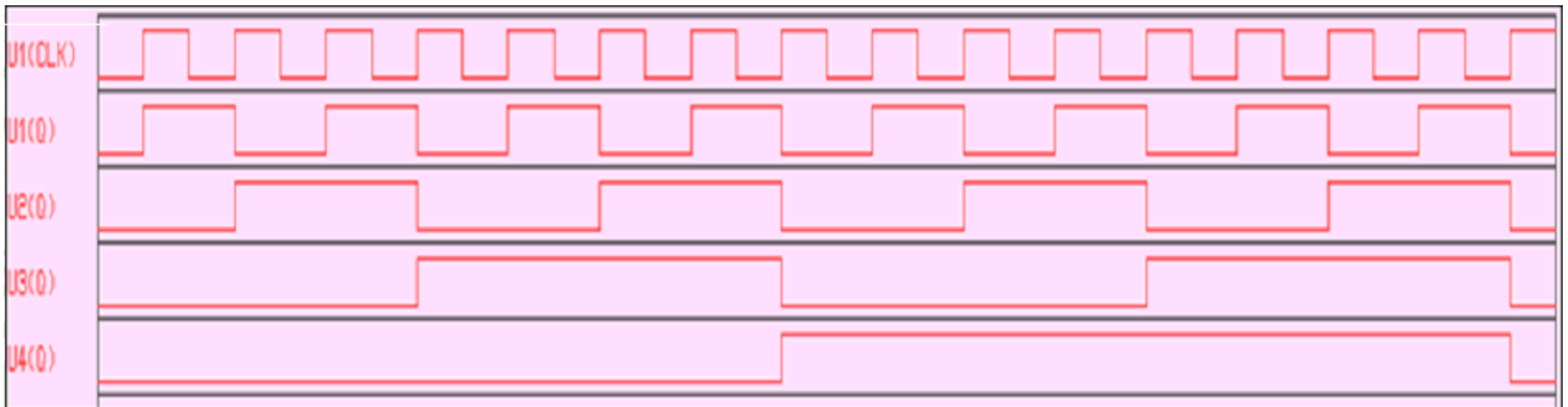
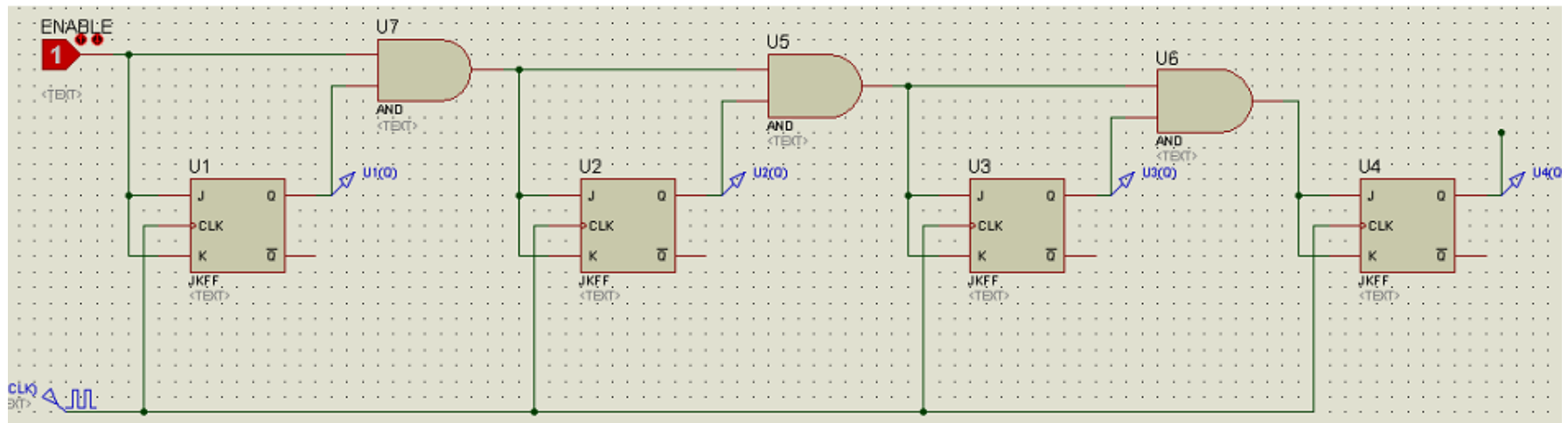




Bộ đếm (Counter)

80

- Mạch đếm đồng bộ nhị phân 4 bit (0 → 15)



Số xung vào	Ngõ ra sau khi có xung vào				Trị thập phân ra
----------------	-------------------------------	--	--	--	---------------------

81	Q3	Q2	Q1	Q0	
----	----	----	----	----	--

Xoá	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	10
11	1	0	1	1	11
12	1	1	0	0	12
13	1	1	0	1	13
14	1	1	1	0	14
15	1	1	1	1	15
16	0	0	0	0	0
17	0	0	0	1	1

Ck —

Q₀ 0

Q₁ 0

Q₂ 0

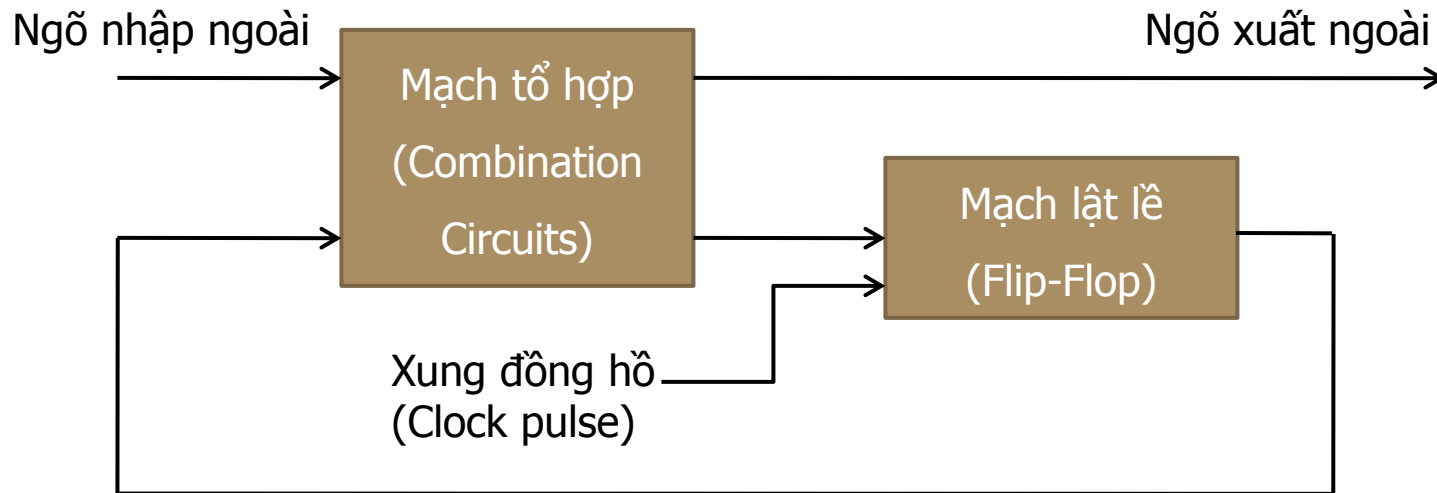
Q₃ 0

↑ 0

Xoá

Mạch tuần tự đồng bộ

82

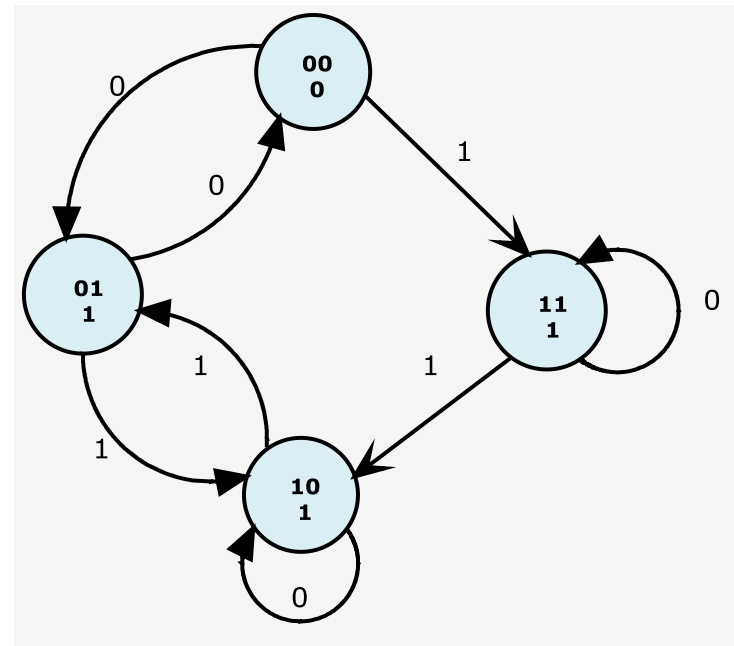
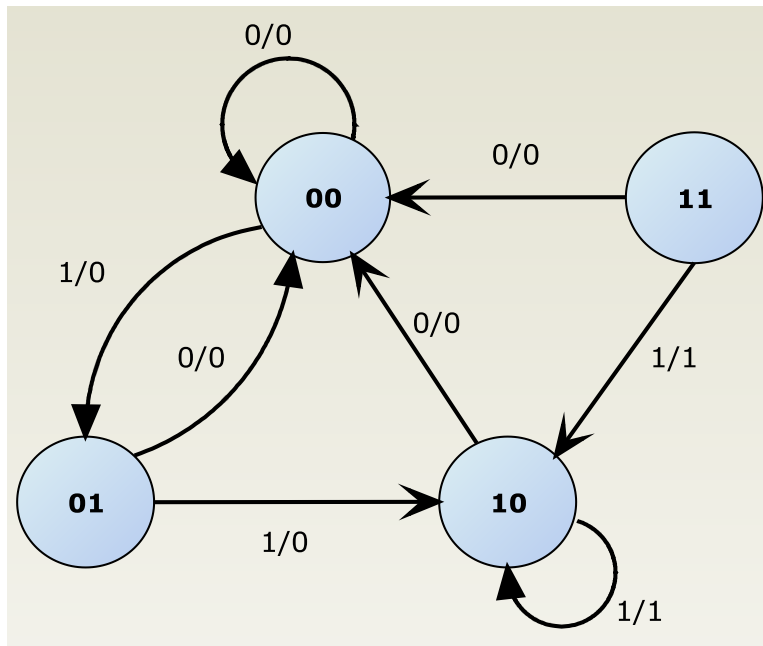


- Mạch tuần tự được xác định bởi:
 - Các ngõ nhập ngoài
 - Các ngõ xuất ngoài
 - Trạng thái nhị phân của mạch lật
- Trạng thái kế của mạch lật = $F(\text{Trạng thái hiện tại}, \text{Các ngõ nhập ngoài})$
- **Thiết kế mạch tuần tự → Xác định dạng mạch lật và các Input của chúng**

Thiết kế mạch tuần tự – Bước 1

83

- Đầu tiên phải **xác định dùng dạng mạch lật gì** (RS / JK / D / T)
- Lập **lược đồ các trạng thái** mạch lật dựa trên đặc tả mạch ban đầu
- Có 2 cách biểu diễn



(Bước 1 – tiếp tục)

84

- Thay vì dùng lược đồ trạng thái, ta cũng có thể lập **bảng trạng thái** mạch lật

Trạng thái hiện tại $Q(t)$			Giá trị ngõ nhập ngoài	Trạng thái kế $Q(t + 1)$		
Ngõ xuất mạch lật 1	...	Ngõ xuất mạch lật n	x	Ngõ xuất mạch lật 1	...	Ngõ xuất mạch lật n

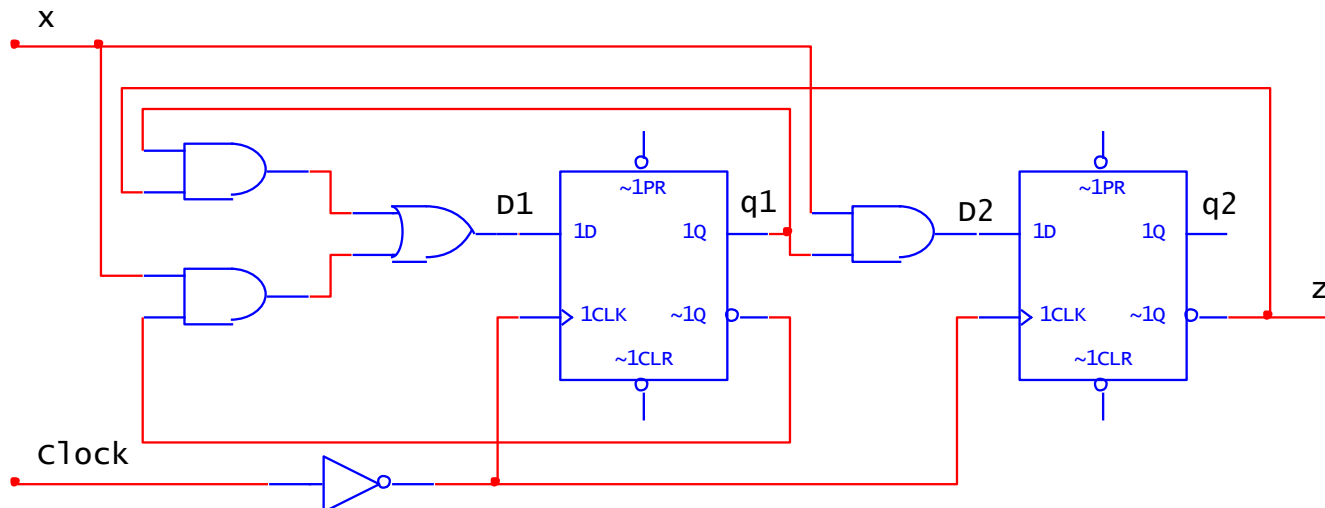
- Trạng thái kế của mạch lật: Dựa trên mô tả đề bài

Thiết kế mạch tuần tự – Bước 2

85

□ Lập **bảng kích thích**

- Nhiệm vụ là phải xác định được **làm thế nào để có được ngõ nhập vào mạch lật từ ngõ nhập ngoài x**
- Lưu ý ngõ nhập vào mạch lật **!=** ngõ nhập ngoài
 - Ví dụ: $x \neq D1, D2$



Bảng kích thích

86

Mạch lật RS / SR			
Q(t)	Q(t+1)	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

Mạch lật JK			
Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Mạch lật D		
Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

Mạch lật T		
Q(t)	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

Thiết kế mạch tuần tự – Bước 3

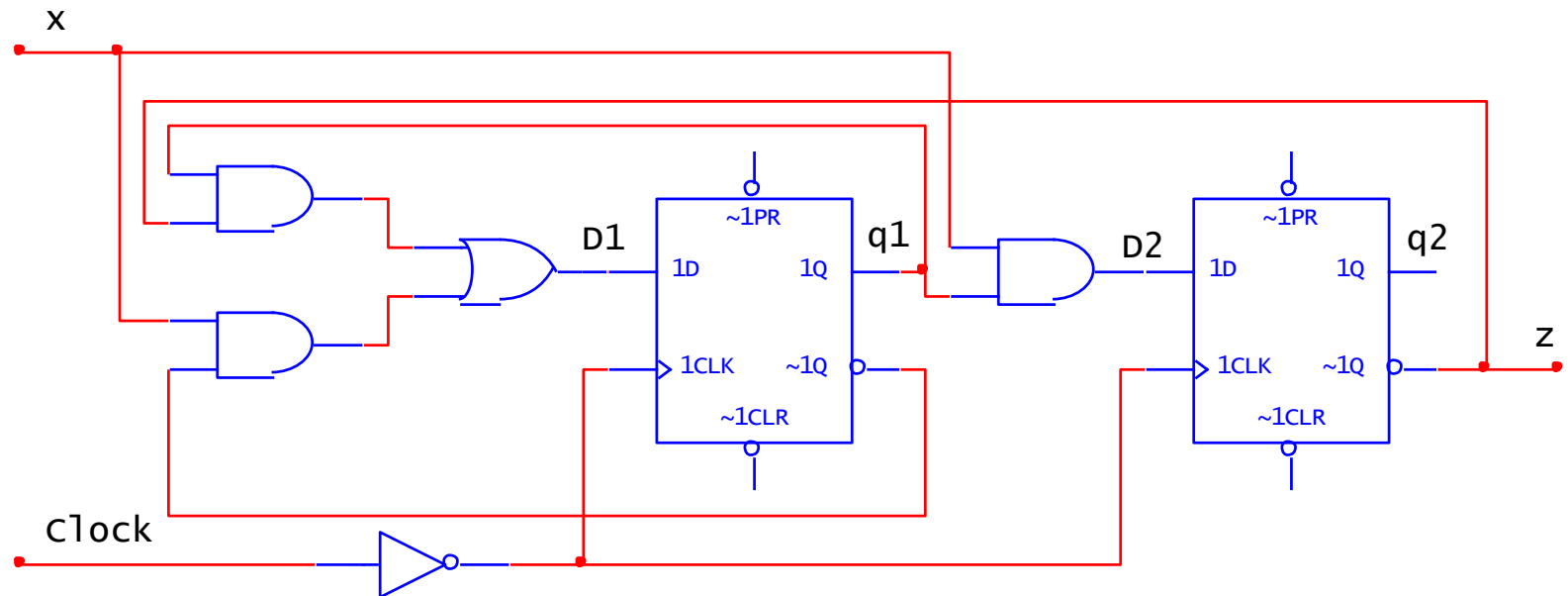
87

- Tìm phương trình đại số xác định ngõ nhập mạch lật từ bảng kích thích (Hàm ngược)
 - Có n mạch lật $\rightarrow n$ ngõ ra mạch lật $A_1 \dots A_n$
 - Suy ra phương trình ngõ nhập mạch lật có $n + 1$ biến bao gồm:
 - n biến $A_1 \dots A_n$
 - 1 biến x (ngõ nhập ngoài)
 - Dùng biểu đồ Karnaugh + bảng kích thích để xác định phương trình hàm ngõ nhập mạch lật

Thiết kế mạch tuần tự – Bước 4

88

- Vẽ sơ đồ mạch dựa trên phương trình hàm ngõ nhập



Bài tập minh hoạ

89

- Xem ví dụ minh hoạ tại giáo trình “Kiến trúc máy tính” – Thầy Nguyễn Minh Tuấn, trang 42-45

Một số bài tập thiết kế mạch

Bài 1 – Digital Clock v.1

90

- Thiết kế đồng hồ với mặt số thể hiện các số từ 0 đến 7 và 2 nút bấm A, B. Nếu bấm nút A, số thể hiện tăng lên 1. Nếu bấm nút B, số thể hiện giảm đi 1
- Cần: Adder, MUX

Một số bài tập thiết kế mạch

Bài 2 – Digital Clock v.2

91

- Thiết kế đồng hồ bấm giây với mặt số thể hiện các số từ 00 đến 63 và 2 nút bấm A, B. Bấm nút A để start / stop. Khi đồng hồ đang ở trạng thái stop, bấm nút B sẽ xoá về 0
- Cần: Counter, MUX

Một số bài tập thiết kế mạch

Bài 3 – Digital Clock v.3

92

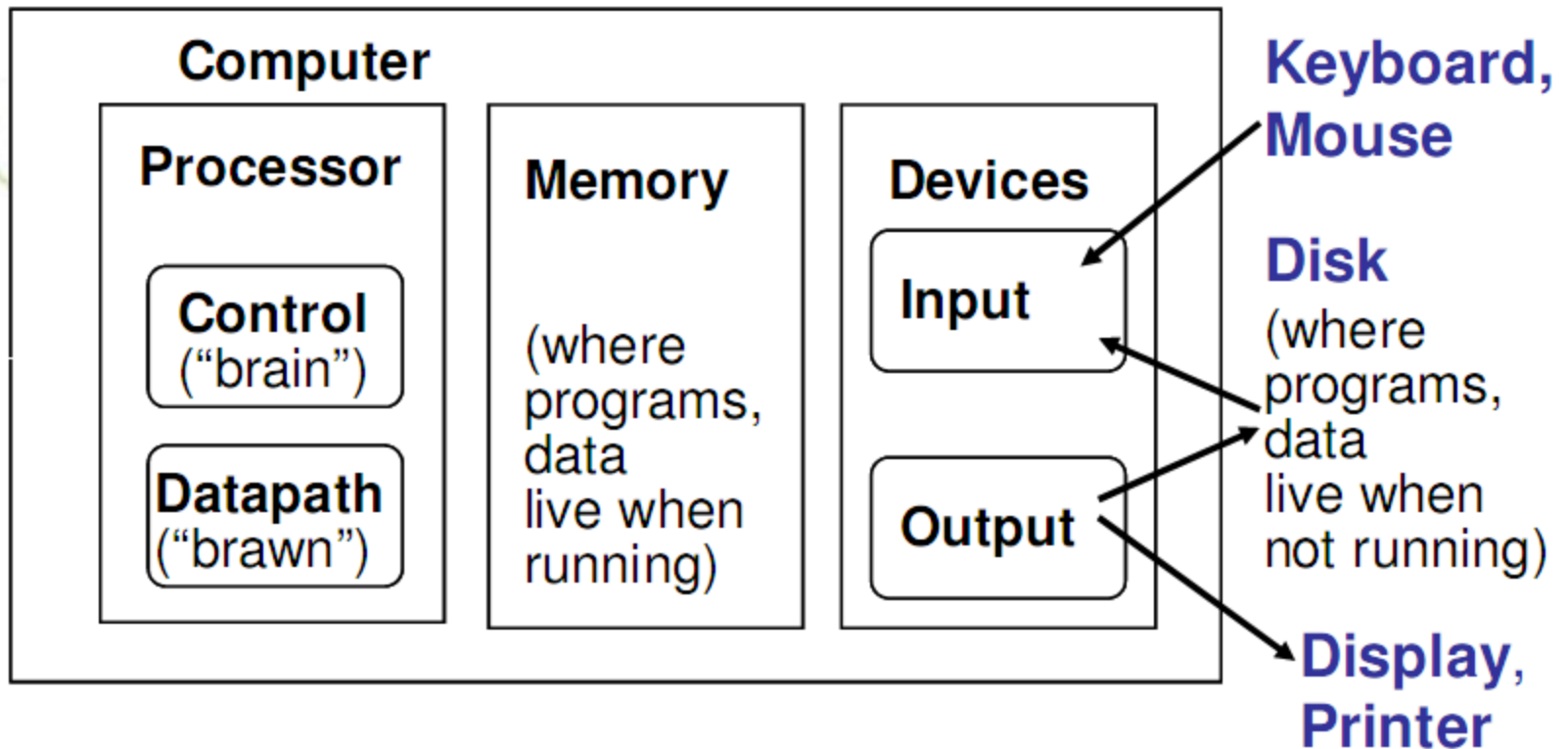
- Thiết kế đồng hồ bấm giây với mặt số thể hiện các số từ 00 đến 63 và 3 nút bấm A, B, C. Bấm nút A để start / stop. Khi đồng hồ đang ở trạng thái stop, bấm nút B sẽ tăng lên 1, bấm nút C sẽ giảm đi 1, bấm đồng thời B và C sẽ xoá về 00

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Vũ Minh Trí – vmtri@fit.hcmus.edu.vn

5 thành phần cơ bản của máy tính

2

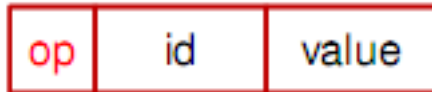


Bộ vi xử lý (CPU)

3

- Datapath
 - ▣ Registers
 - ▣ ALU
- Control unit
- **Stalling:** CPU = {Registers, ALU, Control unit, Internal bus}

AND R, value
OR R, value

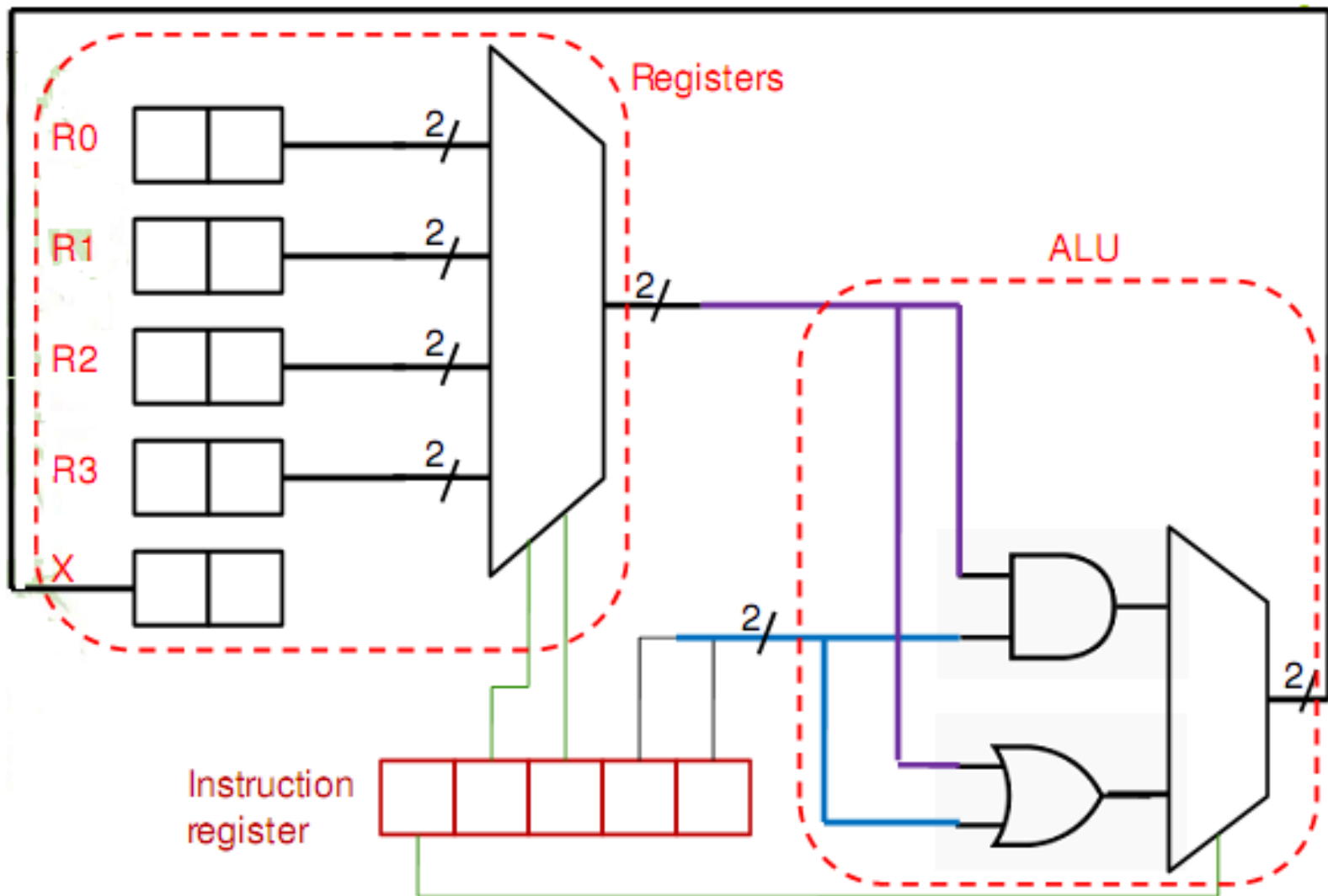


$$X = R_{id} \text{ op value}$$



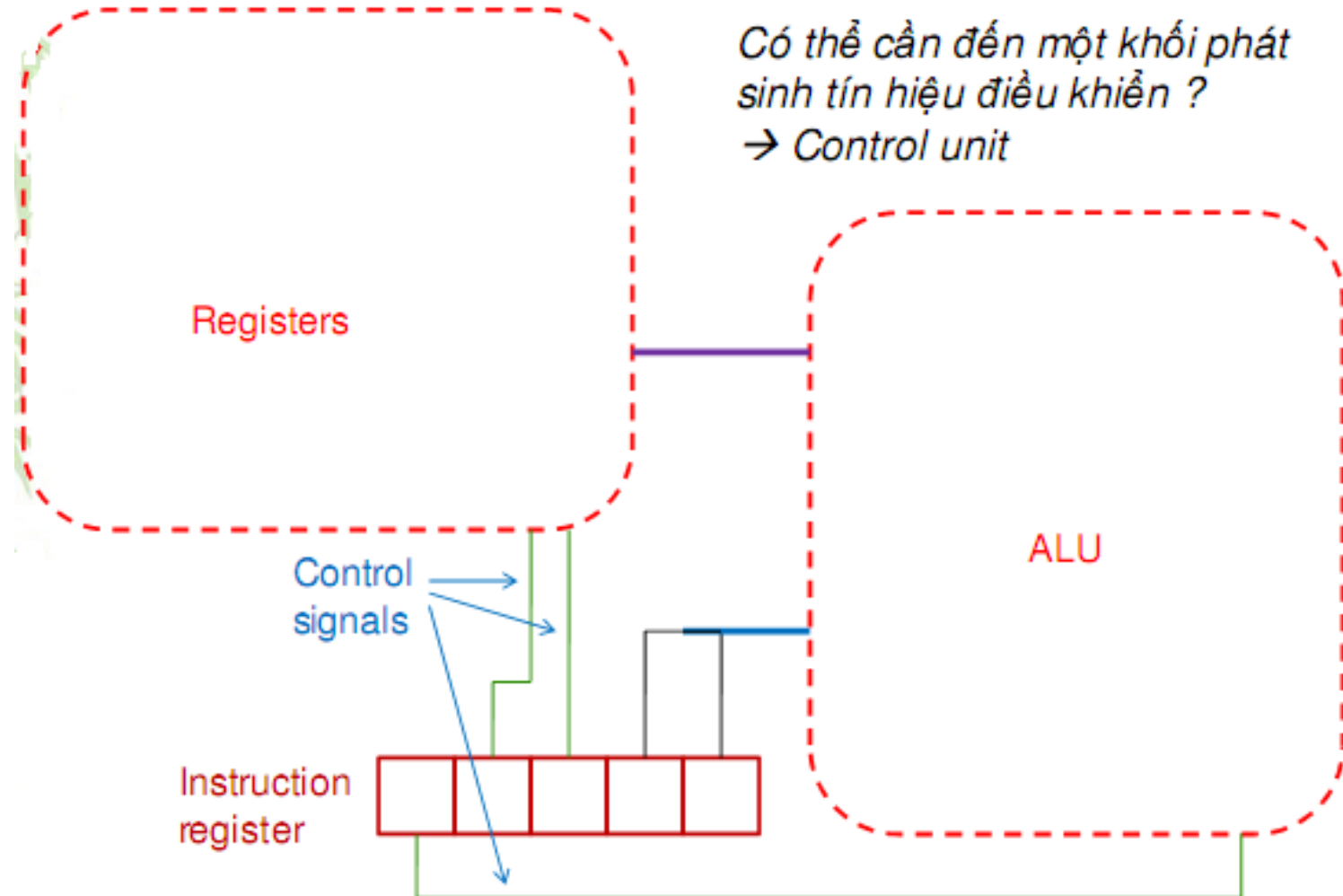
$$X = R1 \text{ OR } 2$$

4



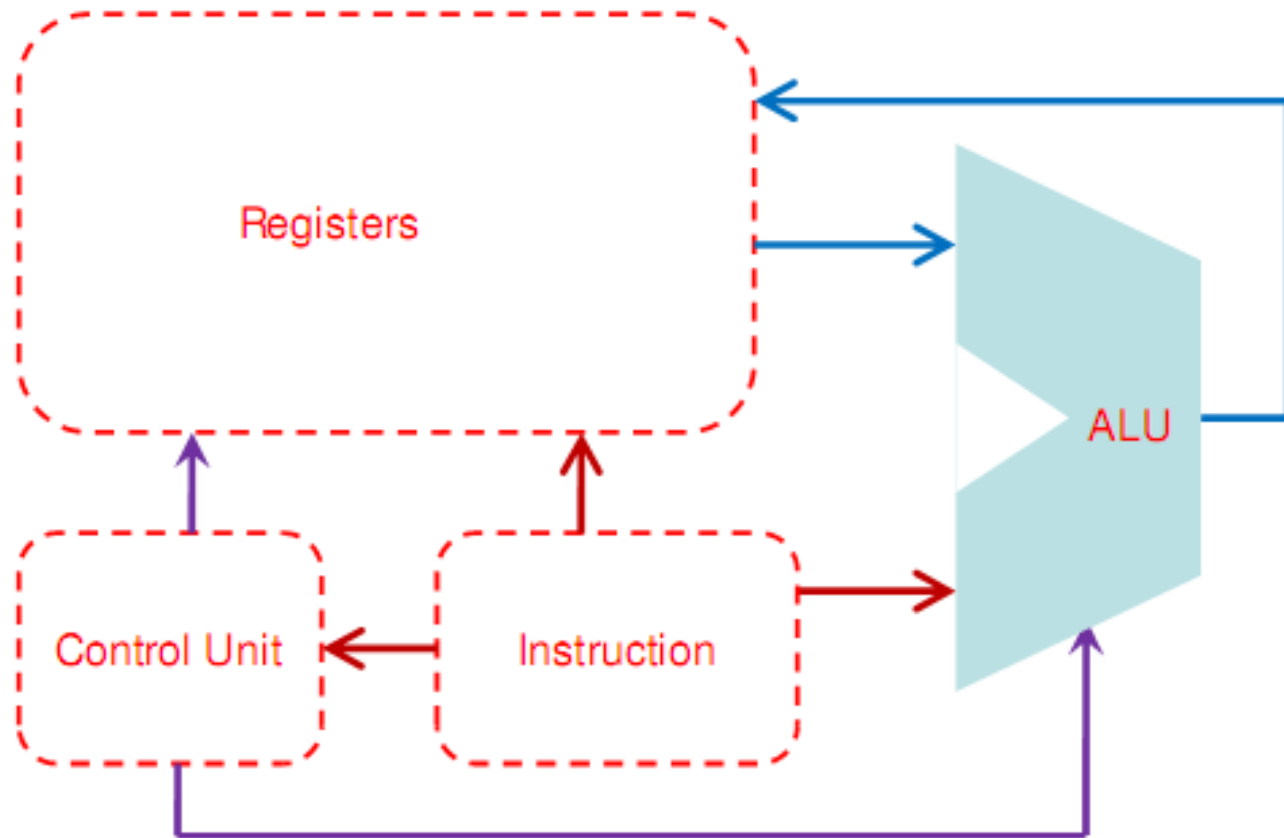
Control signals

5



Datapath & Control unit

6



MIPS thu gọn

7

- Lệnh truy xuất bộ nhớ: **lw, sw**
 - Lệnh số học – luận lý: **add, sub, and, or, slt**
 - Lệnh rẽ nhánh: **beq, j**
- Thiết kế bộ xử lý (Datapath và Control) cho tập lệnh MIPS thu gọn này ?

Một số lưu ý

8

- **Bất kỳ câu lệnh nào muốn thực thi cũng phải qua 2 bước đầu tiên:**
 - Gửi địa chỉ lệnh chứa trong thanh ghi PC (Program counter) đến bộ nhớ lệnh để lấy nội dung câu lệnh từ bộ nhớ
 - Xác định toán hạng trong câu lệnh → Đọc các thanh ghi chứa toán hạng có địa chỉ tương ứng
- **Các bước tiếp theo phụ thuộc vào từng nhóm lệnh khác nhau**
- Tập lệnh MIPS thu gọn có các bước thực thi giống nhau ở khá nhiều điểm, khác biệt chủ yếu nằm ở các bước thực thi cuối của câu lệnh

Instruction format

9

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

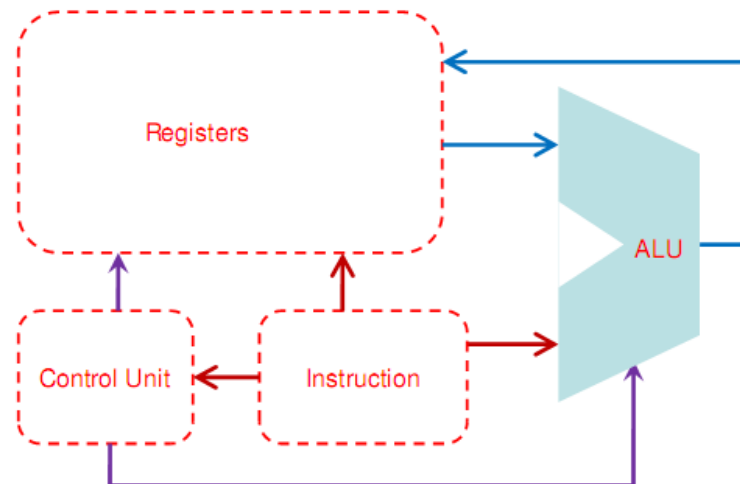
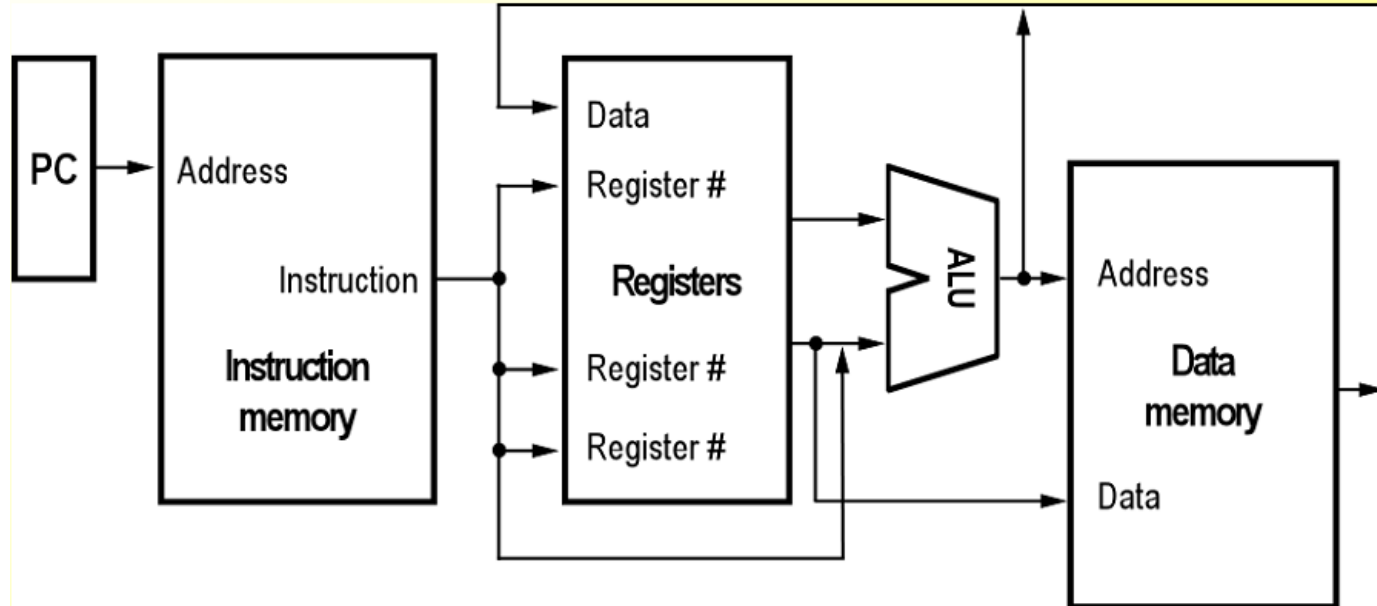
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

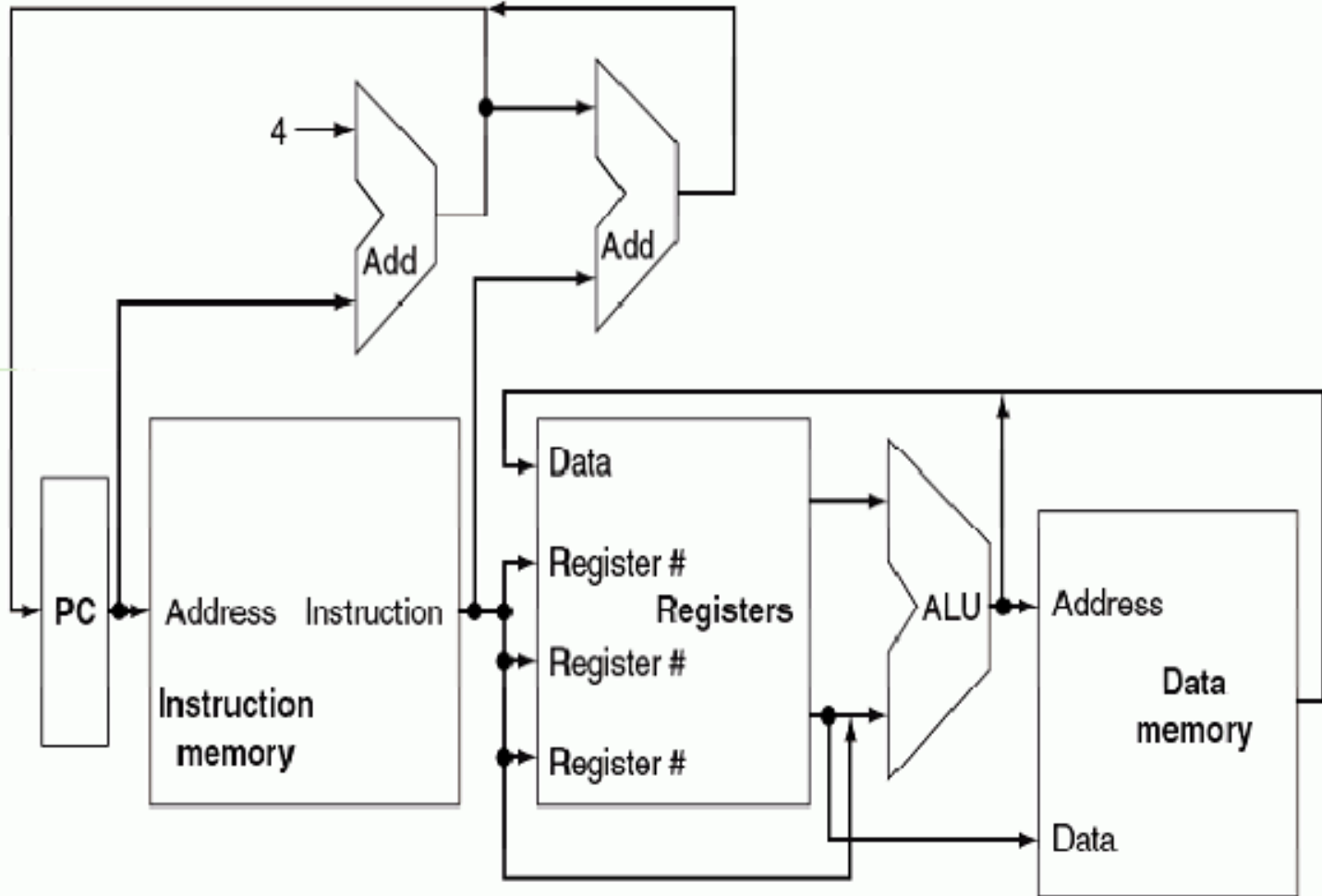
Sơ đồ thực thi tổng quát

10



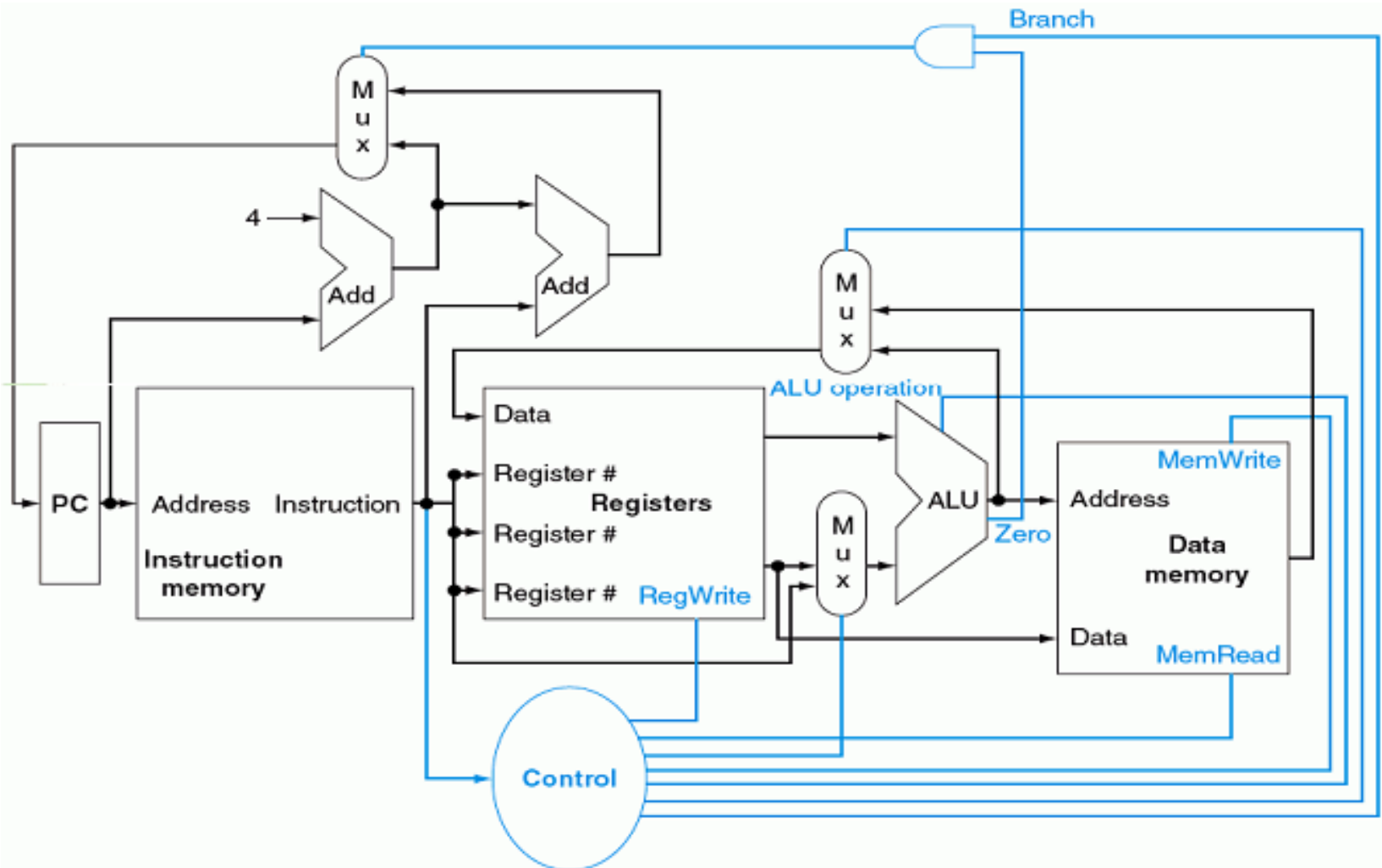
Dịch chuyển lệnh tiếp theo...

11



Sử dụng MUX để điều khiển

12



Xây dựng đường đi dữ liệu (Datapath)

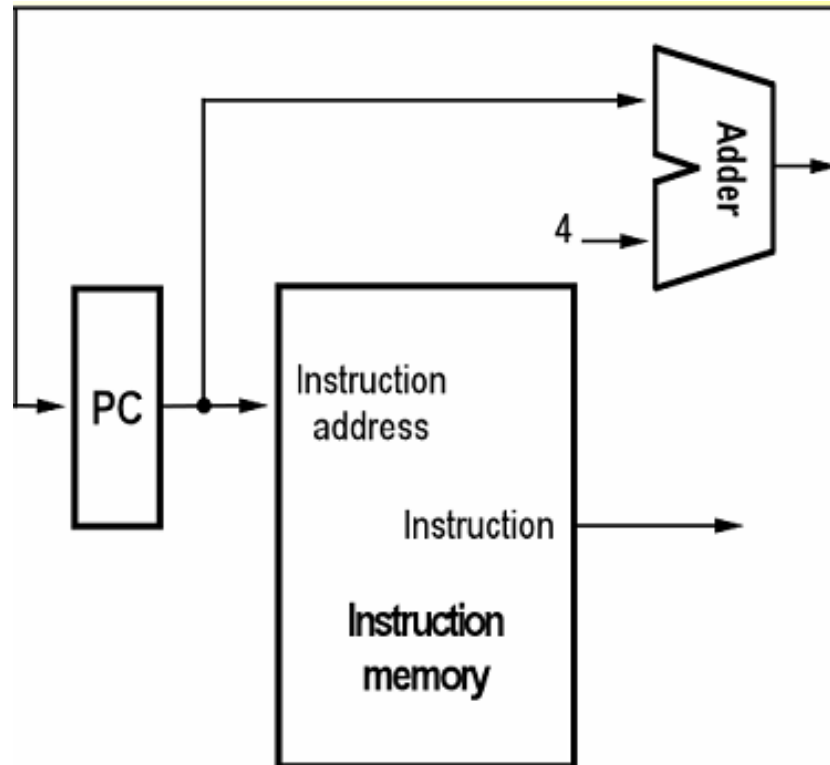
13

- Phương thức xây dựng Datapath:
 - ▣ Xác định **kiến trúc của các phần tử cần thiết** cho câu lệnh
 - ▣ Xây dựng dần các **phân khúc** cho Datapath ứng với **từng công đoạn** trong thực thi câu lệnh
 - ▣ Tiến đến xây dựng **hoàn chỉnh** Datapath cho câu lệnh

Kiến trúc các phần tử cần thiết

14

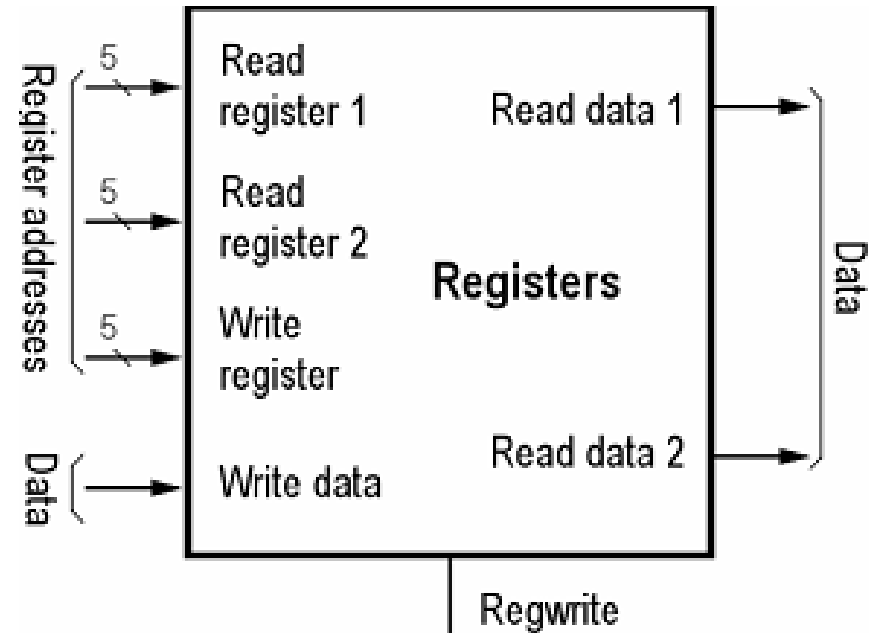
- Dịch chuyển lệnh:



Kiến trúc các phần tử cần thiết

15

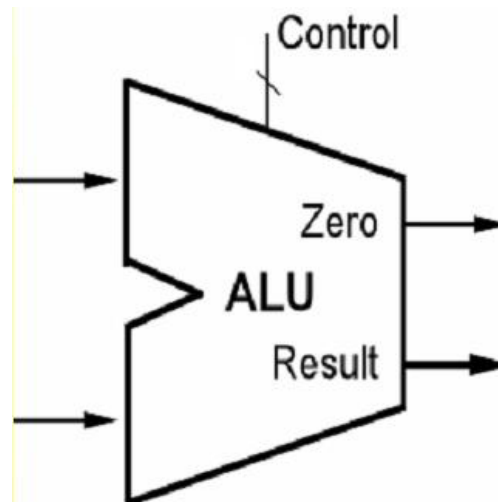
- Tập thanh ghi (register files)
 - 3 ngõ nhận địa chỉ thanh ghi
 - 1 ngõ ghi dữ liệu
 - 2 ngõ đọc dữ liệu (output)
 - 1 tín hiệu điều khiển ghi



Kiến trúc các phần tử cần thiết

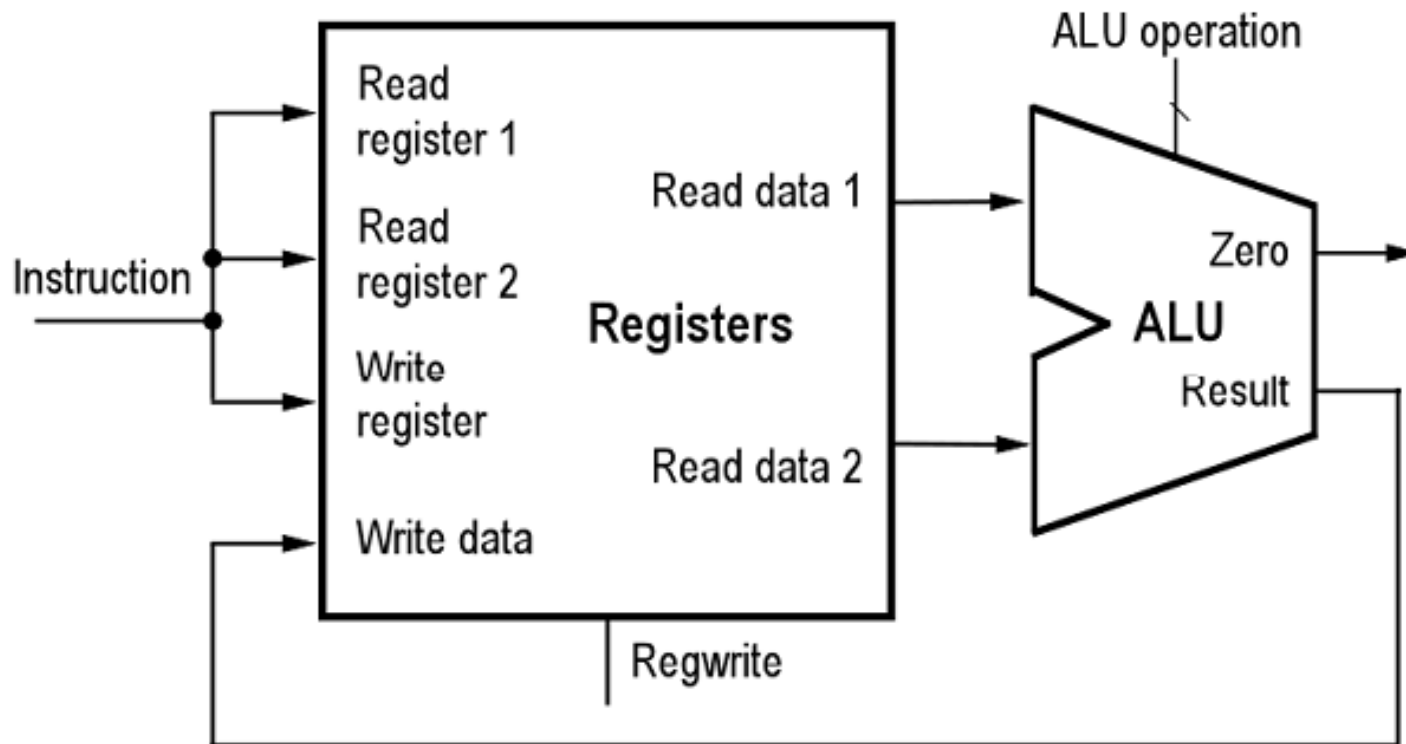
16

- Đơn vị số học – luận lý (ALU – Arithmetic Logic Unit)
 - 2 ngõ vào toán hạng (32-bit)
 - 1 ngõ ra kết quả (32 bit) và 1 bit zero (để chứa kết quả so sánh bằng)
 - 1 tín hiệu điều khiển (4 bit)



Register + ALU

17



Datapath cho I,J-format (lw, sw, beq, j) ?

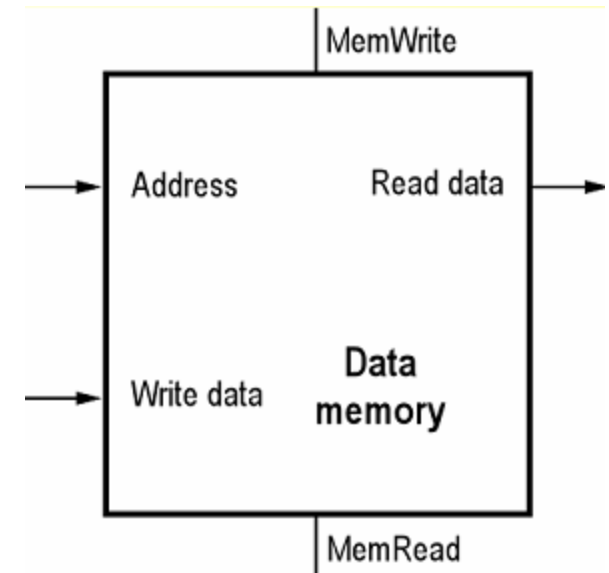
18

- Cần thêm 2 thành phần cơ bản:
 - ▣ Bộ nhớ dữ liệu (Data memory unit)
 - ▣ Bộ mở rộng dấu (Sign extended unit)

Datapath cho I,J-format (lw, sw, beq, j)

19

- Bộ nhớ dữ liệu (Data memory unit)
 - 1 ngõ nhận địa chỉ ô nhớ
 - 1 ngõ nhận dữ liệu cần ghi
 - 1 ngõ dữ liệu đọc (output)
 - 2 tín hiệu điều khiển đọc / ghi



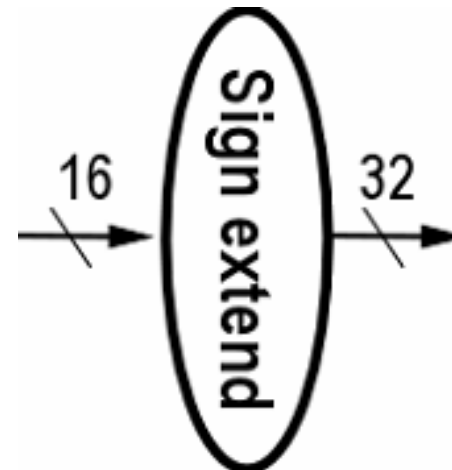
Datapath cho I-format (lw, sw, beq)

20

- Bộ mở rộng dấu (Sign extended unit)

- 1 ngõ nhập dữ liệu 16-bit

- 1 ngõ ra dữ liệu 32-bit

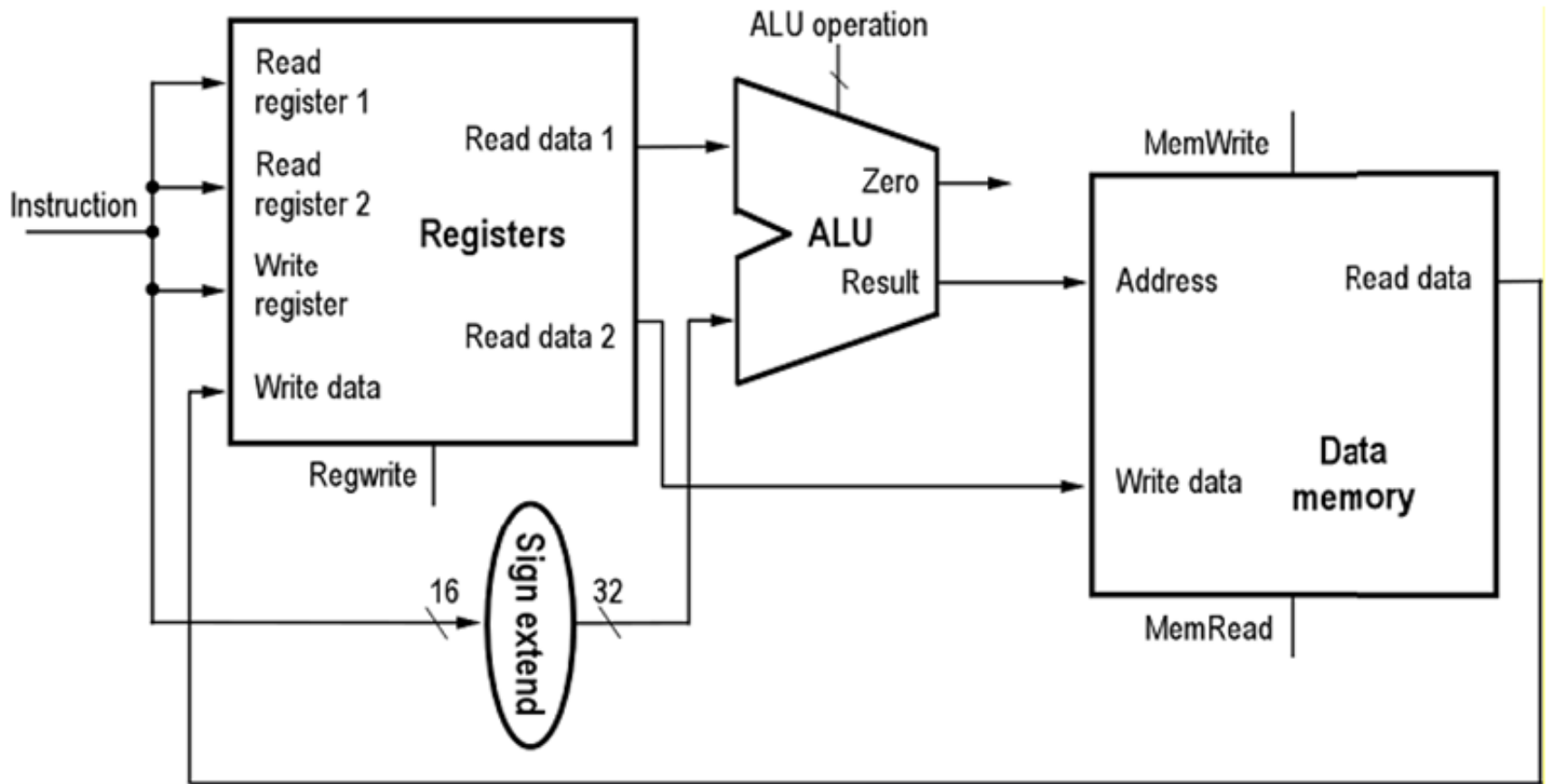


- lw \$s1, 4(\$s0) → 4: 16 bit → 04: 32 bit (sign-extended)

- beq \$s0, \$s1, target_label → target-label: 16 bit → target-label: 32 bit (sign-extended)

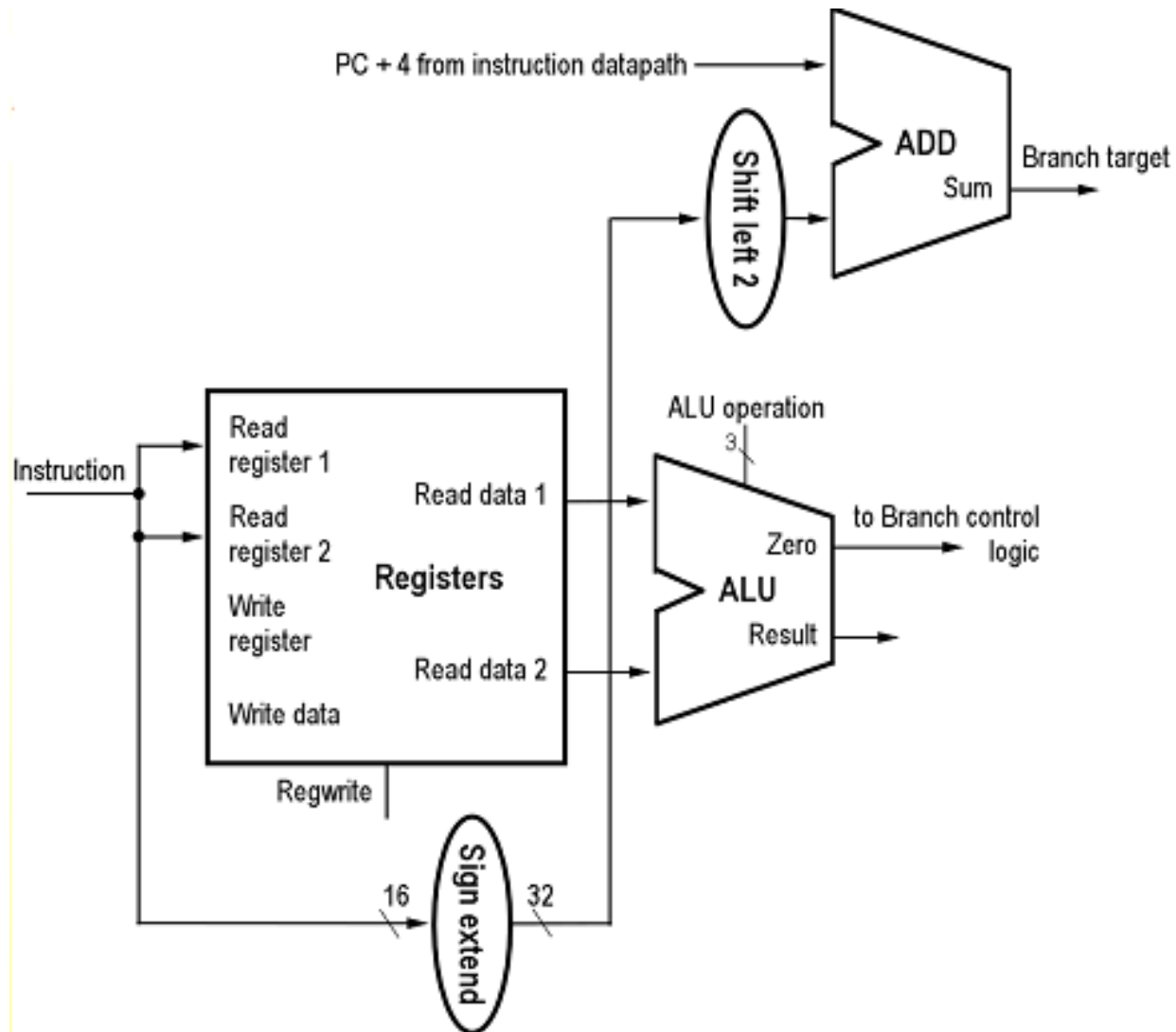
Datapath cho lệnh bộ nhớ (lw,sw)

21



Datapath cho lệnh rẽ nhánh (beq,j)

22



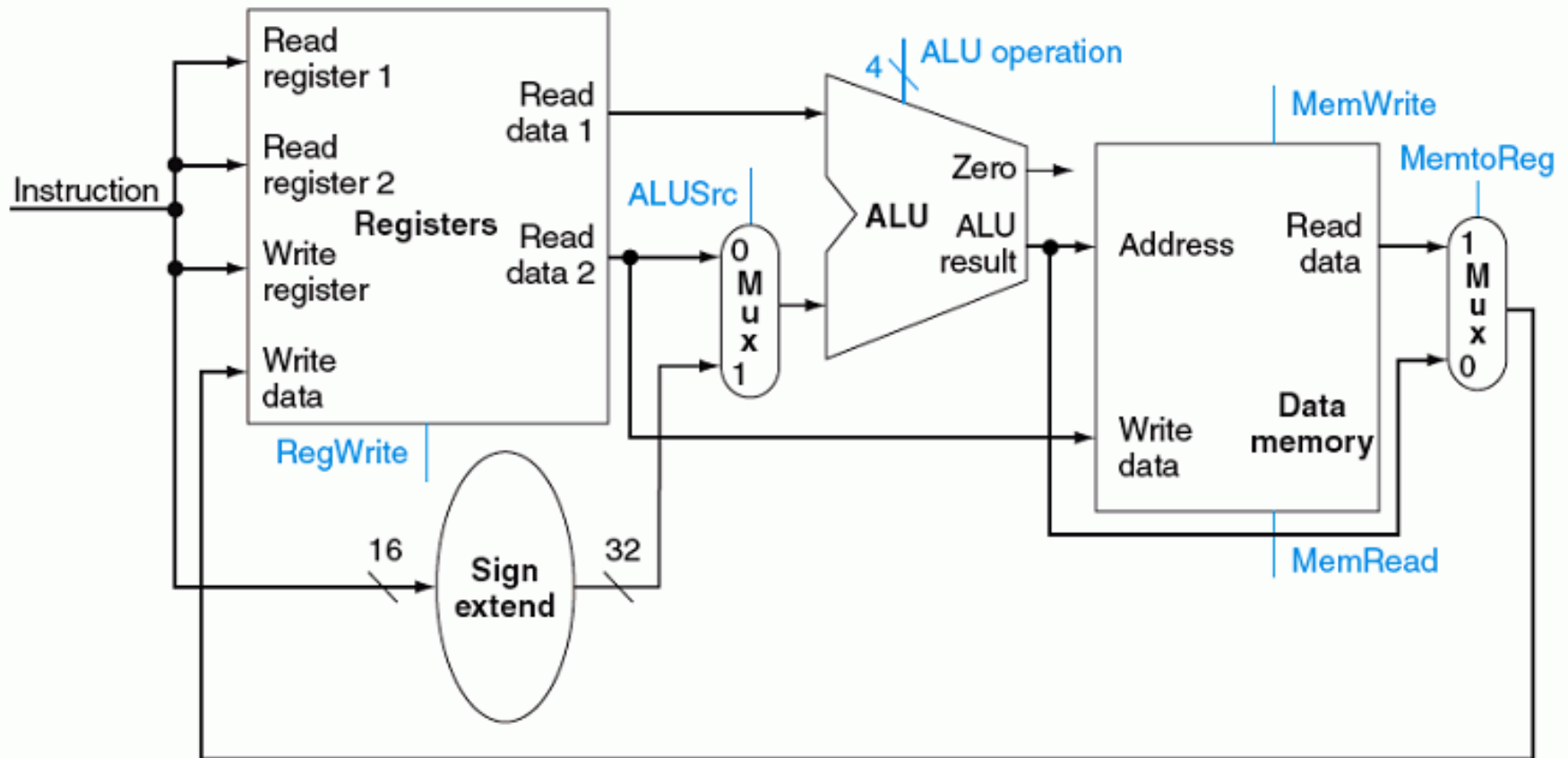
Datapath cho R-format ?

23

- Làm sao xây dựng Datapath cho R-format “xài chung” Datapath của I và J-format?
- Cần những bộ MUX đóng vai trò data selector để chia sẻ và lựa chọn những phần tử kiến trúc giữa những nhóm lệnh khác nhau
- Lưu ý: Hiện tại chúng ta chỉ xét CPU theo kiến trúc đơn chu kỳ (single cycle) – Mọi câu lệnh chỉ thực thi trong 1 chu kỳ clock

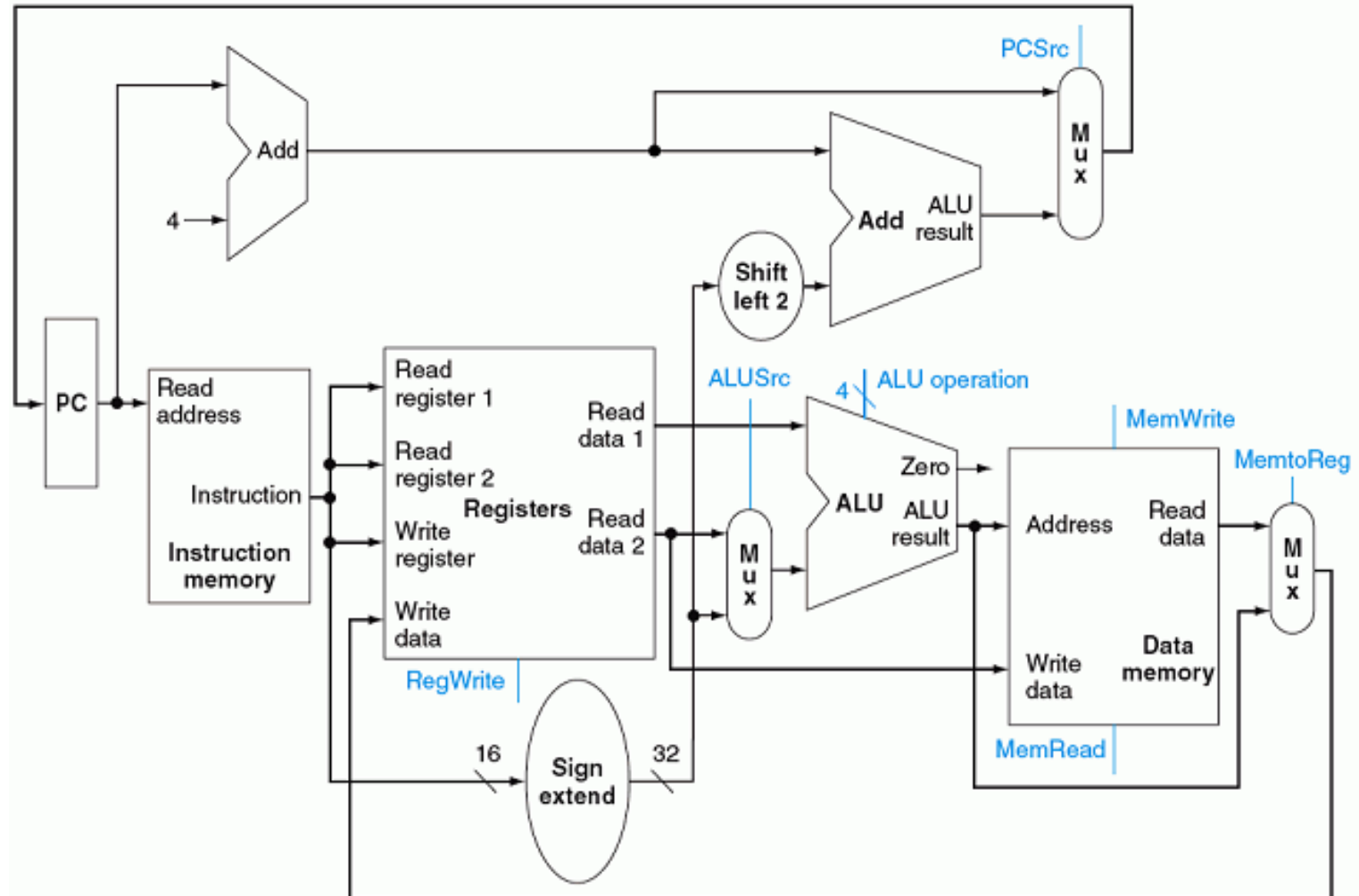
Datapath cho R-format

24



Datapath cho I,J,R-format

25



Tín hiệu điều khiển

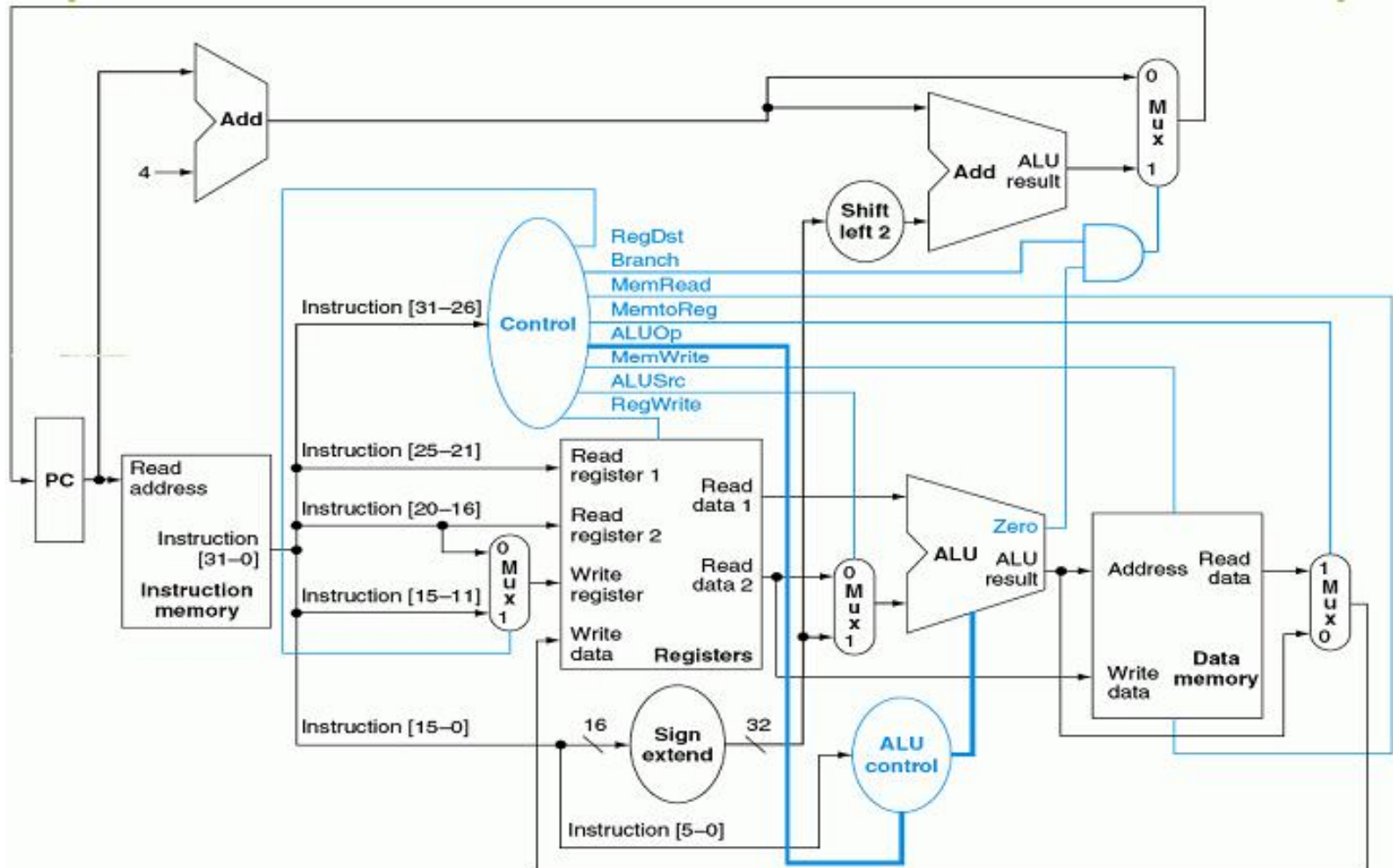
26

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Control unit ?

27

- ALU cần tín hiệu điều khiển hoạt động từ ALU Control



ALU Control Unit

28

- Các tín hiệu điều khiển ALU (4 bit):

ALU control Input	Function
0000	and
0001	or
0010	add
0110	sub
0111	slt
1100	nor

ALU Control Unit

29

Instruction (Control Unit → ALU Control)			ALU control input (to ALU)
Operation	ALU Opcode	Function	
lw	00	xx xx xx	0010 (add)
sw	00	xx xx xx	0010 (add)
beq	01	xx xx xx	0110 (subtract)
add (R-type)	10	10 00 00	0010 (add)
subtract (R-type)	10	10 00 10	0110 (subtract)
and (R-type)	10	10 01 00	0000 (and)
or (R-type)	10	10 01 01	0001 (or)
slt (R-type)	10	10 10 10	0111 (slt)

Bảng chân trị 4-bit ALU Control

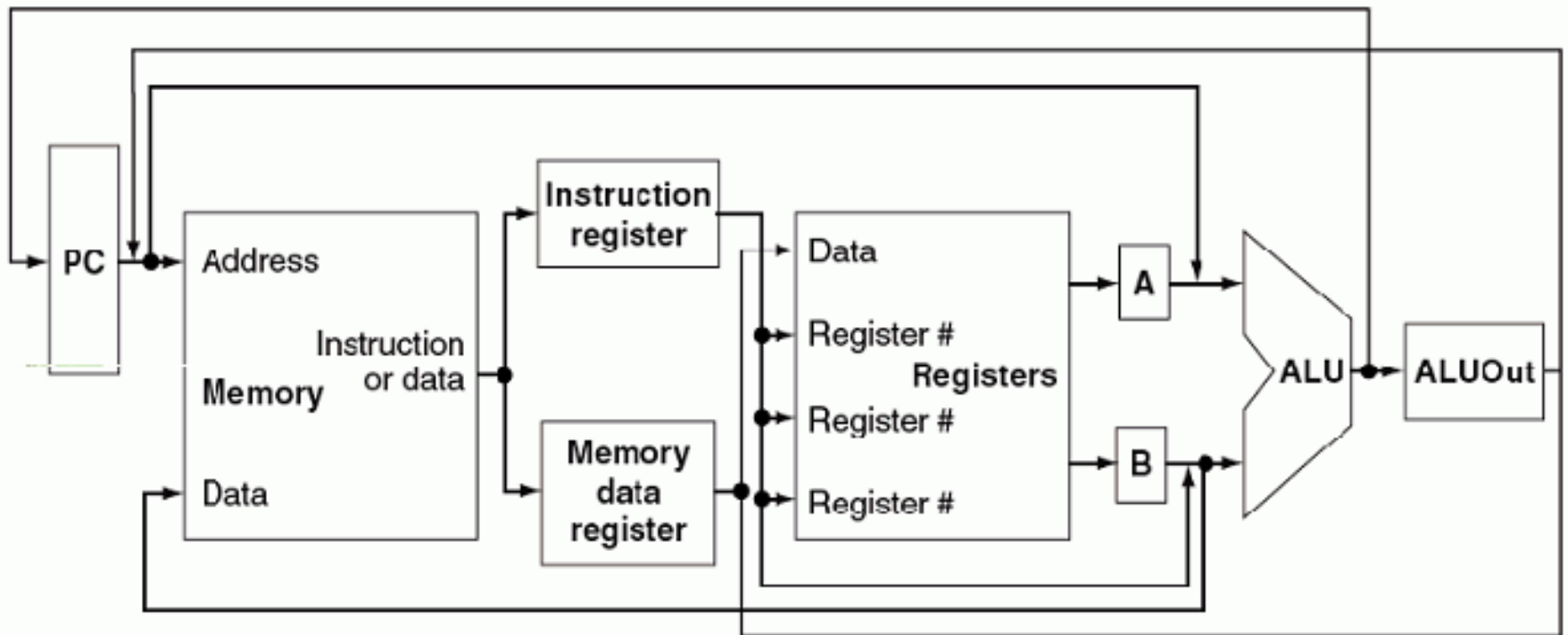
30

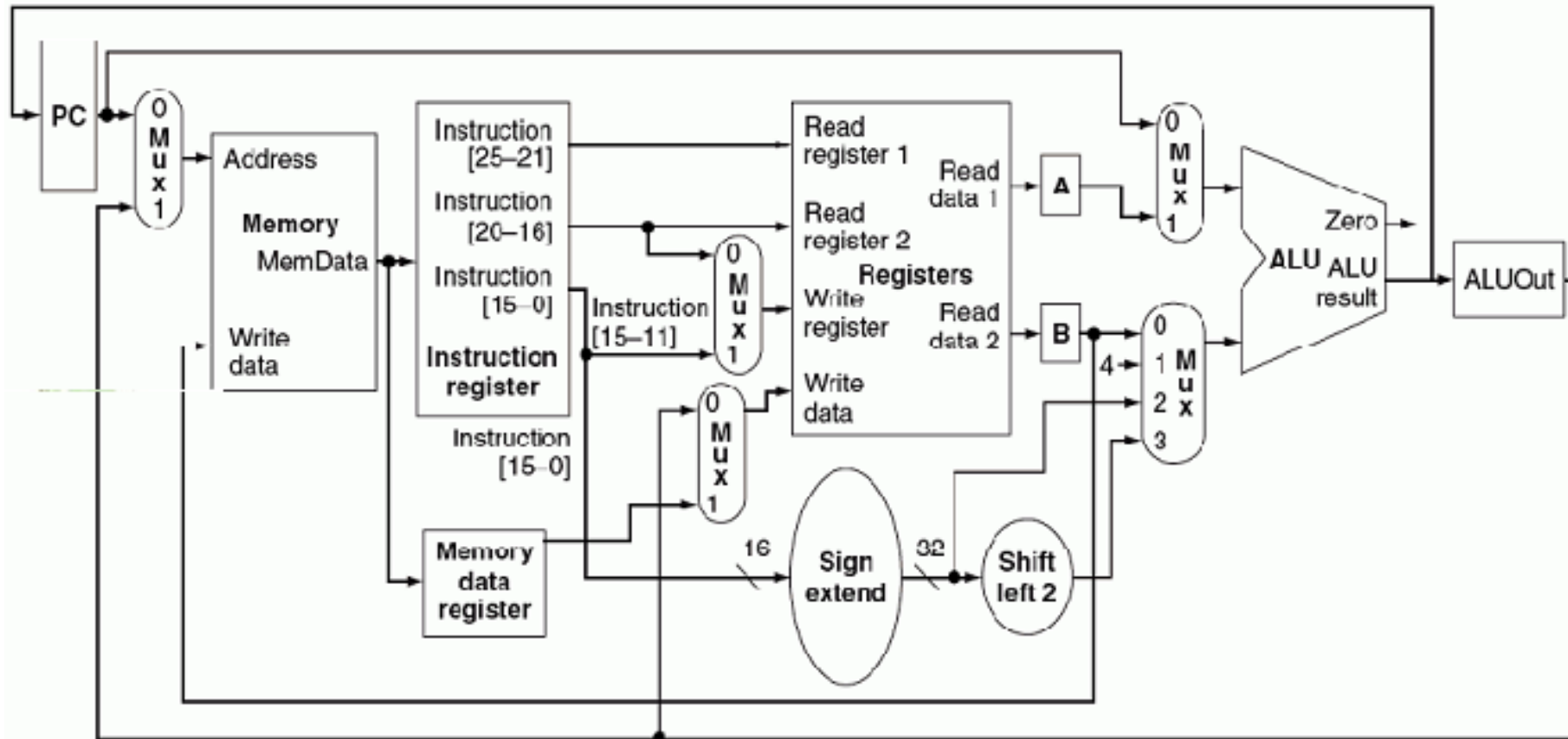
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

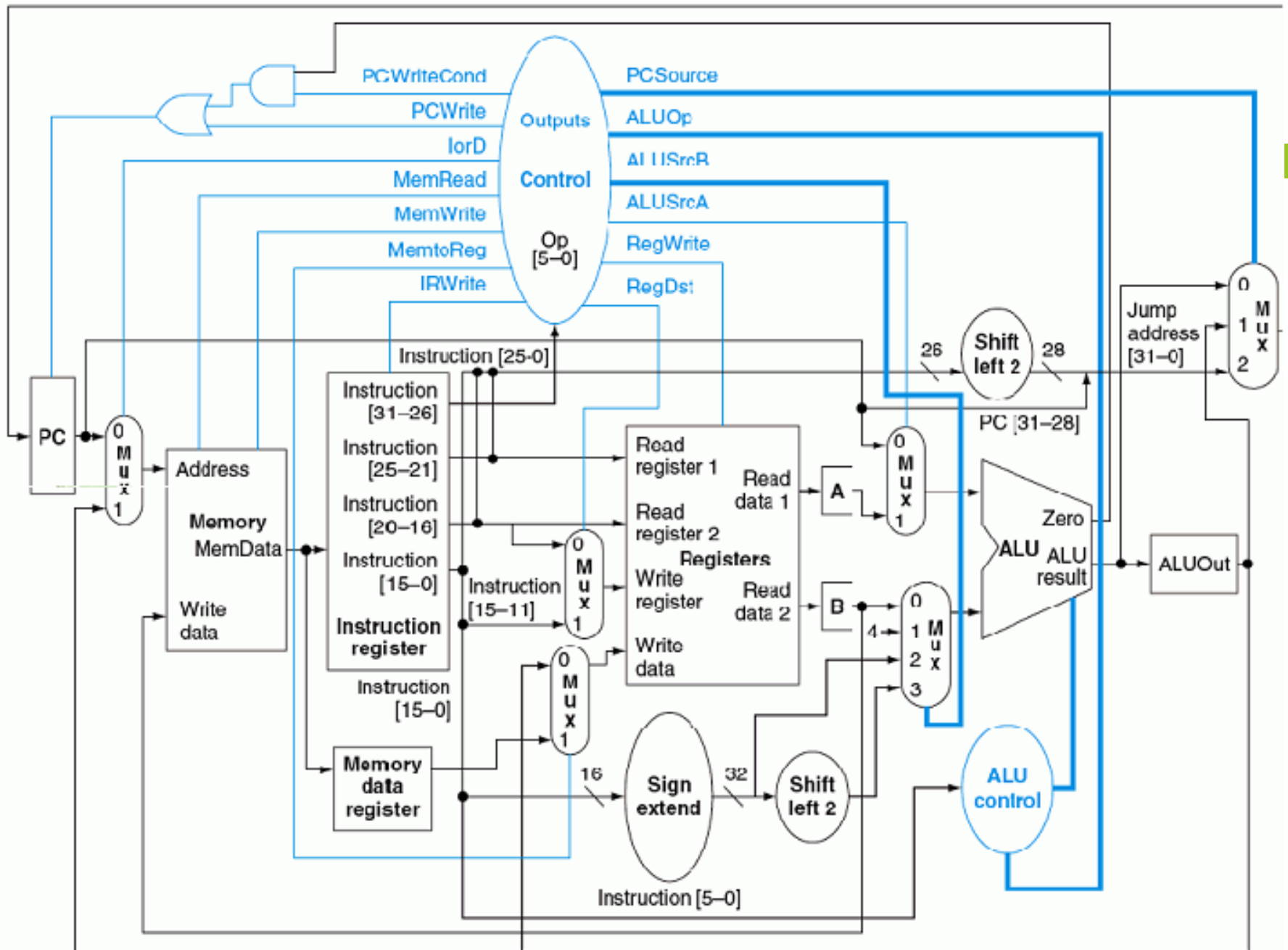
CPU đa chu kỳ (multiple-cycle)

31

- Trong thực tế không sử dụng CPU single-cycle vì các lý do:
 - Thời gian thực hiện các câu lệnh luôn khác nhau → Phải chọn chu kỳ hoạt động của CPU bằng với chu kỳ thực thi câu lệnh dài nhất !
 - Khả năng trùng lặp các phần tử chức năng cao
- Ở CPU đa chu kỳ (multiple-cycle), quá trình thực thi 1 câu lệnh diễn ra thành nhiều chu kỳ clock
- Một số khác biệt so với single-cycle:
 - Tinh chỉnh thời gian thực thi từng câu lệnh theo gián đồ trạng thái
 - Có thể sử dụng 1 bộ nhớ chung cho cả câu lệnh lẫn dữ liệu
 - Thêm vào 1 số thanh ghi để chứa dữ liệu/kết quả trung gian

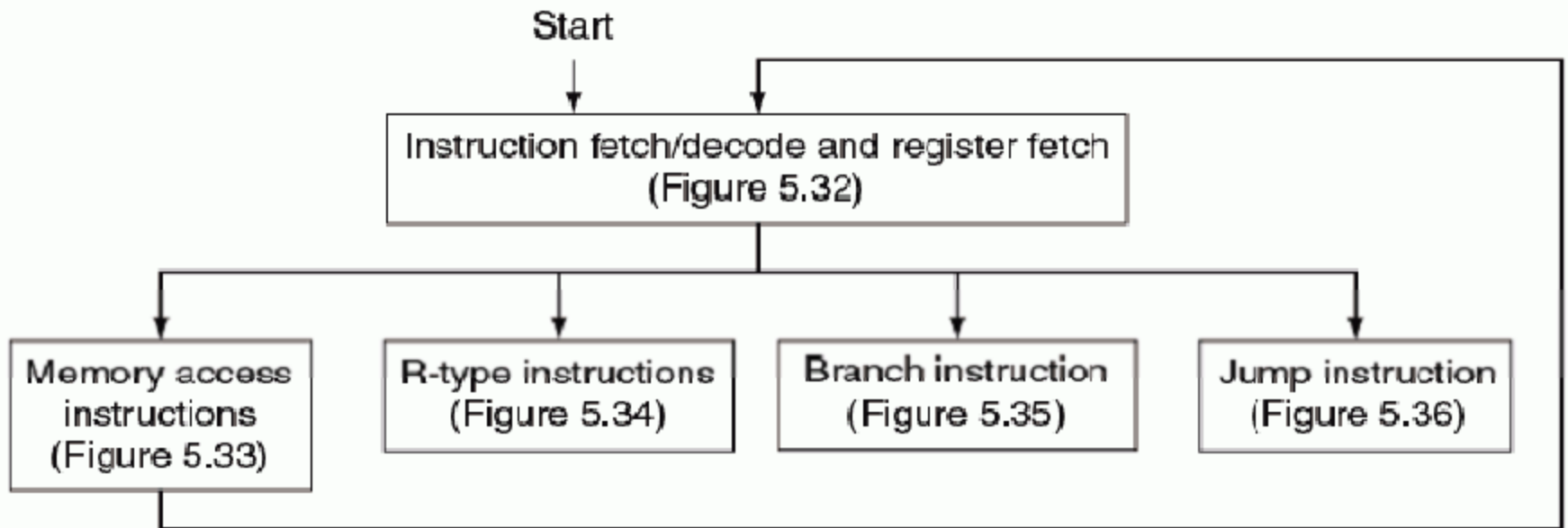


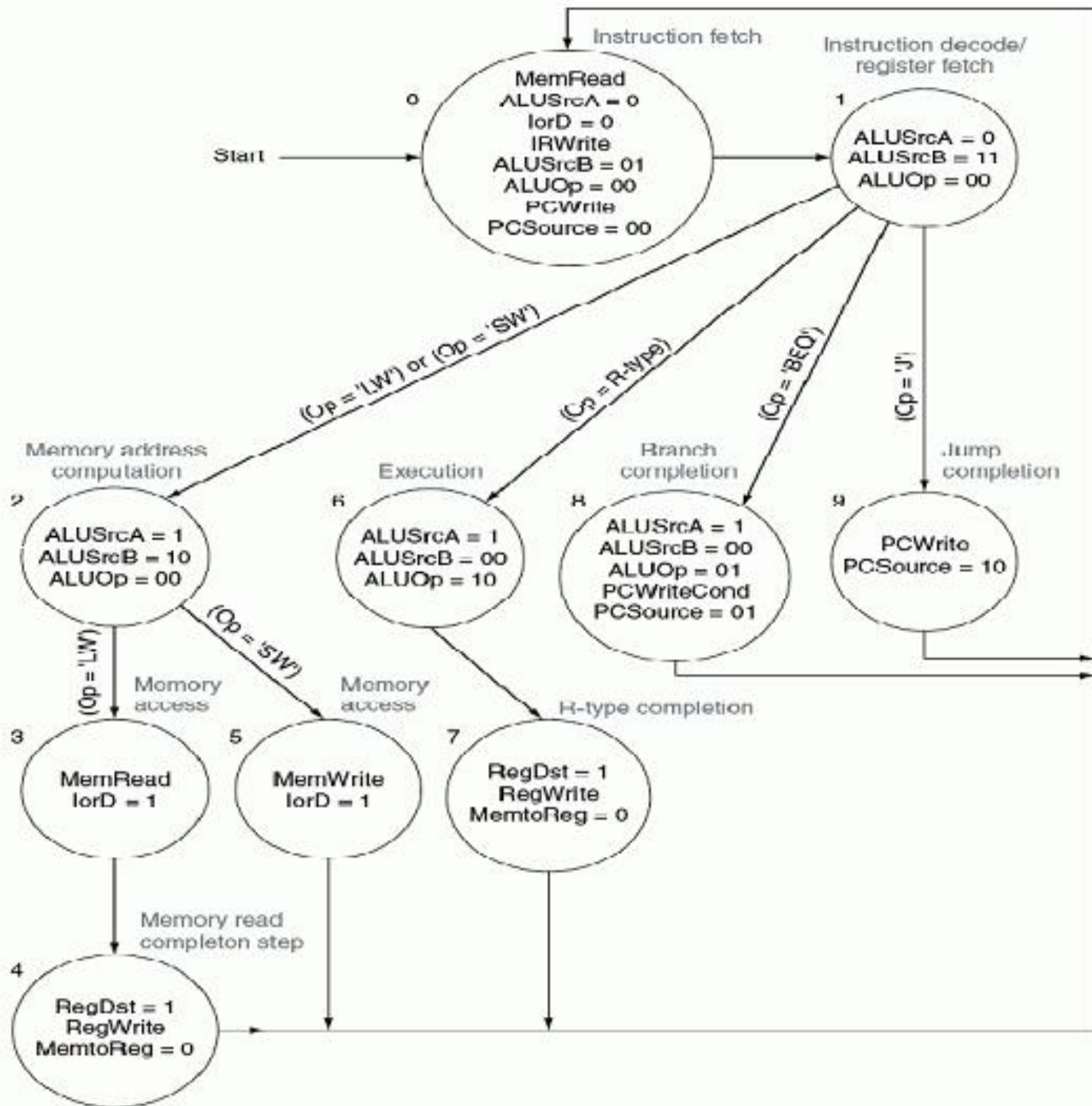




Quy trình thực thi lệnh

35





Homework

37

- Sách Petterson & Hennessy: Đọc chương 5

KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Võ Minh Trí – vmtri@fit.hcmus.edu.vn

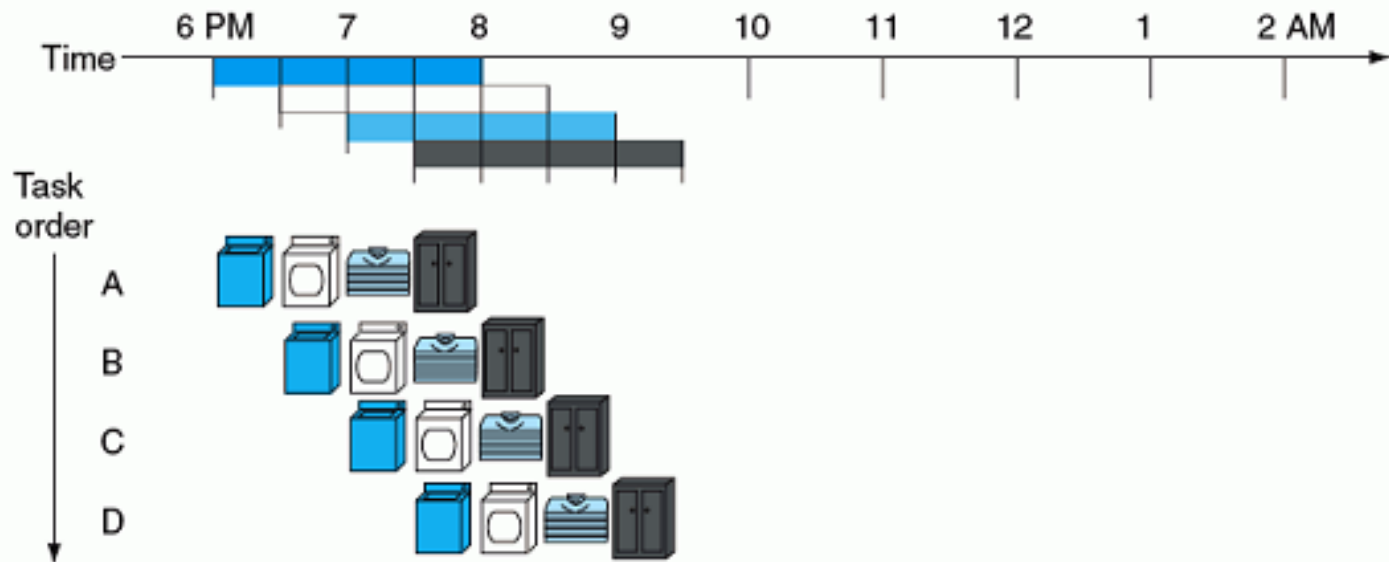
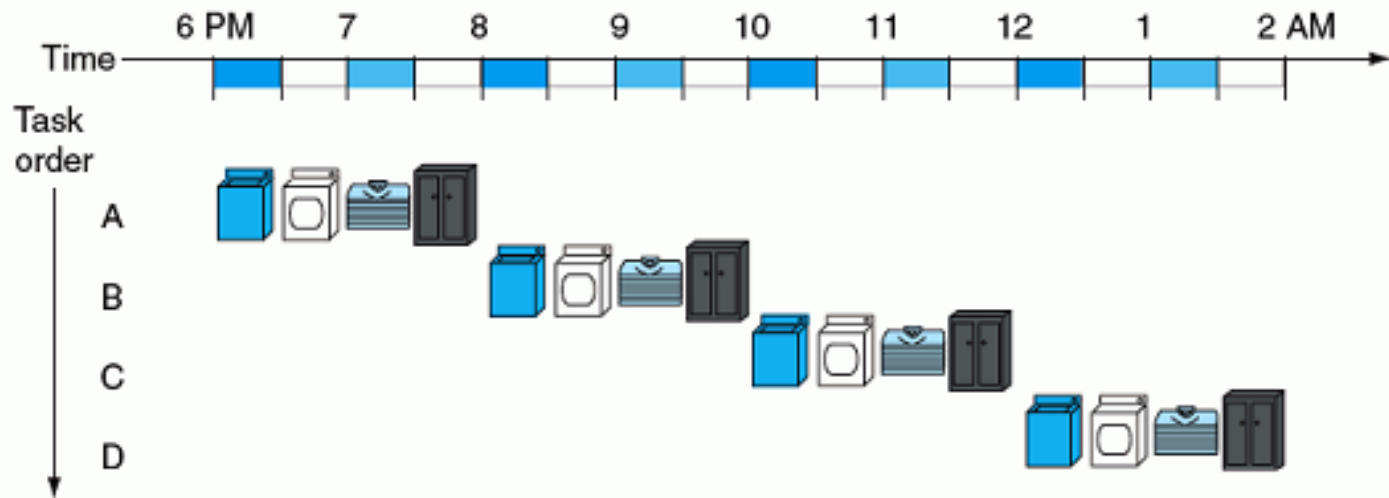
Cải thiện tốc độ?

2

- Có 2 cách tiếp cận phổ biến:
 - **Latency:** Thời gian hoàn thành 1 công việc nhất định
 - Ví dụ: Thời gian để đọc 1 sector từ đĩa gọi là disk access time hoặc disk latency
 - **Throughput:** Số lượng công việc có thể hoàn thành trong 1 khoảng thời gian nhất định

Giải pháp giặt ủi

3



Pipeline

4

- Pipeline không phải là giải pháp giúp tăng tốc theo kiểu Latency, mà là **Throughput trên toàn bộ công việc được giao**
 - Trên cùng 1 lượng tài nguyên không đổi, các công việc sẽ được tiến hành song song thay vì tuần tự, mỗi công việc chạy trong 1 pipeline (đường ống)
- *Pipelining là một kỹ thuật thực hiện lệnh trong đó các lệnh thực hiện theo kiểu "gõ đầu" nhau (overlap) nhằm tận dụng những khoảng thời gian rỗi giữa các công đoạn, qua đó làm tăng tốc độ xử lý lệnh*

Pipeline

5

- Khả năng tăng tốc phụ thuộc vào số lượng đường ống (pipeline) sử dụng
- Thời gian để cho chảy đầy (fill) đường ống và Thời gian để làm khô (drain) sẽ làm giảm khả năng tăng tốc
 - Ví dụ giặt ủi trên nếu không tính thời gian fill và drain thì tăng tốc 4 lần, còn nếu tính thì chỉ tăng tốc được 2.3 lần

Pipeline

6

- Giả sử một máy giặt giặt mất 20 phút, gấp đồ mất 20 phút. Vậy khi dùng giải pháp pipeline sẽ nhanh hơn bình thường bao nhiêu?
- Tổng thời gian cho giải pháp pipeline sẽ bị giới hạn bởi thời gian thực thi của đường ống chậm nhất
- Độ dài không cân bằng giữa các đường ống sẽ làm giảm khả năng tăng tốc

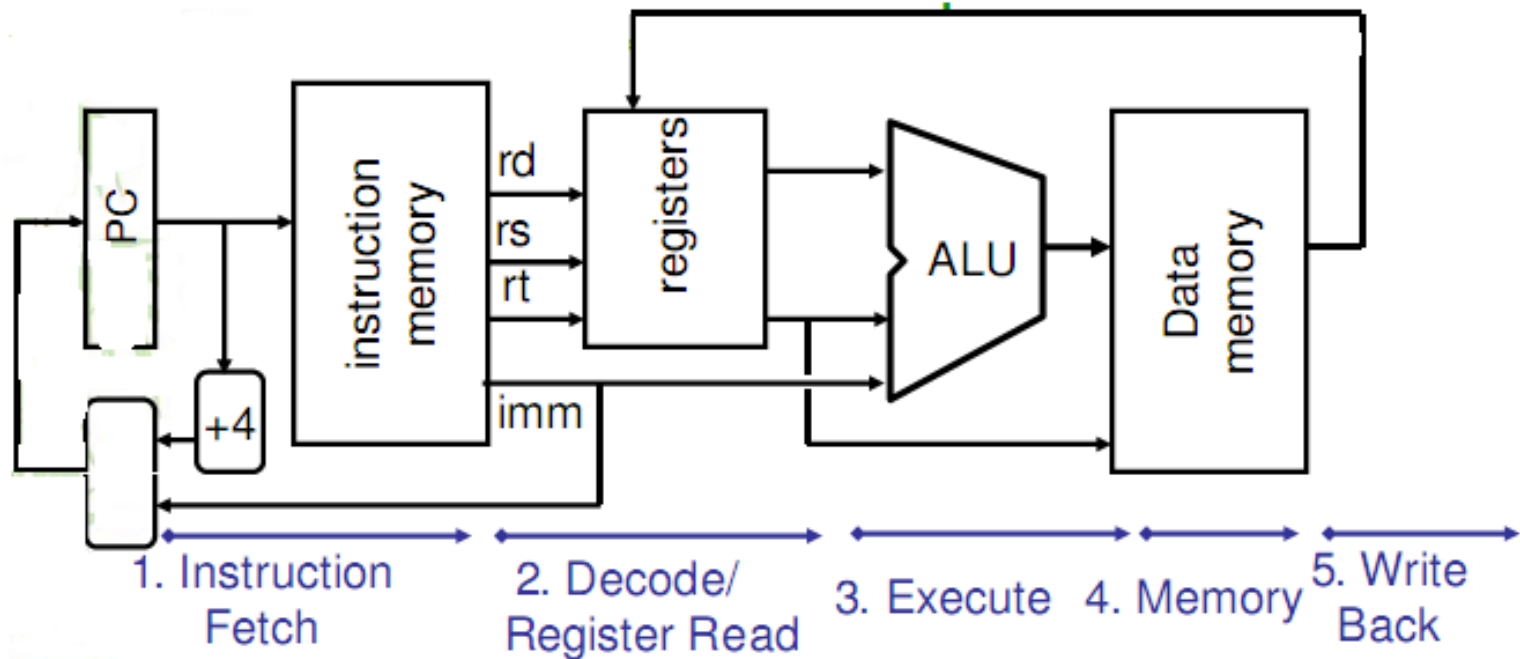
Các bước thực thi lệnh trong MIPS

7

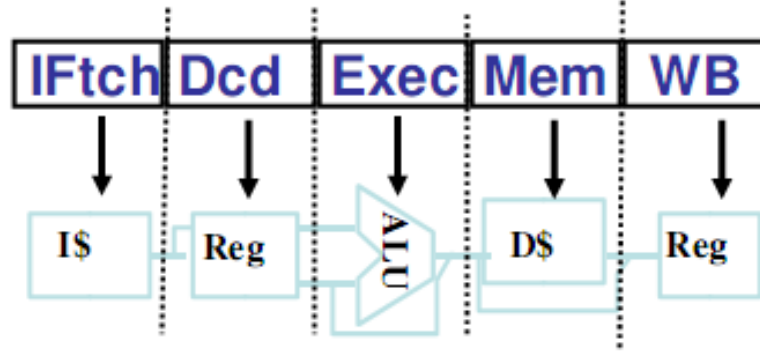
- **IFtch**: Instruction Fetch, Increment PC (Nạp lệnh)
- **Dcd**: Instruction Decode, Read Registers (Giải mã lệnh)
- **Exec**: (Thực thi)
 - Mem-ref: Calculate Address (Tính toán địa chỉ toán hạng)
 - Arith-log: Perform Operation (Tính toán số học, luận lý)
- **Mem**: (Lưu chuyển với bộ nhớ)
 - Load: Read Data from Memory
 - Store: Write Data to Memory
- **WB**: Write Data Back to Register (Lưu dữ liệu vào thanh ghi)

Datapath

8



- Use datapath figure to represent pipeline



Ý tưởng Pipeline

9

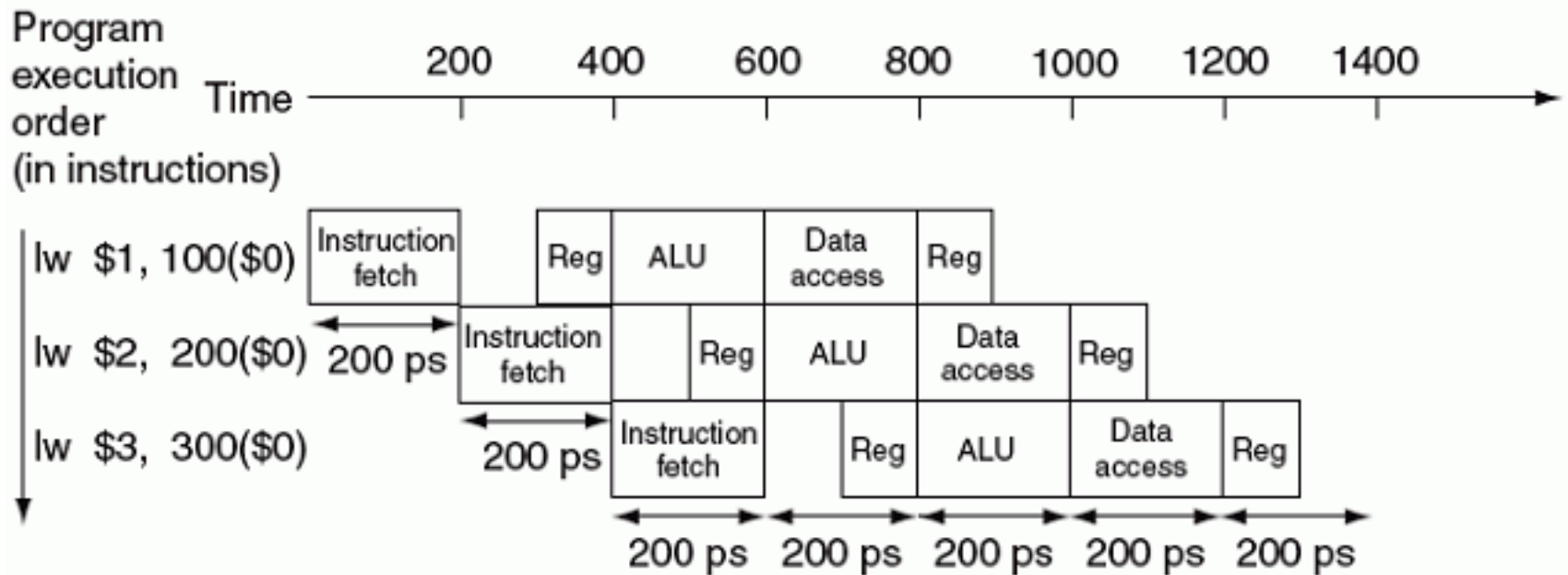
Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Ví dụ

10

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps



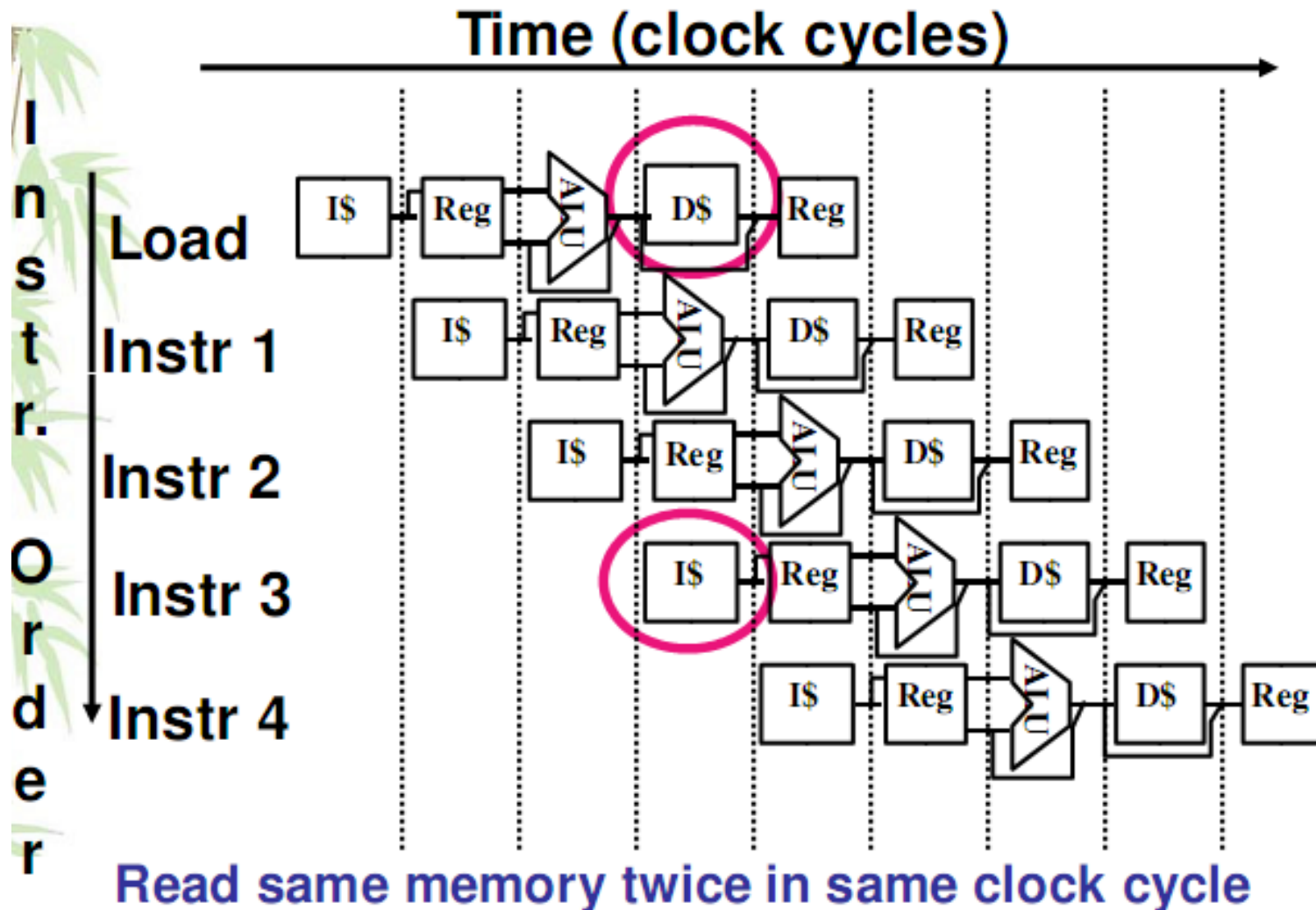
Các trở ngại (Hazards) của pipeline

11

- **Structural hazards**: do nhiều lệnh dùng chung một tài nguyên tại 1 thời điểm
 - **Data hazard**: lệnh sau sử dụng dữ liệu kết quả của lệnh trước
 - **Control hazard**: do rẽ nhánh gây ra, lệnh sau phải đợi kết quả rẽ nhánh của lệnh trước
- Gây ra hiện tượng “stalls” hoặc “bubbles” trong pipeline

Structural hazards #1: Single memory

12



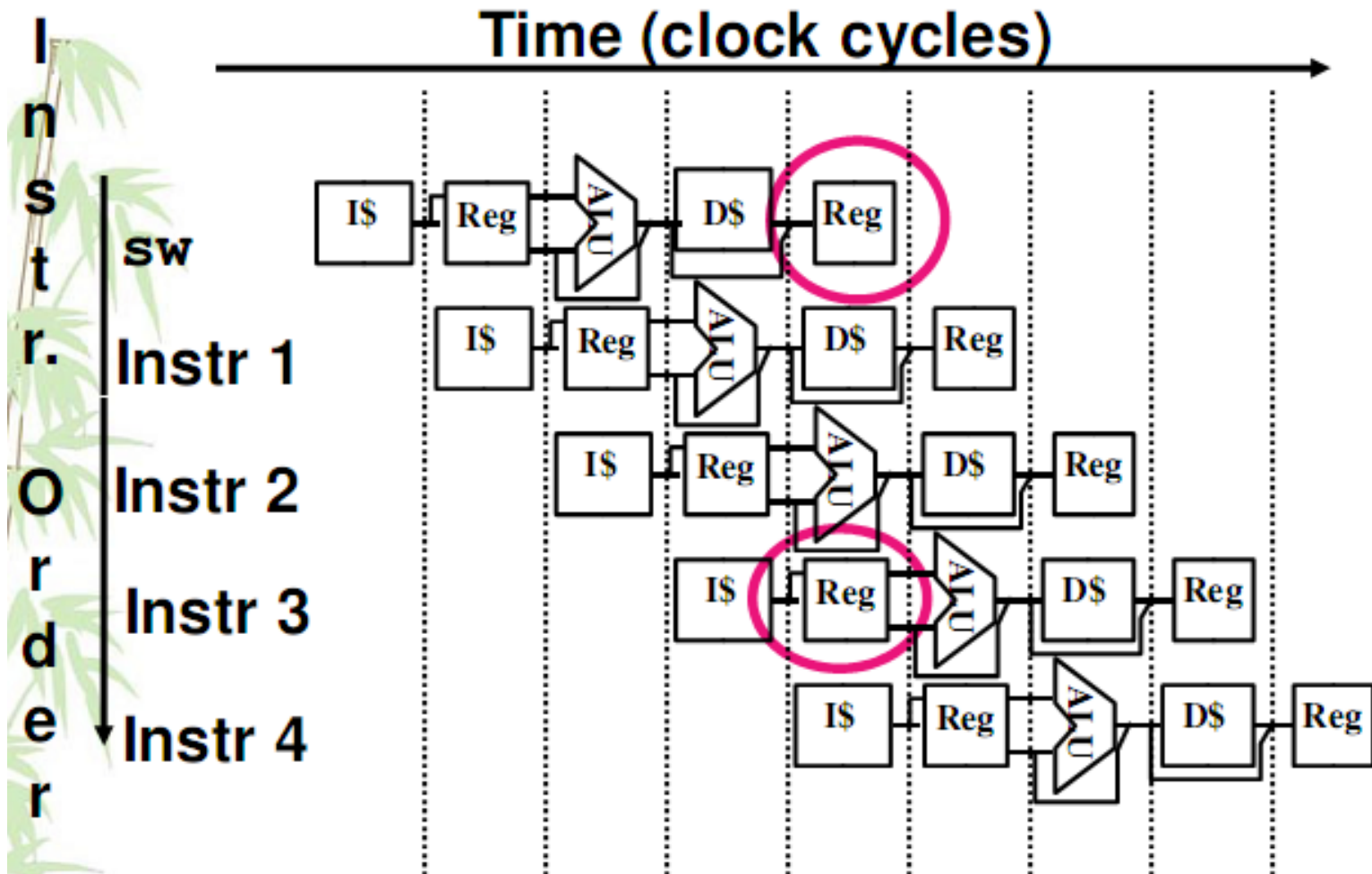
Structural hazards #1: Single memory

13

- Giải pháp:
 - Tạo 2 bộ nhớ đệm **Cache Level 1** trên CPU
 - **L1 Instruction Cache** và **L1 Data Cache**
 - Cần những phần cứng phức tạp hơn để điều khiển khi không có cả 2 bộ nhớ đệm này

Structural hazards #2: Registers

14



Can we read and write to registers simultaneously?

Structural hazards #2:

Registers

15

- Có 2 giải pháp khác nhau được dùng:
 - RegFile có tốc độ truy cập rất nhanh (thường ít hơn 1 nửa thời gian thực thi trên ALU tính trên 1 chu kỳ clock)
 - Write vào RegFile trong suốt nửa đầu chu kỳ clock
 - Read từ RegFile trong nửa chu kỳ clock còn lại
 - Tạo RegFile với 2 ngõ Read và Write độc lập

Control hazard: Rẽ nhánh

17

- Chúng ta phải đặt điều kiện rẽ nhánh vào trong ALU
 - Do vậy sẽ có ít nhất 2 lệnh sau phần rẽ nhánh sẽ được fetch, bất kể điều kiện rẽ nhánh có thực hiện hay không
- Nếu chúng ta không thực hiện rẽ nhánh → Cứ thực thi theo trình tự bình thường
- Ngược lại, đừng thực thi bất kỳ lệnh nào sau điều kiện rẽ nhánh, cứ nhảy đến label tương ứng

Control hazard: Rẽ nhánh

18

- **Giải pháp ban đầu:** Trì hoãn (stall) cho đến khi điều kiện rẽ nhánh được thực hiện
 - ▣ **Chèn những lệnh rác "no-op"** (chẳng thực hiện việc gì, chỉ để trì hoãn thời gian) hoặc **hoãn việc nạp (fetch) sang lệnh kế (trong 2 chu kỳ clock)**
 - ▣ **Nhược điểm:** Điều kiện rẽ nhánh phải làm đến 3 chu kỳ clock

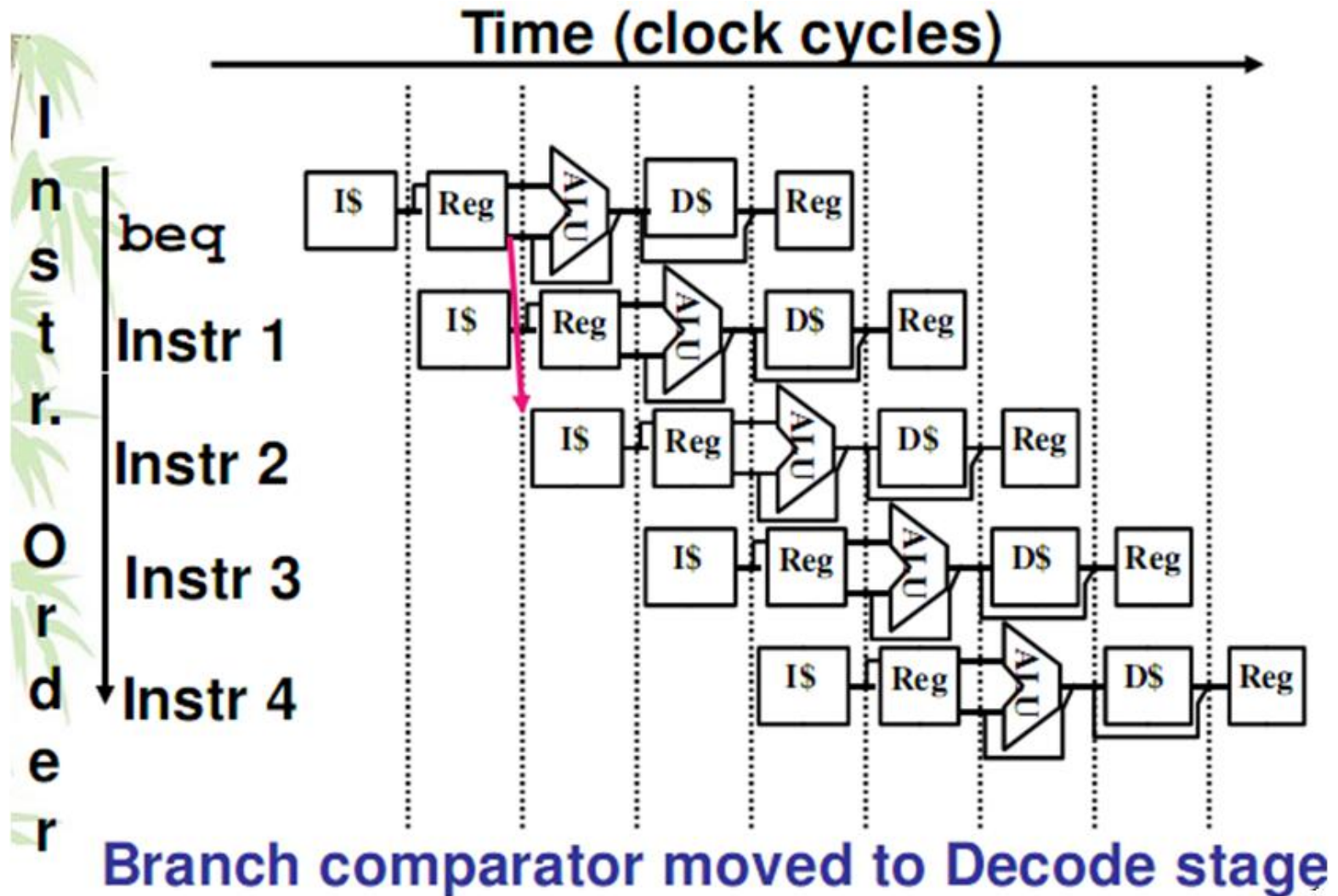
Control hazard: Optimization 1

19

- Giải pháp tối ưu hoá 1:
 - Chèn thêm các phép so sánh rẽ nhánh đặc biệt tại Stage 2 (decode)
 - Ngay sau khi lệnh được decode, lập tức quyết định giá trị mới cho thanh ghi PC
 - Lợi ích: Bởi vì điều kiện rẽ nhánh đã làm xong trong stage 2, nên chỉ có 1 lệnh không cần thiết được nạp → chỉ cần 1 no-op là đủ

Minh hoạ

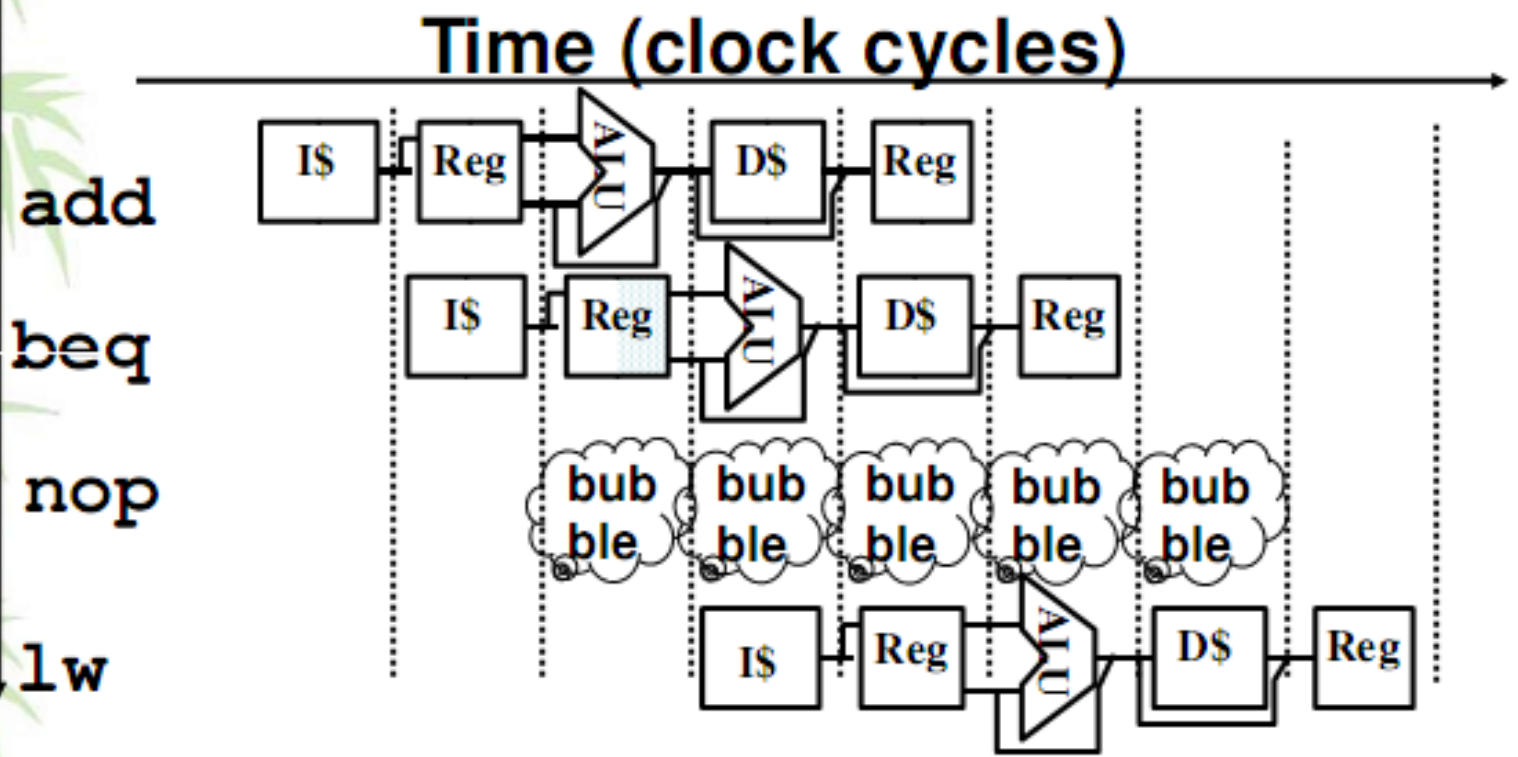
20



Minh hoạ

- User inserting no-op instruction

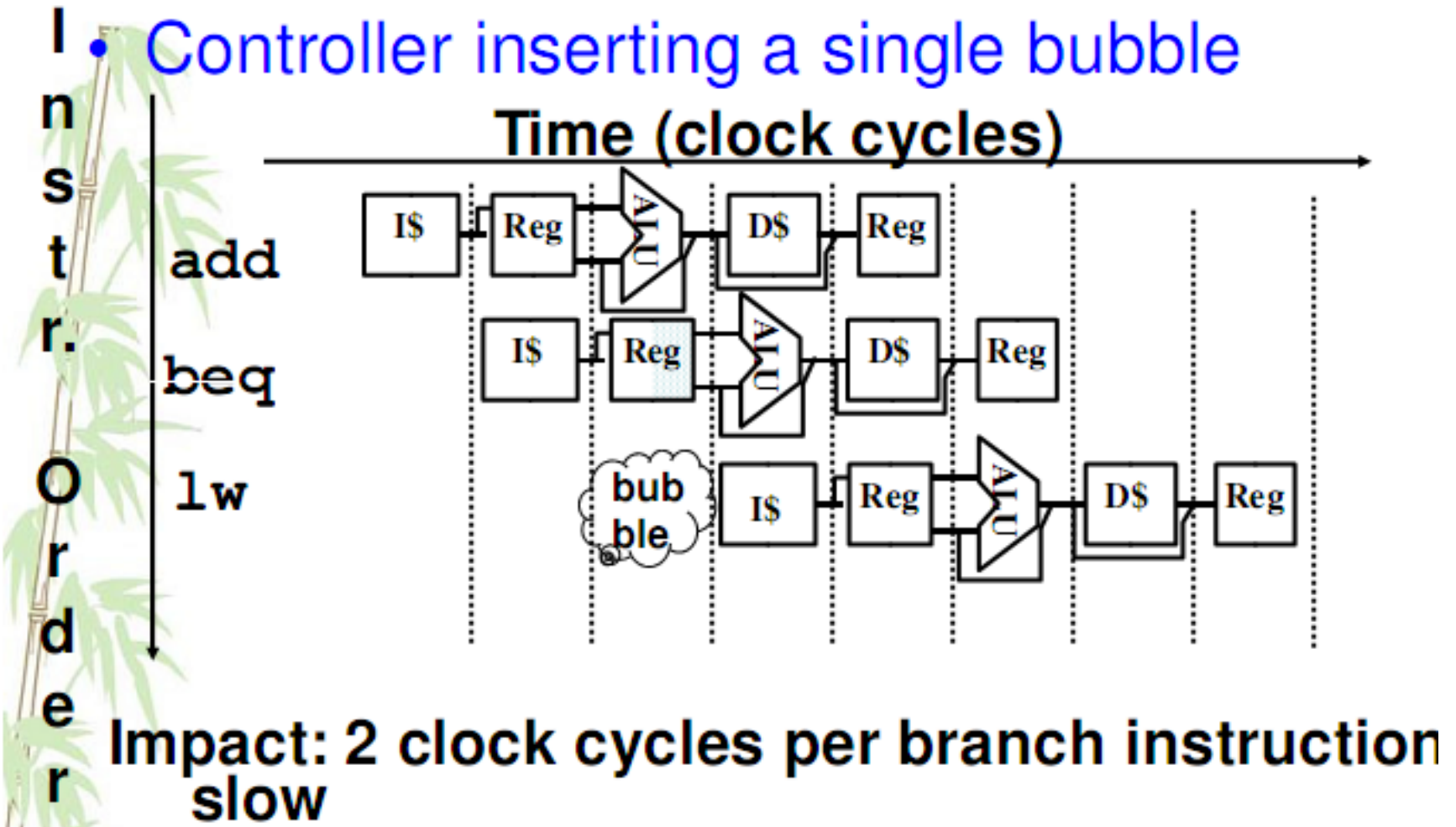
I
n
s
t
r.
O
r
d
e
r



Impact: 2 clock cycles per branch instruction slow

Minh hoạ

- Controller inserting a single bubble



Control hazard: Optimization 2

23

- **Giải pháp tối ưu hoá 2: Tái định nghĩa rẽ nhánh**
 - Định nghĩa cũ: Nếu chúng ta thực hiện rẽ nhánh thì sẽ không có bất kỳ lệnh nào sau lệnh rẽ nhánh được làm một cách “vô tình” (không mong muốn)
 - **Định nghĩa mới:** Bất cứ khi nào thực hiện rẽ nhánh, một lệnh ngay sau lệnh rẽ nhánh sẽ lập tức được thực thi (gọi là branch-delay slot)
 - **Ý nghĩa:** Chúng ta luôn thực thi 1 lệnh ngay phía sau lệnh rẽ nhánh

Control hazard: Optimization 2

24

- Lưu ý về Branch-Delay Slot:
 - Trường hợp xấu nhất: có thể luôn phải đặt 1 lệnh no-op vào trong branch-delay slot
 - Trường hợp tốt hơn: có thể tìm được 1 lệnh trước lệnh rẽ nhánh để đặt trong branch-delay slot mà vẫn không làm ảnh hưởng chương trình
 - Thủ công: Tái cấu trúc thứ tự lệnh là cách làm phổ biến
 - Tự động: Compiler phải rất thông minh để tìm lệnh làm điều này

Nondelayed vs. Delayed

25

Nondelayed Branch

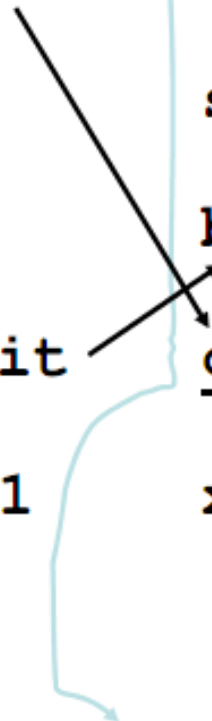
```
or $8, $9, $10  
add $1, $2, $3  
sub $4, $5, $6  
beq $1, $4, Exit  
xor $10, $1, $11
```

Exit:

Delayed Branch

```
add $1, $2, $3  
sub $4, $5, $6  
beq $1, $4, Exit  
or $8, $9, $10  
xor $10, $1, $11
```

Exit:



Data hazards

26

- Xem xét dãy lệnh sau:

add \$t0, \$t1, \$t2

sub \$t4, \$t0, \$t3

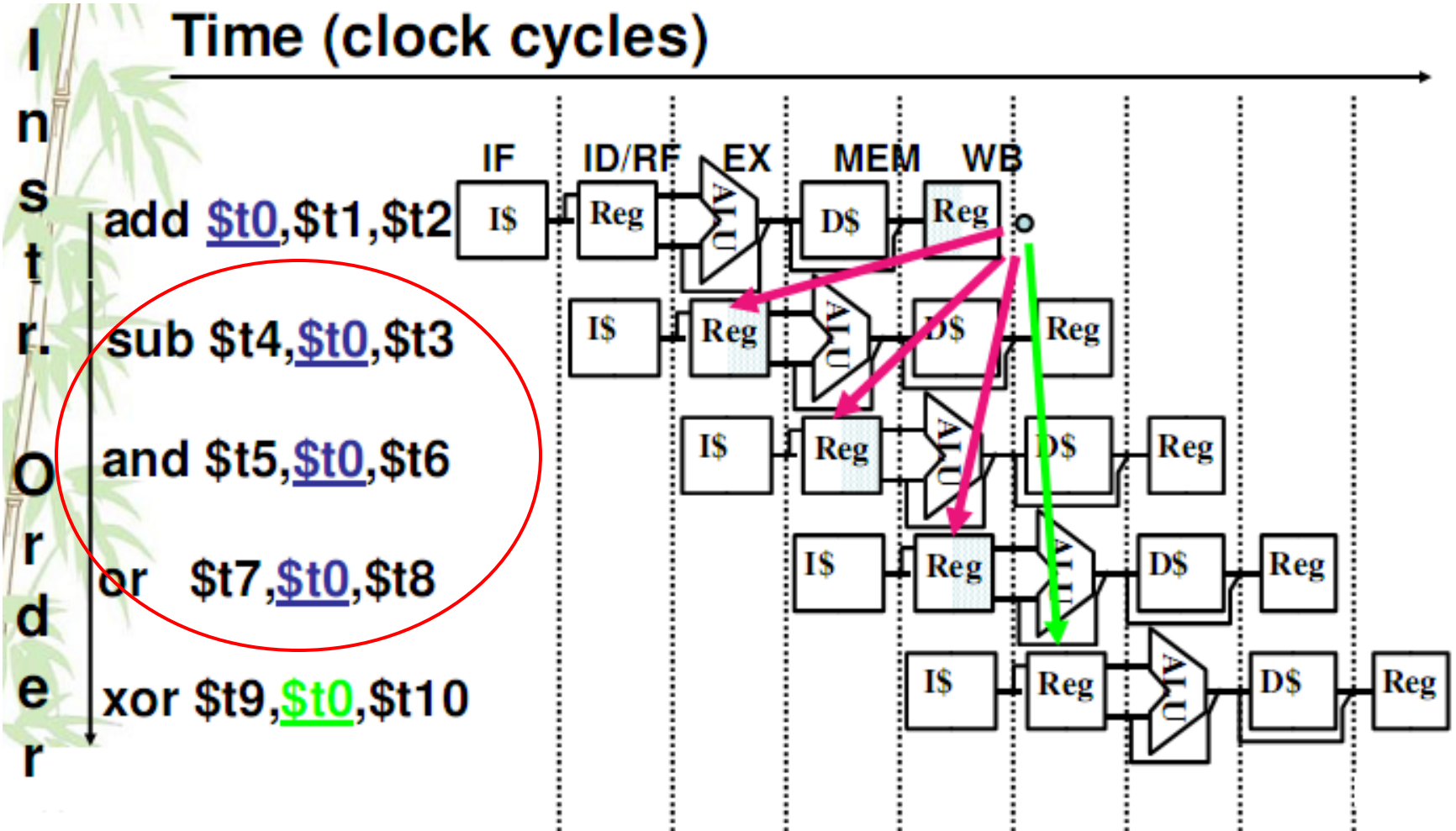
and \$t5, \$t0, \$t6

or \$t7, \$t0, \$t8

xor \$t9, \$t0, \$t10

Data hazards

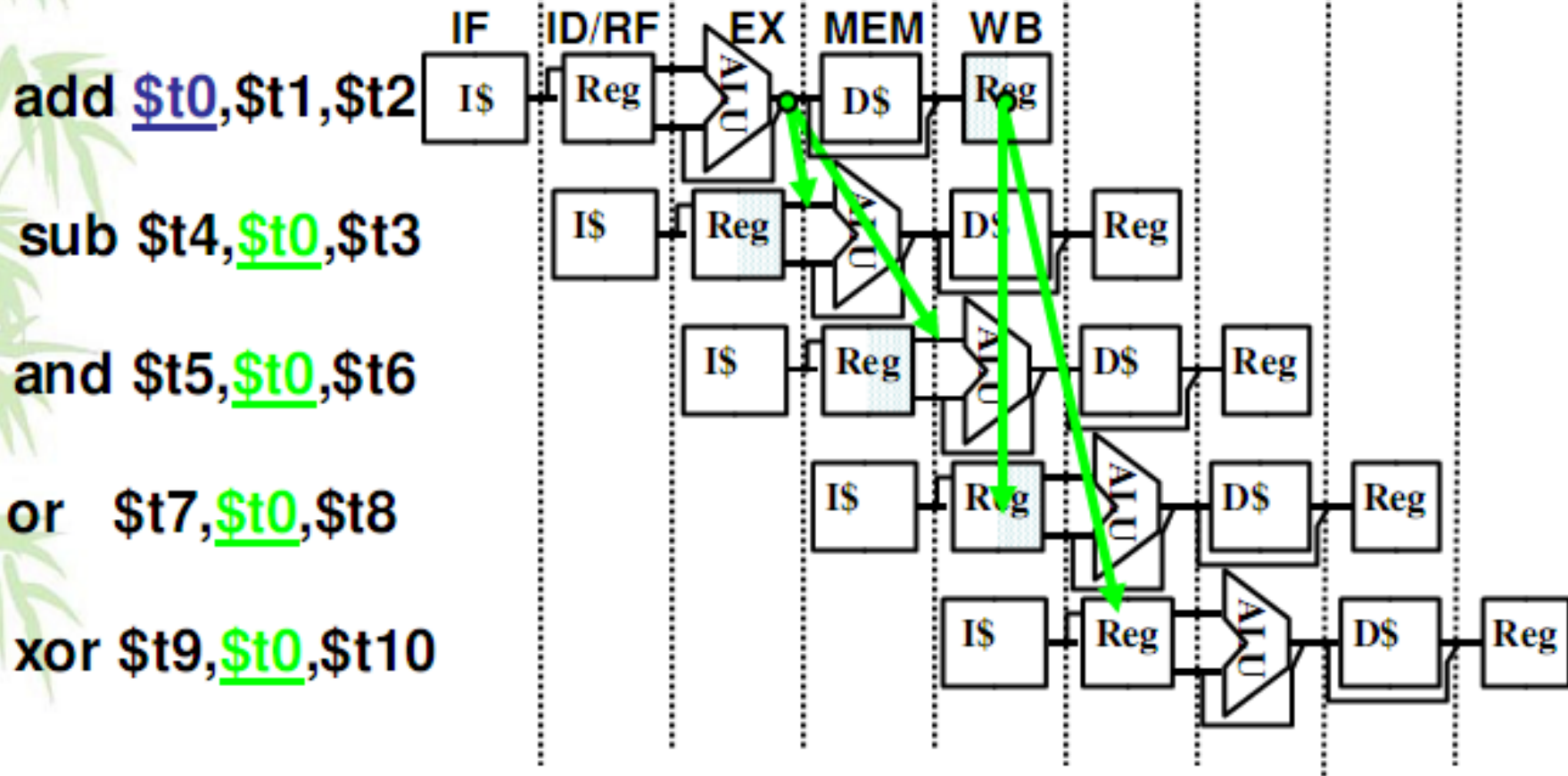
27



Giải pháp Data hazards: Forwarding

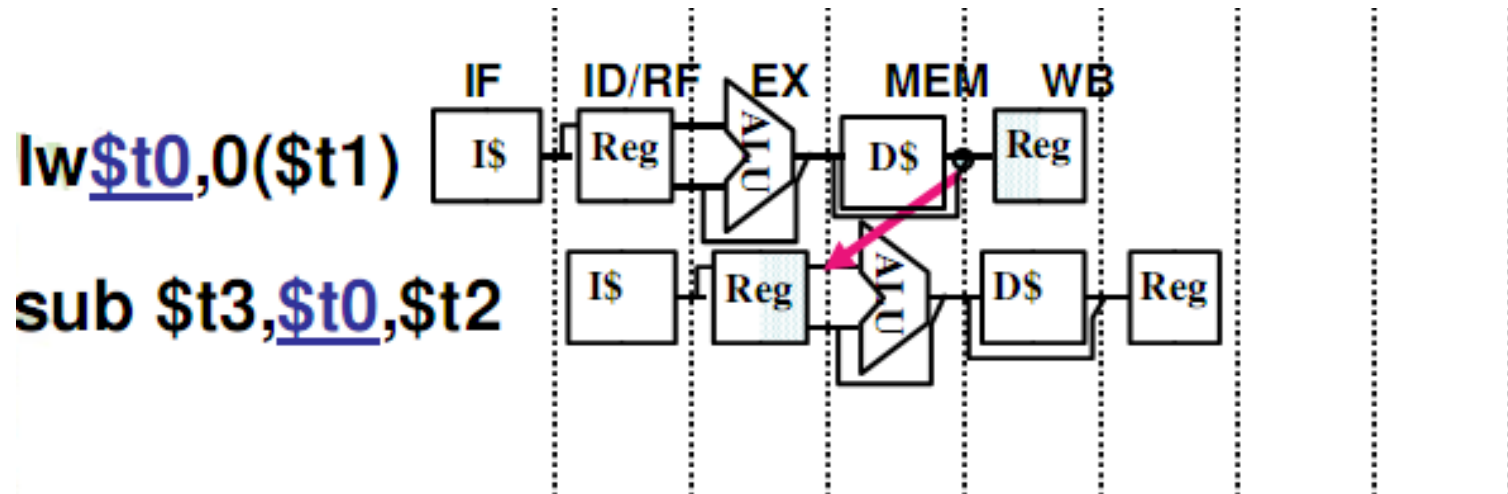
28

- Forward result from one stage to another



Forwarding không giải quyết được...

29



- Giải pháp: Phải trì hoãn lệnh sub lại (stall) sau đó mới dùng Forwarding được

Data hazards: Loads

30

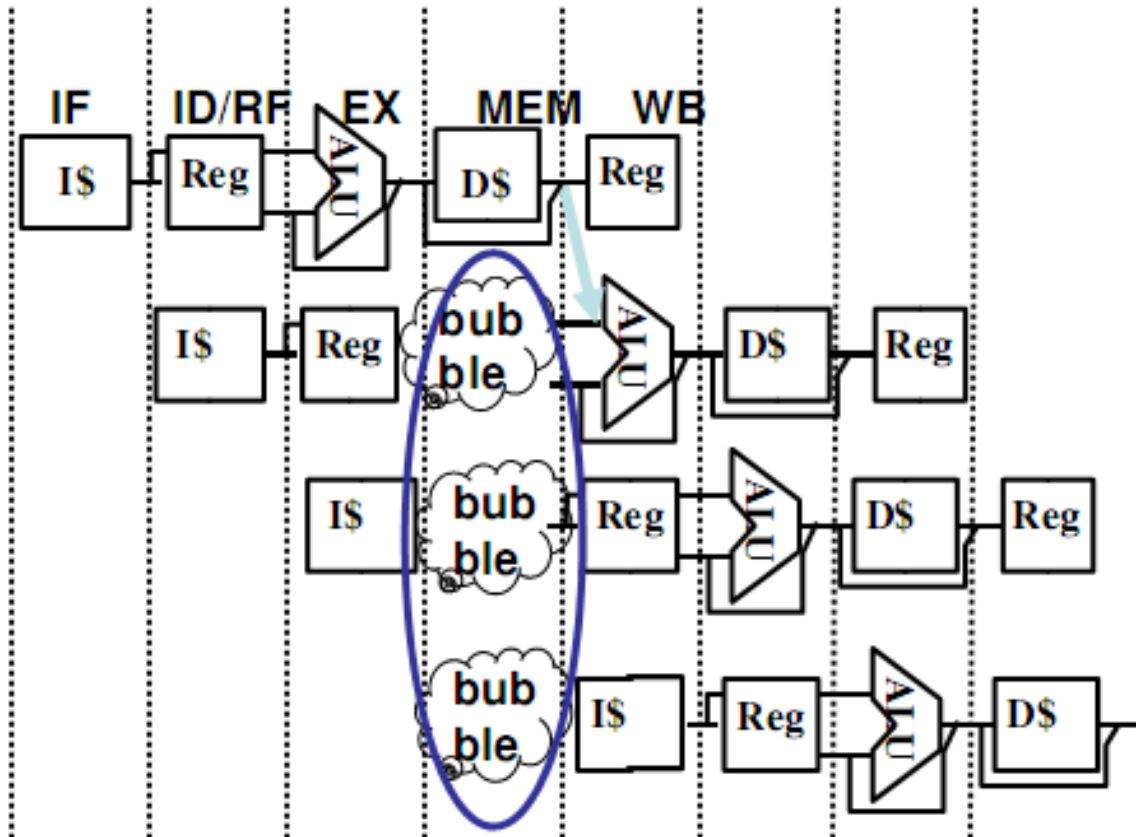
- Hardware stalls pipeline
 - Called “interlock”

lw \$t0, 0(\$t1)

sub \$t3, \$t0, \$t2

and \$t5, \$t0, \$t4

or \$t7, \$t0, \$t6



Data hazards: Loads

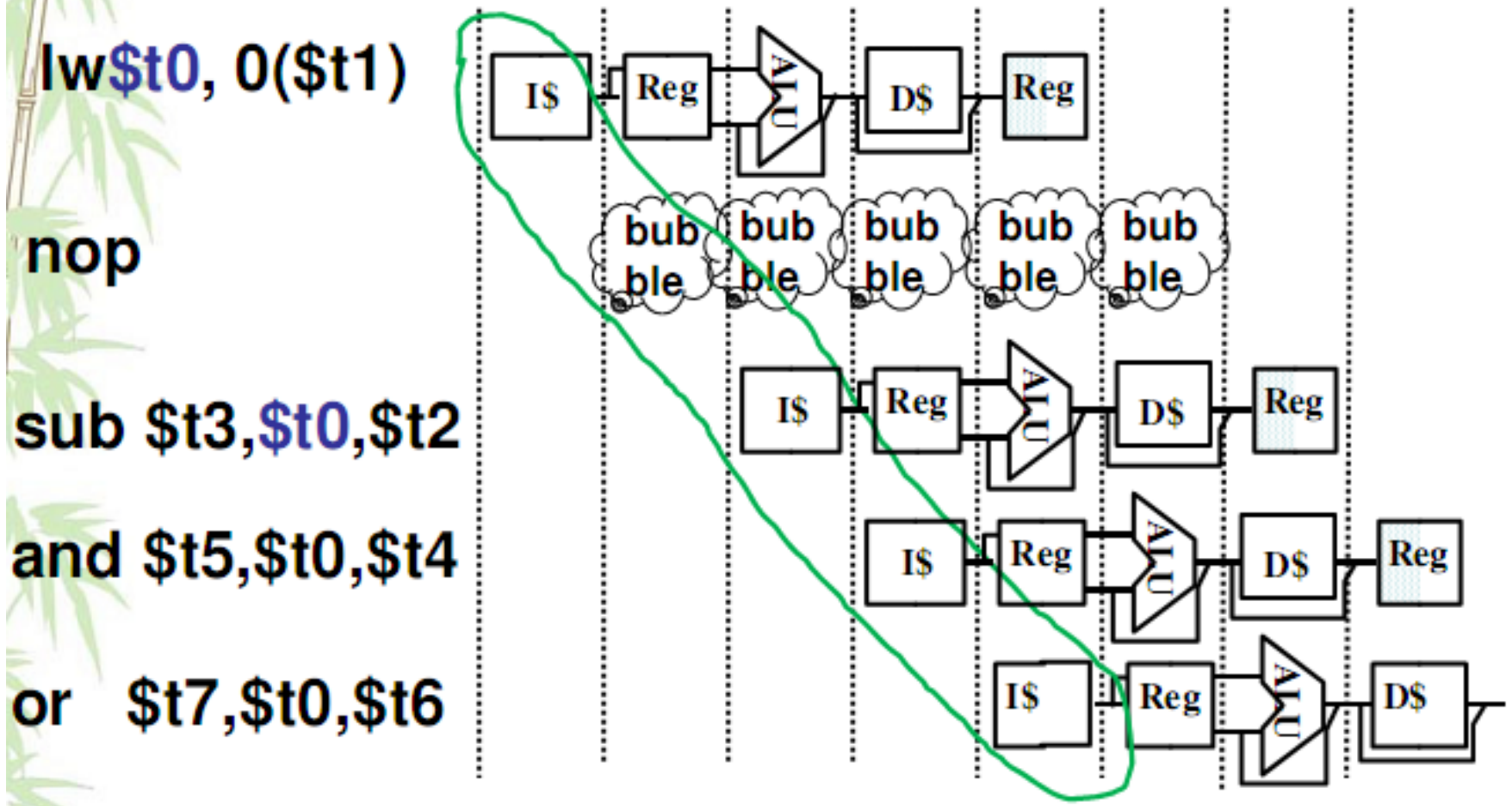
31

- Vị trí lệnh (instruction slot) sau một load được gọi là “load delay slot”
- Nếu lệnh đó dùng kết quả của load, thì hardware interlock có thể sẽ hoãn (stall) nó đúng 1 chu kỳ clock
- Nếu sau load là 1 lệnh không liên quan, thì không cần trì hoãn (stall) lệnh đó

Data hazards: Loads

32

- Stall is equivalent to nop



Homework

33

- Sách Petterson & Hennessy: Đọc 6.1

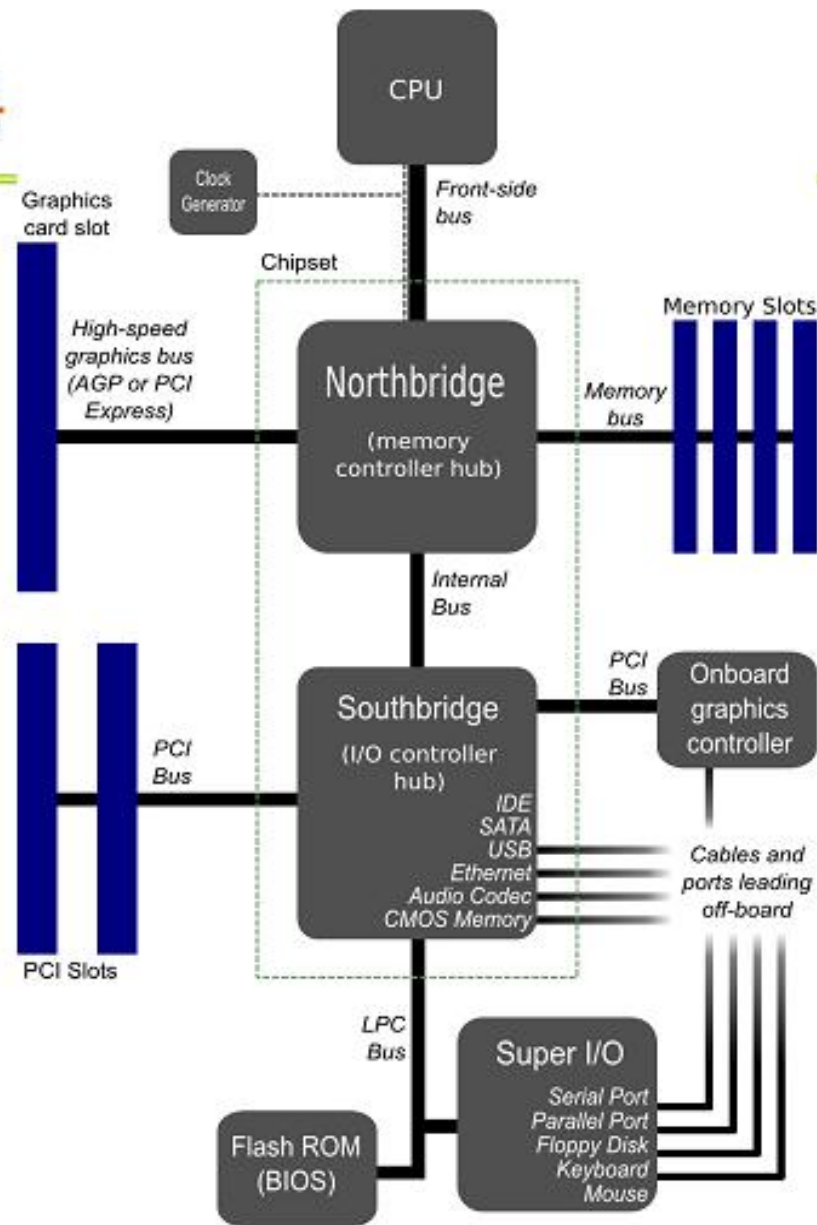
KIẾN TRÚC MÁY TÍNH & HỢP NGỮ

ThS Vũ Minh Trí – vmtri@fit.hcmus.edu.vn



I/O Device Speeds

• Device	Behavior	Partner	Data Rate (KBytes/s)
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice output	Output	Human	5.00
Floppy disk	Storage	Machine	50.00
Laser Printer	Output	Human	100.00
Magnetic Disk	Storage	Machine	10,000.00
Wireless Network	I or O	Machine	10,000.00
Graphics Display	Output	Human	30,000.00
Wired LAN Network	I or O	Machine	125,000.00





I/O devices

- Để giao tiếp với các thiết bị bên ngoài:
 - Làm sao kết nối với nhiều loại thiết bị ?
 - Làm sao để truyền nhận tín hiệu điều khiển và dữ liệu
 - Làm sao để các chương trình giao tiếp với thiết bị ?
- Hai cách tổ chức:
 - Port-mapped I/O
 - Memory-mapped I/O



Port-mapped I/O

- Có thể cần instruction riêng cho I/O
- Sử dụng không gian địa chỉ riêng cho các thiết bị. Mỗi thiết bị được gán một (hoặc một vài) port



Memory-mapped I/O

- Không cần thêm instruction riêng
- Dùng chung không gian địa chỉ bộ nhớ. Mỗi thiết bị được cấp một vùng địa chỉ.
- Làm việc với thiết bị giống như làm việc với bộ nhớ.

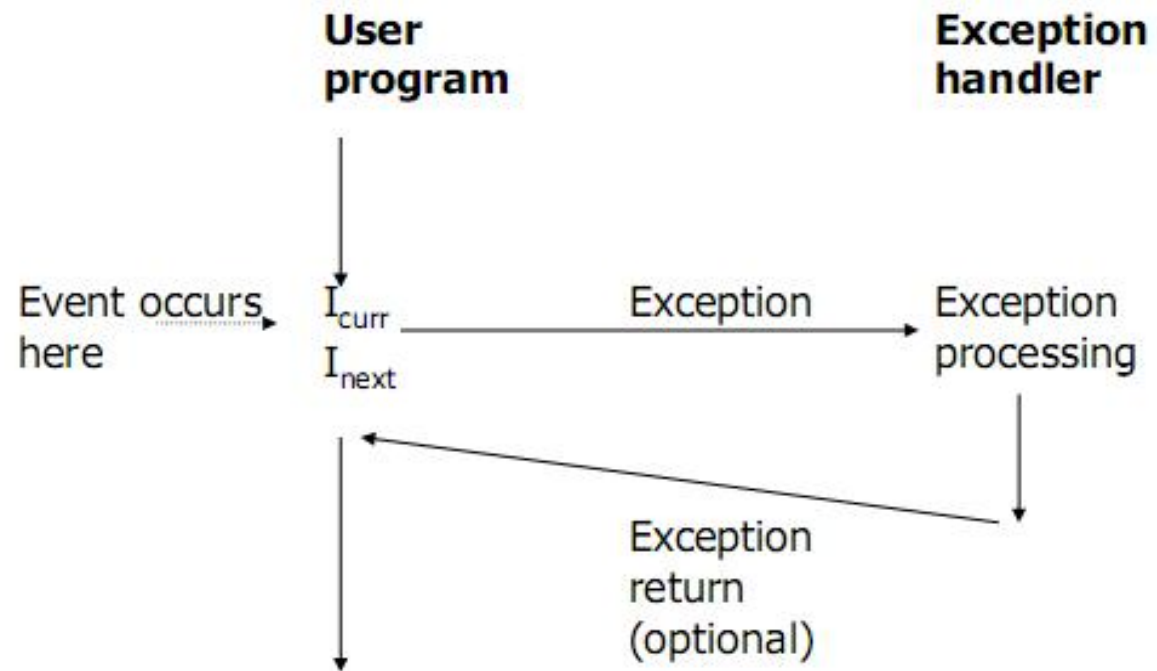


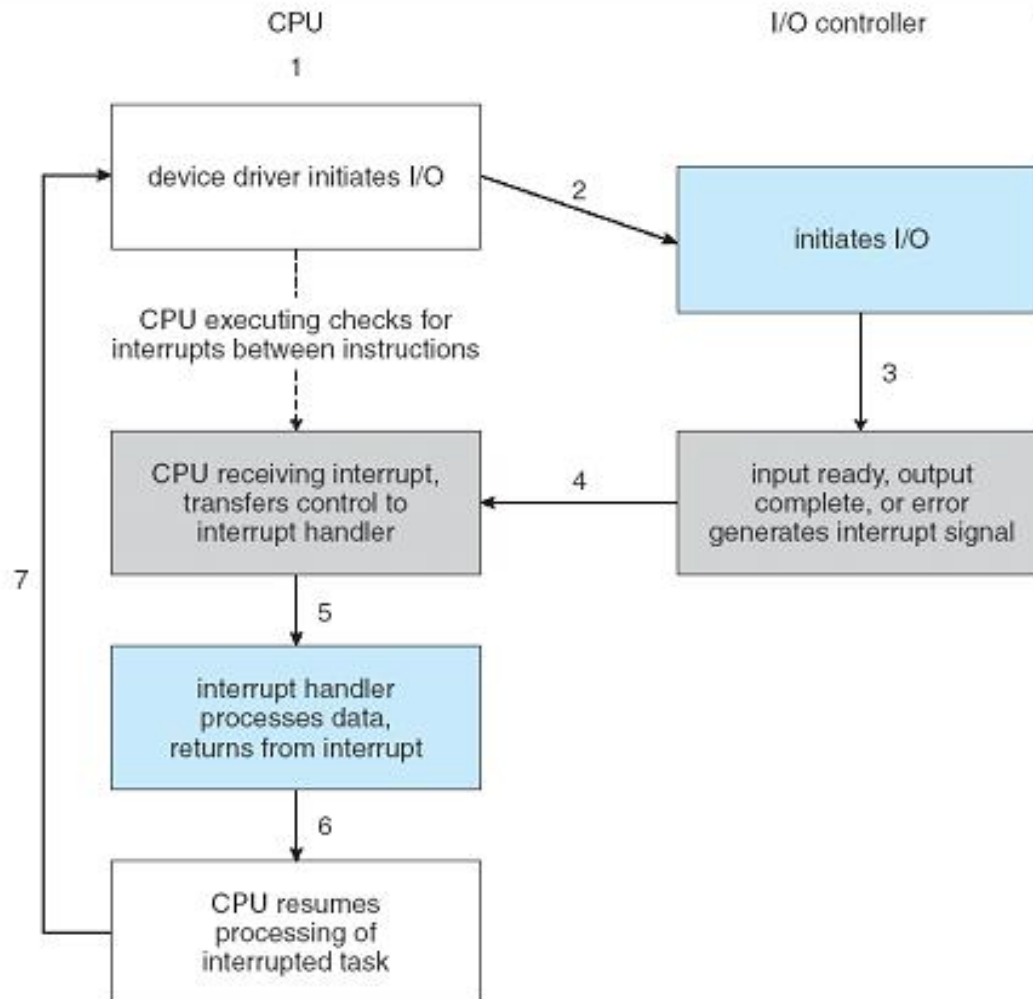
Giao tiếp với thiết bị

- Được thực hiện thông qua
 - Control register
 - Data register
- Cơ chế:
 - Polling: CPU chủ động kiểm tra trạng thái của thiết bị
 - Interrupt-driven: thiết bị chủ động thông báo trạng thái với CPU
 - DMA: giao tiếp không qua CPU
- Interrupt:
 - một trong 4 loại exception (interrupt, trap, fault, abort)



Exceptions







Interrupt Service Handler

- Làm cách nào để đến được đoạn code xử lý interrupt ?
 - Centralized dispatch
 - Vectored dispatch



Bài tập về nhà

- Đọc phần 8.4 và 8.5 sách P&H
- Xem về RAID trên trang:
http://raid.com/04_00.html
- Tìm hiểu về một số loại bus thông dụng:
 - PCI / PCIe / AGP / USB / IDE / SATA
 - So sánh tốc độ (bps)