



# Chương 3: Tập lệnh máy tính

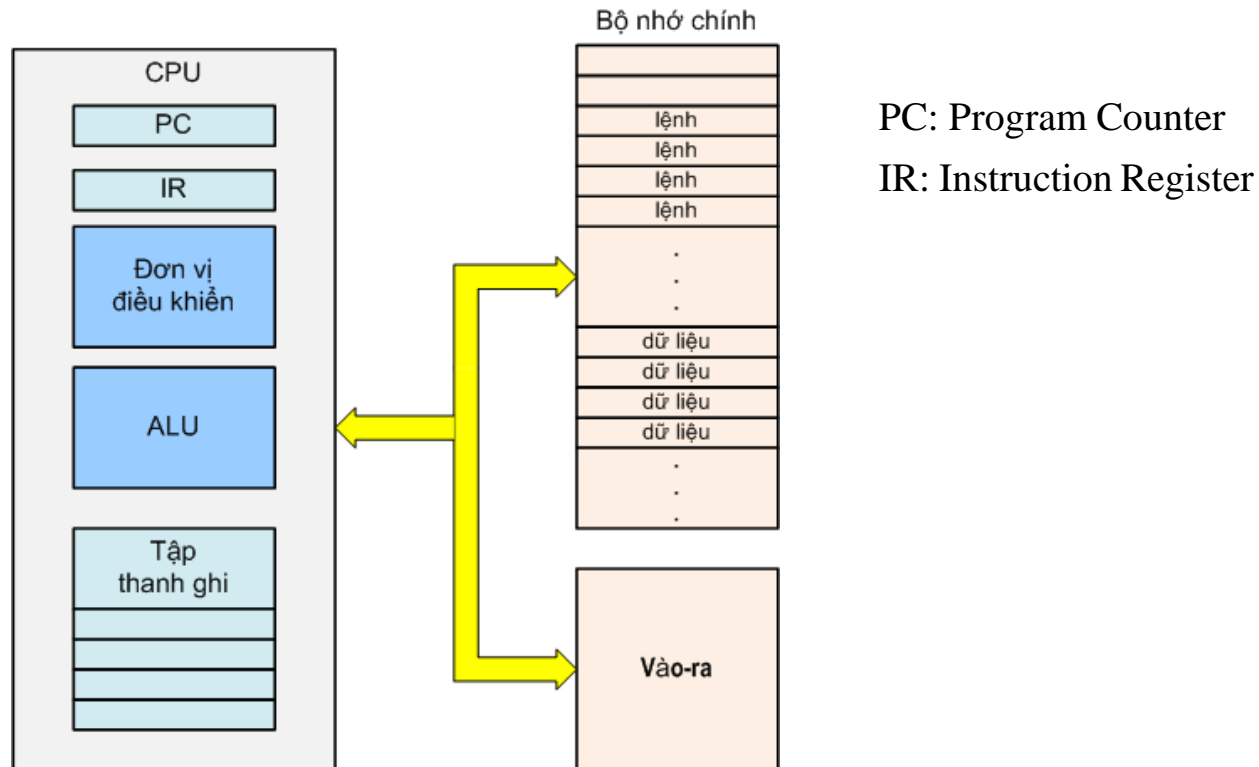
# Chương 3: Nội dung chính

---

- Giới thiệu về tập lệnh
- Khuôn dạng và các thành phần của lệnh
- Các dạng toán hạng lệnh
- Các chế độ địa chỉ
- Một số dạng lệnh thông dụng
- Cơ chế ống lệnh

# Giới thiệu chung

## Mô hình lập trình của máy tính



# Giới thiệu chung về tập lệnh

---

- Mỗi bộ xử lý có một tập lệnh xác định
- Tập lệnh thường có hàng chục đến hàng trăm lệnh
- Mỗi lệnh là một chuỗi số nhị phân mà bộ xử lý hiểu được để thực hiện một thao tác xác định.
- Các lệnh được mô tả bằng các ký hiệu gọi nhớ dạng text: chính là các lệnh của hợp ngữ (assembly language)

# Giới thiệu chung

---

- Lệnh máy tính là một từ nhị phân (binary word) mà thực hiện một nhiệm vụ cụ thể:
  - Lệnh được lưu trong bộ nhớ
  - Lệnh được đọc từ bộ nhớ vào CPU để giải mã và thực hiện
  - Mỗi lệnh có chức năng riêng của nó
- Tập lệnh gồm nhiều lệnh, có thể được chia thành các nhóm theo chức năng:
  - Chuyển dữ liệu (data movement)
  - Tính toán (computational)
  - Điều kiện và rẽ nhánh (conditioning & branching)
  - Các lệnh khác ...

# Giới thiệu chung

---

- Quá trình thực hiện/ chạy lệnh được chia thành các pha hay giai đoạn (stage). Mỗi lệnh có thể được thực hiện theo 4 giai đoạn:
  - Đọc lệnh IF(Instruction Fetch): lệnh được đọc từ bộ nhớ vào CPU
  - Giải mã lệnh ID(Instruction Decode): CPU giải mã lệnh
  - Chạy lệnh IE(Instruction Execution): CPU thực hiện lệnh
  - Ghi WB(Write Back): kết quả lệnh (nếu có) được ghi vào thanh ghi hoặc bộ nhớ

# Chu kỳ thực hiện lệnh

---

- Chu kỳ thực hiện lệnh (instruction execution cycle) là khoảng thời gian mà CPU thực hiện xong một lệnh
  - Một chu kỳ thực hiện lệnh gồm một số giai đoạn thực hiện lệnh
  - Một giai đoạn thực hiện lệnh có thể gồm một số chu kỳ máy
  - Một chu kỳ máy có thể gồm một số chu kỳ đồng hồ

# Chu kỳ thực hiện lệnh

---

- Một chu kỳ thực hiện lệnh có thể gồm các thành phần sau:
  - Chu kỳ đọc lệnh
  - Chu kỳ đọc bộ nhớ (memory read)
  - Chu kỳ ghi bộ nhớ (memory write)
  - Chu kỳ đọc thiết bị ngoại vi (I/O read)
  - Chu kỳ ghi thiết bị ngoại vi (I/O write)
  - Chu kỳ bus rỗi (bus idle)



# Địa chỉ byte nhớ và word nhớ

Dữ liệu hoặc lệnh	Địa chỉ byte (theo Hexa)
byte (8-bit)	0x0000 0000
byte	0x0000 0001
byte	0x0000 0002
byte	0x0000 0003
byte	0x0000 0004
byte	0x0000 0005
byte	0x0000 0006
byte	0x0000 0007
.	
.	
.	
byte	0xFFFF FFFB
byte	0xFFFF FFFC
byte	0xFFFF FFDD
byte	0xFFFF FFDE
byte	0xFFFF FFFF

$2^{32}$  bytes

Dữ liệu hoặc lệnh	Địa chỉ word (theo Hexa)
word (32-bit)	0x0000 0000
word	0x0000 0004
word	0x0000 0008
word	0x0000 000C
word	0x0000 0010
word	0x0000 0014
word	0x0000 0018
.	
.	
.	
word	0xFFFF FFF4
word	0xFFFF FFF8
word	0xFFFF FFFC

$2^{30}$  words

# Các thành phần của lệnh máy

Mã lệnh	Địa chỉ của các toán hạng	
Opcode	Addresses of Operands	
Opcode	Des addr.	Source addr.

- Mã lệnh (operation code □ opcode): mã hóa cho thao tác mà bộ xử lý phải thực hiện
- Địa chỉ toán hạng: chỉ ra nơi chứa các toán hạng mà thao tác sẽ tác động:
  - Toán hạng nguồn (source operand): dữ liệu vào của thao tác
  - Toán hạng đích (destination operand): dữ liệu ra của thao tác

# Chế độ địa chỉ toán hạng

---

- Toán hạng của lệnh có thể là:
  - Một giá trị cụ thể nằm ngay trong lệnh
  - Nội dung của thanh ghi
  - Nội dung của ngăn nhớ hoặc cổng vào-ra
- Phương pháp định địa chỉ (addressing modes) là cách thức địa chỉ hóa trong trường địa chỉ của lệnh để xác định nơi chứa toán hạng

# Các chế độ địa chỉ

---

- Chế độ địa chỉ là cách thức CPU tổ chức các toán hạng
  - Chế độ địa chỉ cho phép CPU kiểm tra dạng và tìm các toán hạng của lệnh
- Một số chế độ địa chỉ tiêu biểu:
  - Chế độ địa chỉ tức thì (Immediate)
  - Chế độ địa chỉ trực tiếp (Direct)
  - Chế độ địa chỉ gián tiếp qua thanh ghi (Register Indirect)
  - Chế độ địa chỉ gián tiếp qua bộ nhớ (Memory Indirect)
  - Chế độ địa chỉ chỉ số (Indexed)
  - Chế độ địa chỉ tương đối (Relative)

# Chế độ địa chỉ tức thì

---

- Giá trị của toán hạng nguồn có sẵn trong lệnh (hằng số)
- Toán hạng đích có thể là thanh ghi hoặc một vị trí bộ nhớ
- Ví dụ:

LOAD  $R_1$ , #1000;       $1000 \rightarrow R_1$   
giá trị 1000 được tải vào thanh ghi  $R_1$

LOAD B, #500;       $500 \rightarrow M[B]$   
Giá trị 500 được tải vào vị trí B trong bộ nhớ

# Chế độ địa chỉ tức thì

---

Mã lệnh		Toán hạng
---------	--	-----------

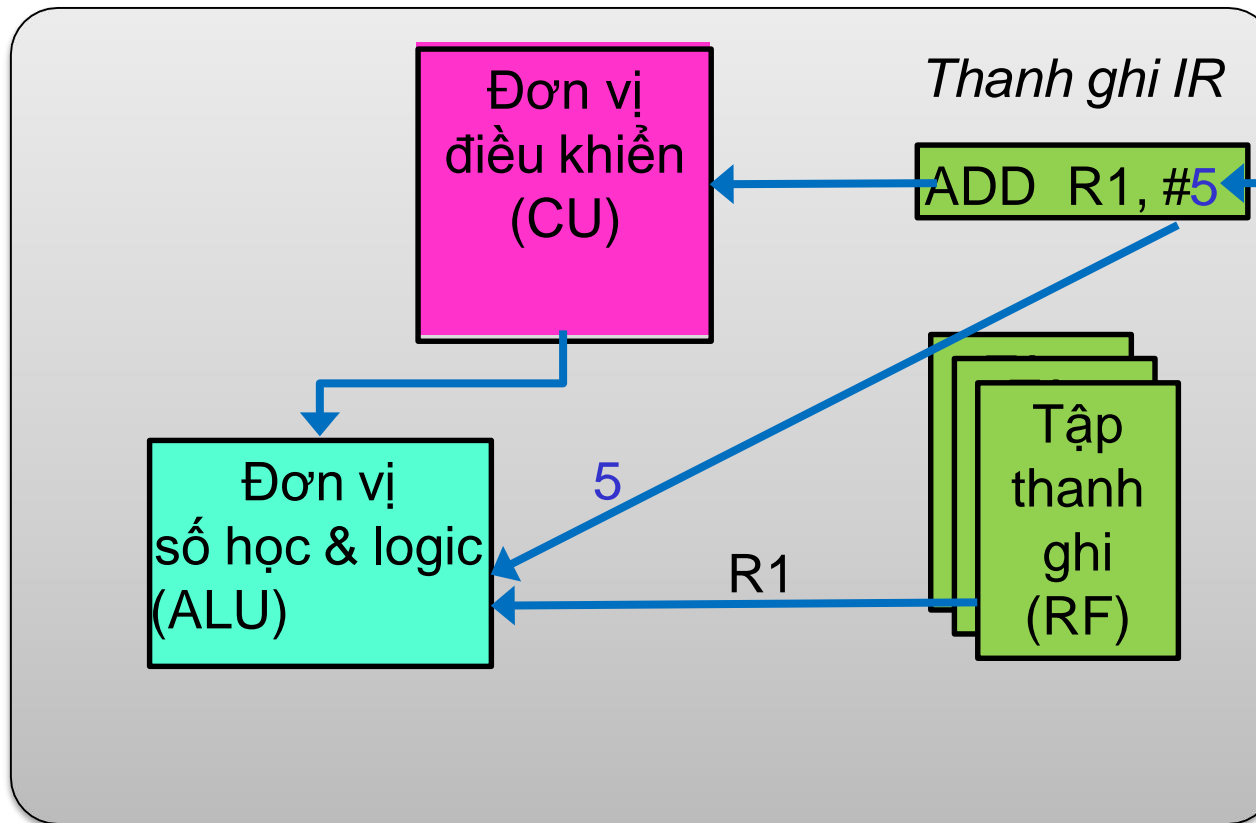
- Toán hạng là hằng số nằm ngay trong lệnh
- Chỉ có thể là toán hạng nguồn
- Ví dụ:

ADD R1, #5 // R1 = R1+5

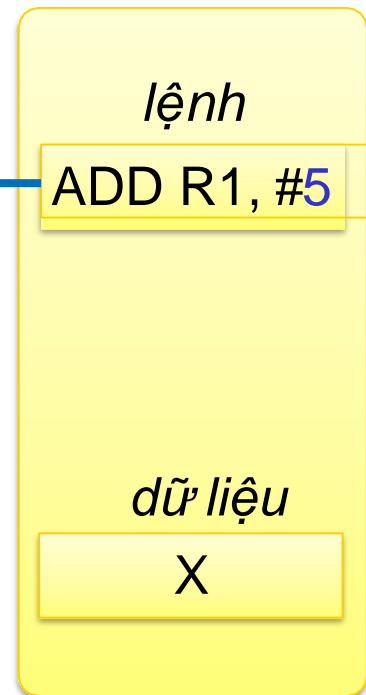
- Không tham chiếu bộ nhớ
- Truy nhập toán hạng rất nhanh
- Dải giá trị của toán hạng bị hạn chế

# Chế độ địa chỉ tức thì

## CPU



## Bộ nhớ chính



# Chế độ địa chỉ trực tiếp/ tuyệt đối

---

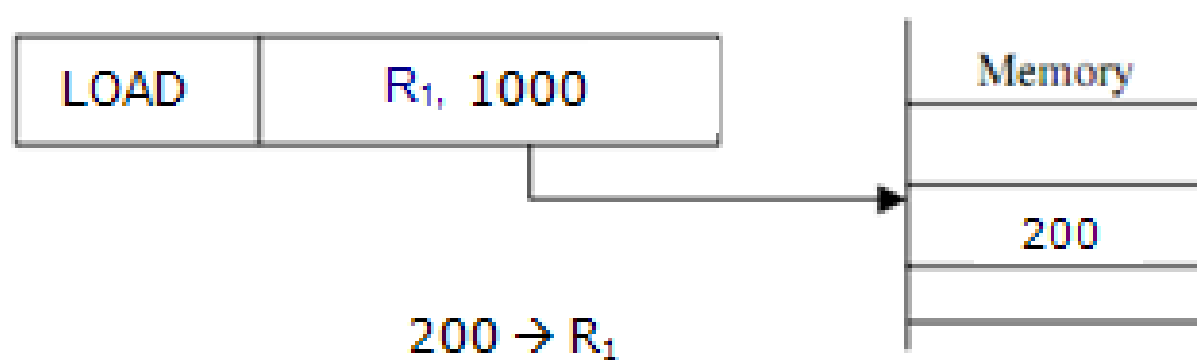
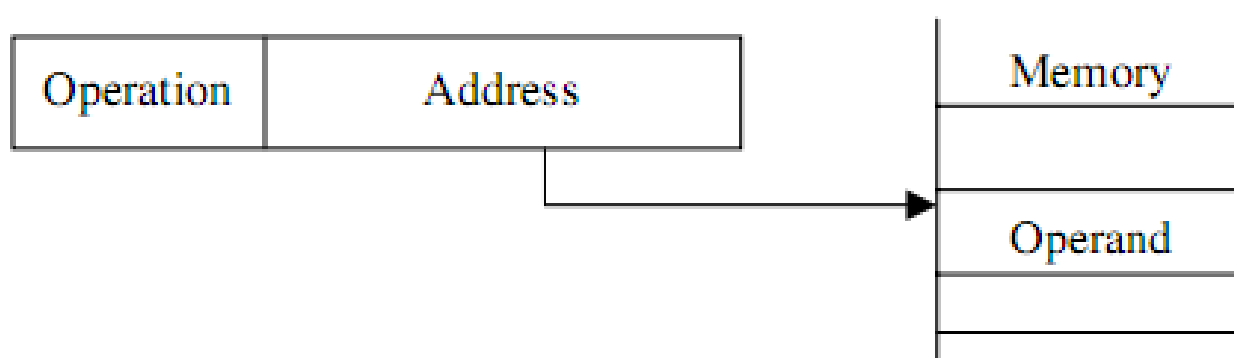
- Một toán hạng là địa chỉ của một vị trí trong bộ nhớ chứa dữ liệu
- Toán hạng kia là thanh ghi hoặc 1 địa chỉ ô nhớ
- Ví dụ:

LOAD R<sub>1</sub>, 1000;      M[1000] → R<sub>1</sub>

giá trị lưu trong vị trí 1000 ở bộ nhớ được tải vào thanh ghi R<sub>1</sub>



# Chế độ địa chỉ trực tiếp/ tuyệt đối



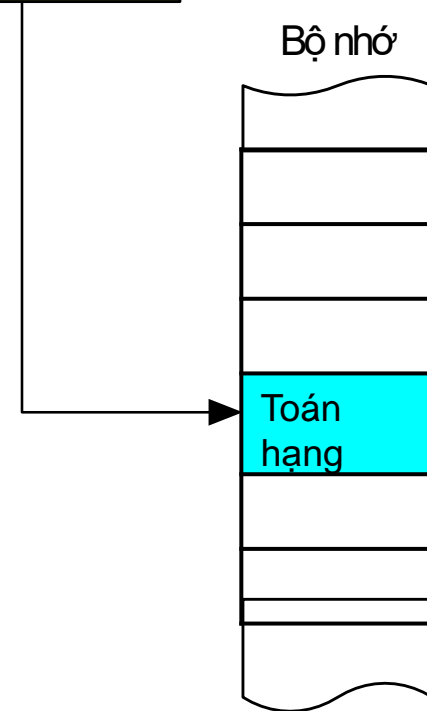
# Chế độ địa chỉ trực tiếp



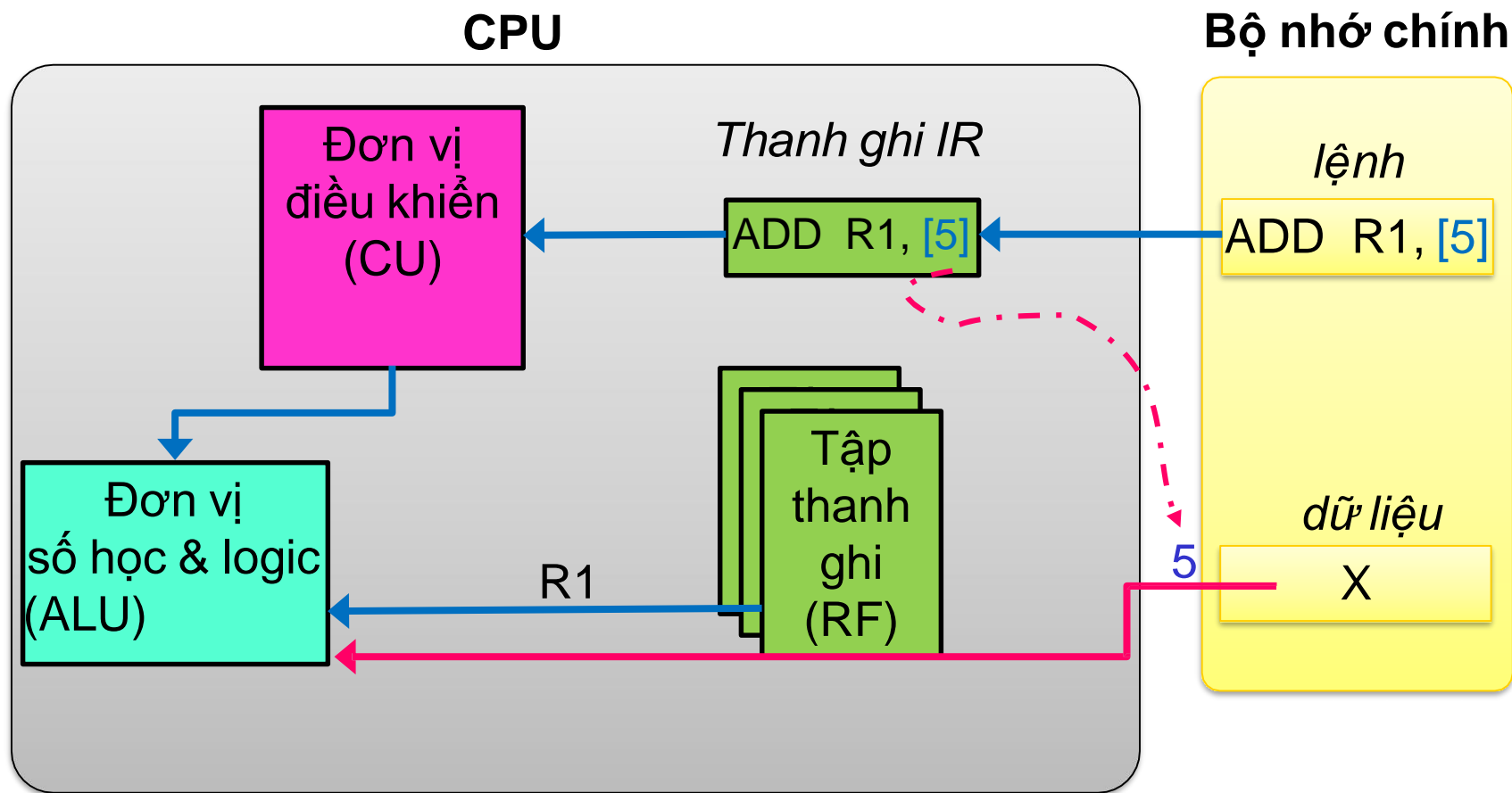
- Toán hạng là ngăn nhớ có địa chỉ được cho trực tiếp trong lệnh
- Ví dụ:

ADD R1, A       $\#R1 = R1 + (A)$

- Cộng nội dung thanh ghi R1 với nội dung của ngăn nhớ có địa chỉ là A
- Tìm toán hạng trong bộ nhớ ở địa chỉ A
- CPU tham chiếu bộ nhớ một lần để truy cập dữ liệu



# Chế độ địa chỉ trực tiếp



# Chế độ địa chỉ gián tiếp

---

- Một thanh ghi hoặc một vị trí trong bộ nhớ được sử dụng để lưu địa chỉ của toán hạng

- Gián tiếp thanh ghi:

$\text{LOAD } R_j, (R_i); \quad M[R_i] \rightarrow R_j$

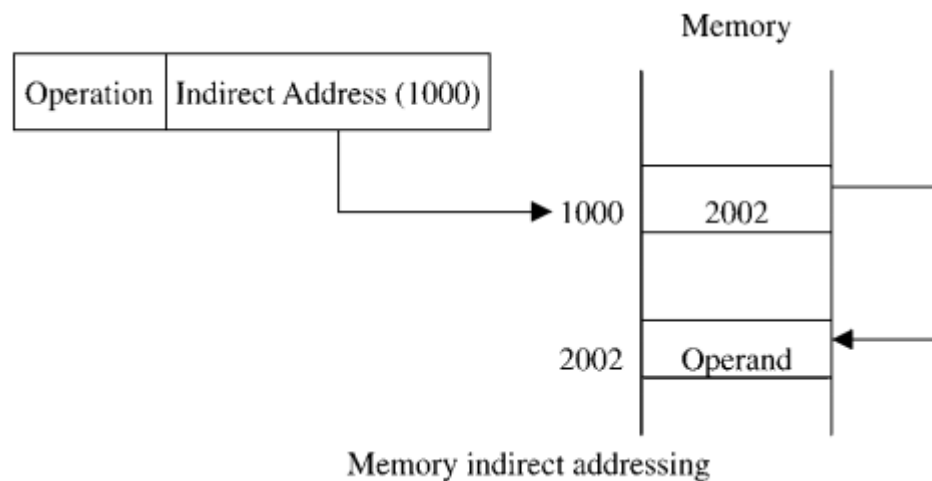
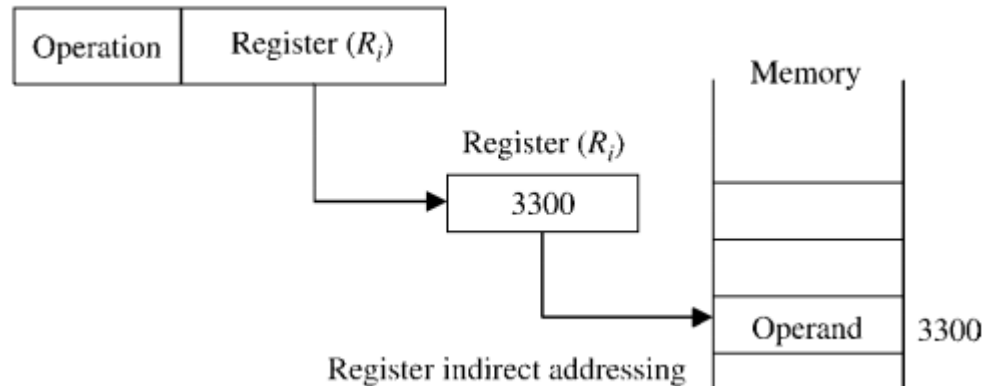
Tải giá trị tại vị trí bộ nhớ có địa chỉ được lưu trong  $R_i$  vào thanh ghi  $R_j$

- Gián tiếp bộ nhớ:

$\text{LOAD } R_i, (1000); \quad M[M[1000]] \rightarrow R_i$

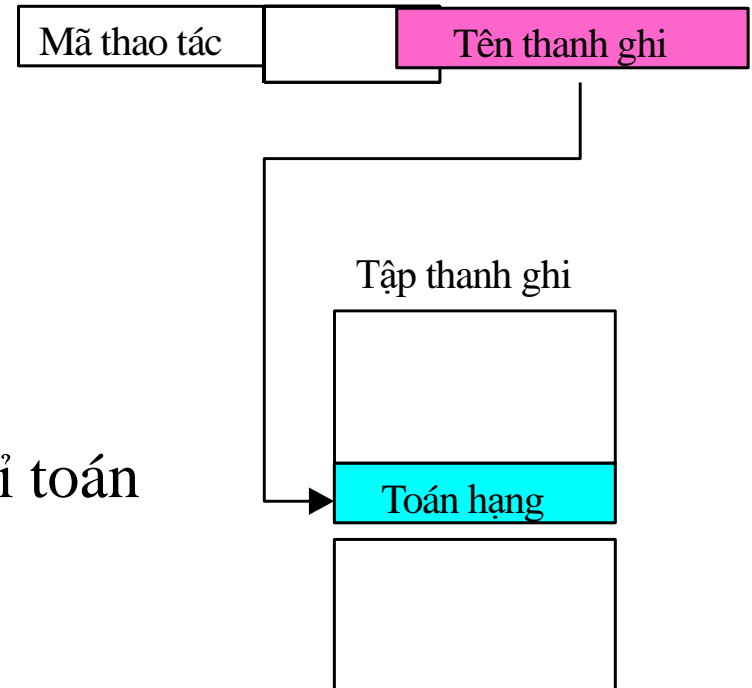
Giá trị của vị trí bộ nhớ có địa chỉ được lưu tại vị trí 1000 vào  $R_i$

# Chế độ địa chỉ gián tiếp



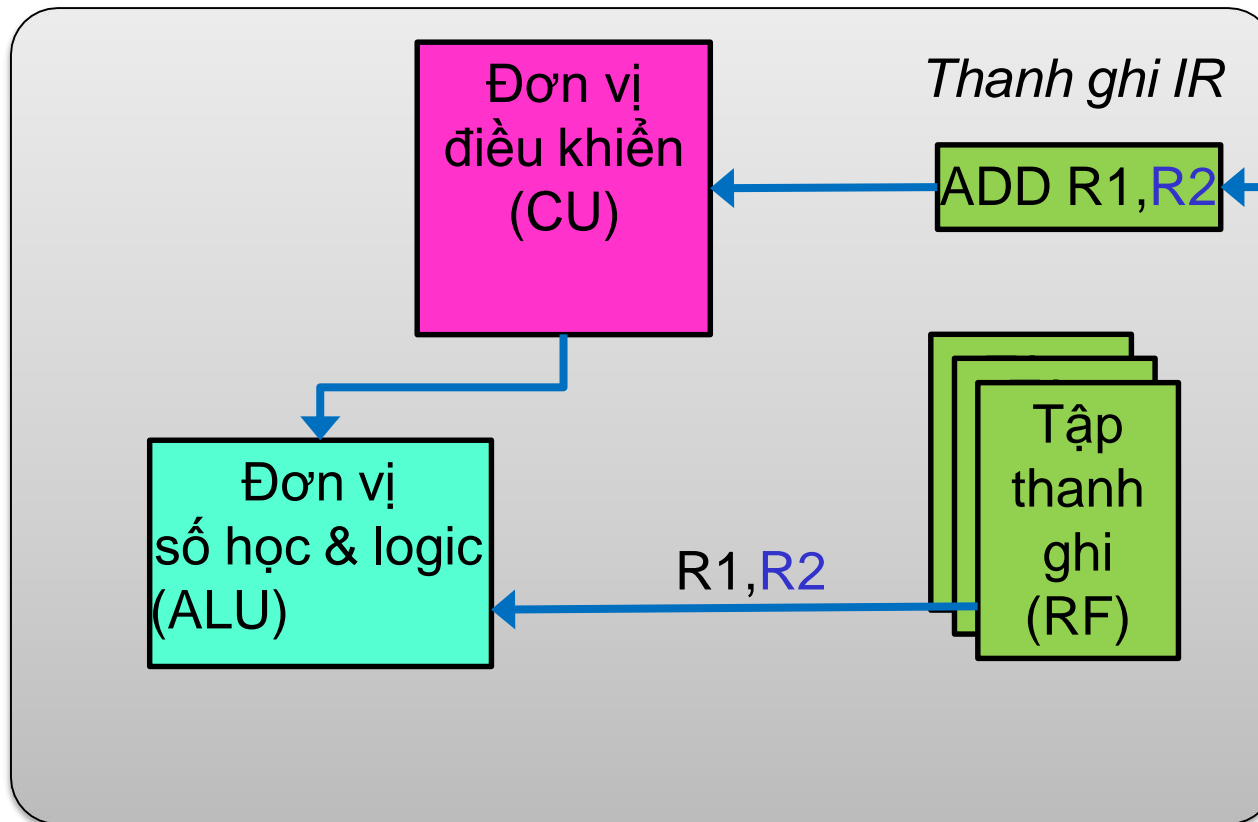
# Định địa chỉ thanh ghi

- Toán hạng nằm trong thanh ghi có tên được chỉ ra trong lệnh
- Ví dụ:  
ADD R1, R2 # R1 = R1 + R2
- Số lượng thanh ghi ít : Trường địa chỉ toán hạng chỉ cần ít bit
- Không tham chiếu bộ nhớ
- Truy nhập toán hạng nhanh
- Tăng số lượng thanh ghi : hiệu quả hơn

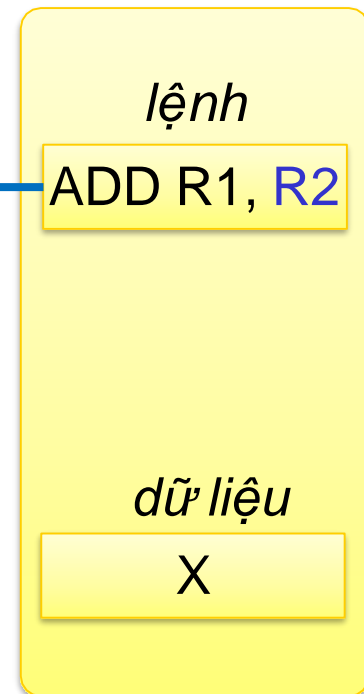


# Chế độ địa chỉ thanh ghi

## CPU

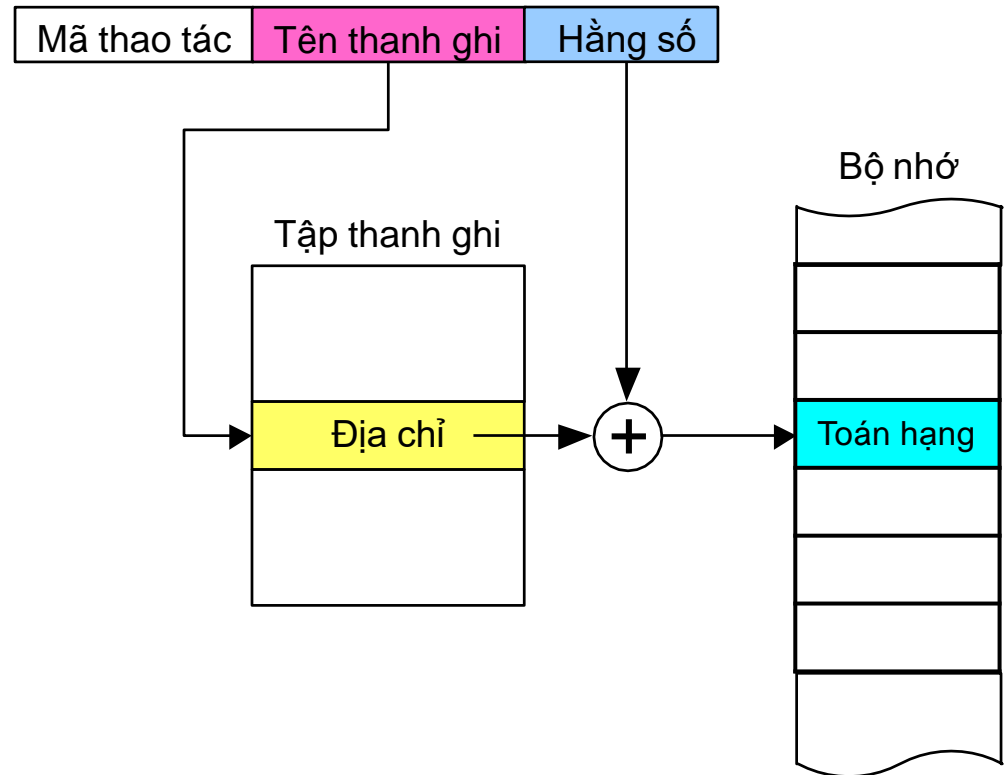


## Bộ nhớ chính



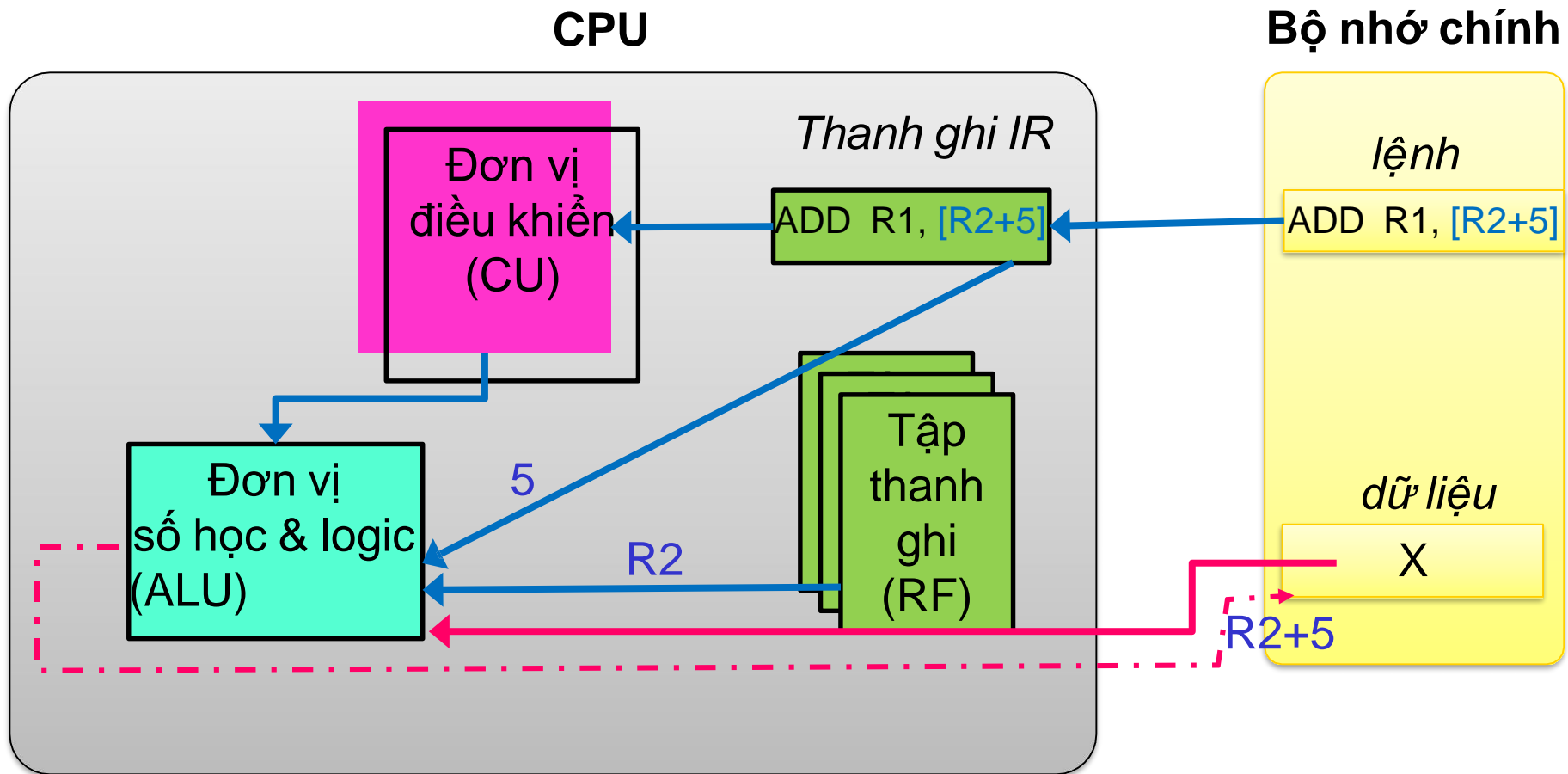
# Chế độ địa chỉ dịch chuyển

- Để xác định toán hạng, Trường địa chỉ chứa hai thành phần:
  - Tên thanh ghi
  - Hằng số (offset)
- Địa chỉ của toán hạng = nội dung thanh ghi + hằng số
- Thanh ghi có thể được ngầm định





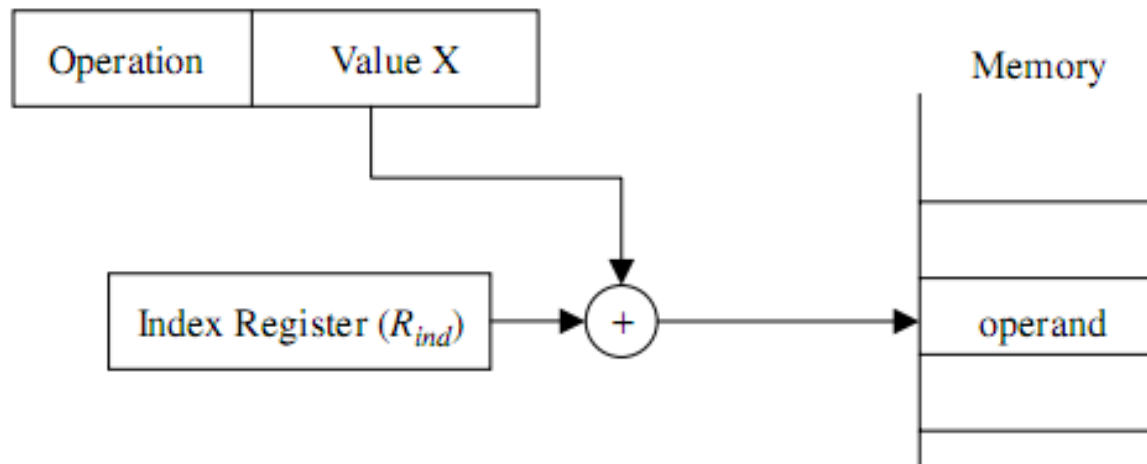
# Chế độ địa chỉ dịch chuyển



# Chế độ địa chỉ chỉ số

- Địa chỉ của toán hạng có được bằng cách cộng thêm hằng số vào nội dung của một thanh ghi, là thanh ghi chỉ số
- Ví dụ

LOAD  $R_i, X(R_{ind}); \quad M[X+R_{ind}] \rightarrow R_i$

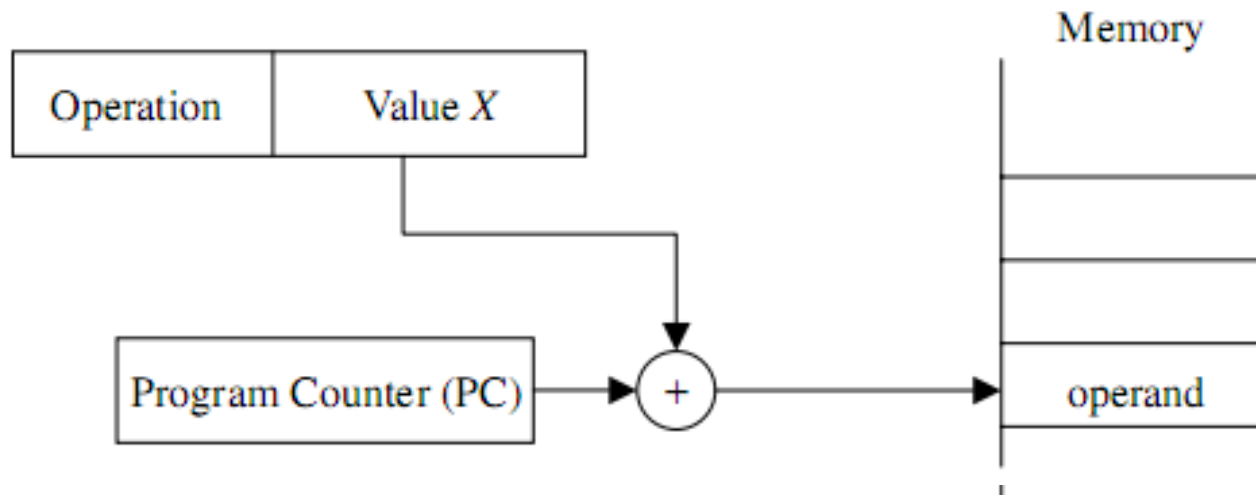


# Chế độ địa chỉ tương đối

- Địa chỉ của toán hạng có được bằng cách cộng thêm hằng số vào nội dung của một thanh ghi, là thanh ghi con đếm chương trình PC

- Ví dụ

LOAD  $R_i, X(PC); M[X+PC] \rightarrow R_i$



# Tổng kết các chế độ địa chỉ

Chế độ địa chỉ	Ý nghĩa	Ví dụ	Thực hiện
Tức thì	Giá trị của toán hạng được chứa trong lệnh	LOAD Ri, #1000	$Ri \leftarrow 1000$
Trực tiếp	Địa chỉ của toán hạng được chứa trong lệnh	LOAD Ri, 1000	$Ri \leftarrow M[1000]$
Gián tiếp thanh ghi	Giá trị của thanh ghi trong lệnh là địa chỉ bộ nhớ chứa toán hạng	LOAD Ri, (Rj)	$Ri \leftarrow M[Rj]$
Gián tiếp bộ nhớ	Địa chỉ bộ nhớ trong lệnh chứa địa chỉ bộ nhớ của toán hạng	LOAD Ri, (1000)	$Ri \leftarrow M[M[1000]]$
Chỉ số	Địa chỉ của toán hạng là tổng của hằng số (trong lệnh) và giá trị của một thanh ghi chỉ số	LOAD Ri, X(Rind)	$Ri \leftarrow M[X + Rind]$
Tương đối	Địa chỉ của toán hạng là tổng của hằng số và giá trị của thanh ghi con đếm chương trình	LOAD Ri, X(PC)	$Ri \leftarrow M[X + PC]$

# Các thành phần của lệnh máy

Mã thao tác	Địa chỉ của các toán hạng	
Opcode	Addresses of Operands	
Opcode	Des addr.	Source addr.

- Mã thao tác (operation code - opcode): mã hóa cho thao tác mà bộ xử lý phải thực hiện
- Địa chỉ toán hạng: chỉ ra nơi chứa các toán hạng mà thao tác sẽ tác động
  - Toán hạng nguồn (source operand): dữ liệu vào của thao tác
  - Toán hạng đích (destination operand): dữ liệu ra của thao tác

# Toán hạng 3 địa chỉ

---

## □ Khuôn dạng:

- opcode addr1, addr2, addr3
- Mỗi địa chỉ addr1, addr2, addr3: tham chiếu tới một ô nhớ hoặc 1 thanh ghi

## □ Ví dụ

1.  $\text{ADD } R_1, R_2, R_3; \quad R_2 + R_3 \rightarrow R_1$   
 $R_2$  cộng  $R_3$  sau đó kết quả đưa vào  $R_1$   
 $R_i$  là các thanh ghi CPU
2.  $\text{ADD } A, B, C; \quad M[B] + M[C] \rightarrow M[A]$   
 $A, B, C$  là các vị trí trong bộ nhớ

# Toán hạng 2 địa chỉ

---

## □ Khuôn dạng:

- opcode addr1, addr2
- Mỗi địa chỉ addr1, addr2: tham chiếu tới 1 thanh ghi hoặc 1 vị trí trong bộ nhớ

## □ Ví dụ

1.  $\text{ADD } R_1, R_2; \quad R_1 + R_2 \rightarrow R_1$   
 $R_1$  cộng  $R_2$  sau đó kết quả đưa vào  $R_1$   
 $R_i$  là các thanh ghi CPU
2.  $\text{ADD } A, B; \quad M[A] + M[B] \rightarrow M[A]$   
 $A, B$  là các vị trí trong bộ nhớ

# Toán hạng 1 địa chỉ

---

## □ Khuôn dạng:

- opcode addr
- addr: tham chiếu tới 1 thanh ghi hoặc 1 vị trí trong bộ nhớ
- Khuôn dạng này sử dụng  $R_{acc}$  (thanh ghi tích lũy) mặc định cho địa chỉ thứ 2

## □ Ví dụ

1.  $ADD R_1; \quad R_1 + R_{acc} \rightarrow R_{acc}$   
 $R_1$  cộng  $R_{acc}$  sau đó kết quả đưa vào  $R_{acc}$   
 $R_i$  là các thanh ghi CPU
2.  $ADD A; \quad M[A] + R_{acc} \rightarrow R_{acc}$   
 $A$  là vị trí trong bộ nhớ



# Toán hạng 1.5 địa chỉ

---

## □ Khuôn dạng:

- opcode addr1, addr2
- Một địa chỉ tham chiếu tới 1 ô nhớ và địa chỉ còn lại tham chiếu tới 1 thanh ghi
- Là dạng hỗn hợp giữa các toán hạng thanh ghi và vị trí bộ nhớ

## □ Ví dụ

1. ADD  $R_1, B$ ;       $M[B] + R_1 \rightarrow R_1$

# Toán hạng 0 địa chỉ

---

- Được thực hiện trong các lệnh mà thực hiện các thao tác ngăn xếp: push & pop

- Ví dụ:

- push a

- push b

- add pop c

- có nghĩa là :  $c = a + b$

# Một số dạng lệnh thông dụng

---

- Các lệnh vận chuyển dữ liệu
- Các lệnh số học và logic
- Các lệnh điều khiển chương trình
- Các lệnh vào/ ra

# Lệnh vận chuyển dữ liệu

---

- Chuyển dữ liệu giữa các phần của máy tính
  - Giữa các thanh ghi trong CPU  
MOVE Ri, Rj ; Rj -> Ri
  - Giữa thanh ghi CPU và một vị trí trong bộ nhớ  
MOVE Rj, 1000; M[1000] -> Rj
  - Giữa các vị trí trong bộ nhớ  
MOVE 1000, (Rj) ; M[Rj] -> M[1000]

# Một số lệnh vận chuyển dữ liệu thông dụng

- ❑ MOVE: chuyển dữ liệu giữa thanh ghi – thanh ghi, ô nhớ - thanh ghi, ô nhớ - ô nhớ
- ❑ LOAD: nạp nội dung 1 ô nhớ vào 1 thanh ghi
- ❑ STORE: lưu nội dung 1 thanh ghi ra 1 ô nhớ
- ❑ PUSH: đẩy dữ liệu vào ngăn xếp
- ❑ POP: lấy dữ liệu ra khỏi ngăn xếp

# Lệnh số học và logic

---

- Thực hiện các thao tác số học và logic giữa các thanh ghi và nội dung ô nhớ

- Ví dụ:

ADD R1, R2, R3;  $R2 + R3 \rightarrow R1$

SUBTRACT R1, R2, R3;  $R2 - R3 \rightarrow R1$

# Các lệnh tính toán số học thông dụng

---

- ADD: cộng 2 toán hạng
- SUBTRACT: trừ 2 toán hạng
- MULTIPLY: nhân 2 toán hạng
- DIVIDE: chia số học
- INCREMENT: tăng 1
- DECREMENT: giảm 1

# Các lệnh logic thông dụng

---

- NOT: phủ định
- AND: và
- OR: hoặc
- XOR: hoặc loại trừ
- COMPARE: so sánh
- SHIFT: dịch
- ROTATE: quay



# Lệnh điều khiển/ tuần tự

---

- Được dùng để thay đổi trình tự các lệnh được thực hiện:
  - Các lệnh rẽ nhánh (nhảy) có điều kiện (conditional branching/ jump)
  - Các lệnh rẽ nhánh (nhảy) không điều kiện (unconditional branching/ jump)
  - CALL và RETURN: lệnh gọi thực hiện và trở về từ chương trình con
- Đặc tính chung của các lệnh này là quá trình thực hiện lệnh của chúng làm thay đổi giá trị PC
- Sử dụng các cờ ALU để xác định các điều kiện

# Một số lệnh điều khiển thông dụng

---

- ❑ **BRANCH – IF – CONDITION:** chuyển đến thực hiện lệnh ở địa chỉ mới nếu điều kiện là đúng
- ❑ **JUMP:** chuyển đến thực hiện lệnh ở địa chỉ mới
- ❑ **CALL:** chuyển đến thực hiện chương trình con
- ❑ **RETURN:** trở về (từ chương trình con) thực hiện tiếp chương trình gọi

# Một số lệnh điều khiển thông dụng

---

LOAD R1, #100

LAP:

ADD R0, (R2)

DECREMENT R1

BRANCH\_IF >0 LAP

# Các lệnh vào/ ra

---

- ❑ Được dùng để truyền dữ liệu giữa máy tính và các thiết bị ngoại vi
- ❑ Các thiết bị ngoại vi giao tiếp với máy tính thông qua các cổng. Mỗi cổng có một địa chỉ dành riêng
- ❑ Hai lệnh I/O cơ bản được sử dụng là các lệnh INPUT và OUTPUT
  - Lệnh INPUT được dùng để chuyển dữ liệu từ thiết bị ngoại vi vào tới bộ vi xử lý
  - Lệnh OUTPUT dùng để chuyển dữ liệu từ VXL ra thiết bị đầu ra

# Các ví dụ

---

CLEAR R0;

MOVE R1, #100;

CLEAR R2;

LAP:

ADD R0, 1000(R2);

INCREMENT R2;

DECREMENT R1;

BRANCH\_IF>0 LAP;

STORE 2000, R0;

# Các ví dụ

---

CLEAR R0;

$R0 \leftarrow 0$

MOVE R1, #100;

$R1 \leftarrow 100$

CLEAR R2;

$R2 \leftarrow 0$

LAP:

ADD R0, 1000(R2);

$R0 \leftarrow R0 + M[R2 + 1000]$

INCREMENT R2;

$R2 \leftarrow R2 + 1$

DECREMENT R1;

$R1 \leftarrow R1 - 1$

BRANCH\_IF>0 LAP;

go to LAP if  $R1 > 0$

STORE 2000, R0;

$M[2000] \leftarrow R0$

# BÀI TẬP

---

1. Cho đoạn lệnh sau:

ADD R2, (R0);

SUBTRACT R2, (R1);

MOVE 500(R0), R2;

LOAD R2, #5000;

STORE 100(R2), R0;

Biết  $R0=1500$ ,  $R1=4500$ ,  $R2=1000$ ,  $M[1500]=3000$ ,  $M[4500]=500$

a. Chỉ rõ chế độ địa chỉ của từng lệnh

b. Hãy chỉ ra giá trị của thanh ghi và tại vị trí trong bộ nhớ qua mỗi lệnh thực hiện.

# BÀI TẬP

---

2. Cho đoạn lệnh sau:

MOVE R0, #100;

CLEAR R1;

CLEAR R2;

LAP:

ADD R1, 2000(R2);

ADD R2, #2;

DECREMENT R0;

BRANCH\_IF>0 LAP;

STORE 3000, R1;

- a. Hãy giải thích ý nghĩa của từng lệnh
- b. Chỉ ra chế độ địa chỉ của từng lệnh (đối với các lệnh có 2 toán hạng)
- c. Đoạn lệnh trên thực hiện công việc gì?



# BÀI TẬP

---

- Cho một mảng gồm 10 số, được lưu trữ liên tiếp nhau trong bộ nhớ, bắt đầu từ vị trí ô nhớ 1000. Viết đoạn chương trình tính tổng các số dương trong mảng đó và lưu kết quả vào ô nhớ 2000.

# CISC và RISC

---

- CISC: Complex Instruction Set Computer:
  - Máy tính với tập lệnh phức tạp
  - Các bộ xử lý truyền thống: Intel x86, Motorola 680x0
- RISC: Reduced Instruction Set Computer:
  - Máy tính với tập lệnh thu gọn
  - SunSPARC, Power PC, MIPS, ARM ...
  - RISC đối nghịch với CISC
  - Kiến trúc tập lệnh tiên tiến

# Các đặc trưng của RISC

---

- Số lượng lệnh ít
- Hầu hết các lệnh truy nhập toán hạng ở các thanh ghi
- Truy nhập bộ nhớ bằng các lệnh LOAD/STORE
- Thời gian thực hiện lệnh là một chu kỳ máy
- Các lệnh có độ dài cố định (32 bit)
- Số lượng dạng lệnh ít ( $\leq 4$ )
- CPU có tập thanh ghi lớn
- Có ít phương pháp định địa chỉ toán hạng ( $\leq 4$ )
- Hỗ trợ các thao tác của ngôn ngữ bậc cao

# Kiến trúc tập lệnh MIPS

---

- MIPS- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
- Được phát triển ở đại học Stanford, sau đó được thương mại hóa bởi công ty MIPS Technologies
- Năm 2012 : MIPS Technologies được bán cho Imagination Technologies (imgteh.com)
- Kiến trúc RISC
- Chiếm thị phần lớn trong các sản phẩm nhúng
- Điển hình cho nhiều kiến trúc tập lệnh hiện đại

Các phần tiếp theo trong chương này sẽ nghiên cứu kiến trúc tập lệnh MIPS 32-bit

*Tài liệu: MIPS Reference Data Sheet và Chapter 2 – COD*

□ [MIPS Data sheet Doc](#)

# Phép hợp ngữ và các toán hạng

---

- Thực hiện phép cộng: 3 toán hạng
  - Là phép toán phổ biến nhất
  - Hai toán hạng nguồn và một toán hạng đích

**add a, b, c**

**# a = b + c**

- Hầu hết các lệnh số học/logic có dạng trên
- Các lệnh số học sử dụng toán hạng thanh ghi hoặc hằng số

# Tập thanh ghi của MIPS

---

- MIPS có tập 32 thanh ghi 32-bit
  - Được sử dụng thường xuyên
  - Được đánh số từ 0 đến 31 (mã hóa bằng 5-bit)
- Chương trình hợp dịch Assembler đặt tên:
  - Bắt đầu bằng dấu \$
  - \$t0, \$t1, ..., \$t9 chứa các giá trị tạm thời
  - \$s0, \$s1, ..., \$s7 cất các biến
- Qui ước gọi dữ liệu trong MIPS:
  - Dữ liệu 32-bit được gọi là “word”
  - Dữ liệu 16-bit được gọi là “halfword”

# Tập thanh ghi của MIPS

Tên thanh ghi	Số hiệu thanh ghi	Công dụng
\$zero	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2-3	procedure return values
\$a0-\$a3	4-7	procedure arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved variables
\$t8-\$t9	24-25	more temporaries
\$k0-\$k1	26-27	OS temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	procedure return address

# Toán hạng thanh ghi

---

- Lệnh add, lệnh sub (subtract) chỉ thao tác với toán hạng thanh ghi

- add \$rd, \$rs, \$rt      # (rd) = (rs)+(rt)

- sub \$rd, \$rs, \$rt      # (rd) = (rs)-(rt)

- Ví dụ mã C:

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

- giả thiết: f, g, h, i, j nằm ở \$s0, \$s1, \$s2, \$s3, \$s4

- Được dịch thành mã hợp ngữ MIPS:

add \$t0, \$s1, \$s2      # \$t0 = \$s1 + \$s2 = g + h

add \$t1, \$s3, \$s4      # \$t1 = \$s3 + \$s4 = i + j

sub \$s0, \$t0, \$t1      # \$s0 = \$t0 - \$t1 = f = (g+h)-(i+j)



# Toán hạng ở bộ nhớ

---

- Muốn thực hiện phép toán số học với toán hạng ở bộ nhớ, cần phải:
  - Nạp (load) giá trị từ bộ nhớ vào thanh ghi
  - Thực hiện phép toán trên thanh ghi
  - Lưu (store) kết quả từ thanh ghi ra bộ nhớ
- Bộ nhớ được đánh địa chỉ theo byte
  - MIPS sử dụng 32-bit để đánh địa chỉ cho các byte nhớ và các cổng vào-ra
  - Không gian địa chỉ: **0x00000000 – 0xFFFFFFFF**
  - Mỗi word có độ dài 32-bit chiếm 4-byte trong bộ nhớ, địa chỉ của các word là bội của 4 (địa chỉ của byte đầu tiên)
- MIPS cho phép lưu trữ trong bộ nhớ theo kiểu đầu to (big-endian) hoặc kiểu đầu nhỏ (little-endian)

# Địa chỉ byte nhớ và word nhớ

Dữ liệu hoặc lệnh	Địa chỉ byte (theo Hexa)
byte (8-bit)	0x0000 0000
byte	0x0000 0001
byte	0x0000 0002
byte	0x0000 0003
byte	0x0000 0004
byte	0x0000 0005
byte	0x0000 0006
byte	0x0000 0007
.	
.	
.	
byte	0xFFFF FFFB
byte	0xFFFF FFFC
byte	0xFFFF FFDD
byte	0xFFFF FFDE
byte	0xFFFF FFFF

$2^{32}$  bytes

Dữ liệu hoặc lệnh	Địa chỉ word (theo Hexa)
word (32-bit)	0x0000 0000
word	0x0000 0004
word	0x0000 0008
word	0x0000 000C
word	0x0000 0010
word	0x0000 0014
word	0x0000 0018
.	
.	
.	
word	0xFFFF FFF4
word	0xFFFF FFF8
word	0xFFFF FFFC

$2^{30}$  words

# Lệnh load và lệnh store

---

- Để đọc word dữ liệu 32-bit từ bộ nhớ đưa vào thanh ghi, sử dụng lệnh *load word*

*lw rt, imm(rs)      # (rt) = mem[(rs)+imm]*

- rs: thanh ghi chứa địa chỉ cơ sở (base address)
- imm (immediate): hằng số (offset)
- địa chỉ của word dữ liệu cần đọc = địa chỉ cơ sở + hằng số
- rt: thanh ghi đích, chứa word dữ liệu được đọc vào

- Để ghi word dữ liệu 32-bit từ thanh ghi đưa ra bộ nhớ sử dụng lệnh *store word*

*sw rt, imm(rs)      # mem[(rs)+imm] =(rt)*

- rt: thanh ghi nguồn, chứa word dữ liệu cần ghi ra bộ nhớ
- rs: thanh ghi chứa địa chỉ cơ sở (base address)
- imm: hằng số (offset)
- địa chỉ nơi ghi chỉ cơ sở + hằng số

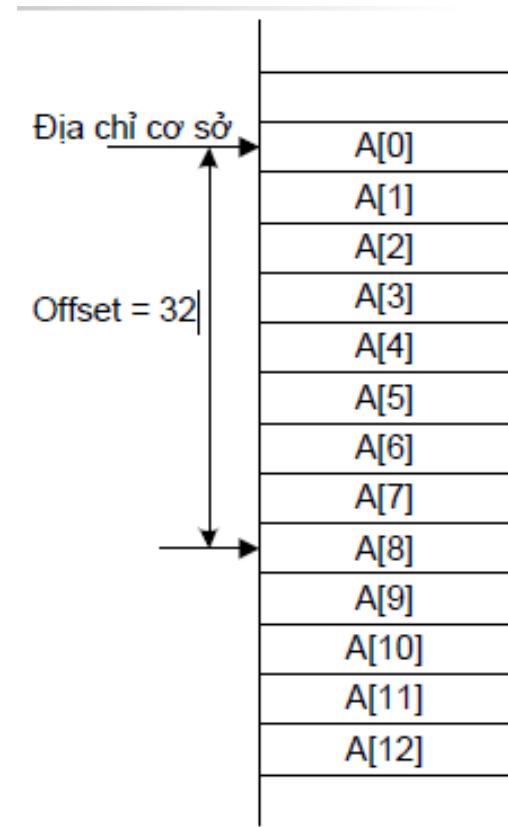
# Ví dụ toán hạng bộ nhớ

## ■ Mã C:

// A là mảng các phần tử 32-bit

$g = h + A[8];$

- gở \$s1, hở \$s2,
- \$s3 chứa địa chỉ cơ sở của mảng A



# Ví dụ toán hạng bộ nhớ

## ■ Mã C:

// A là mảng các phần tử 32-bit

$g = h + A[8];$

- $g$  ở  $\$s1$ ,  $h$  ở  $\$s2$ ,

- $\$s3$  chứa địa chỉ cơ sở của mảng  $A$

## ■ Mã hợp ngữ MIPS:

- # Chỉ số 8, do đó offset = 32

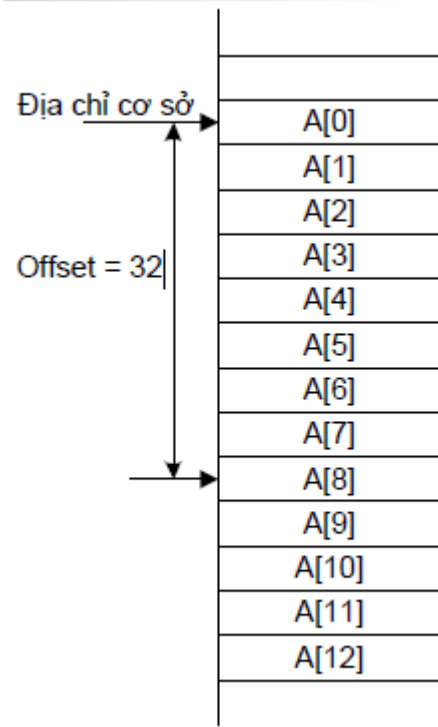
lw  $\$t0, 32(\$s3)$  # load word  $A[8]$

add  $\$s1, \$s2, \$t0$  #  $g = h + A[8]$

offset

base register

(Chú  $\therefore$  offset phải là hằng số, có thể dương hoặc âm)

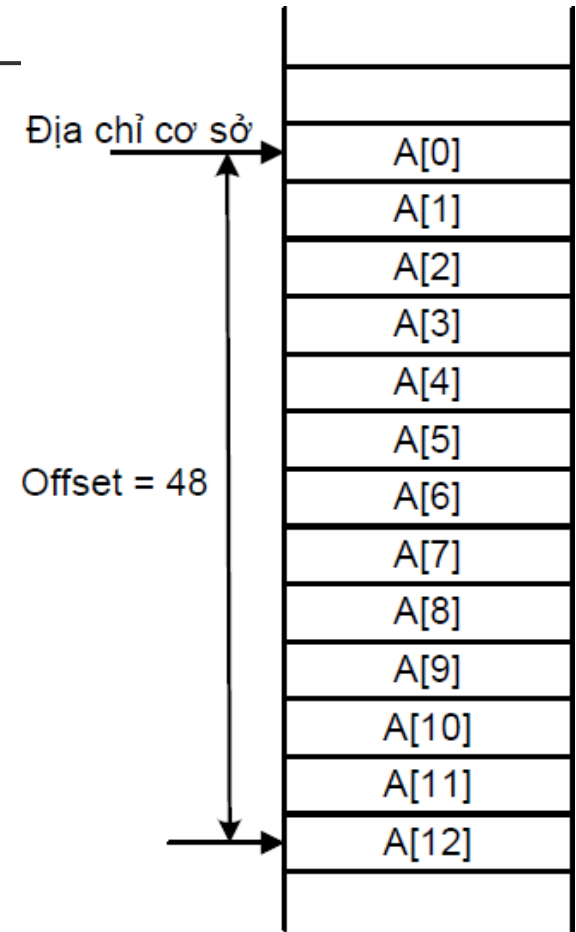


# Ví dụ toán hạng bộ nhớ (tiếp)

- Mã C:

$A[12] = h + A[8];$

- $h$  ở  $\$s2$ ,
- $\$s3$  chứa địa chỉ cơ sở của mảng  $A$



# Ví dụ toán hạng bộ nhớ (tiếp)

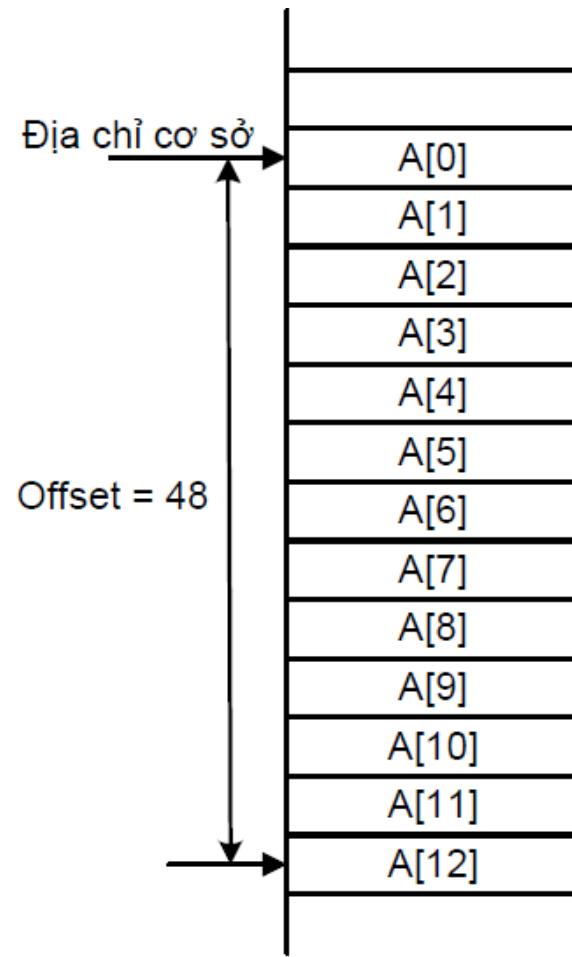
## ■ Mã C:

$A[12] = h + A[8];$

- $h$  ở  $\$s2$ ,
- $\$s3$  chứa địa chỉ cơ sở của mảng  $A$

## ■ Mã hợp ngữ MIPS:

```
1 lw $t0, 32($s3)    # t0 = A[8]
2 add $t0, $s2, $t0   # t0 = h + A[8]
3 sw $t0, 48($s3),    # A[12] = h + A[8]
```



# Thanh ghi với Bộ nhớ

---

- Truy nhập thanh ghi nhanh hơn bộ nhớ
- Thao tác dữ liệu trên bộ nhớ yêu cầu nạp (load) và lưu (store).
  - Cần thực hiện nhiều lệnh hơn
- Chương trình dịch sử dụng các thanh ghi cho các biến nhiều nhất có thể
  - Chỉ sử dụng bộ nhớ cho các biến ít được sử dụng
  - Cần tối ưu hóa sử dụng thanh ghi



# Toán hạng tức thì (immediate)

---

- Dữ liệu hằng số được xác định ngay trong lệnh

`addi $s3, $s3, 4`                      `# $s3 = $s3+4`

- Không có lệnh trừ (subi) với giá trị tức thì

- Sử dụng hằng số âm để thực hiện phép trừ

`addi $s2, $s1, -1`                      `# $s2 = $s1-1`

# Xử lý với số nguyên

- Số nguyên có dấu (biểu diễn bằng bù hai):
  - Với  $n$  bit, dải biểu diễn:  $[-2^{n-1}, +(2^{n-1}-1)]$
  - Các lệnh **add**, **sub**, **addi** dành cho số nguyên có dấu
- Số nguyên không dấu:
  - Với  $n$  bit, dải biểu diễn:  $[0, 2^n - 1]$
  - Các lệnh **addu**, **subu** dành cho số nguyên không dấu
- Quy ước biểu diễn hằng số nguyên trong hợp ngữ MIPS:
  - số thập phân: 12; 3456; -18
  - số Hexa (bắt đầu bằng **0x**): 0x12; 0x3456; 0x1AB6

# Hằng số Zero

---

- Thanh ghi 0 của MIPS (\$zero hay \$0) luôn chứa hằng số 0
  - Không thể thay đổi giá trị
- Hữu ích cho một số thao tác thông dụng
  - Chẳng hạn, chuyển dữ liệu giữa các thanh ghi  
`add $t2, $s1, $zero`                      #  $\$t2 = \$s1$

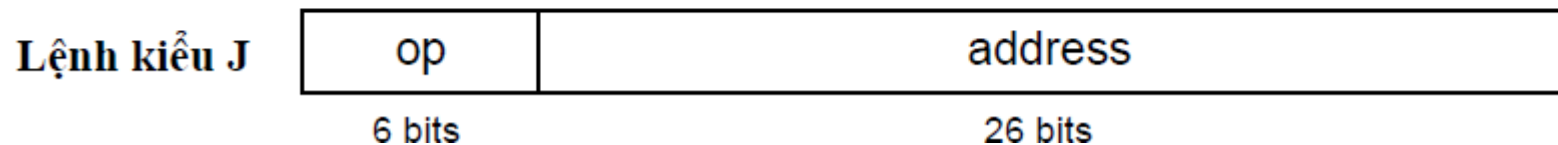
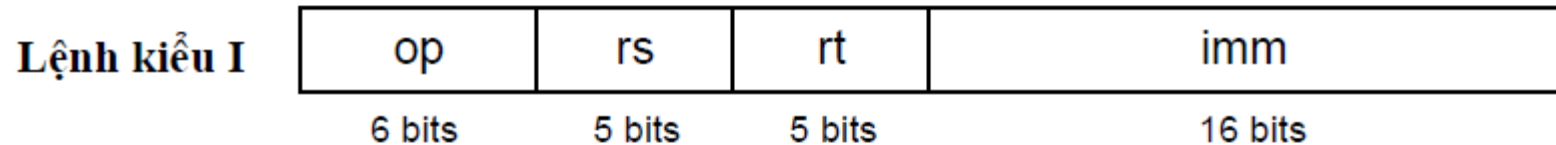
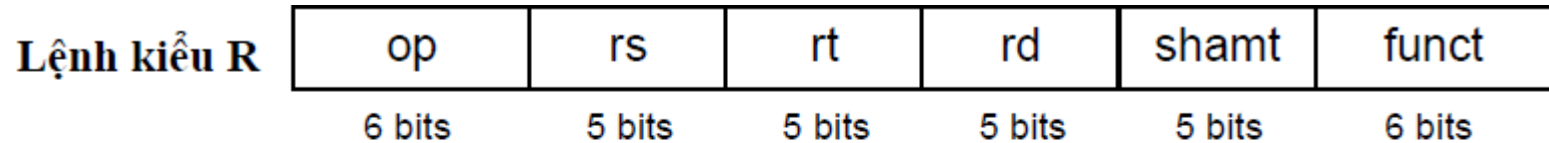
# Mã máy (Machine code)

---

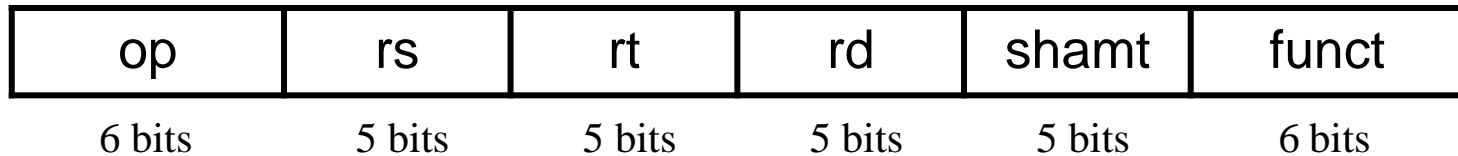
- Các lệnh được mã hóa dưới dạng nhị phân được gọi là mã máy
- Các lệnh của MIPS:
  - Được mã hóa bằng các từ lệnh 32-bit
  - Mỗi lệnh chiếm 4-byte trong bộ nhớ, do vậy địa chỉ của lệnh trong bộ nhớ là bội của 4
  - Có ít dạng lệnh
- Số hiệu thanh ghi
  - $\$t0 - \$t7$  là các thanh ghi 8 – 15
  - $\$t8 - \$t9$  là các thanh ghi 24 – 25
  - $\$s0 - \$s7$  là các thanh ghi 16 – 23

# Các dạng lệnh của MIPS

---



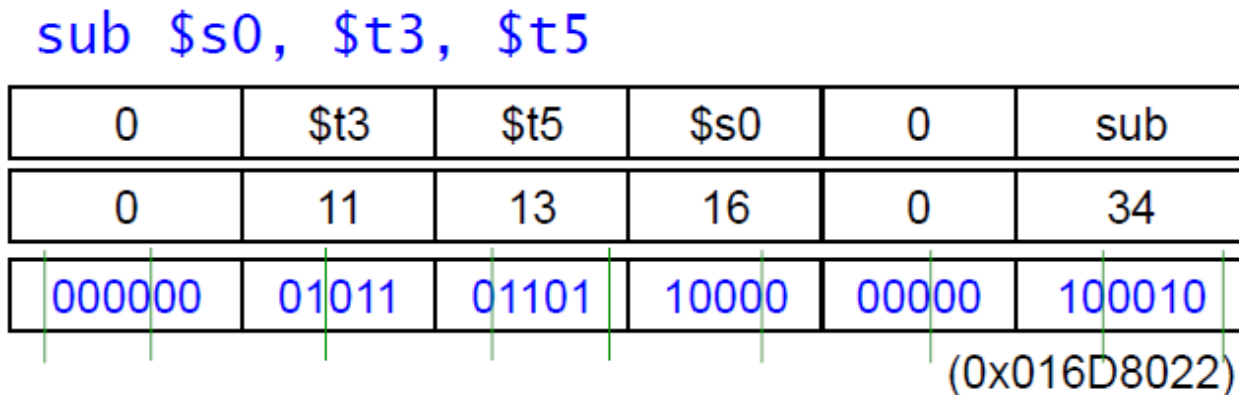
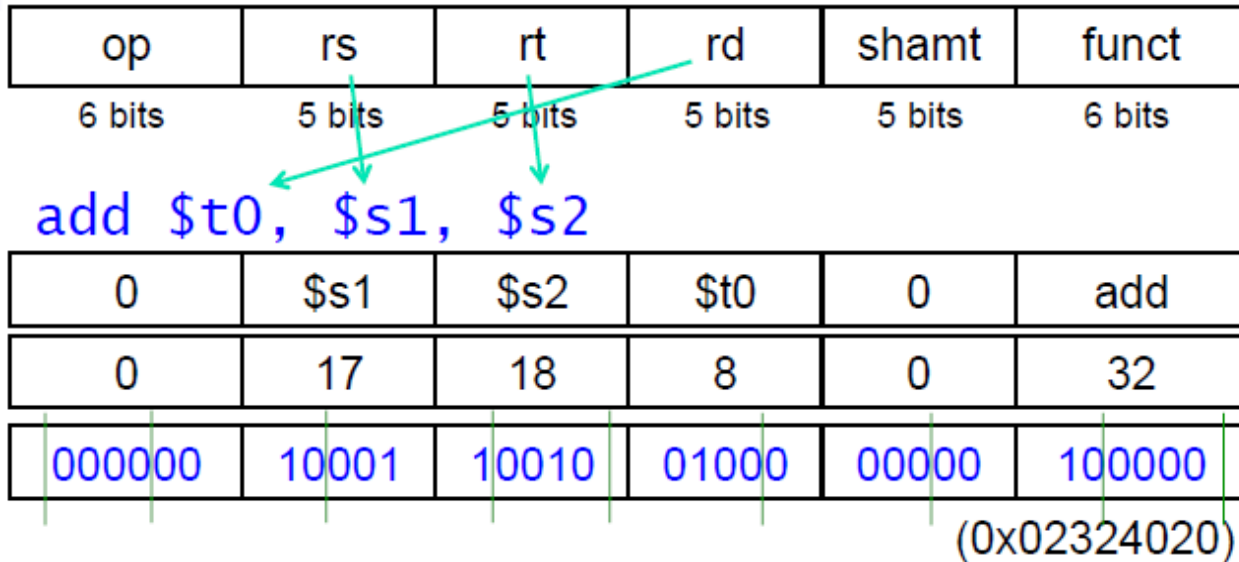
# Lệnh dạng R (Register)



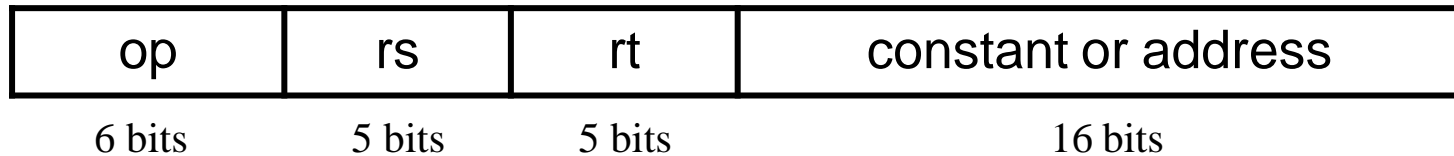
## ■ Các trường của lệnh

- op: operation code (opcode): mã thao tác
  - với các lệnh kiểu R, op = 000000
- rs: số hiệu thanh ghi nguồn thứ nhất
- rt: số hiệu thanh ghi nguồn thứ hai
- rd: số hiệu thanh ghi đích
- shamt (shift amount): số bit được dịch, chỉ dùng cho lệnh dịch bit, với các lệnh khác shamt = 00000
- funct: function code (extends opcode): mã hàm

# Ví dụ mã máy của lệnh add, sub



# Lệnh dạng I (Immediate)



- Dùng cho các lệnh số học với toán hạng tức thì và các lệnh **load/store** ( nạp/lưu)

- rs: số hiệu thanh ghi nguồn (addi) hoặc thanh ghi cơ sở (lw, sw)
- rt: số hiệu thanh ghi đích (addi, lw) hoặc thanh ghi nguồn (sw)
- imm (immediate): hằng số nguyên 16-bit

addi rt, rs, imm # (rt) = (rs)+SignExtImm

lw rt, imm(rs) # (rt) = mem[(rs)+SignExtImm]

sw rt, imm(rs) # mem[(rs)+SignExtImm] = (rt)

(SignExtImm: hằng số imm 16-bit được mở rộng theo kiểu số có dấu thành 32-bit)



# Mở rộng bit cho hằng số theo số có dấu

- Với các lệnh addi, lw, sw cần cộng nội dung thanh ghi với hằng số:
  - Thanh ghi có độ dài 32-bit
  - Hằng số imm 16-bit, cần mở rộng thành 32-bit theo kiểu số có dấu (Sign-extended)
- Ví dụ mở rộng số 16-bit thành 32-bit theo kiểu số có dấu:

+5 = 

0000	0000	0000	0101
------	------	------	------

 16-bit

+5 = 

0000	0000	0000	0000	0000	0000	0000	0101
------	------	------	------	------	------	------	------

 32-bit

-12 = 

1111	1111	1111	0100
------	------	------	------

 16-bit

-12 = 

1111	1111	1111	1111	1111	1111	1111	0100
------	------	------	------	------	------	------	------

 32-bit

# Ví dụ mã máy của lệnh addi

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
addi \$s0, \$s1, 5			
8	\$s1	\$s0	5
8	17	16	5
001000	10001	10000	0000 0000 0000 0101

(0x22300005)

addi \$t1, \$s2, -12			
8	\$s2	\$t1	-12
8	18	9	-12
001000	10010	01001	1111 1111 1111 0100

(0x2249FFF4)

# Ví dụ mã máy lệnh load và lệnh store

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
lw \$t0, 32(\$s3)			
35	\$s3	\$t0	32
35	19	8	32
100011	10011	01000	0000 0000 0010 0000
(0x8E680020)			

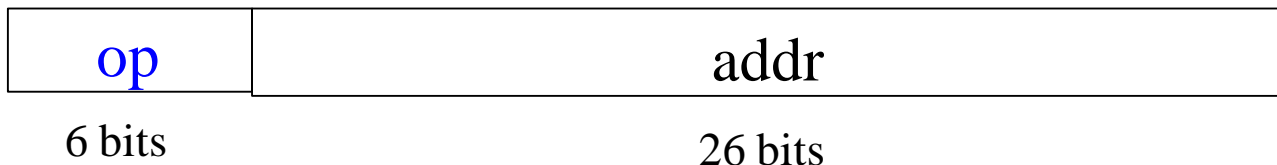
sw \$s1, 4(\$t1)			
43	\$t1	\$s1	4
43	9	17	4
101011	01001	10001	0000 0000 0000 0100
(0xAD310004)			

# Lệnh kiểu J (Jump)

---

- Toán hạng 26-bit địa chỉ
- Được sử dụng cho các lệnh nhảy
  - j (jump): opcode = 000010
  - jal (jump and link): opcode = 000011

## J-Type



# Cơ bản về lập trình hợp ngữ

---

1. Các lệnh logic
2. Nạp hằng số vào thanh ghi
3. Tạo các cấu trúc điều khiển
4. Lập trình mảng dữ liệu
5. Chương trình con
6. Dữ liệu ký tự
7. Lệnh nhân và lệnh chia
8. Các lệnh với số dấu phẩy động

# Các lệnh logic

- Các lệnh logic để thao tác trên các bit của dữ liệu

Phép toán logic	Toán tử trong C	Toán tử trong Java	Lệnh của MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR			or, ori
Bitwise XOR	^	^	xor, xori
Bitwise NOT	~	~	nor

# Ví dụ lệnh logic kiểu R

Nội dung các thanh ghi nguồn

\$s1	0100	0110	1010	0001	1100	0000	1011	0111
\$s2	1111	1111	1111	1111	0000	0000	0000	0000

Mã hợp ngữ

Kết quả thanh ghi đích

and \$s3, \$s1, \$s2

\$s3

--	--	--	--	--	--	--	--

or \$s4, \$s1, \$s2

\$s4

--	--	--	--	--	--	--	--

xor \$s5, \$s1, \$s2

\$s5

--	--	--	--	--	--	--	--

nor \$s6, \$s1, \$s2

\$s6

--	--	--	--	--	--	--	--

# Ví dụ lệnh logic kiểu R

## Nội dung các thanh ghi nguồn

\$s1	0100	0110	1010	0001	1100	0000	1011	0111
\$s2	1111	1111	1111	1111	0000	0000	0000	0000

## Mã hợp ngữ

## Kết quả thanh ghi đích

and \$s3, \$s1, \$s2

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
------	------	------	------	------	------	------	------	------

or \$s4, \$s1, \$s2

\$s4	1111	1111	1111	1111	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

xor \$s5, \$s1, \$s2

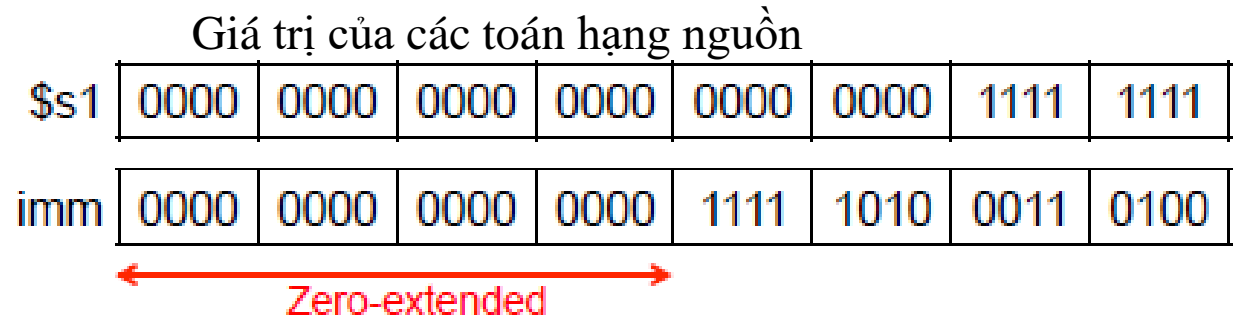
\$s5	1011	1001	0101	1110	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

nor \$s6, \$s1, \$s2

\$s6	0000	0000	0000	0000	0011	1111	0100	1000
------	------	------	------	------	------	------	------	------



# Ví dụ lệnh logic kiểu I




Mã hợp ngữ			Kết quả thanh ghi đích							
andi	\$s2,\$s1,0xFA34	\$s2								
ori	\$s3,\$s1,0xFA34	\$s3								
xori	\$s4,\$s1,0xFA34	\$s4								

*Chú ý:* Với các lệnh logic kiểu I, hằng số imm 16-bit được mở rộng thành 32-bit theo số không dấu (zero-extended)

# Ví dụ lệnh logic kiểu I

Giá trị của các toán hạng nguồn

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100
								

Mã hợp ngữ

Kết quả thanh ghi đích

andi	\$s2,\$s1,0xFA34	\$s2	0000	0000	0000	0000	0000	0000	0011	0100
ori	\$s3,\$s1,0xFA34	\$s3	0000	0000	0000	0000	1111	1010	1111	1111
xori	\$s4,\$s1,0xFA34	\$s4	0000	0000	0000	0000	1111	1010	1100	1011

# Ý nghĩa của các phép toán logic

---

- Phép AND dùng để giữ nguyên một số bit trong word, xóa các bit khác về 0
- Phép OR dùng để giữ nguyên một số bit trong word, thiết lập các bit còn lại lên 1
- Phép XOR dùng để giữ nguyên một số bit trong word, đảo giá trị các bit còn lại
- Phép NOT dùng để đảo các bit trong word
  - Đổi 0 thành 1, và đổi 1 thành 0
  - MIPS không có lệnh NOT, nhưng có lệnh NOR với 3 toán hạng
  - $a \text{ NOR } b == \text{NOT} ( a \text{ OR } b )$

`nor $t0, $t1, $zero`

`# $t0 = not ($t1)`

# Lệnh logic dịch bit

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *shamt*: chỉ ra dịch bao nhiêu vị trí (shift amount)
- *rs*: không sử dụng, thiết lập = 00000
- Thanh ghi đích *rd* nhận giá trị thanh ghi nguồn *rt* đã được dịch trái hoặc dịch phải, *rt* không thay đổi nội dung
- sll - shift left logical (dịch trái logic)
  - Dịch trái các bit và điền các bit 0 vào bên phải
  - Dịch trái  $i$  bits là nhân với  $2^i$  (nếu kết quả trong phạm vi biểu diễn 32-bit)
- srl - shift right logical (dịch phải logic)
  - Dịch phải các bit và điền các bit 0 vào bên trái
  - Dịch phải  $i$  bits là chia cho  $2^i$  (chỉ với số nguyên không dấu)

# Ví dụ lệnh dịch trái sll

- Lệnh hợp ngữ:

sll \$t2, \$s0, 4 # \$t2 = \$s0 << 4

- Mã máy

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

(0x00105100)

- Ví dụ kết quả thực hiện lệnh

\$s0	0000	0000	0000	0000	0000	0000	0000	1101	= 13
\$t2	0000	0000	0000	0000	0000	0000	1101	0000	= 208 (13x16)

*Chú ý: Nội dung thanh ghi \$s0 không bị thay đổi*

# Ví dụ lệnh dịch phải srl

- Lệnh hợp ngữ:

srl     \$s2, \$s1, 2            # \$s2 = \$s1 >> 2

- Mã máy

op	rs	rt	rd	shamt	funct
0	0	17	18	2	2
000000	00000	10001	10010	00010	000010

(0x00119082)

- Ví dụ kết quả thực hiện lệnh

\$s1    

0000	0000	0000	0000	0000	0000	0101	0110
------	------	------	------	------	------	------	------

    = 86

\$s2    

0000	0000	0000	0000	0000	0000	0001	0101
------	------	------	------	------	------	------	------

    = 21  
[86/4]

*Chú ý: Nội dung thanh ghi \$s0 không bị thay đổi*

# Nạp hằng số vào thanh ghi

---

- Trường hợp hằng số 16-bit □ sử dụng lệnh **addi**:
  - Ví dụ: nạp hằng số 0x4f3c vào thanh ghi \$s0:  
**addi \$s0, \$0, 0x4f3c** *#\$s0 = 0x4F3C*
- Trong trường hợp hằng số 32-bit □ sử dụng lệnh **lui** và lệnh **ori**:

**lui rt, constant\_hi16bit**

- Copy 16 bit cao của hằng số vào 16 bit trái của rt
- Xóa 16 bits bên phải của rt về 0

**ori rt,rt,constant\_low16bit**

- Đưa 16 bit thấp của hằng số 32 bit vào thanh ghi rt

# Lệnh lui (*load upper immediate*)

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

lui \$s0, 0x21A0

15	0	\$s0	0x21A0
15	0	16	0x21A0

Lệnh mã máy

001111	00000	10000	0010 0001 1010 0000
--------	-------	-------	---------------------

(0x3C1021A0)

Nội dung \$s0 sau khi lệnh được thực hiện:

\$s0

0010	0001	1010	0000	0000	0000	0000	0000
------	------	------	------	------	------	------	------



# Ví dụ khởi tạo thanh ghi 32-bit

- Nạp vào các thanh ghi \$s0 giá trị 32-bit sau:

0010 0001 1010 0000 0100 0000 0011 1011 = 0x21A0 403B

lui \$s0, 0x21A0 # nạp 0x21A0 vào nửa cao  
# của thanh ghi \$s0

ori \$s0, \$s0, 0x403B # nạp 0x403B vào nửa thấp  
# của thanh ghi \$s0

\$s0	0010	0001	1010	0000	0000	0000	0000
	0000	0000	0000	0000	0100	0000	0011 1011

or

Nội dung \$s0 sau khi thực hiện lệnh ori

\$s0	0010	0001	1010	0000	0100	0000	0011 1011
------	------	------	------	------	------	------	-----------

# Tạo các cấu trúc điều khiển

---

- Các cấu trúc rẽ nhánh
  - Cấu trúc `if`
  - Cấu trúc `if/else`
  - Cấu trúc `switch/case`
- Các cấu trúc lặp
  - Cấu trúc `while`
  - Cấu trúc `do while`
  - Cấu trúc `for`

# Các lệnh rẽ nhánh và lệnh nhảy

---

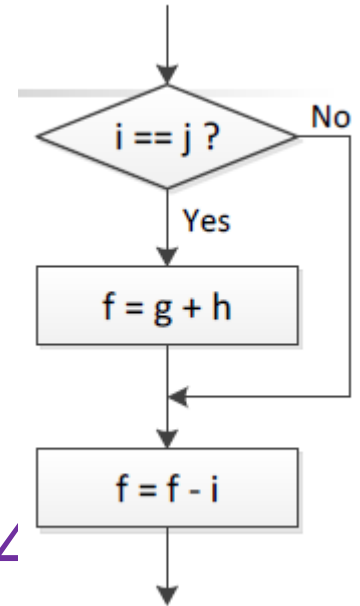
- Các lệnh rẽ nhánh: beq, bne
  - Rẽ nhánh đến lệnh được đánh nhãn nếu điều kiện là đúng, ngược lại, thực hiện tuần tự
  - beq rs, rt, L1
    - branch on equal
    - nếu (rs == rt) rẽ nhánh đến lệnh ở nhãn L1;
  - bne rs, rt, L1
    - branch on not equal
    - nếu (rs != rt) rẽ nhánh đến lệnh ở nhãn L1;
- Lệnh nhảy j
  - j L1
  - nhảy (jump) không điều kiện đến lệnh ở nhãn L1

# Dịch câu lệnh if

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4



# Dịch câu lệnh if

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4

- Mã MIPS:

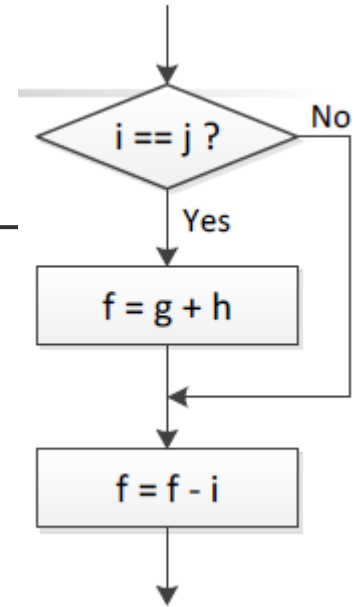
```
# $s0 = f, $s1 = g, $s2 = h
```

```
# $s3 = i, $s4 = j
```

```
    bne $s3, $s4, L1    # Nếu i≠j
```

```
    add $s0, $s1, $s2    # thì f=g+h
```

```
L1: sub $s0, $s0, $s3    # f=f-i
```

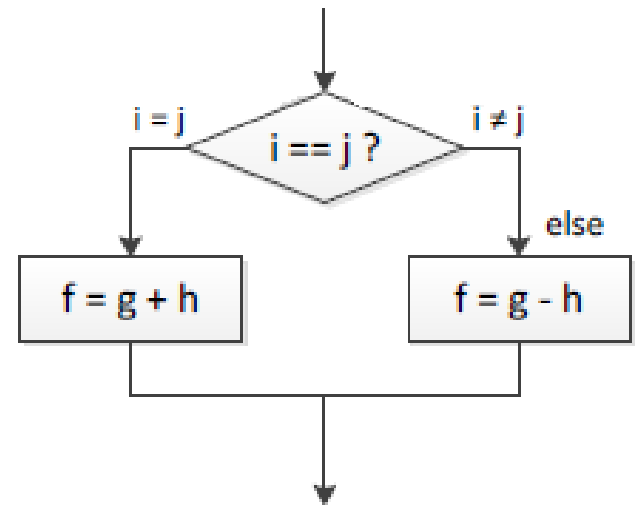


# Dịch câu lệnh if/else

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4

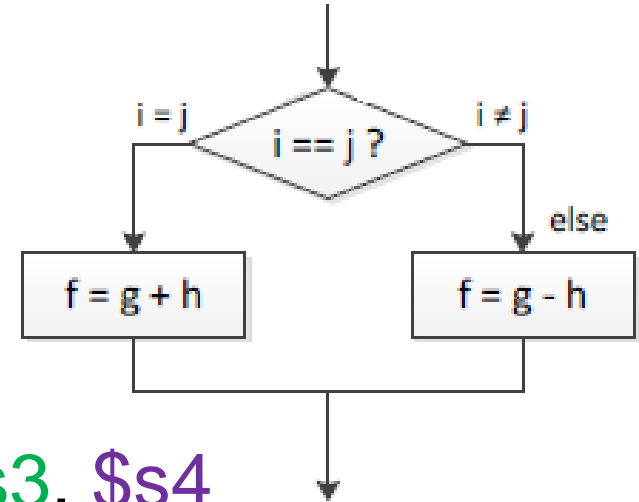


# Dịch câu lệnh if/else

## ■ Mã C:

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

■ f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4



## ■ Mã MIPS:

```
bne $s3, $s4, Else    # Nếu i=j
add $s0, $s1, $s2     # thì f=g+h
j Exit                # thoát
Else: sub $s0, $s1, $s2 # nếu khác
                        # thì f=g-h
```

Exit: ...

# Dịch câu lệnh switch/case

---

Mã C:

```
switch (amount) {  
    case 20: fee = 2; break;  
    case 50: fee = 3; break;  
    case 100: fee = 5; break;  
    default: fee = 0;  
}
```

// tương đương với sử dụng các câu lệnh if/else

```
if(amount == 20) fee = 2;  
else if (amount == 50) fee = 3;  
else if (amount == 100) fee = 5;  
else fee = 0;
```



# Dịch câu lệnh switch/case

Mã hợp ngữ MIPS

# \$s0 = amount, \$s1 = fee

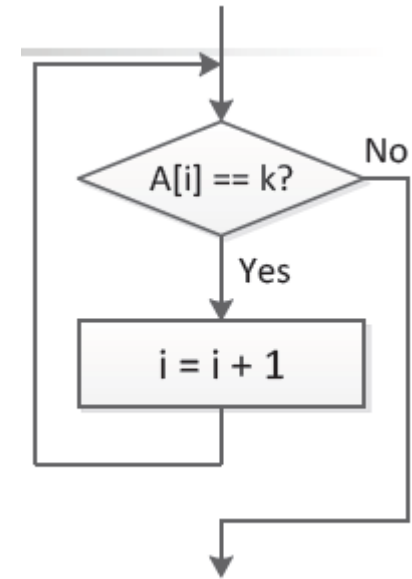
```
case20: addi $t0, $0, 20          # $t0 = 20
        bne $s0, $t0, case50     # amount == 20? if not, skip to case50
        addi $s1, $0, 2          # if so, fee = 2
        j done                   # and break out of case
case50:  addi $t0, $0, 50         # $t0 = 50
        bne $s0, $t0, case100    # amount == 50? if not, skip to case100
        addi $s1, $0, 3          # if so, fee = 3
        j done                   # and break out of case
case100: addi $t0, $0, 100        # $t0 = 100
        bne $s0, $t0, default    # amount == 100? if not, skip to default
        addi $s1, $0, 5          # if so, fee = 5
        j done                   # and break out of case
default: add $s1, $0, $0         # fee = 0
done:
```

# Dịch câu lệnh vòng lặp While

- Mã C:

```
while (A[i] == k) i += 1;
```

- $i$  ở  $\$s3$ ,  $k$  ở  $\$s5$ ,
- địa chỉ của mảng  $A$  ở  $\$s6$



# Dịch câu lệnh vòng lặp While

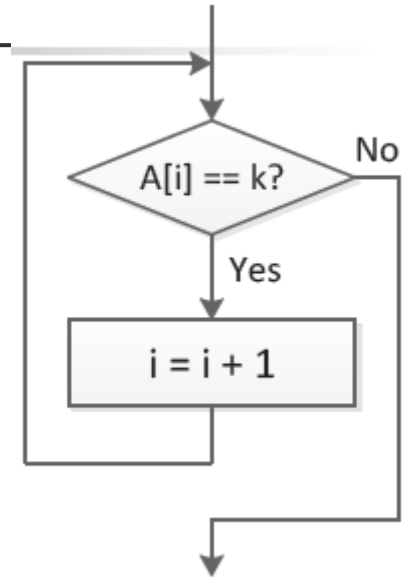
- Mã C:

```
while (A[i] == k) i += 1;
```

- $i$  ở  $\$s3$ ,  $k$  ở  $\$s5$ ,
- địa chỉ của mảng  $A$  ở  $\$s6$

- Mã MIPS được dịch:

```
Loop:  sll    $t1, $s3, 2        # $t1 = 4 * i
        add   $t1, $t1, $s6      # $t1 trở tới A[i]
        lw    $t0, 0($t1)        # $t0 = A[i]
        bne   $t0, $s5, Exit     # nếu A[i] != k
        addi  $s3, $s3, 1        # thì i = i + 1
        j     Loop              # quay lại
Exit:   ...                      # nếu A[i] == k, thoát
```



# Dịch câu lệnh vòng lặp For

---

- Mã C:

```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

# Dịch câu lệnh vòng lặp For

- Mã C:

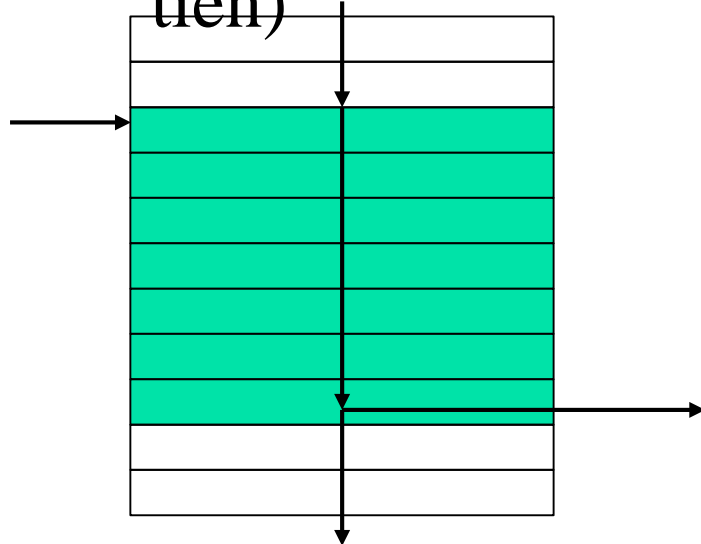
```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i = i+1) {
    sum = sum + i;
}
```

- Mã MIPS được dịch:

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0           # sum = 0
    add  $s0, $0, $0          # i = 0
    addi $t0, $0, 10          # $t0 = 10
for:  beq  $s0, $t0, done      # Nếu i = 10, thoát
    add  $s1, $s1, $s0        # sum = sum + i
    addi $s0, $s0, 1          # i = i+1
    j    for                  # quay lại
done:
```

# Khối lệnh cơ sở

- Khối lệnh cơ sở là dãy các lệnh với
  - Không có lệnh rẽ nhánh nhúng trong đó (ngoại trừ ở cuối)
  - Không có đích rẽ nhánh tới (ngoại trừ ở vị trí đầu tiên)



- Chương trình dịch xác định khối cơ sở để tối ưu hóa
- Các bộ xử lý tiên tiến có thể tăng tốc độ thực hiện khối cơ sở

# Thêm các thao tác điều kiện

---

- Lệnh `slt` (set on less than)

**`slt rd, rs, rt`**

- Nếu ( $rs < rt$ ) thì  $rd = 1$ ; ngược lại  $rd = 0$ ;

- Lệnh `slti`

**`slti rt, rs, constant`**

- Nếu ( $rs < \text{constant}$ ) thì  $rt = 1$ ; ngược lại  $rt = 0$ ;

- Sử dụng kết hợp với các lệnh `beq`, `bne`

`slt $t0, $s1, $s2      # nếu ($s1 < $s2)`

`bne $t0, $zero, L1    # rẽ nhánh đến L1`

`...`

`L1:`

# So sánh số có dấu và không dấu

---

- So sánh số có dấu: `slt`, `slti`
- So sánh số không dấu: `sltu`, `sltiu`
- Ví dụ
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt        $t0, $s0, $s1        # signed`
    - `-1 < +1                → $t0 = 1`
  - `sltu       $t0, $s0, $s1        # unsigned`
    - `+4,294,967,295 > +1        → $t0 = 0`



# Ví dụ mã lệnh slt

---

- Mã C

```
int sum = 0;  
int i;  
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

# Ví dụ mã lệnh slt

---

## ■ Mã hợp ngữ MIPS

# \$s0 = i, \$s1 = sum

```
        addi $s1, $0, 0      # sum = 0
        addi $s0, $0, 1      # i = 1
        addi $t0, $0, 101    # t0 = 101
loop:   slt  $t1, $s0, $t0    # Nếu i >= 101
        beq $t1, $0, done    # thì thoát
        add $s1, $s1, $s0    # nếu i < 101 thì sum = sum + i
        sll $s0, $s0, 1      # i = 2 * i
        j   loop            # lặp lại
done:
```

# Lập trình với mảng dữ liệu

---

- Truy cập số lượng lớn các dữ liệu cùng loại
- Chỉ số (Index): truy cập từng phần tử của mảng
- Kích cỡ (Size): số phần tử của mảng

# Ví dụ về mảng

- Mảng 5-phần tử, mỗi phần tử có độ dài 32-bit, chiếm 4 byte trong bộ nhớ

- Địa chỉ cơ sở =  $0x12348000$   
(địa chỉ của phần tử đầu tiên của mảng `array[0]`)

0x12348000	array[0]
0x12348004	array[1]
0x12348008	array[2]
0x1234800C	array[3]
0x12348010	array[4]

- Bước đầu tiên để truy cập mảng: nạp địa chỉ cơ sở vào thanh ghi

# Ví dụ truy cập các phần tử

---

## ■ Mã C

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

## ■ Mã hợp ngữ MIPS

*# nạp địa chỉ cơ sở của mảng vào \$s0*

lui \$s0, 0x1234                      #0x1234 vào nửa cao của \$s0

ori \$s0, \$s0, 0x8000 #0x8000 vào nửa thấp của \$s0

lw \$t1, 0(\$s0) sll \$t1,              # \$t1 = array[0] # \$t1 =

\$t1, 1 sw \$t1, 0(\$s0)              \$t1 \* 2 # array[0] = \$t1

lw \$t1, 4(\$s0) sll \$t1,              # \$t1 = array[1] # \$t1 =

\$t1, 1 sw \$t1, 4(\$s0)              \$t1 \* 2 # array[1] = \$t1

# Vòng lặp truy cập mảng dữ liệu

---

## ■ Mã C

```
int array[1000];  
int i;  
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;  
// giả sử địa chỉ cơ sở của mảng = 0x23b8f000
```

## ■ Mã hợp ngữ MIPS

```
#$s0 = array base address (0x23b8f000), $s1 = i
```

# Vòng lặp truy cập mảng dữ liệu (tiếp)

```
# Mã hợp ngữ MIPS
# $s0 = array base address (0x23b8f000), $s1 = i
# khởi tạo các thanh ghi
    lui $s0, 0x23b8                # $s0 = 0x23b80000
    ori $s0, $s0, 0xf000          # $s0 = 0x23b8f000
    addi $s1, $0, 0               # i = 0
    addi $t2, $0, 1000            # $t2 = 1000
# vòng lặp
    loop: slt $t0, $s1, $t2        # i < 1000?
           beq $t0, $0, done        # if not then done
           sll $t0, $s1, 2          # $t0 = i*4
           add $t0, $t0, $s0        # address of array[i]
           lw  $t1, 0($t0)          # $t1 = array[i]
           sll $t1, $t1, 3          # $t1 = array[i]*8
           sw  $t1, 0($t0)          # array[i] = array[i]*8
           addi $s1, $s1, 1         # i = i + 1
           j  loop                  # repeat
done:
```

# 5. Chương trình con - thủ tục

---

- Các bước yêu cầu:
  1. Đặt các tham số vào các thanh ghi
  2. Chuyển điều khiển đến thủ tục
  3. Thực hiện các thao tác của thủ tục
  4. Đặt kết quả vào thanh ghi cho chương trình đã gọi thủ tục
  5. Trở về vị trí đã gọi



# Sử dụng các thanh ghi

---

- \$a0 – \$a3: các tham số (các thanh ghi 4 – 7)
- \$v0, \$v1: giá trị kết quả (các thanh ghi 2 và 3)
- \$t0 – \$t9: các giá trị tạm thời
  - Có thể được ghi lại bởi thủ tục được gọi
- \$s0 – \$s7: cất giữ các biến
  - Cần phải cất/khôi phục bởi thủ tục được gọi
- \$gp: global pointer - con trỏ toàn cục cho dữ liệu tĩnh (thanh ghi 28)
- \$sp: stack pointer -con trỏ ngăn xếp (thanh ghi 29)
- \$fp: frame pointer – con trỏ khung (thanh ghi 30)
- \$ra: return address – địa chỉ trở về (thanh ghi 31)

# Các lệnh gọi thủ tục

---

- Gọi thủ tục: jump and link

`jal ProcedureLabel`

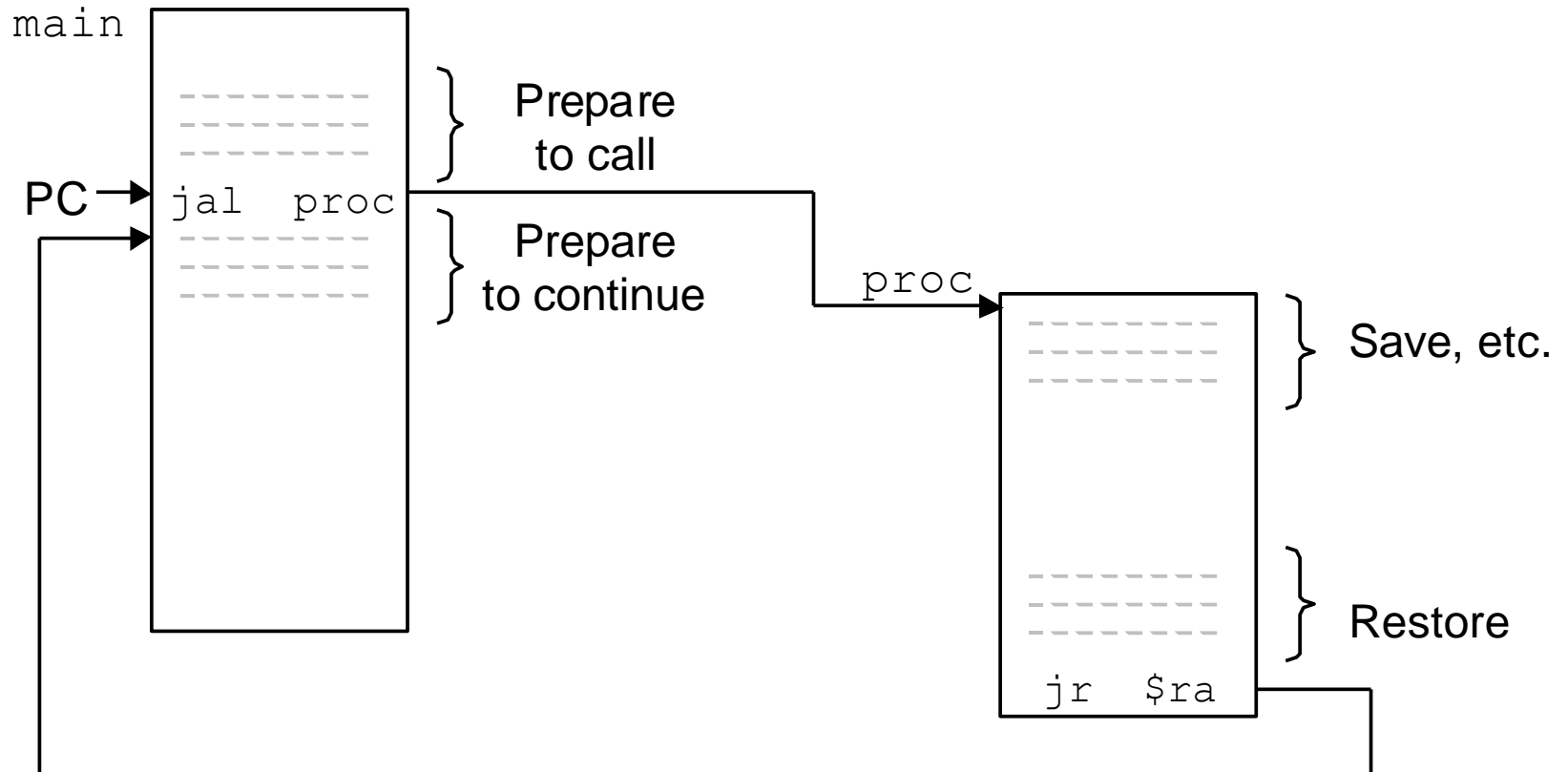
- Địa chỉ của lệnh kế tiếp (địa chỉ trở về) được cất ở thanh ghi \$ra
- Nhảy đến địa chỉ của thủ tục

- Trở về từ thủ tục: jump register

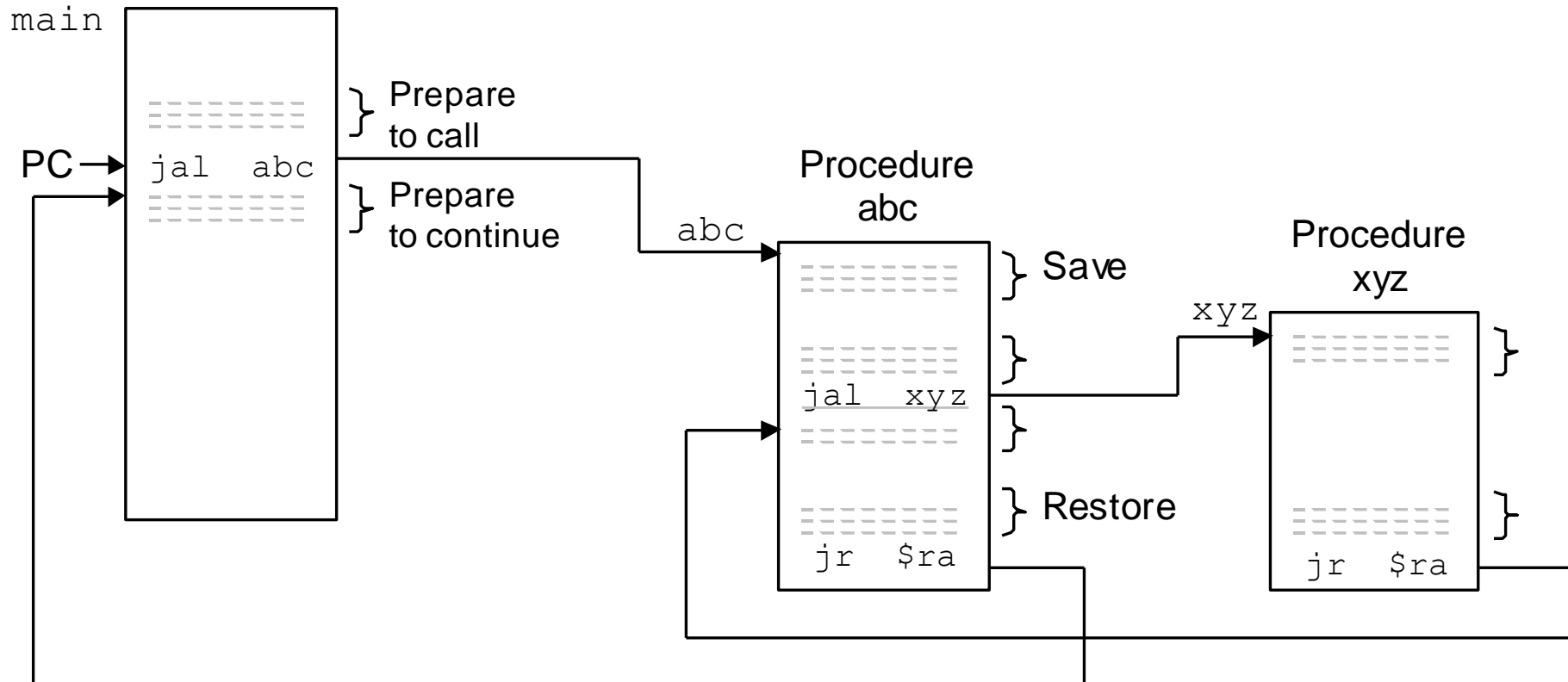
`jr $ra`

- Copy nội dung thanh ghi \$ra (đang chứa địa chỉ trở về) trả lại cho bộ đếm chương trình PC

# Minh họa gọi Thủ tục



# Gọi thủ tục lồng nhau



# Ví dụ Thủ tục lá

---

- Thủ tục lá là thủ tục không có lời gọi thủ tục khác
- Mã C:

```
int leaf_example (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Các tham số g, h, i, j ở \$a0, \$a1, \$a2, \$a3
- f ở \$s0 (do đó, cần cất \$s0 ra ngăn xếp)
- \$t0 và \$t1 được thủ tục sử dụng để chứa các giá trị tạm thời, cũng cần cất trước khi sử dụng.
- Kết quả ở \$v0

# Mã hợp ngữ MIPS

fact:		
	addi \$sp, \$sp, -8	# dành stack cho 2 mục
	sw \$ra, 4(\$sp)	# cất địa chỉ trở về
	sw \$a0, 0(\$sp)	# cất tham số n
	slti \$t0, \$a0, 1	# kiểm tra $n < 1$
	beq \$t0, \$zero, L1	
	addi \$v0, \$zero, 1	# nếu đúng, kết quả là 1
	addi \$sp, \$sp, 8	# lấy 2 mục từ stack
	jr \$ra	# và trở về
L1:	addi \$a0, \$a0, -1	# nếu không, giảm n
	jal fact	# gọi đệ qui
	lw \$a0, 0(\$sp)	# khôi phục n ban đầu
	lw \$ra, 4(\$sp)	# và địa chỉ trở về
	addi \$sp, \$sp, 8	# lấy 2 mục từ stack
	mul \$v0, \$a0, \$v0	# nhân để nhận kết quả
	jr \$ra	# và trở về

# Ví dụ Thủ tục đệ quy

---

- Là thủ tục có gọi thủ tục khác

- Mã C:

```
int fact (int n)
{
    if (n < 1) return (1);
    else return n * fact(n - 1);
}
```

- Tham số n ở \$a0
- Kết quả ở \$v0

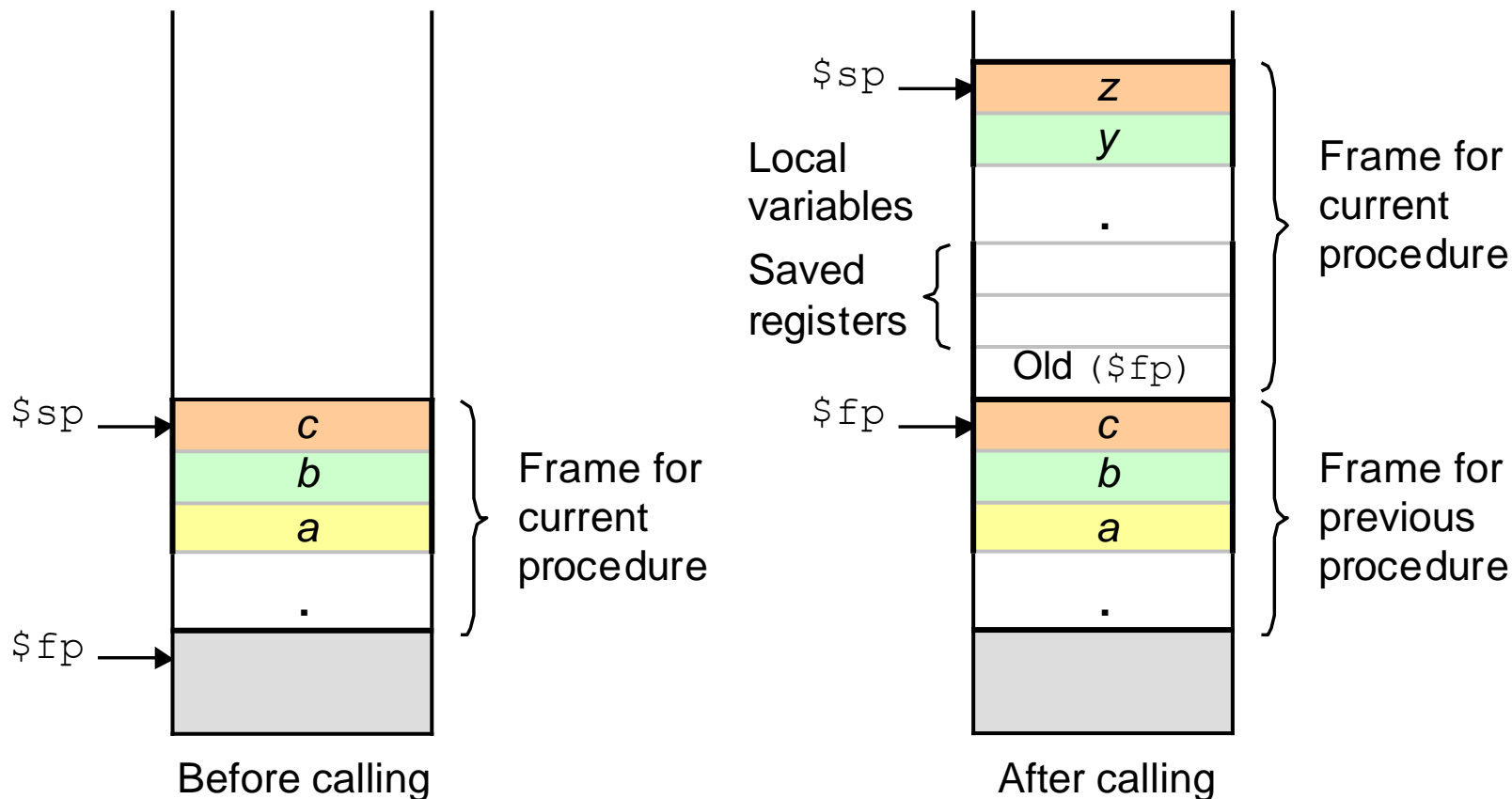
# Ví dụ Thủ tục càn (tiếp)

## ■ Mã MIPS:

fact:		
addi	\$sp, \$sp, -8	# dành stack cho 2 mục
sw	\$ra, 4(\$sp)	# cất địa chỉ trở về
sw	\$a0, 0(\$sp)	# cất tham số n
slti	\$t0, \$a0, 1	# kiểm tra $n < 1$
beq	\$t0, \$zero, L1	
addi	\$v0, \$zero, 1	# nếu đúng, kết quả là 1
addi	\$sp, \$sp, 8	# lấy 2 mục từ stack
jr	\$ra	# và trở về
L1:	addi \$a0, \$a0, -1	# nếu không, giảm n
	jal fact	# gọi đệ qui
lw	\$a0, 0(\$sp)	# khôi phục n ban đầu
lw	\$ra, 4(\$sp)	# và địa chỉ trở về
addi	\$sp, \$sp, 8	# lấy 2 mục từ stack
mul	\$v0, \$a0, \$v0	# nhân để nhận kết quả
jr	\$ra	# và trở về



# Sử dụng Stack khi gọi thủ tục



## 6. Dữ liệu ký tự

---

- Các tập ký tự được mã hóa theo byte
  - ASCII: 128 ký tự
    - 95 ký thị hiển thị , 33 mã điều khiển
  - Latin-1: 256 ký tự
    - ASCII và các ký tự mở rộng
- Unicode: Tập ký tự 16-bit
  - Được sử dụng trong Java, C++, ...
  - Hầu hết các ký tự của các ngôn ngữ trên thế giới và các ký hiệu

# Các thao tác với Byte/Halfword

---

- Có thể sử dụng các phép toán logic
- Nạp/Lưu byte/halfword trong MIPS
- `lb rt, offset(rs)`      `lh rt, offset(rs)`
  - Mở rộng dấu thành 32 bits trong `rt`
- `lbu rt, offset(rs)`      `lhu rt, offset(rs)`
  - Mở rộng zero thành 32 bits trong `rt`
- `sb rt, offset(rs)`      `sh rt, offset(rs)`
  - Chỉ lưu byte/halfword bên phải

# Ví dụ copy String

---

- Mã C:

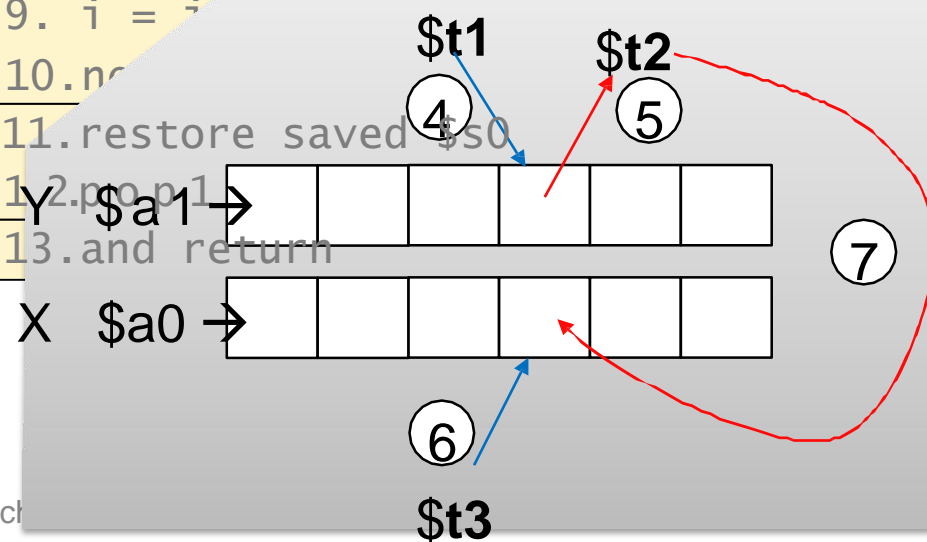
```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

- Các địa chỉ của x, y ở \$a0, \$a1
- i ở \$s0

# Ví dụ Copy String

## ■ MIPS code:

strcpy:		
addi	\$sp, \$sp, -4	# 1. adjust stack for 1 item
sw	\$s0, 0(\$sp)	# 2. save \$s0
add	\$s0, \$zero, \$zero	# 3. i = 0
L1:	add \$t1, \$s0, \$a1	# 4. addr of y[i] in \$t1
lbu	\$t2, 0(\$t1)	# 5. \$t2 = y[i]
add	\$t3, \$s0, \$a0	# 6. addr of x[i] in \$t3
sb	\$t2, 0(\$t3)	# 7. x[i] = y[i]
beq	\$t2, \$zero, L2	# 8. exit loop if y[i] == 0
addi	\$s0, \$s0, 1	# 9. i = i + 1
j	L1	# 10. next iteration
L2:	lw \$s0, 0(\$sp)	# 11. restore saved \$s0
addi	\$sp, \$sp, 4	# 12. pop 1 item
jr	\$ra	# 13. and return



# 7. Các lệnh nhân và chia số nguyên

---

- MIPS có hai thanh ghi 32-bit: HI (high) và LO (low)
- Các lệnh liên quan:
  - `mul rd, rs, rt # rd = rs * rt`
  - `div rd, rs, rt # rd = rs / rt`
  
  - `mult rs, rt # nhân số nguyên có dấu`
  - `multu rs, rt # nhân số nguyên không dấu`
    - Tích 64-bit nằm trong cặp thanh ghi HI/LO
  
  - `div rs, rt # chia số nguyên có dấu`
  - `divu rs, rt # chia số nguyên không dấu`
    - HI: chứa phần dư, LO: chứa thương
  
  - `mfhi rd # Thanh ghi rd Hi`
  - `mflo rd # Thanh ghi rd Lo`

# Lệnh với số dấu phẩy động (FP)

---

- Các thanh ghi số dấu phẩy động
  - 32 thanh ghi 32-bit (single-precision): \$f0, \$f1, ... \$f31
  - Cặp đôi để chứa dữ liệu dạng 64-bit (double-precision): \$f0/\$f1, \$f2/\$f3, ...
- Các lệnh số dấu phẩy động chỉ thực hiện trên các thanh ghi số dấu phẩy động
  - Lệnh load và store với FP
  - lwc1, ldc1, swc1, sdc1
    - Ví dụ: ldc1 \$f8, 32(\$s2)

# Các lệnh với số dấu phẩy động

---

- Các lệnh số học với số FP 32-bit (single-precision)
  - add.s, sub.s, mul.s, div.s
  - VD: add.s \$f0, \$f1, \$f6
- Các lệnh số học với số FP 64-bit (double-precision)
  - add.d, sub.d, mul.d, div.d
  - VD: mul.d \$f4, \$f4, \$f6
- Các lệnh so sánh
  - c.xx.s, c.xx.d (trong đó xx là eq, lt, le, ...)
  - Thiết lập hoặc xóa các bit mã điều kiện
  - VD: c.lt.s \$f3, \$f4
- Các lệnh rẽ nhánh dựa trên mã điều kiện
  - bc1t, bc1f
  - VD: bc1t TargetLabel



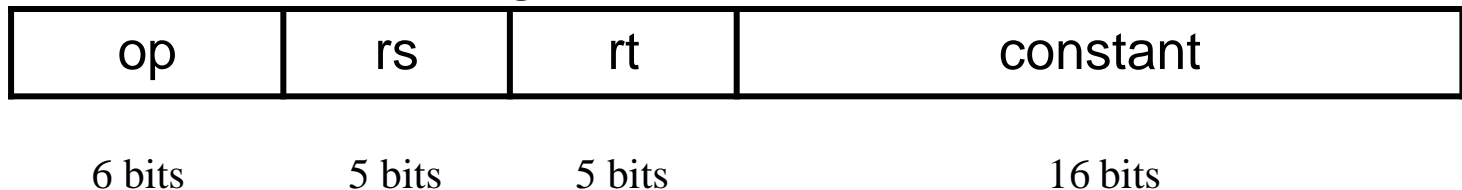
# Phương pháp định địa chỉ của MIPS

- Các lệnh **Branch** chỉ ra:

- Mã thao tác, hai thanh ghi, offset

- Hầu hết các đích rẽ nhánh là rẽ nhánh gần

- Rẽ xuôi hoặc rẽ ngược



- Định địa chỉ tương đối với PC

- PC-relative addressing
- Địa chỉ đích =  $PC + \text{hằng số} \times 4$
- Chú ý: trước đó PC đã được tăng lên
- Hằng số imm 16-bit có giá trị trong dải  $[-2^{15}, +2^{15} - 1]$

# Lệnh beq, bne

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

beq \$s0, \$s1, Exit

bne \$s0, \$s1, Exit

4 or 5	16	17	Exit
--------	----	----	------

khoảng cách tương đối tính theo word

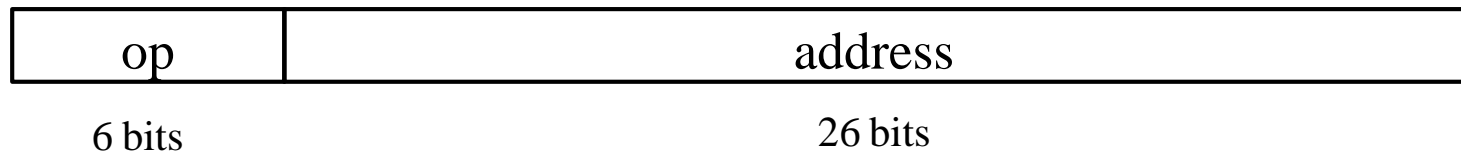
## Lệnh mã máy

beq	000100	10000	10001	0000 0000 0000 0110
-----	--------	-------	-------	---------------------

bne	000101	10000	10001	0000 0000 0000 0110
-----	--------	-------	-------	---------------------

# Địa chỉ hóa cho lệnh Jump

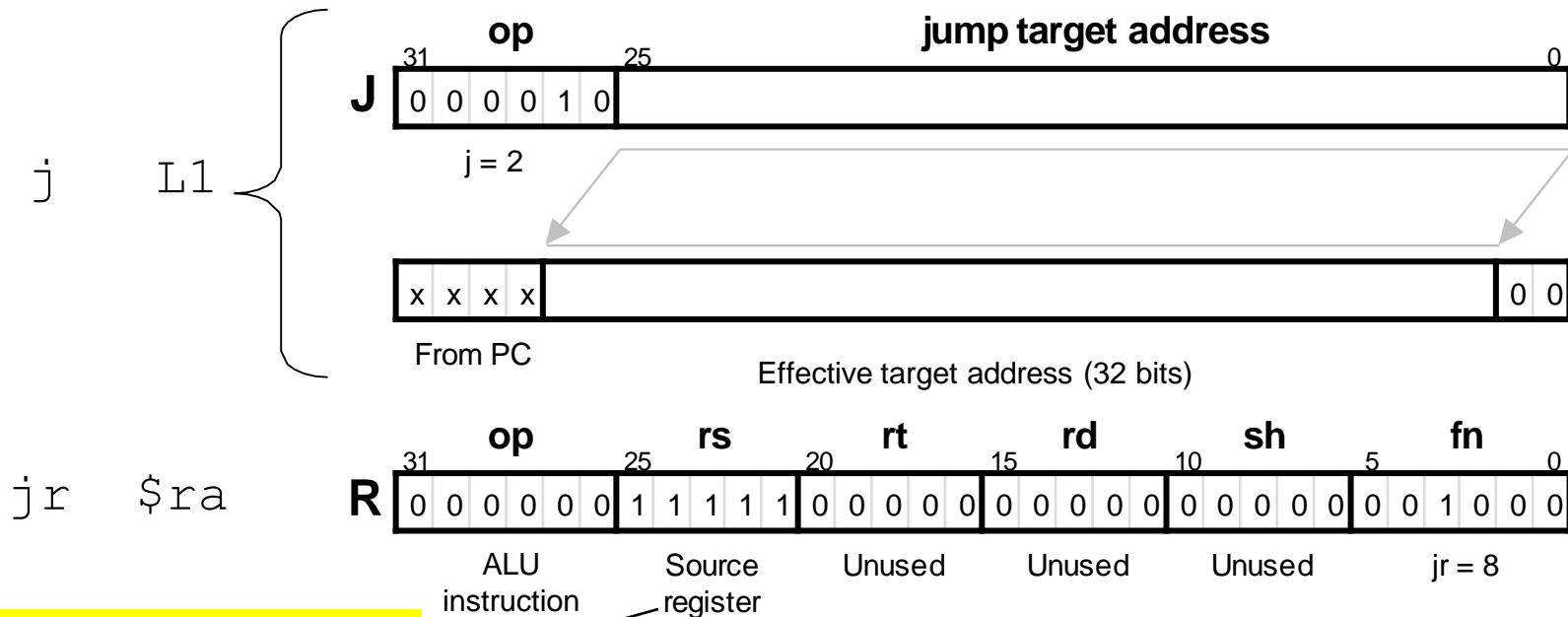
- Đích của lệnh **Jump (jv và jal)** có thể là bất kỳ chỗ nào trong chương trình
  - Cần mã hóa đầy đủ địa chỉ trong lệnh



- Định địa chỉ nhảy (giả) trực tiếp (Pseudo)Direct jump addressing
  - Địa chỉ đích =  $PC_{31..28} : (\text{address} \times 4)$   
 $= PC_{31..28} : \text{address}_{27..2} : 00_{1..0}$

# Ví dụ mã lệnh j và jr

- j L1 # nhảy đến vị trí có nhãn L1
- jr \$ra # nhảy đến vị trí có địa chỉ ở \$ra;  
# \$ra may hold a return address



■ \$ra là thanh ghi \$31  
(return address)

# Ví dụ mã hóa lệnh

```

Loop: sll  $t1, $s3, 2      8000
      add  $t1, $t1, $s6    8004
      lw   $t0, 0($t1)      8008
      bne  $t0, $s5, Exit   8012
      addi $s3, $s3, 1      8016
      j    Loop            8020
Exit:  ...                 8024
    
```

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	0x2000				

# Rẽ nhánh xa

---

- Nếu đích rẽ nhánh là quá xa để mã hóa với offset 16-bit, assembler sẽ viết lại code
- Ví dụ

```
beq $s0, $s1, L1
```

*(lệnh kế tiếp)*

...

L1:

*sẽ được thay bằng đoạn lệnh sau:*

```
bne $s0, $s1, L2
```

```
j L1
```

L2: *(lệnh kế tiếp)*

...

L1:

# Nhắc lại về phương pháp định địa chỉ

## 1. Định địa chỉ tức thì

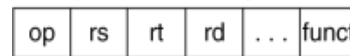
### 1. Immediate addressing



*addi \$t1,\$t2, 6*

## 2. Định địa chỉ thanh ghi

### 2. Register addressing

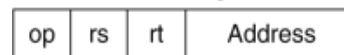


Registers

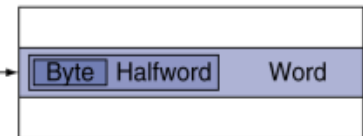
Register

## 3. Định địa chỉ cơ sở

### 3. Base addressing

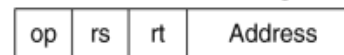


Memory

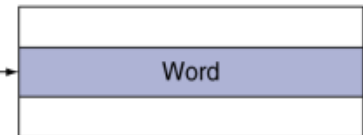
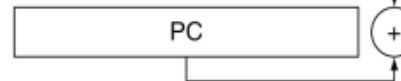


## 4. Định địa chỉ tương đối với PC

### 4. PC-relative addressing

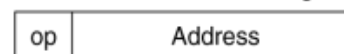


Memory

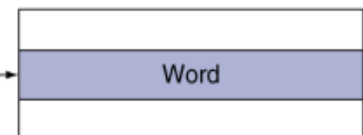
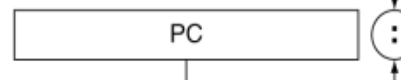


## 5. Định địa chỉ giả trực tiếp

### 5. Pseudodirect addressing

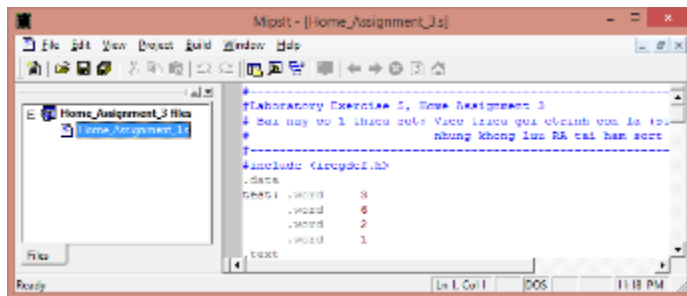


Memory

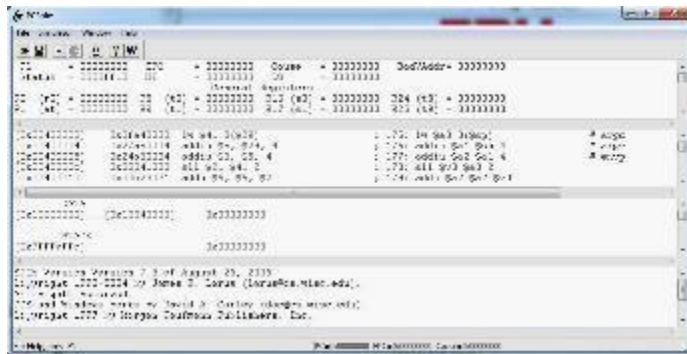


# Dịch và chạy chương trình hợp ngữ

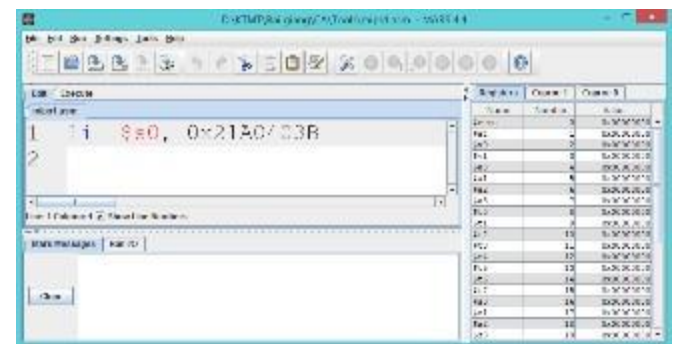
- Các phần mềm lập trình hợp ngữ MIPS:



MipsIT



PCSpin



MARS

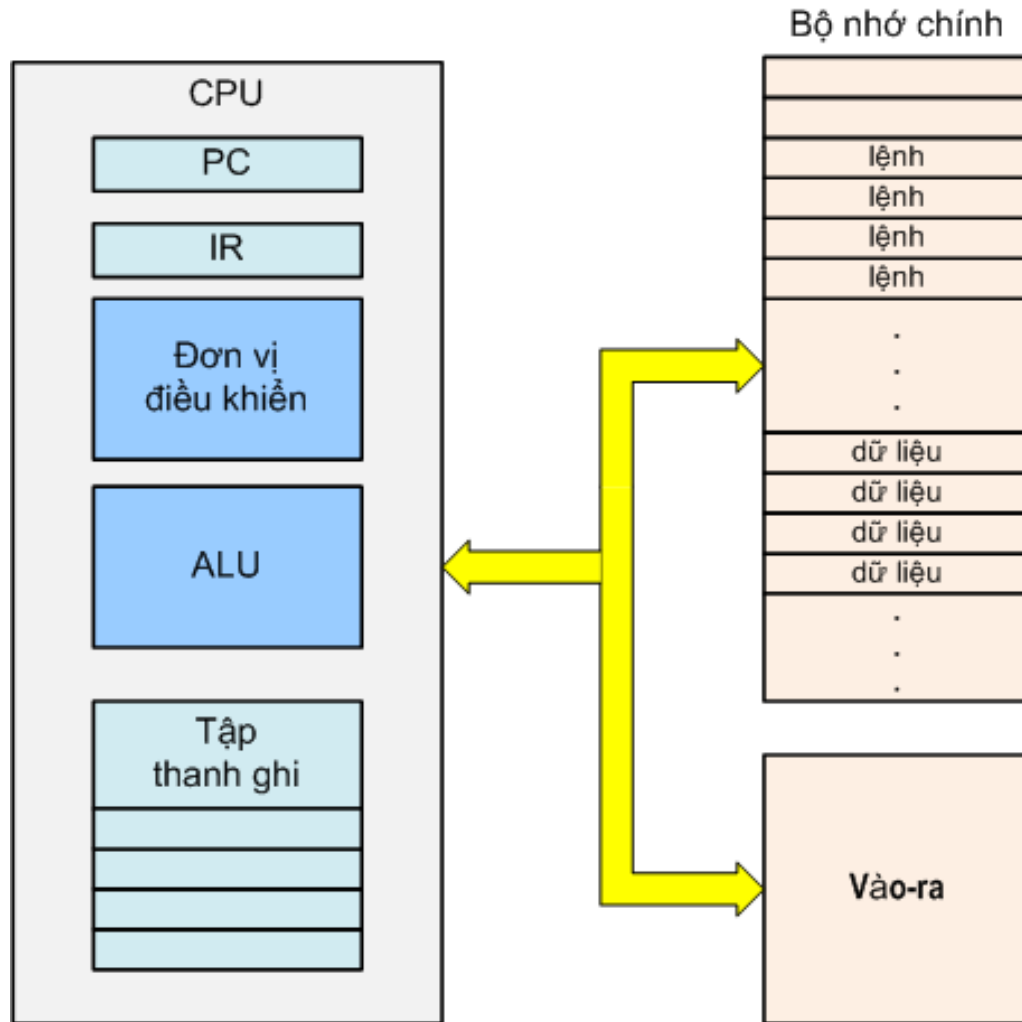
- MIPS Reference Data

[Link download](#)

[Link Mars Online](#)



# Mô hình lập trình của máy tính



PC: Program Counter  
IR: Instruction Register

# Tập thanh ghi

---

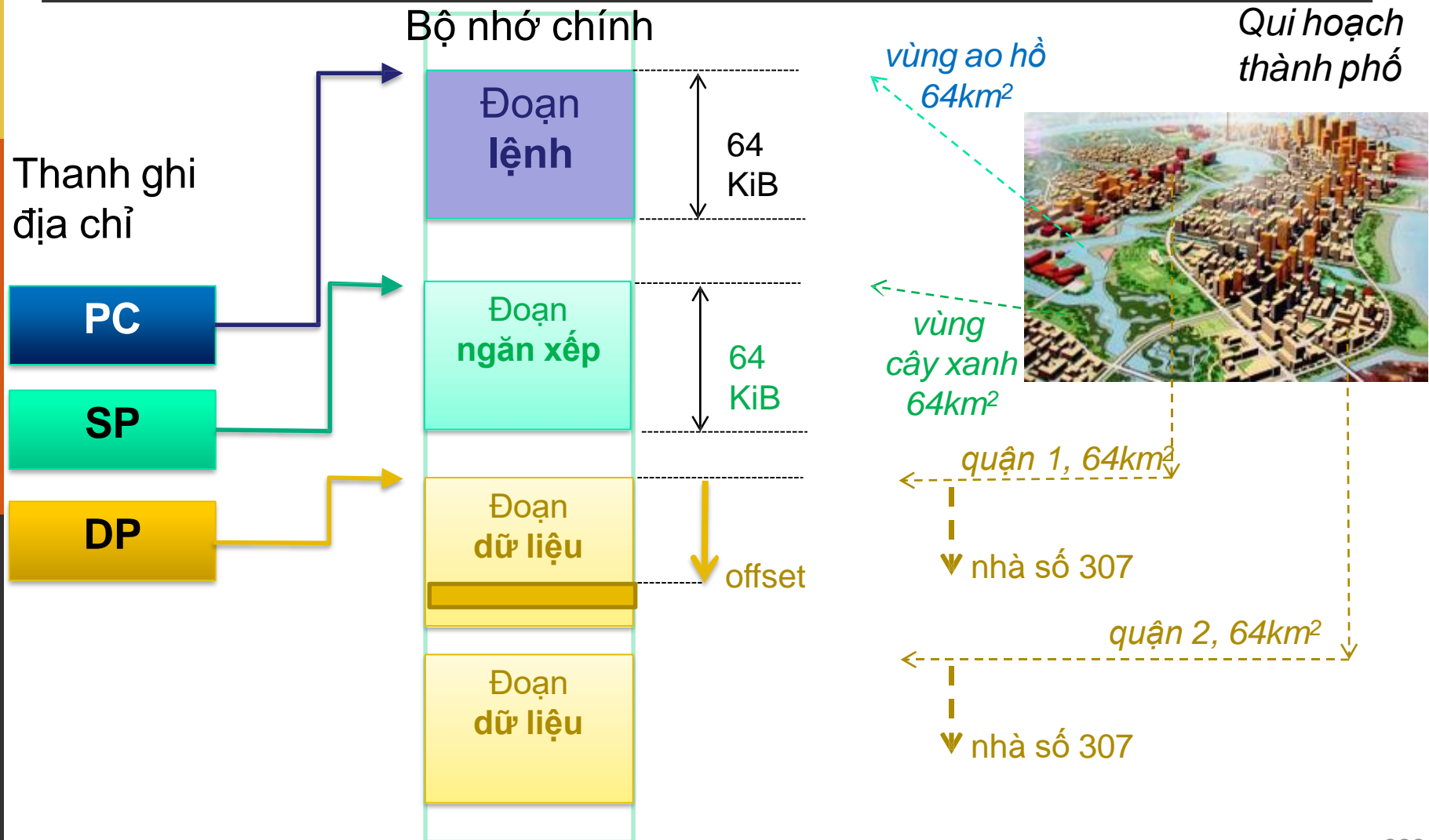
- Chứa các thông tin (dữ liệu, địa chỉ, trạng thái) cho hoạt động điều khiển và xử lý dữ liệu của CPU ở thời điểm hiện tại
- Được coi là mức đầu tiên của hệ thống nhớ
- Số lượng thanh ghi nhiều □ tăng hiệu năng của CPU
- Có hai loại thanh ghi:
  - Các thanh ghi lập trình được
  - Các thanh ghi không lập trình được

# Một số thanh ghi điển hình

---

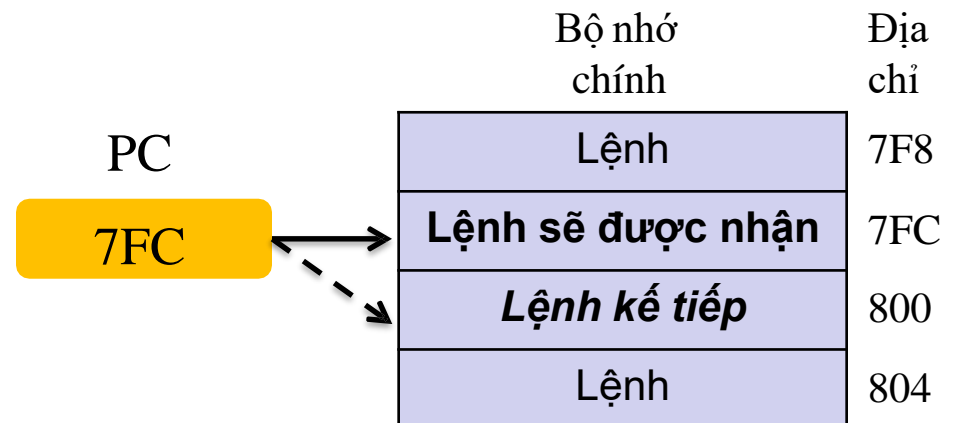
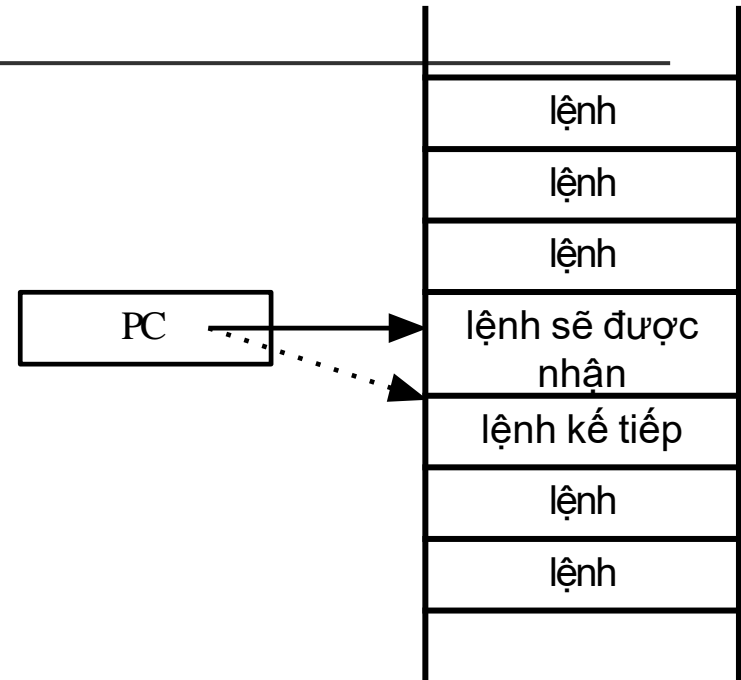
- Các thanh ghi địa chỉ
  - Bộ đếm chương trình PC (Program Counter)
  - Con trỏ dữ liệu DP (Data Pointer)
  - Con trỏ ngăn xếp SP (Stack Pointer)
  - Thanh ghi cơ sở và Thanh ghi chỉ số (Base Register & Index Register)
- Các thanh ghi dữ liệu
- Thanh ghi trạng thái

# Ý nghĩa của các thanh ghi địa chỉ



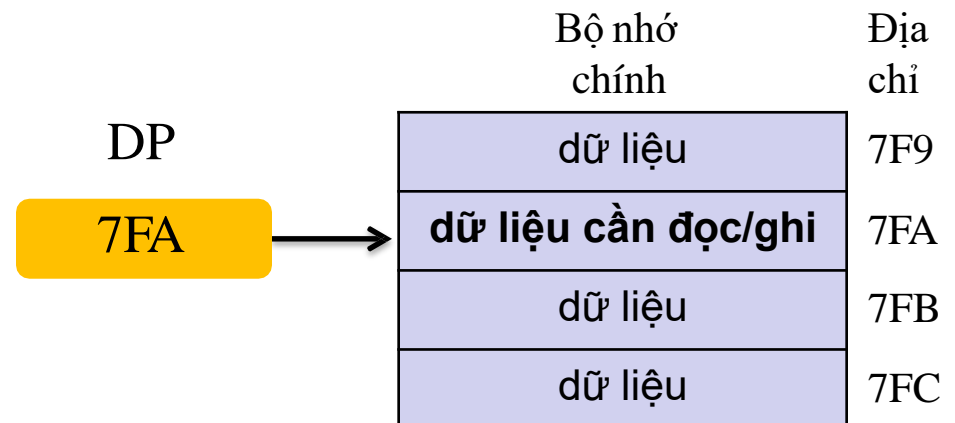
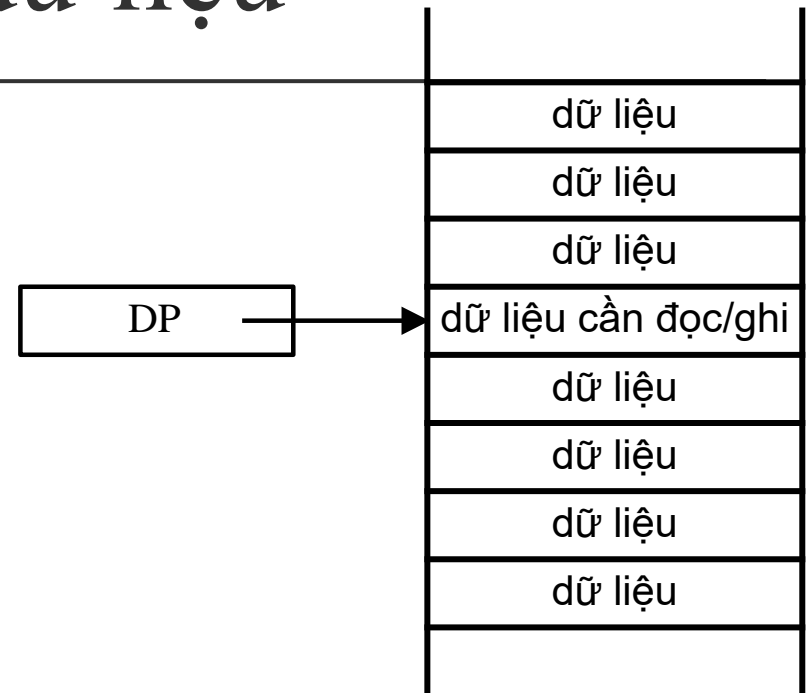
# Bộ đếm chương trình PC

- Còn được gọi là con trỏ lệnh IP (Instruction Pointer)
- Giữ địa chỉ của lệnh tiếp theo sẽ được nhận vào.
- Sau khi một lệnh được nhận vào, nội dung PC **tự động tăng** để trỏ sang lệnh kế tiếp.
- PC tăng bao nhiêu?



# Thanh ghi con trỏ dữ liệu

- Chứa địa chỉ của ngăn nhớ dữ liệu mà CPU muốn truy nhập



# Ngăn xếp (Stack)

---

- Ngăn xếp là vùng nhớ có cấu trúc LIFO (Last In - First Out: vào sau – ra trước)
- Ngăn xếp thường dùng để phục vụ cho chương trình con
- Đáy ngăn xếp là một ngăn nhớ xác định
- Đỉnh ngăn xếp là thông tin nằm ở vị trí trên cùng trong ngăn xếp
- Đỉnh ngăn xếp có thể bị thay đổi

# Con trỏ ngăn xếp SP (Stack Pointer)

- Chứa địa chỉ của ngăn nhớ đỉnh ngăn xếp

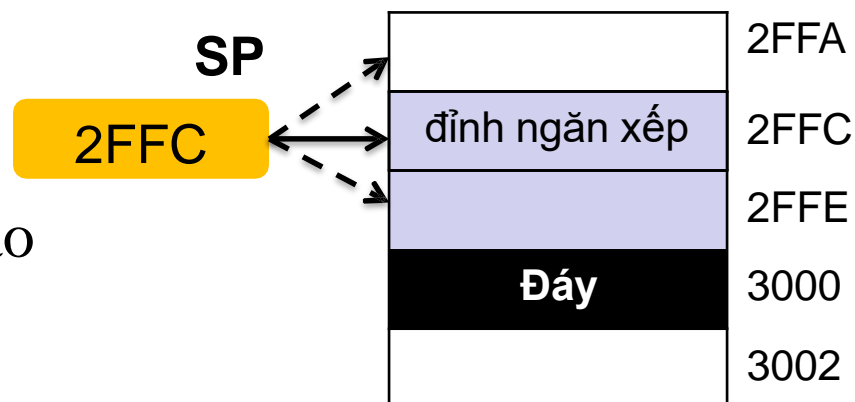
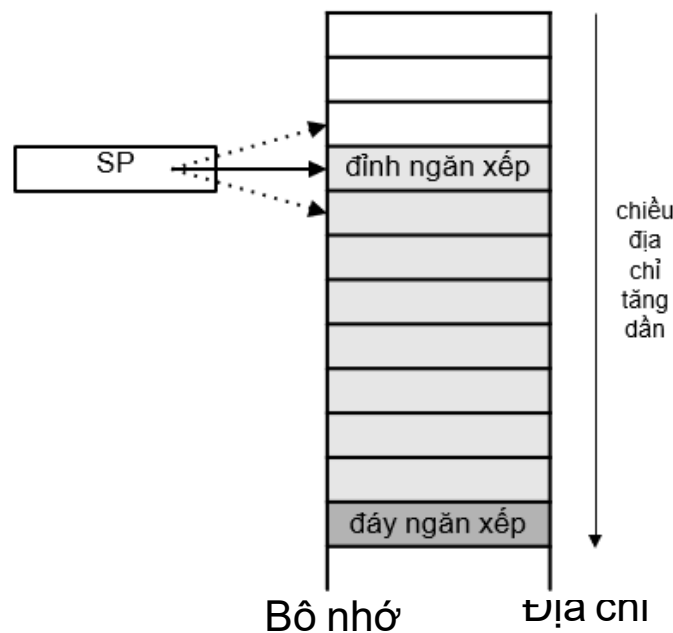
Khi cất một thông tin vào ngăn xếp:

- Nội dung của SP giảm
- Thông tin được cất vào ngăn nhớ được trỏ bởi SP

- Khi lấy một thông tin ra khỏi ngăn xếp:

- Thông tin được đọc từ ngăn nhớ được trỏ bởi SP
- Nội dung của SP tăng

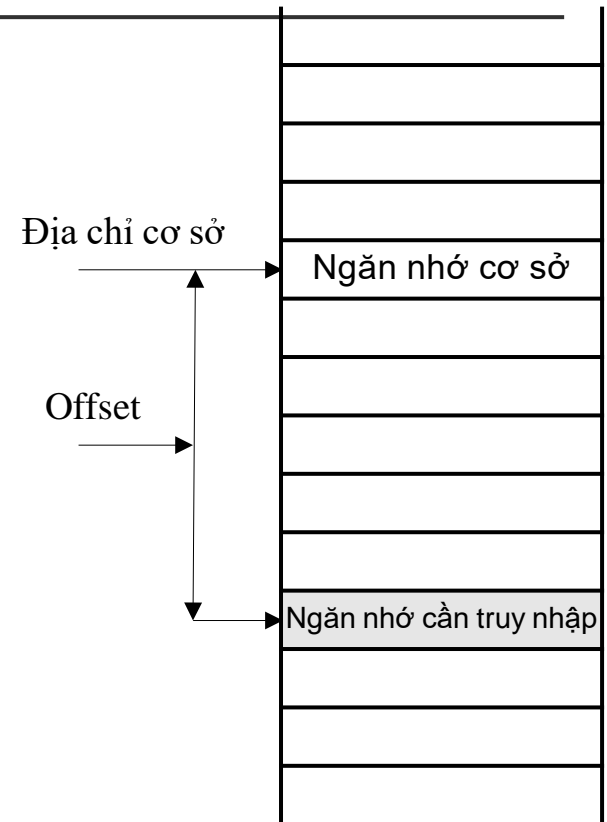
- Khi ngăn xếp rỗng, SP trỏ vào đáy





# Thanh ghi cơ sở và thanh ghi chỉ số

- Để truy nhập một ngăn nhớ có thể sử dụng hai tham số:
  - Địa chỉ cơ sở (base address)
  - Phần dịch chuyển địa chỉ (offset)
  - Địa chỉ của ngăn nhớ cần truy nhập  
= địa chỉ cơ sở + offset
- Có thể sử dụng các thanh ghi để quản lý các tham số này:
  - Thanh ghi cơ sở: chứa địa chỉ cơ sở
  - Thanh ghi chỉ số: chứa phần dịch chuyển địa chỉ



# Các thanh ghi dữ liệu

---

- Chứa các dữ liệu tạm thời hoặc các kết quả trung gian
- Cần có nhiều thanh ghi dữ liệu
- Các thanh ghi số nguyên: 8, 16, 32, 64 bit
- Các thanh ghi số dấu phẩy động

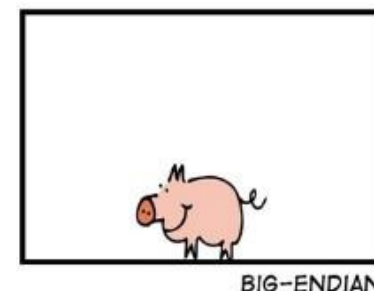
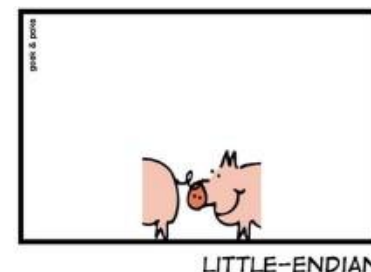
# Thanh ghi trạng thái (Status Register)

---

- Được sử dụng trên một số kiến trúc cụ thể
- Còn gọi là thanh ghi cờ (Flag Register)
- Chứa các thông tin trạng thái của CPU
  - Các cờ phép toán: báo hiệu trạng thái của kết quả phép toán
  - Các cờ điều khiển: biểu thị trạng thái điều khiển của CPU

# Thứ tự lưu trữ các byte trong bộ nhớ

- Bộ nhớ chính thường đánh địa chỉ theo byte
- Hai cách lưu trữ thông tin nhiều byte:
  - **Đầu nhỏ** (*Little-endian*): Byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ lớn.
  - **Đầu to** (*Big-endian*): Byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ lớn.



# Ví dụ lưu trữ dữ liệu 32-bit

0001 1010 0010 1011 0011 1100 0100 1101

1A	2B	3C	4D
----	----	----	----

*nghìn*

*trăm*

*chục*

*đơn vị*

<i>đơn vị</i>	4D	3000
<i>chục</i>	3C	3001
<i>trăm</i>	2B	3002
<i>nghìn</i>	1A	3003

little-endian

<i>nghìn</i>	1A	3000
<i>trăm</i>	2B	3001
<i>chục</i>	3C	3002
<i>đơn vị</i>	4D	3003

big-endian



Debug

Demo [C/C++ Application]

gdb  
Demo

Bre Ex Me Co Se Cal Me

Monitors

&x

&x : 0xBFFFF44C <Hex

0 - 3	4 - 7	8 - B	C - F
A5D41500	74030B0A	03000000	71030B0A
50840408	00000000	D8F4FFBF	D6451400
01000000	04F5FFBF	0CF5FFBF	80F9FFB7

OCCI\_Example.cpp

Demo.c

```

4 Author : Nguyen Duc Tien
5 Version :
6 Copyright : tiennd@soict.hut.edu.vn
7 Description : Hello World in C, Ansi-st
8
9 */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int main(void) {
15     int x = 0x0A0B0371;
16     int y = 0x03;
17     int z = x + y;
18     printf("%i + %i = %i\n", x, y, z);
19     return EXIT_SUCCESS;
20 }
    
```

Expressions

Variables

Expression	Value
(x)= x	0xa0b0371
(x)= y	0x3
(x)= z	0xa0b0374
+ ➔ &x	0xbffff44c
+ ➔ &y	0xbffff448
+ ➔ &z	0xbffff444
+ Add new expression	

Đầu nhỏ/Đầu to?

## - Tương ứng C – Asm

```

Demo.cpp Disassembly
File Edit View Terminal Help
#include <stdio.h>
#include <stdlib.h>

int main(void) {
80483e4:    55                push    %ebp
80483e5:    89 e5             mov     %esp,%ebp
80483e7:    83 e4 f0          and     $0xffffffff0,%esp
80483ea:    83 ec 20          sub     $0x20,%esp
    int x = 0x0A0B0371;
80483ed:    c7 44 24 1c 71 03 0b  movl    $0xa0b0371,0x1c(%esp)
80483f4:    0a
    int y = 0x03;
80483f5:    c7 44 24 18 03 00 00  movl    $0x3,0x18(%esp)
80483fc:    00
    int z = x + y;
80483fd:    8b 44 24 18       mov     0x18(%esp),%eax
8048401:    8b 54 24 1c       mov     0x1c(%esp),%edx
8048405:    8d 04 02          lea     (%edx,%eax,1),%eax
8048408:    89 44 24 14       mov     %eax,0x14(%esp)
    printf("%i + %i = %i\n",x,y,z);
804840c:    b8 00 85 04 08    mov     $0x8048500,%eax
8048411:    8b 54 24 14       mov     0x14(%esp),%edx
8048415:    89 54 24 0c       mov     %edx,0xc(%esp)
8048419:    8b 54 24 18       mov     0x18(%esp),%edx
804841d:    89 54 24 08       mov     %edx,0x8(%esp)
8048421:    8b 54 24 1c       mov     0x1c(%esp),%edx
8048425:    89 54 24 04       mov     %edx,0x4(%esp)
8048429:    89 04 24          mov     %eax,(%esp)
804842c:    e8 eb fe ff ff    call    804831c <printf@plt>
    return EXIT_SUCCESS;
8048431:    b8 00 00 00 00    mov     $0x0,%eax
}
8048436:    c9                leave

```

Linux cmd: \$ objdump -hS

# Lưu trữ của các bộ xử lý điển hình

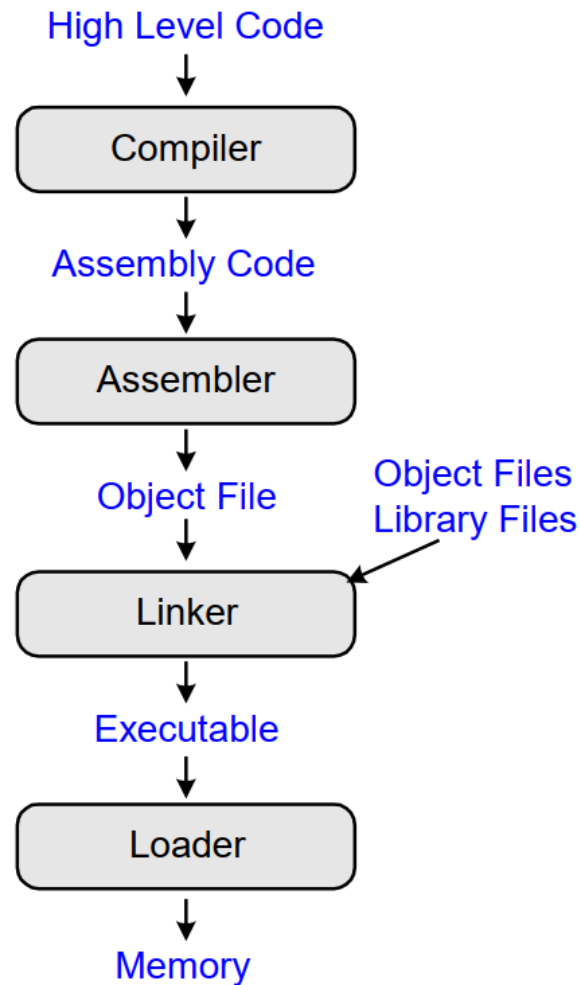
---

- Intel x86: little-endian
- Motorola 680x0, MIPS, SunSPARC: big-endian
- Power PC, Itanium: bi-endian



# Dịch và chạy ứng dụng

---

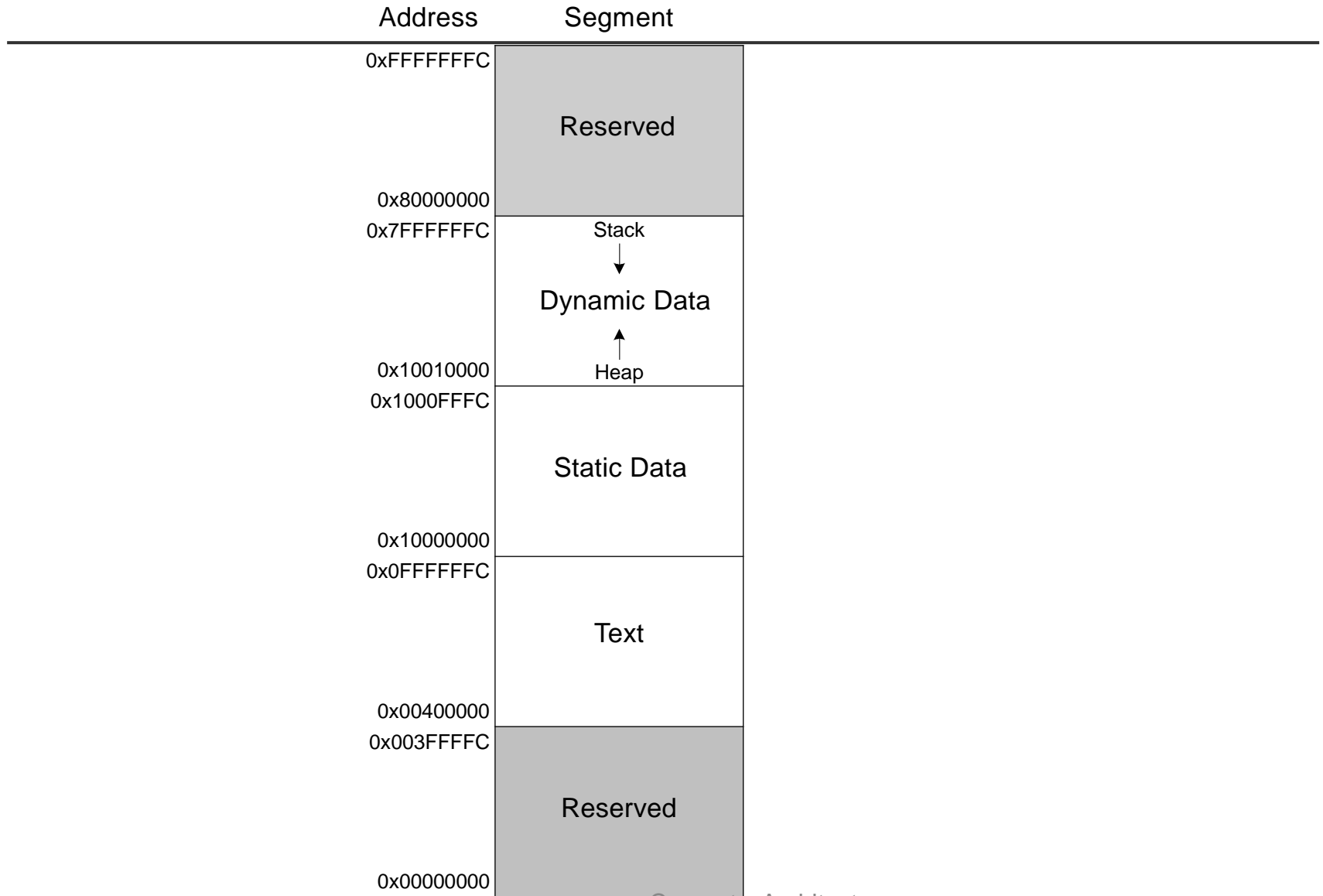


# Chương trình trong bộ nhớ

---

- Các lệnh (instructions)
- Dữ liệu
  - Toàn cục/tĩnh: được cấp phát trước khi chương trình bắt đầu thực hiện
  - Động: được cấp phát trong khi chương trình thực hiện
- Bộ nhớ:
  - $2^{32} = 4$  gigabytes (4 GiB)
  - Từ địa chỉ 0x00000000 đến 0xFFFFFFFF

# Bản đồ bộ nhớ của MIPS



# Ví dụ: Mã C

---

```
int f, g, y;    // global
                variables
```

```
int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}
```

```
int sum(int a, int b) {
    return (a + b);
}
```

# Ví dụ chương trình hợp ngữ

---

```
.data
f:
g:
y:
.text
main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)    # store $ra
    addi $a0, $0, 2     # $a0 = 2
    sw   $a0, f          # f = 2
    addi $a1, $0, 3     # $a1 = 3
    sw   $a1, g          # g = 3
    jal  sum             # call sum
    sw   $v0, y          # y = sum()
    lw   $ra, 0($sp)    # restore $ra
    addi $sp, $sp, 4     # restore $sp
    jr   $ra            # return to OS
sum:
    add  $v0, $a0, $a1   # $v0 = a + b
    jr   $ra            # return
```

# Bảng ký hiệu

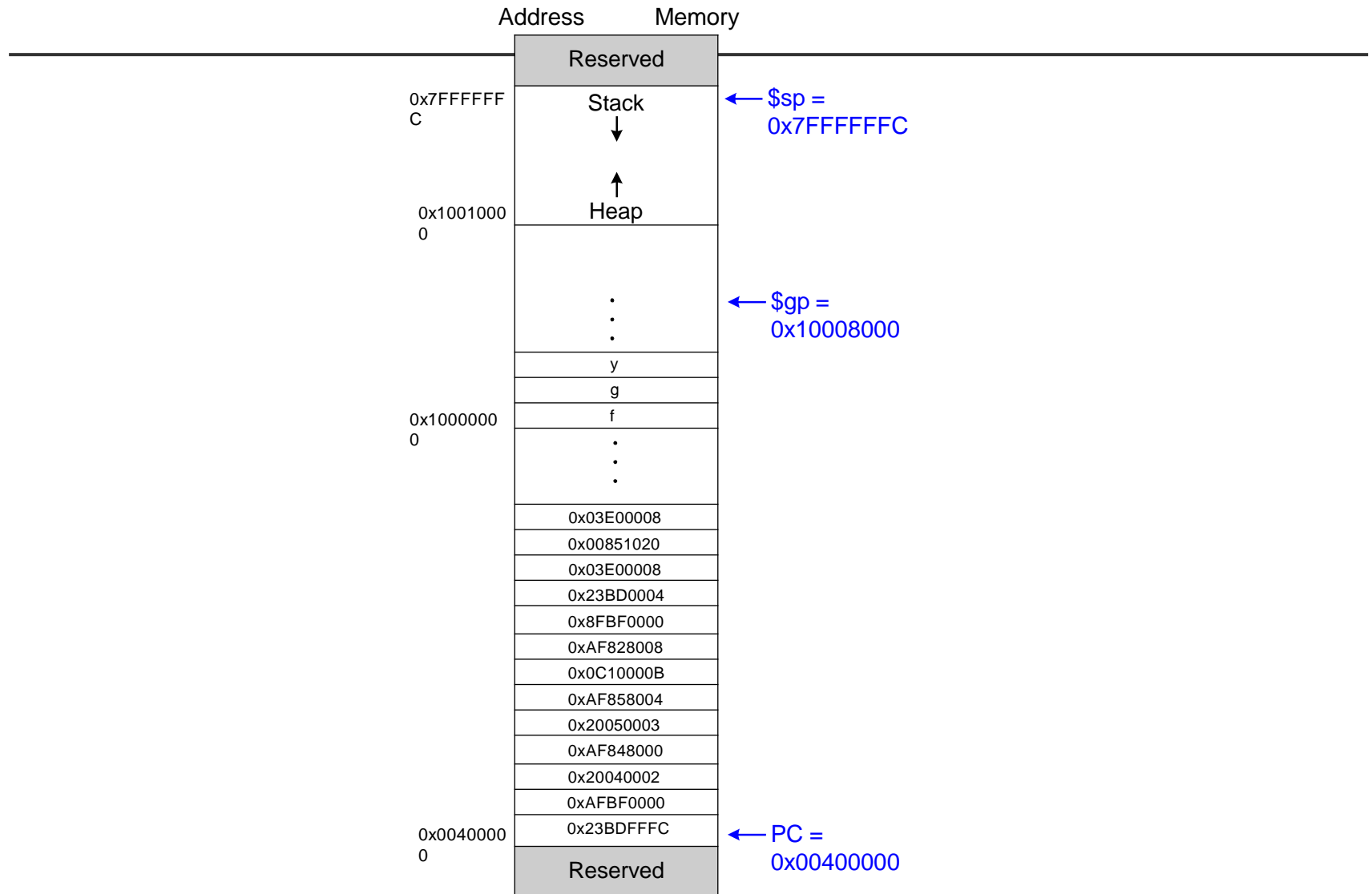
---

Ký hiệu	Địa chỉ
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

# Chương trình thực thi

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	Address	Instruction	
	0x00400000	0x23BDFFFC	addi \$sp, \$sp, -4 sw \$ra, 0 (\$sp) addi \$a0, \$0, 2
	0x00400004	0xAFBF0000	sw \$a0, 0x8000 (\$gp) addi \$a1, \$0, 3
	0x00400008	0x20040002	sw \$a1, 0x8004 (\$gp) jal 0x0040002C
	0x0040000C	0xAF848000	sw \$v0, 0x8008 (\$gp) lw \$ra, 0 (\$sp)
	0x00400010	0x20050003	addi \$sp, \$sp, -4 jr \$ra
	0x00400014	0xAF858004	add \$v0, \$a0, \$a1 jr \$ra
	0x00400018	0x0C10000B	
	0x0040001C	0xAF828008	
	0x00400020	0x8FBF0000	
	0x00400024	0x23BD0004	
	0x00400028	0x03E00008	
	0x0040002C	0x00851020	
	0x00400030	0x03E00008	
Data segment	Address	Data	
	0x10000000	f g	
	0x10000004	y	
	0x10000008		

# Chương trình trong bộ nhớ





# Ví dụ lệnh giả (Pseudoinstruction)

---

Pseudoinstruction	MIPS Instructions
<code>li \$s0, 0x1234AA77</code>	<code>lui \$s0, 0x1234</code> <code>ori \$s0, 0xAA77</code>
<code>mul \$s0, \$s1, \$s2</code>	<code>mult \$s1, \$s2</code> <code>mflo \$s0</code>
<code>clear \$t0</code>	<code>add \$t0, \$0, \$0</code>
<code>move \$s1, \$s2</code>	<code>add \$s2, \$s1, \$0</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>

# Phụ lục: Kiến trúc tập lệnh Intel x86(\*)

---

- Sự tiến hóa của các bộ xử lý Intel
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

# Kiến trúc tập lệnh Intel x86

---

- i486 (1989): pipelined, on-chip caches and FPU
  - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
  - Later versions added MMX (Multi-Media eXtension) instructions
  - The infamous FDIIV bug
- Pentium Pro (1995), Pentium II (1997)
  - New microarchitecture
- Pentium III (1999)
  - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
  - New microarchitecture
  - Added SSE2 instructions
- Intel Core (2006)
  - Added SSE4 instructions, virtual machine support

# Các thanh ghi cơ bản của x86

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

# Các phương pháp định địa chỉ cơ bản

## ■ Hai toán hạng của lệnh

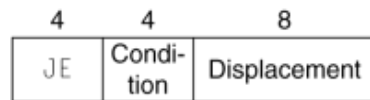
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

## ■ Các phương pháp định địa chỉ

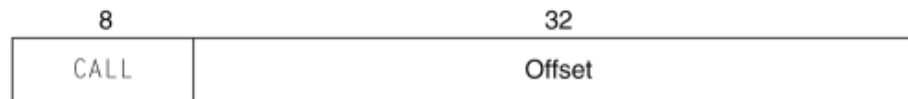
- Address in register
- $\text{Address} = R_{\text{base}} + \text{displacement}$
- $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$  (scale = 0, 1, 2, or 3)
- $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

# Mã hóa lệnh x86

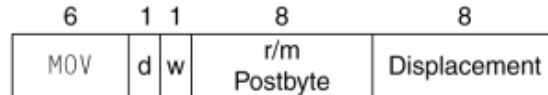
a. JE EIP + displacement



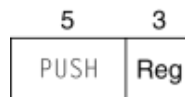
b. CALL



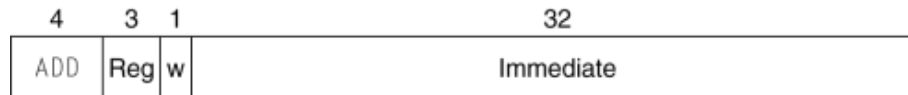
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42





# CPU pipeline

# Nội dung chính

---

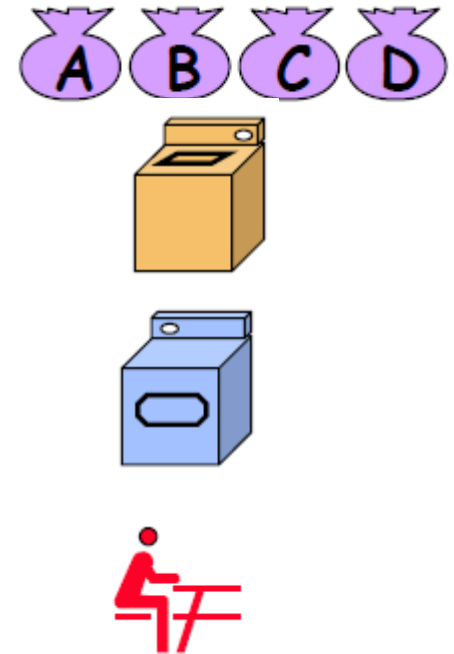
- Giới thiệu về CPU pipeline
- Các vấn đề của pipeline
- Xử lý xung đột dữ liệu và tài nguyên
- Xử lý rẽ nhánh (branch)
- Super pipeline



# Pipeline – Ví dụ thực tế

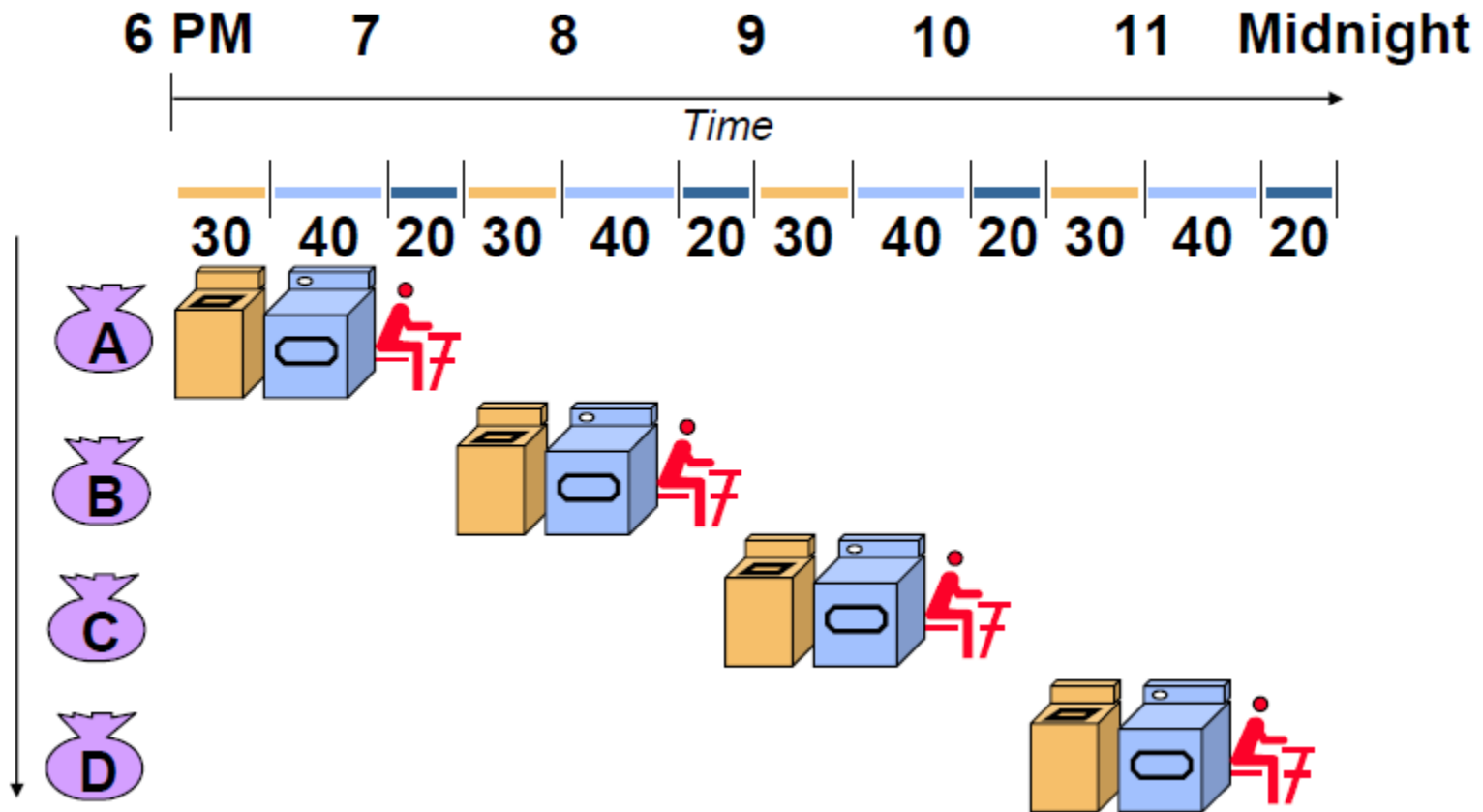
---

- ❑ Bài toán giặt: A, B, C, D có 4 túi quần áo cần giặt, làm khô, gấp
- ❑ Giặt tốn 30 phút
- ❑ Sấy khô: 40 phút
- ❑ Gấp: 20 phút



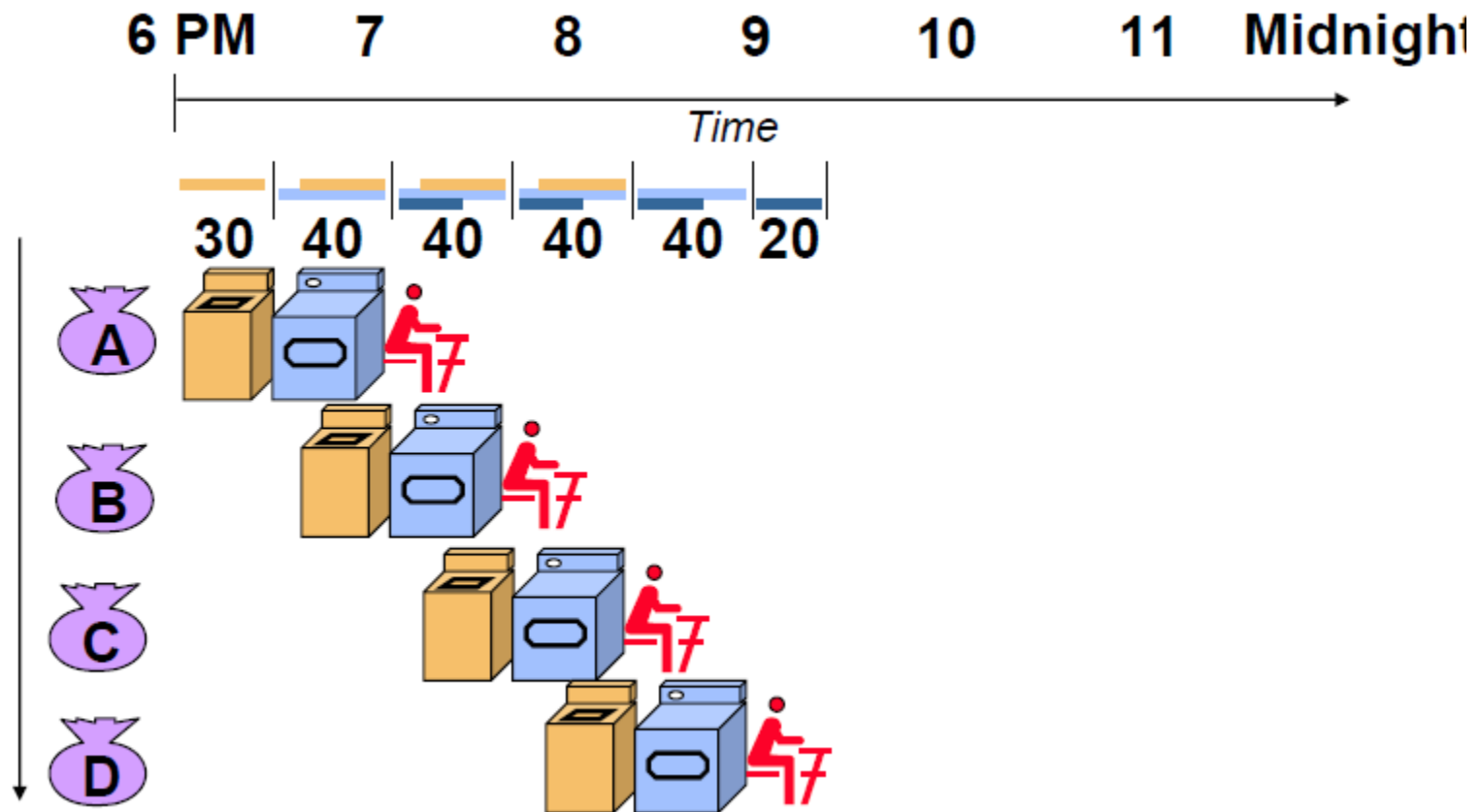
# Pipeline – Ví dụ thực tế

Thực hiện tuần tự



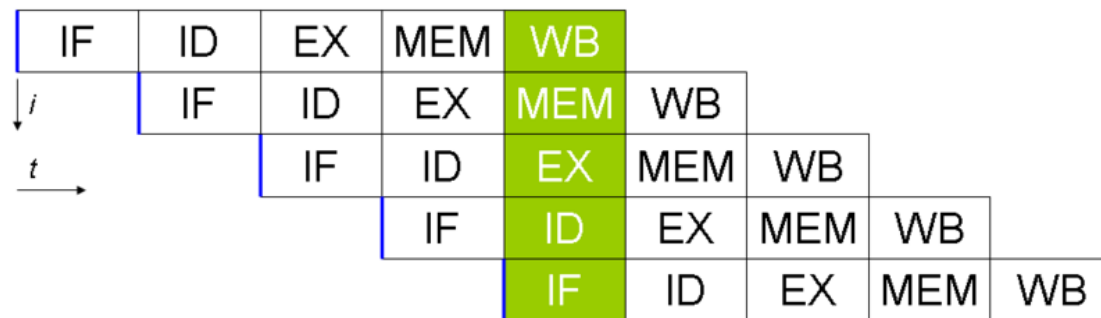
# Pipeline – Ví dụ thực tế

## Áp dụng pipeline

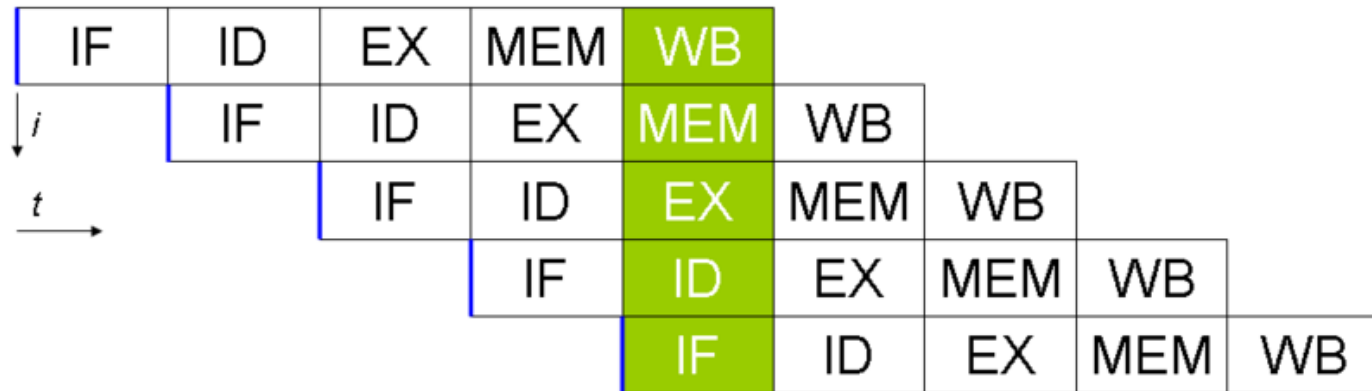


# Giới thiệu về CPU Pipeline – Nguyên lý

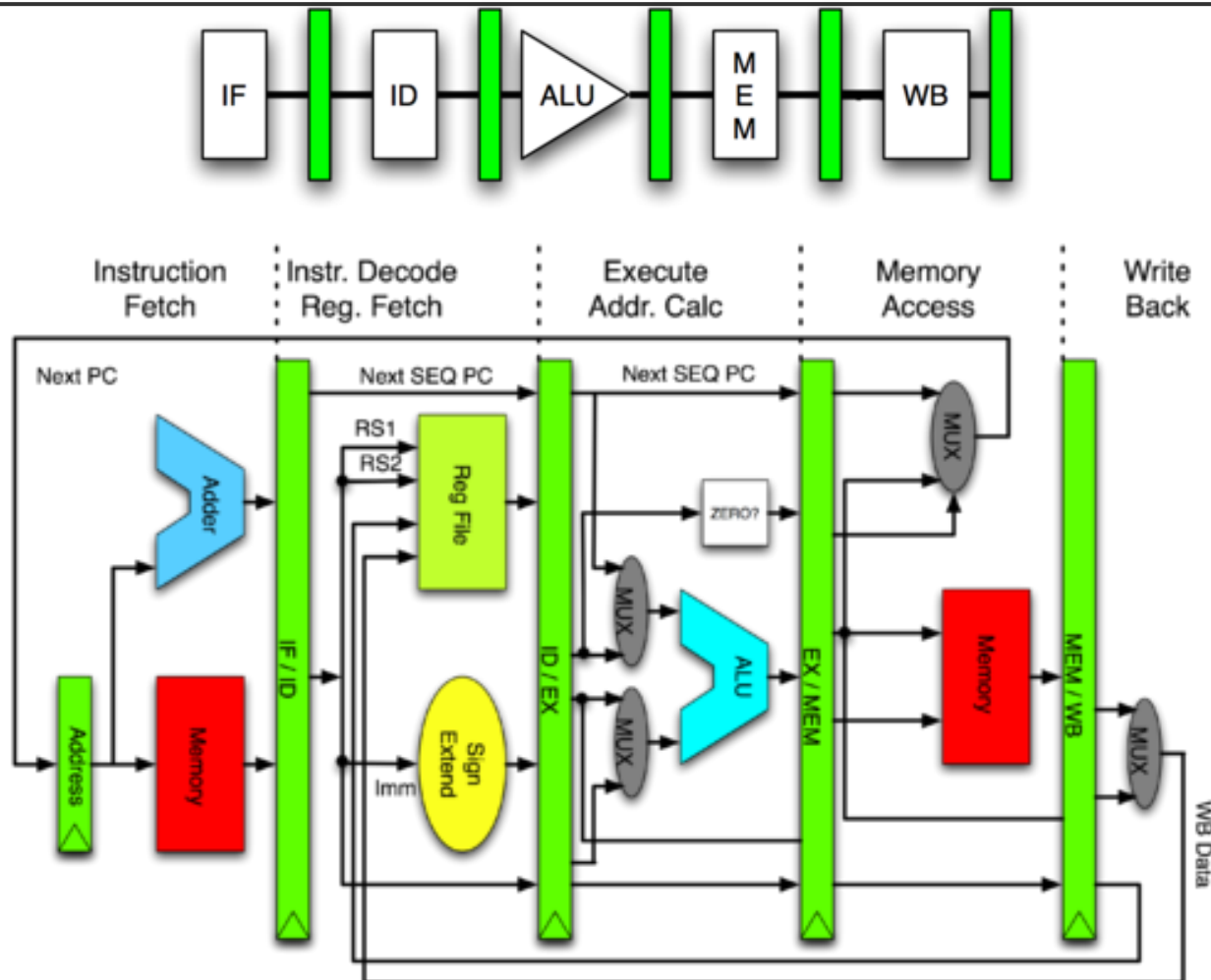
- ❑ Quá trình thực hiện lệnh được chia thành các giai đoạn
- ❑ 5 giai đoạn của hệ thống load – store:
  - Instruction fetch (IF): lấy lệnh từ bộ nhớ (hoặc cache)
  - Instruction Decode (ID): giải mã lệnh và lấy các toán hạng
  - Execute (EX): thực hiện lệnh: nếu là lệnh truy cập bộ nhớ thì tính toán địa chỉ bộ nhớ
  - Memory access (MEM): đọc/ ghi bộ nhớ ; nếu không truy cập bộ nhớ thì không có
  - Write back (WB): lưu kết quả vào thanh ghi
- ❑ Cải thiện hiệu năng bằng cách tăng số lượng lệnh vào xử lý



# Giới thiệu về CPU Pipeline – Nguyên lý



# Giới thiệu về CPU Pipeline – Nguyên lý



# Giới thiệu về CPU Pipeline – Đặc điểm

---

- ❑ Pipeline là kỹ thuật song song ở mức lệnh (ILP: Instruction Level Parallelism)
- ❑ Một pipeline là đầy đủ nếu nó luôn nhận một lệnh mới tại mỗi chu kỳ đồng hồ
- ❑ Một pipeline là không đầy đủ nếu có nhiều giai đoạn trễ trong quá trình xử lý
- ❑ Số lượng giai đoạn của pipeline phụ thuộc vào thiết kế CPU:
  - 2, 3, 5 giai đoạn: pipeline đơn giản
  - 14 giai đoạn: Pen II, Pen III
  - 20 – 31 giai đoạn: Pen IV
  - 12 -15 giai đoạn: Core

# Giới thiệu về CPU Pipeline

## Số lượng giai đoạn

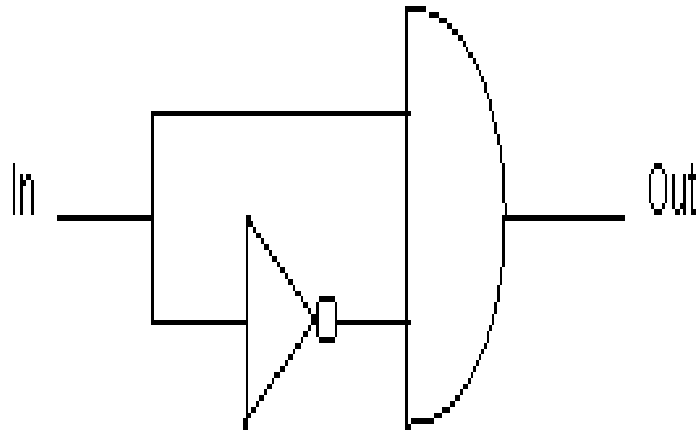
---

- Thời gian thực hiện của các giai đoạn:
  - Mọi giai đoạn nên có thời gian thực hiện bằng nhau
  - Các giai đoạn chậm nên chia ra
- Lựa chọn số lượng giai đoạn:
  - Theo lý thuyết, số lượng giai đoạn càng nhiều thì hiệu năng càng cao
  - Nếu pipeline dài mà rỗng vì một số lý do, sẽ mất nhiều thời gian để làm đầy pipeline



# Các vấn đề (rủi ro: hazard) của Pipeline

---



- ❑ Đầu ra mong muốn luôn là 0 (false)
  - ❑ Nhưng trong một số trường hợp, đầu ra là 1 (true)
- ⇒ Hazard

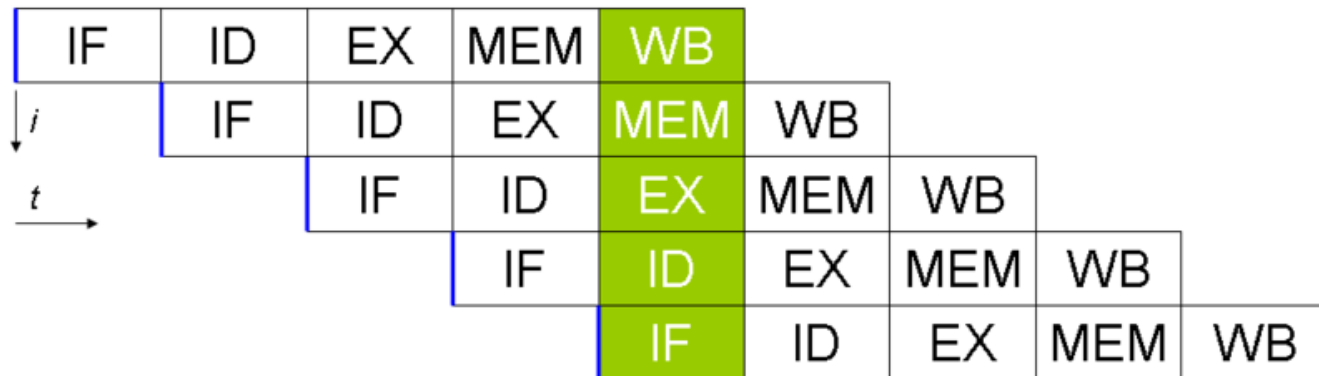
# Các vấn đề của Pipeline

---

- Vấn đề xung đột tài nguyên (resource conflict)
  - Xung đột truy cập bộ nhớ
  - Xung đột truy cập thanh ghi
- Xung đột/ tranh chấp dữ liệu (data hazard)
  - Hầu hết là RAW hay Read After Write Hazard
- Các lệnh rẽ nhánh (Branch Instruction)
  - Không điều kiện
  - Có điều kiện
  - Gọi thực hiện và trở về từ chương trình con

# Xung đột tài nguyên

- ❑ Tài nguyên không đủ
- ❑ Ví dụ: nếu bộ nhớ chỉ hỗ trợ một thao tác đọc/ ghi tại một thời điểm, pipeline yêu cầu 2 truy cập bộ nhớ 1 lúc (đọc lệnh tại giai đoạn IF và đọc dữ liệu tại ID)  
-> nảy sinh xung đột



# Xung đột tài nguyên

---

## □ Giải pháp:

- Nâng cao khả năng tài nguyên
- Memory/ cache: hỗ trợ nhiều thao tác đọc/ ghi cùng lúc
- Chia cache thành cache lệnh và cache dữ liệu để cải thiện truy nhập

# Xung đột dữ liệu

---

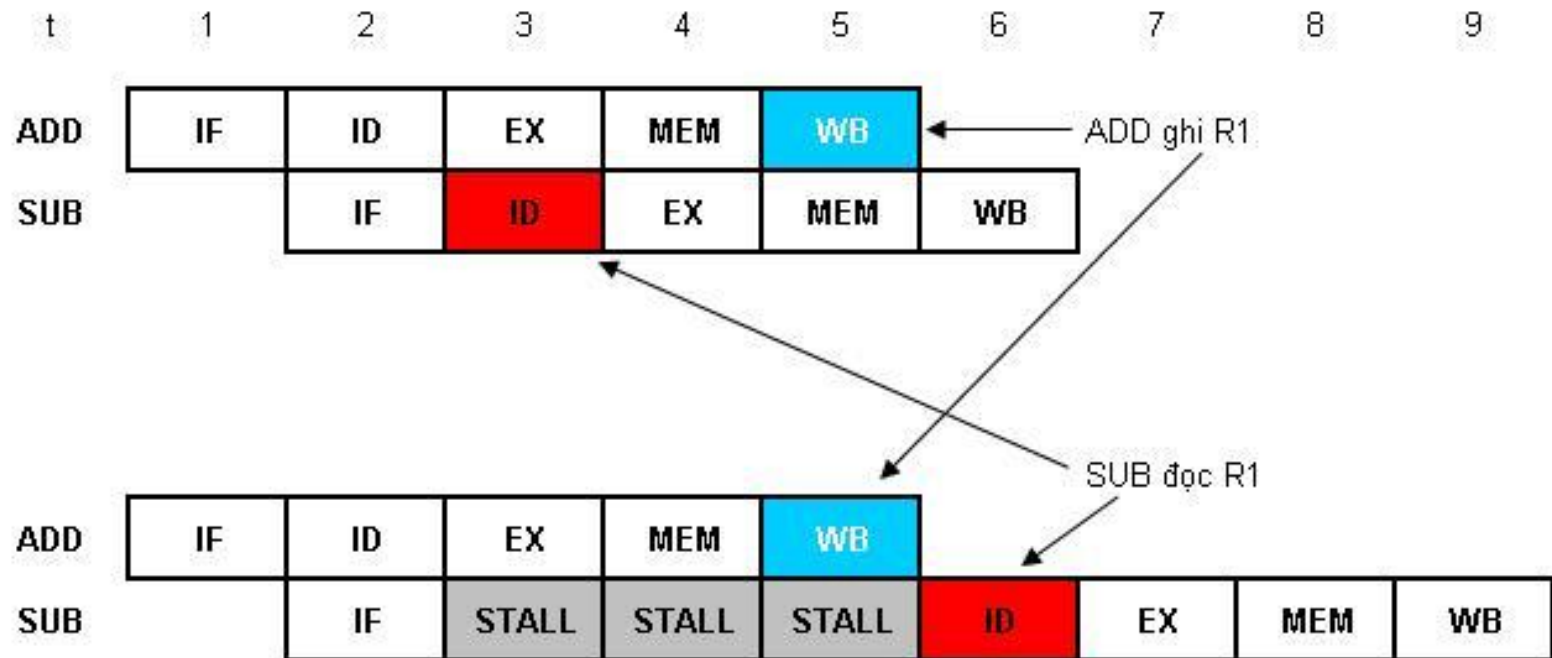
- Xét 2 lệnh sau:

ADD R1, R1, R3;  $R1 \leftarrow R1 + R3$

SUB R4, R1, R2;  $R4 \leftarrow R1 - R2$

- SUB sử dụng kết quả lệnh ADD: có phụ thuộc dữ liệu giữa 2 lệnh này
  - SUB đọc R1 tại giai đoạn 2 (ID); trong khi đó ADD lưu kết quả tại giai đoạn 5 (WB)
    - SUB đọc giá trị cũ của R1 trước khi ADD lưu trữ giá trị mới vào R1
- ⇒ *Dữ liệu chưa sẵn sàng cho các lệnh phụ thuộc tiếp theo*

# Pipeline hazard – xung đột dữ liệu



ADD R1, R1, R3;  $R1 \leftarrow R1 + R3$

SUB R4, R1, R2;  $R4 \leftarrow R1 - R2$

# Hướng khắc phục xung đột dữ liệu

---

- Nhận biết nó xảy ra
- Ngưng pipeline (stall): phải làm trể hoặc ngưng pipeline bằng cách sử dụng một vài phương pháp tới khi có dữ liệu chính xác
- Sử dụng compiler để nhận biết RAW và:
  - Chèn các lệnh NO-OP vào giữa các lệnh có RAW
  - Thay đổi trình tự các lệnh trong chương trình và chèn các lệnh độc lập dữ liệu vào vị trí giữa 2 lệnh có RAW
- Sử dụng phần cứng để xác định RAW (có trong các CPUs hiện đại) và dự đoán trước giá trị dữ liệu phụ thuộc

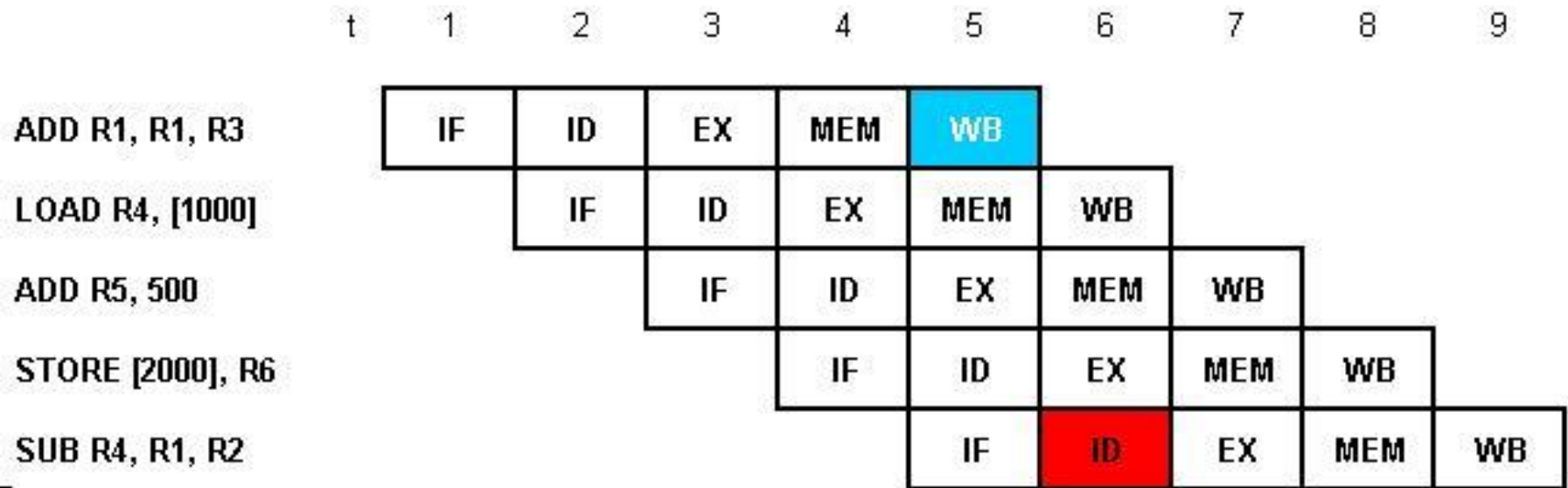
# Hướng khắc phục xung đột dữ liệu

t	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
NO-OP		NO-OP	NO-OP	NO-OP	NO-OP	NO-OP			
NO-OP			NO-OP	NO-OP	NO-OP	NO-OP	NO-OP		
NO-OP				NO-OP	NO-OP	NO-OP	NO-OP	NO-OP	
SUB					IF	ID	EX	MEM	WB

- ❑ Làm trễ quá trình thực hiện lệnh SUB bằng cách chèn 3 NO-OP



# Hướng khắc phục xung đột dữ liệu



- Chèn 3 lệnh độc lập dữ liệu vào giữa ADD và SUB

# Ví dụ

---

□ Viết chương trình tính:

$a = b + c;$

$d = e + f;$

# Bài Tập

---

- Xác định lỗi và bố trí lại các câu lệnh tránh trì hoãn khi thiết kế pipeline cho các câu lệnh sau:

load  $R_1, 0(R_0)$

load  $R_2, 4(R_0)$

Add  $R_3, R_1, R_2$

Store  $R_3, 12(R_0)$

load  $R_4, 8(R_0)$

Add  $R_5, R_1, R_4$

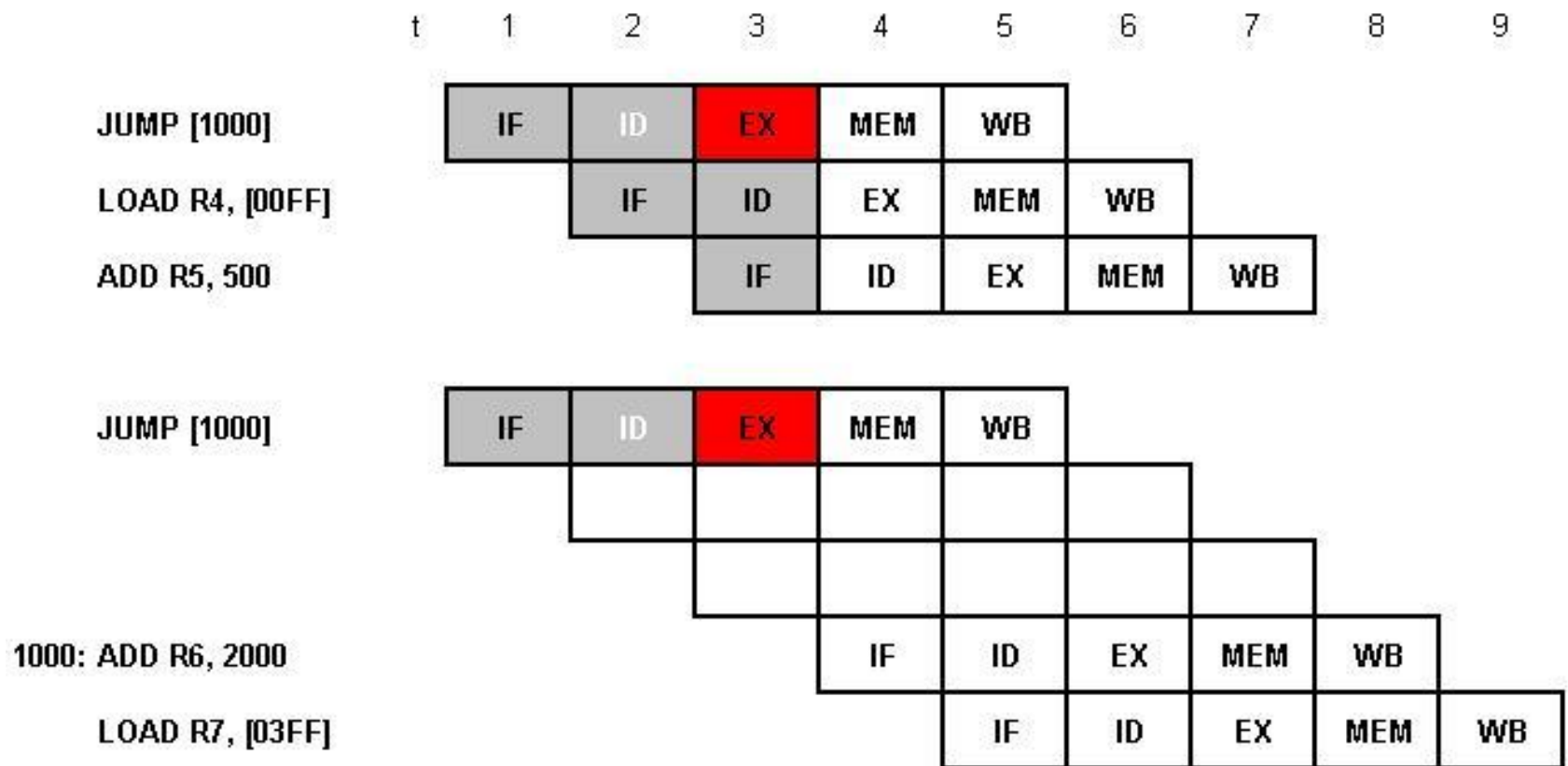
Store  $R_5, 16(R_0)$

# Quản lý các lệnh rẽ nhánh trong pipeline

---

- Tỷ lệ các lệnh rẽ nhánh chiếm khoảng 10 - 30%. Các lệnh rẽ nhánh có thể gây ra:
  - Gián đoạn trong quá trình chạy bình thường của chương trình
  - Làm cho Pipeline rỗng nếu không có biện pháp ngăn chặn hiệu quả
- Với các CPU mà pipeline dài (P4 với 31 giai đoạn) và nhiều pipeline chạy song song, vấn đề rẽ nhánh càng trở nên phức tạp hơn vì:
  - Phải đẩy mọi lệnh đang thực hiện ra ngoài pipeline khi gặp lệnh rẽ nhánh
  - Tải mới các lệnh từ địa chỉ rẽ nhánh vào pipeline. Tiêu tốn nhiều thời gian để điền đầy pipeline

# Quản lý các lệnh rẽ nhánh



- Khi 1 lệnh rẽ nhánh được thực hiện, các lệnh tiếp theo bị đẩy ra khỏi pipeline và các lệnh mới được tải

# Giải pháp quản lý các lệnh rẽ nhánh

---

- Đích rẽ nhánh (branch target)
- Rẽ nhánh có điều kiện (conditional branches)
  - Làm chậm rẽ nhánh (delayed branching)
  - Dự báo rẽ nhánh (branch prediction)

# Đích rẽ nhánh

---

- Khi một lệnh rẽ nhánh được thực hiện, lệnh tiếp theo được lấy là lệnh ở địa chỉ đích rẽ nhánh (target) chứ không phải lệnh tại vị trí tiếp theo lệnh nhảy

JUMP <Address>  
ADD R1, R2

Address: SUB R3, R4

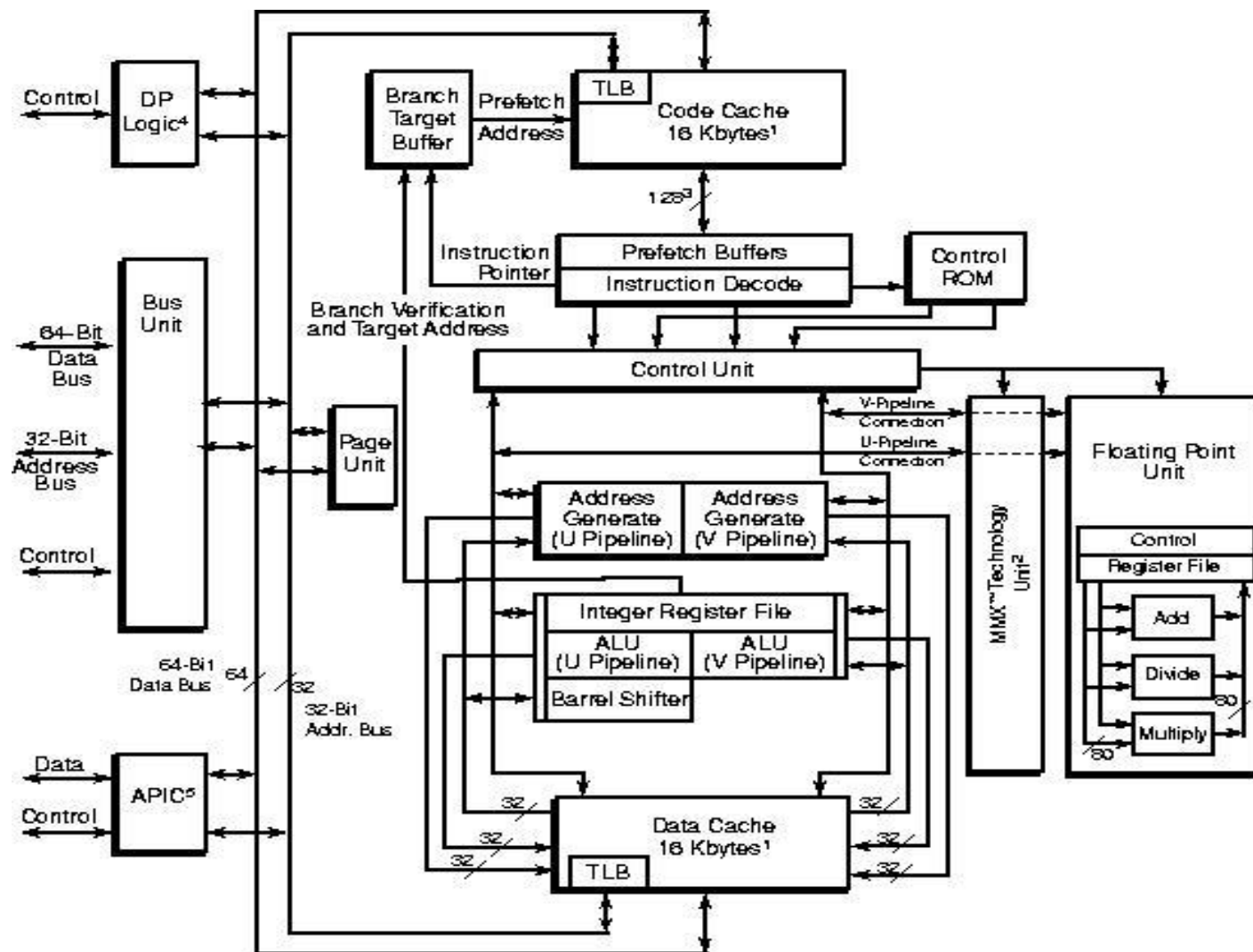
# Đích rẽ nhánh

---

- Các lệnh rẽ nhánh được xác định tại giai đoạn ID, vậy có thể biết trước chúng bằng cách giải mã trước
  - Sử dụng đệm đích rẽ nhánh (BTB: branch target buffer) để lưu vết của các lệnh rẽ nhánh đã được thực thi:
    - Địa chỉ đích của các lệnh rẽ nhánh đã được thực hiện
    - Lệnh đích của các lệnh rẽ nhánh đã được thực hiện
  - Nếu các lệnh rẽ nhánh được sử dụng lại (trong vòng lặp):
    - Các địa chỉ đích của chúng lưu trong BTB có thể được dùng mà không cần tính lại
    - Các lệnh đích có thể dùng trực tiếp không cần load lại từ bộ nhớ
- ⇒ Điều này có thể vì địa chỉ và lệnh đích thường không thay đổi



# Đích rẽ nhánh của PIII



# Lệnh rẽ nhánh có điều kiện

---

- Khó quản lý các lệnh rẽ nhánh có điều kiện hơn vì:
  - Có 2 lệnh đích để lựa chọn
  - Không thể xác định được lệnh đích tới khi lệnh rẽ nhánh được thực hiện xong
  - Sử dụng BTB riêng rẽ không hiệu quả vì phải đợi tới khi có thể xác định được lệnh đích.

# Lệnh nhảy có điều kiện – các chiến lược

- Làm chậm rẽ nhánh
- Dự đoán rẽ nhánh

# Làm chậm rẽ nhánh

---

## □ Dựa trên ý tưởng:

- Lệnh rẽ nhánh không làm rẽ nhánh ngay lập tức
- Mà nó sẽ bị làm chậm một vài chu kỳ đồng hồ phụ thuộc vào độ dài của pipeline

## □ Đặc điểm:

- Hoạt động tốt trên các vi xử lý RISC trong đó các lệnh có thời gian xử lý bằng nhau
- Pipeline ngắn (thông thường là 2 giai đoạn)
- Lệnh sau lệnh nhảy luôn được thực hiện, không phụ thuộc vào kết quả lệnh rẽ nhánh

# Làm chậm rẽ nhánh

---

## □ Cài đặt:

- Sử dụng compiler để chèn NO-OP vào vị trí ngay sau lệnh rẽ nhánh, hoặc
- Chuyển một lệnh độc lập từ trước tới ngay sau lệnh rẽ nhánh

# Làm chậm rẽ nhánh

---

## ❑ Xét các lệnh:

ADD R2, R3, R4  
CMP R1,0  
JNE somewhere

## ❑ Chèn NO-OP vào vị trí ngay sau lệnh rẽ nhánh

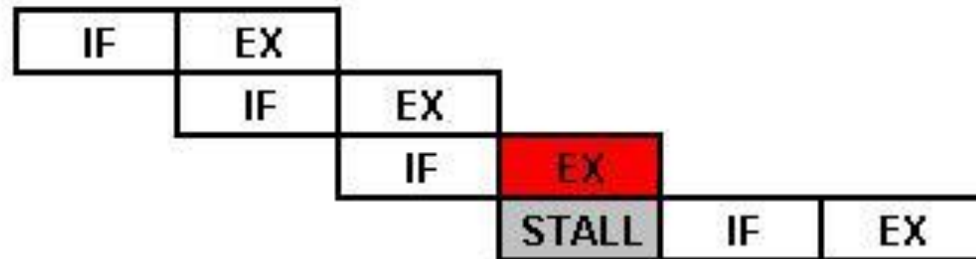
ADD R2, R3, R4  
CMP R1,0  
JNE somewhere  
NO-OP

## ❑ Chuyển một lệnh độc lập từ trước tới ngay sau lệnh rẽ nhánh

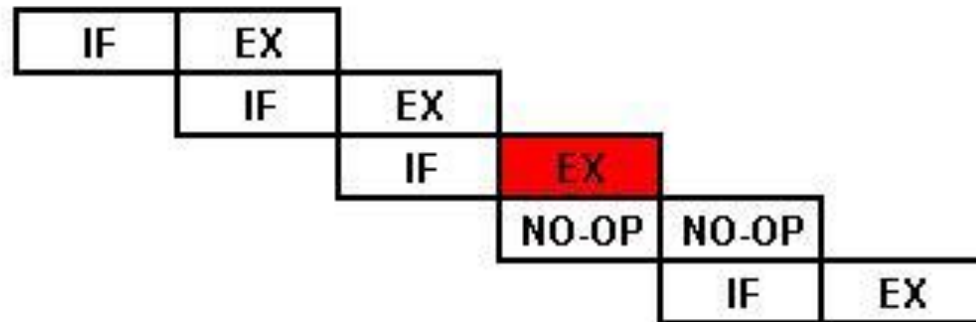
CMP R1,0  
JNE somewhere  
ADD R2, R3, R4

# Làm chậm rẽ nhánh

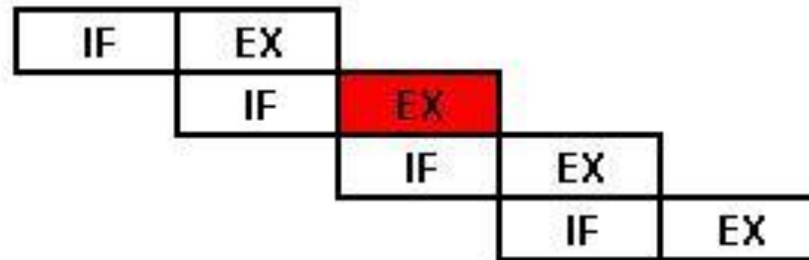
ADD R2, R3, R4  
CMP R1,0  
JNE somewhere  
SUB R5, R6, R7



ADD R2, R3, R4  
CMP R1,0  
JNE somewhere  
NO-OP  
SUB R5, R6, R7



CMP R1,0  
JNE somewhere  
ADD R2, R3, R4  
SUB R5, R6, R7



# Làm chậm rẽ nhánh – các nhận xét

---

- ❑ Dễ cài đặt nhờ tối ưu trình biên dịch (complier)
- ❑ Không cần phần cứng đặc biệt
- ❑ Nếu chỉ chèn NO-OP làm giảm hiệu năng khi pipeline dài
- ❑ Thay các lệnh NO-OP bằng các lệnh độc lập có thể làm giảm số lượng NO-OP cần thiết tới 70%



# Làm chậm rẽ nhánh – các nhận xét

---

- ❑ Làm tăng độ phức tạp mã chương trình (code)
- ❑ Cần lập trình viên và người xây dựng trình biên dịch có mức độ hiểu biết sâu về pipeline vi xử lý => hạn chế lớn
- ❑ Giảm tính khả chuyển (portable) của mã chương trình vì các chương trình phải được viết hoặc biên dịch lại trên các nền VXL mới

# Dự đoán rẽ nhánh

---

- Có thể dự đoán lệnh đích của lệnh rẽ nhánh:
  - Dự đoán đúng: nâng cao hiệu năng
  - Dự đoán sai: đẩy các lệnh tiếp theo đã load và phải load lại các lệnh tại đích rẽ nhánh
  - Trường hợp xấu của dự đoán là 50% đúng và 50% sai

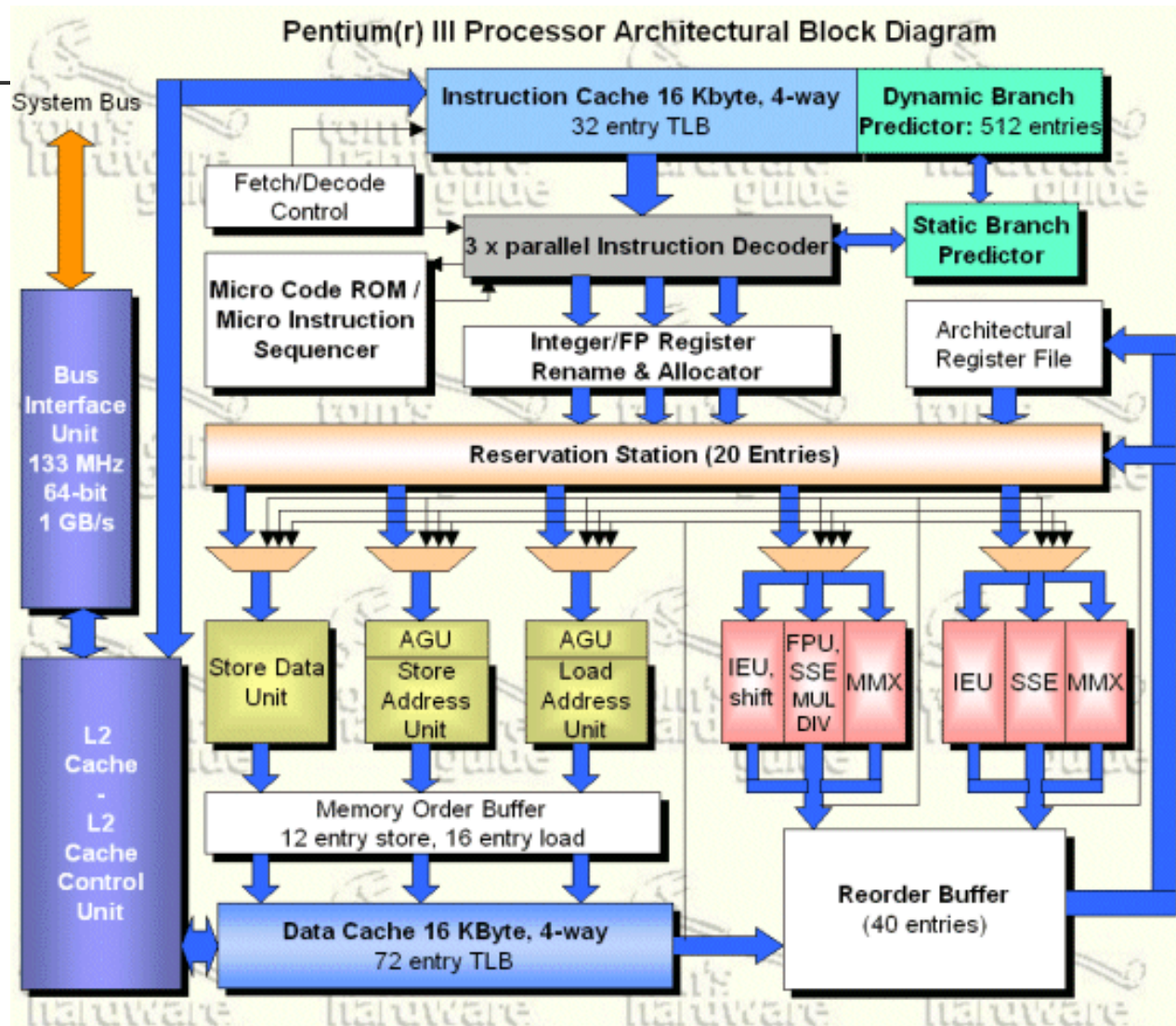
# Dự đoán rẽ nhánh

---

## □ Các cơ sở để dự đoán:

- Đối với các lệnh nhảy ngược (backward):
  - Thường là một phần của vòng lặp
  - Các vòng lặp thường được thực hiện nhiều lần
- Đối với các lệnh nhảy xuôi (forward), khó dự đoán hơn:
  - Có thể là kết thúc lệnh loop
  - Có thể là nhảy có điều kiện

# Branch Prediction – Intel PIII



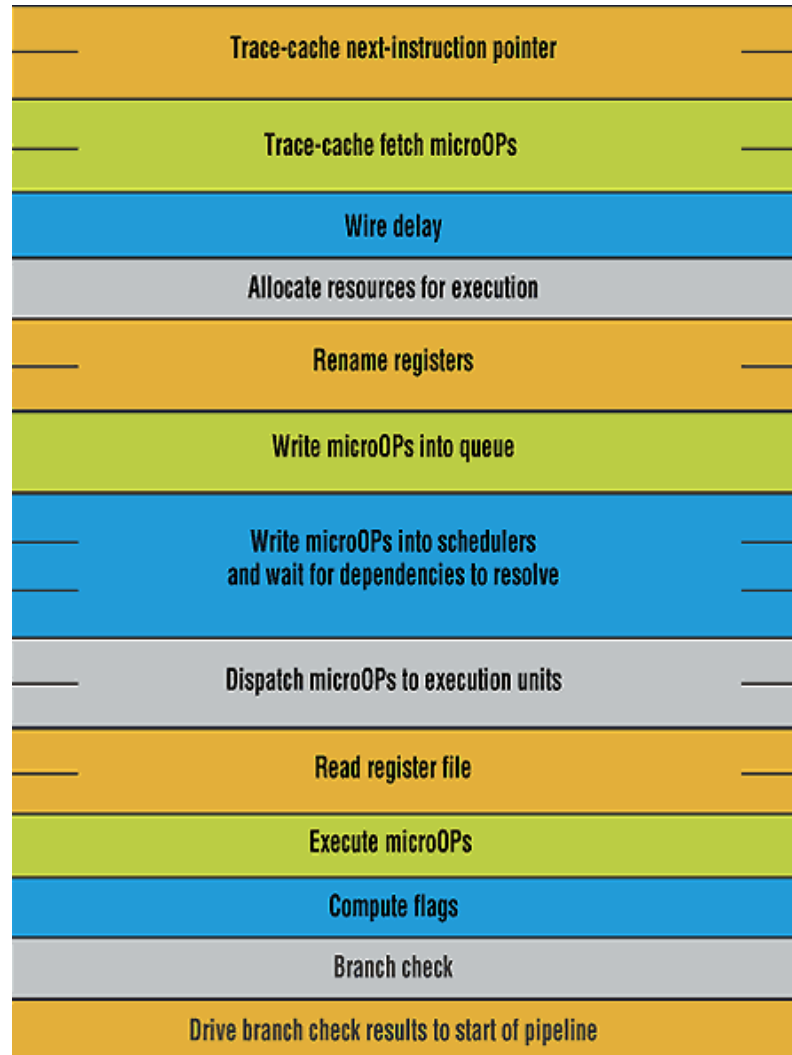
# Siêu pipeline (superpipelining)

- ❑ Siêu pipeline là kỹ thuật cho phép:
  - Tăng độ sâu ống lệnh
  - Tăng tốc độ đồng hồ
  - Giảm thời gian trễ cho từng giai đoạn thực hiện lệnh
- ❑ Ví dụ: nếu giai đoạn thực hiện lệnh bởi ALU kéo dài -> chia thành một số giai đoạn nhỏ -> giảm thời gian chờ cho các giai đoạn ngắn
- ❑ Pentium 4 siêu ống với 20 giai đoạn

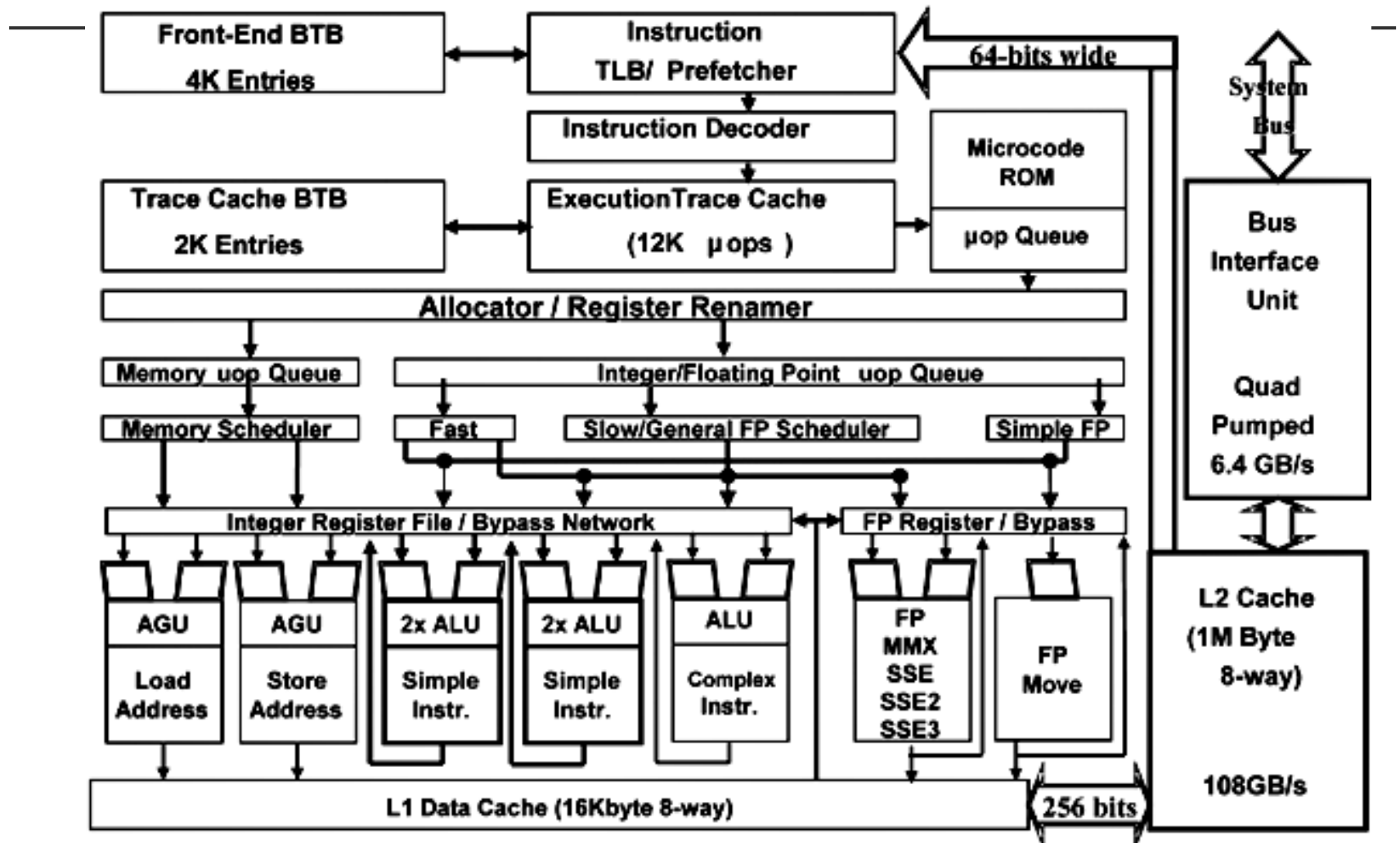


# Pentium 4 siêu ống với 20 giai đoạn

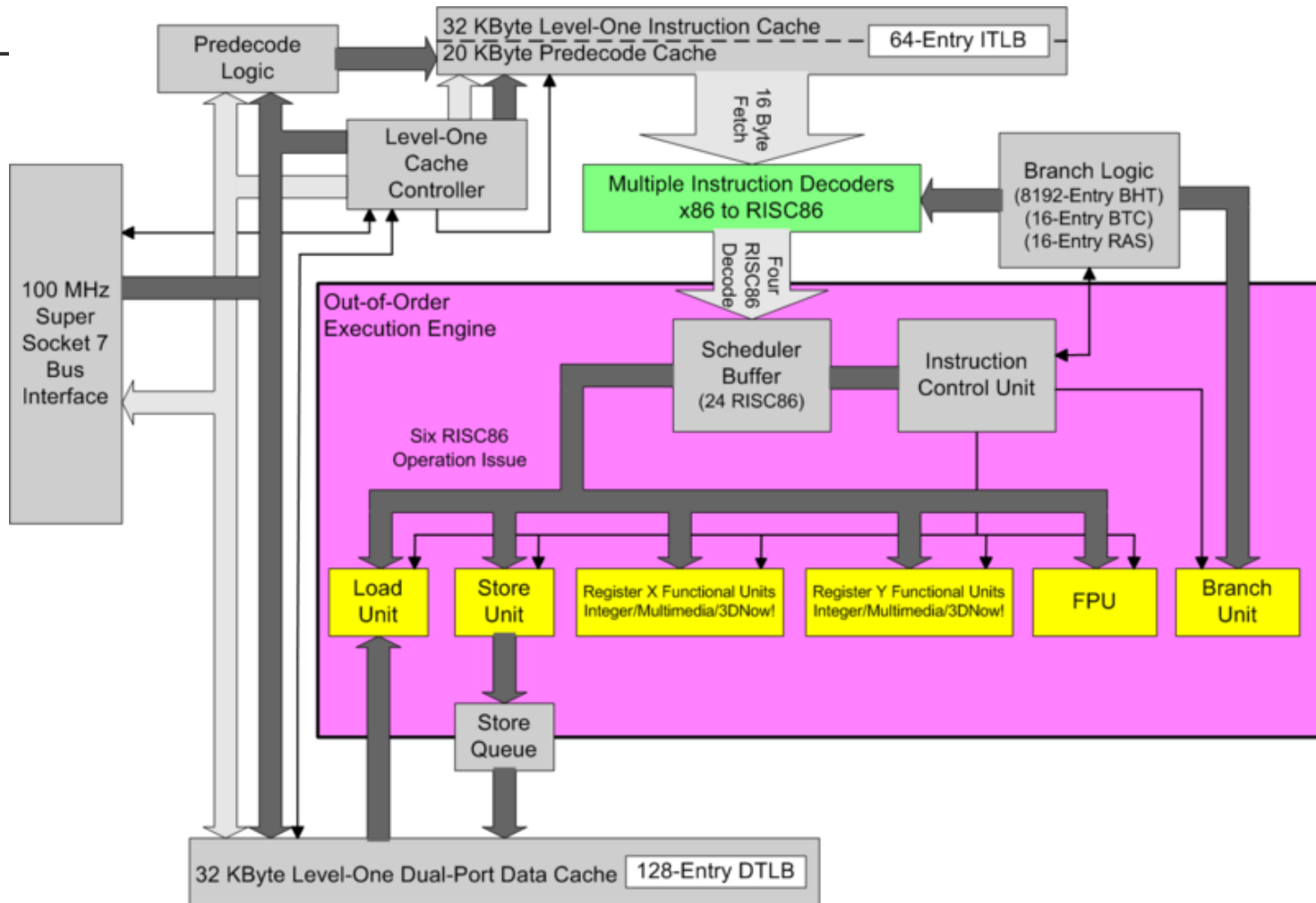
---



# Branch Prediction – Intel P4



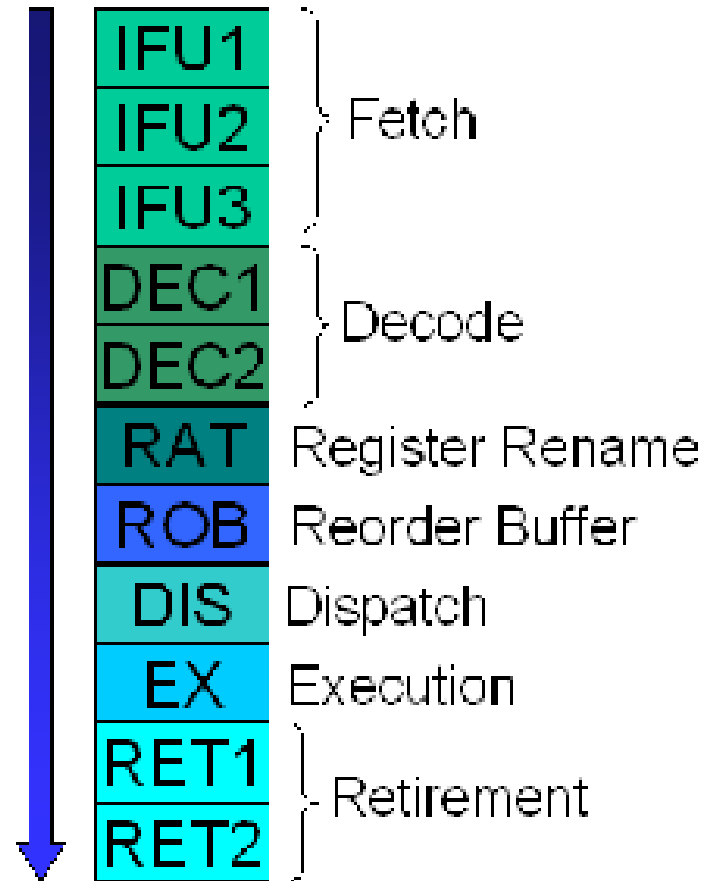
# AMD K6-2 pipeline



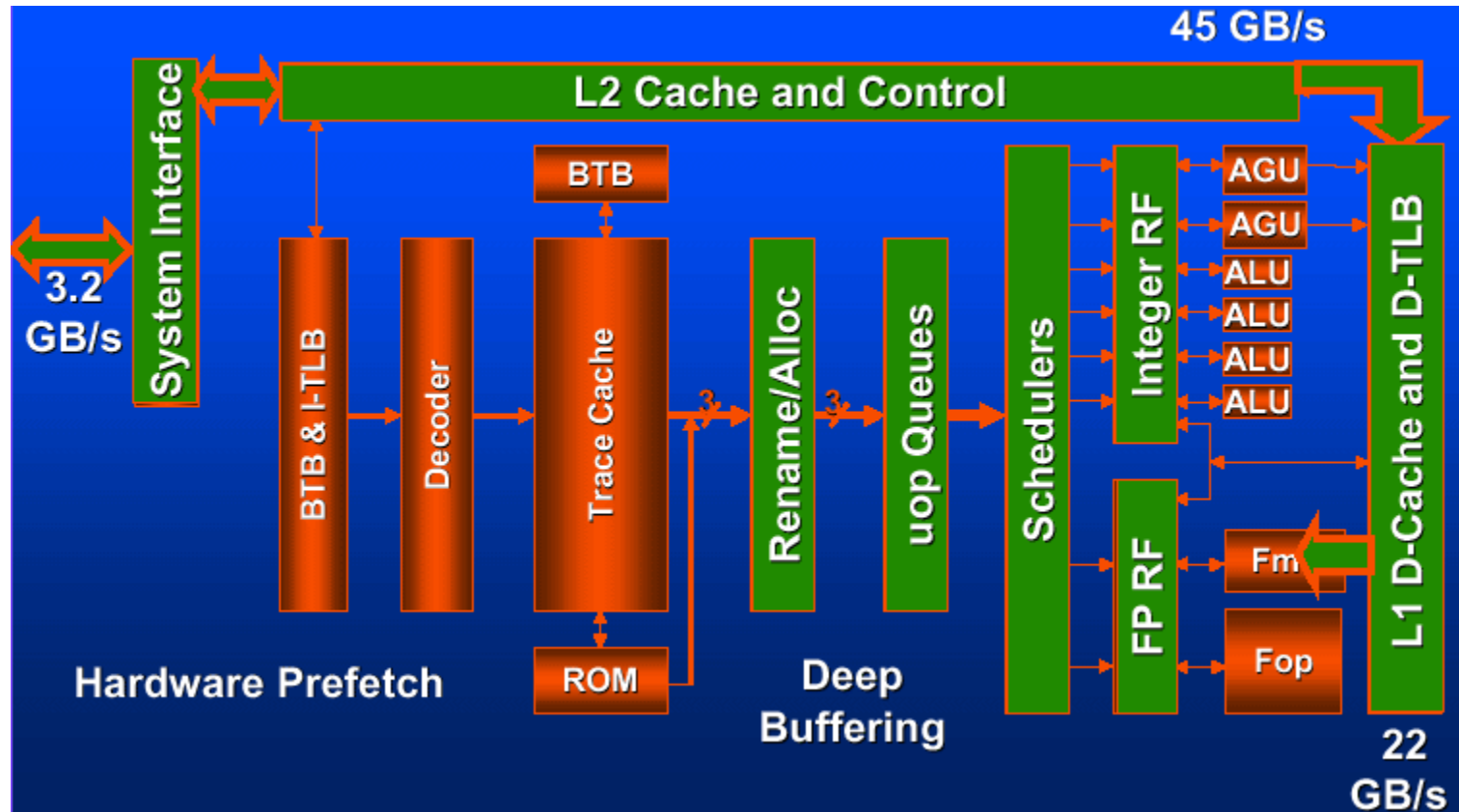


# Pipeline –Pen III, M

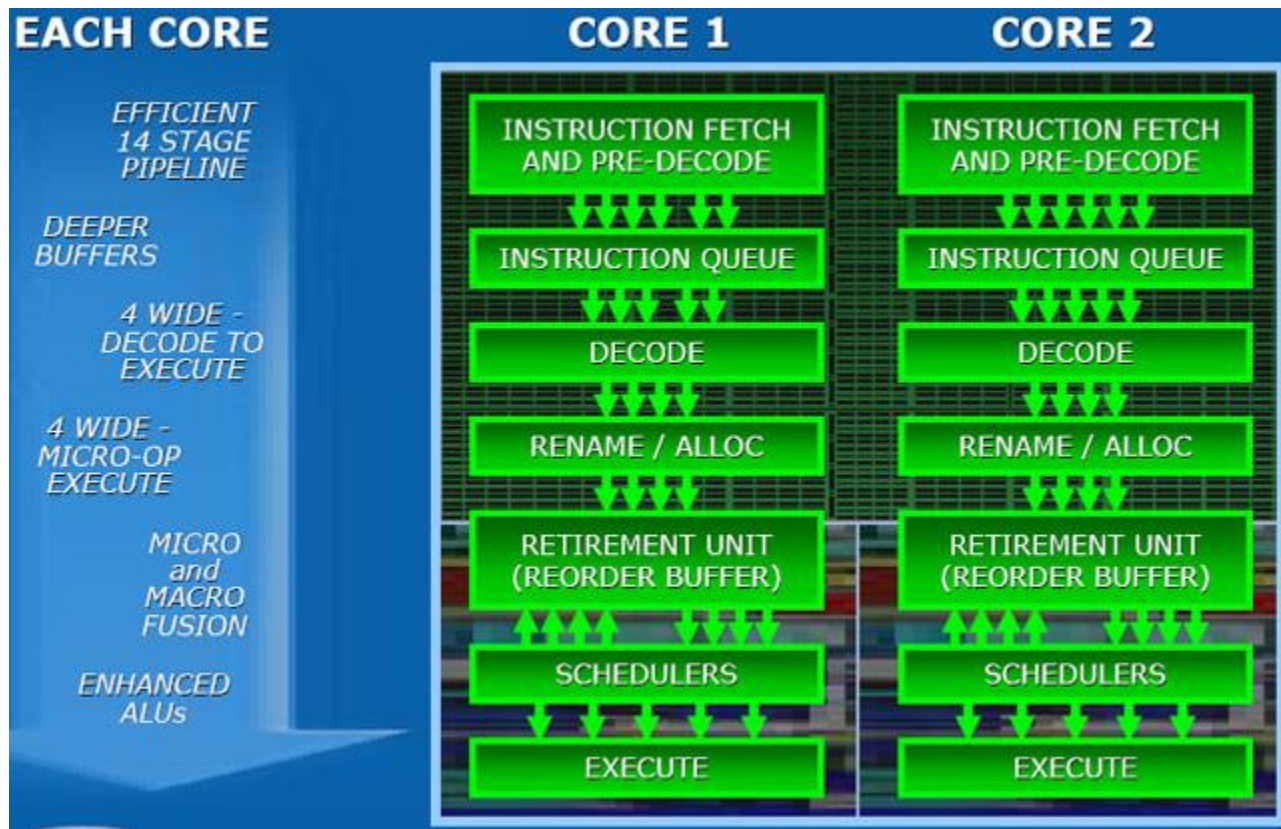
---



# Intel Pen 4 Pipeline

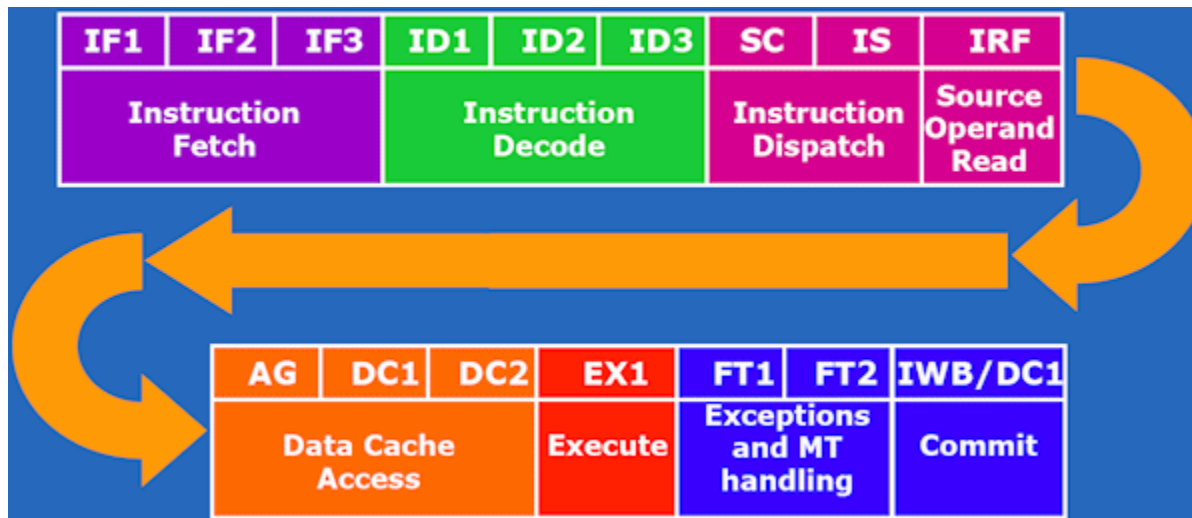


# Intel Core 2 Duo pipeline



# Intel Atom 16-stage pipeline

---



# Intel Core 2 Duo – Super Pipeline

