



Khoa  
**CÔNG NGHỆ THÔNG TIN**  
ĐH Khoa học Tự nhiên TP HCM

# Bài 05: Kiến trúc MIPS

**Phạm Tuấn Sơn**

**[ptson@fit.hcmus.edu.vn](mailto:ptson@fit.hcmus.edu.vn)**



# Mục tiêu

- Sau bài này, SV có khả năng:
  - Giải thích quan điểm thiết kế bộ lệnh MIPS
  - Có khả năng lập trình hợp ngữ MIPS





# Bộ lệnh

- Công việc cơ bản nhất của bộ xử lý là xử lý các lệnh máy (*instruction*).
- Tập hợp các lệnh mà một bộ xử lý nào đó cài đặt gọi là bộ lệnh (*Instruction Set*).
- Các bộ xử lý khác nhau cài đặt các bộ lệnh khác nhau.
  - Ví dụ: Pentium 4 (Intel), MIPS R3000 (MIPS Technology Inc), ARM2 (ARM), PowerPC 601 (IBM), SPARC V8 (Sun),...
- Câu hỏi
  - Một chương trình thực thi (.exe) chạy trên bộ xử lý Pentium 3 (Intel) có thể chạy được trên bộ xử lý Pentium 4 (Intel) không ?
  - Một chương trình thực thi (.exe) chạy trên một bộ xử lý của Intel có thể chạy được trên bộ xử lý của AMD ?



# Kiến trúc bộ lệnh

- Các bộ xử lý khác nhau có cùng kiến trúc bộ lệnh (Instruction Set Architecture - ISA) có thể thực thi cùng một chương trình
- x86 (máy tính cá nhân – PC, laptop, netbook)
  - x86-32 (IA-32/ i386): Intel 80386, Intel 80486, Intel Pentium, AMD Am386, AMD Am486, AMD K5, AMD K6,...
  - x86-64: Intel 64 (Intel Pentium D, Intel Core 2, Intel Core i7, Intel Atom,...), AMD64 (AMD Athlon 64, AMD Phenom , ...)
- IA-64: Pentium Itanium (máy chủ - server)
- MIPS (hệ thống nhúng – embedded system và siêu máy tính – supercomputer)
  - MIPS32: R2000, R3000, R6000,...
  - MIPS64: R4000, R5000, R8000,...
- Ngoài ra, PowerPC (máy chủ, hệ thống nhúng), SPARC (máy chủ), ARM (hệ thống nhúng), ...



## 4 nguyên tắc thiết kế bộ lệnh MIPS

- Cấu trúc lệnh đơn giản và có quy tắc  
(Simplicity favors regularity)
- Lệnh và bộ lệnh càng nhỏ gọn càng xử lý nhanh  
(Smaller is faster)
- Tăng tốc độ xử lý cho những trường hợp thường xuyên xảy ra  
(Make the common case fast)
- Thiết kế tốt đòi hỏi sự thỏa hiệp tốt  
(Good design demands good compromises)



# Một số khảo sát và nhận xét

- MIPS chỉ cần hỗ trợ 32 thanh ghi là đủ, đánh số từ \$0 - \$31
- Mỗi thanh ghi có kích thước 32 bit (4 byte)
- Các phép toán luận lý và số học như

$$a = b + c$$

$$a = b \& c$$

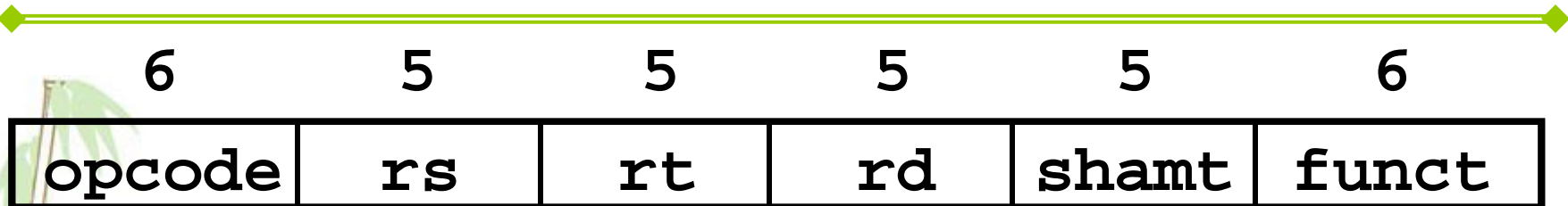
$$a = b \ll 3$$

gồm:

- Loại phép toán
- 2 toán hạng nguồn + 1 toán hạng đích
  - Để đơn giản và thao tác nhanh, các toán hạng là địa chỉ thanh ghi (không là địa chỉ bộ nhớ)
  - Trong phép dịch, toán hạng thứ 2 là hằng số
- MIPS hỗ trợ nhiều loại lệnh khác nhau: lệnh tính toán số học, luận lý, lệnh thao tác bộ nhớ, lệnh rẽ nhánh,...
- Để đơn giản và dễ dàng trong việc truy xuất bộ nhớ, tất cả các lệnh đều có chiều dài 32 bit
  - Trong MIPS, nhóm 32 bit được gọi là một từ (*word*)
- Từ đó, MIPS đưa ra cấu trúc lệnh như slide sau



# Cấu trúc lệnh R-Format (1/2)



- opcode: mã thao tác, cho biết loại lệnh gì
- funct: dùng kết hợp với opcode để xác định lệnh làm gì (trường hợp các lệnh có cùng mã thao tác opcode)
  - *Tại sao 2 trường opcode và funct không nằm liền nhau ?*
- shamt: trường này chứa số bit cần dịch trong các lệnh dịch.
  - Trường này có kích thước 5 bit, nghĩa là biểu diễn được các số từ 0-31 (đủ để dịch các bit trong 1 thanh ghi 32 bit).
  - Nếu không phải lệnh dịch thì trường này có giá trị 0.
  - *Tại sao không dùng rt làm số bit dịch ?*



## Cấu trúc R-Format (2/2)

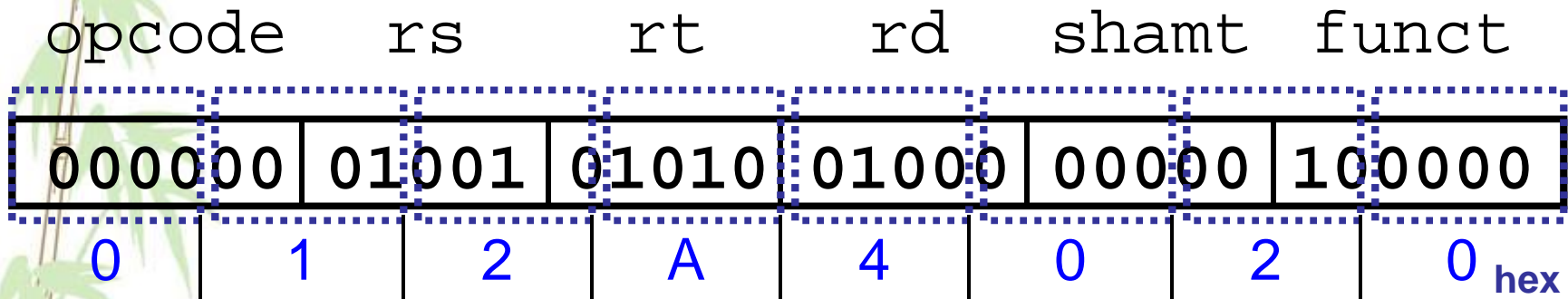
- rs (Source Register): chứa địa chỉ thanh ghi nguồn thứ 1
- rt (Target Register): chứa địa chỉ thanh ghi nguồn thứ 2 (sai tên ?)
- rd (Destination Register): chứa địa chỉ thanh ghi đích
  - Mỗi trường có kích thước 5 bit, nghĩa là biểu diễn được các số từ 0-31 (đủ để biểu diễn 32 thanh ghi của MIPS)





## Ví dụ cấu trúc lệnh R-Format

Lệnh máy (dưới dạng nhị phân)



Giá trị thập phân tương ứng của từng trường

0	9	10	8	0	32
---	---	----	---	---	----

opcode = 0 } Xác định thao tác cộng (các lệnh theo cấu trúc  
funct = 32 } R-Format có trường mã thao tác opcode = 0)

rd = 8 (toán hạng đích là thanh ghi \$8)

rs = 9 (toán hạng nguồn thứ 1 là thanh ghi \$9)

rt = 10 (toán hạng nguồn thứ 2 là thanh ghi \$10)

shamt = 0 (không phải lệnh dịch)

$$\$8 = \$9 + \$10$$



# Lệnh hợp ngữ số học và luận lý

- Cú pháp: `opt opr, opr1, opr2`

– Trong đó:

`opt` – Tên thao tác (toán tử)

`opr` – Thanh ghi (toán hạng đích) chứa kết quả

`opr1` – Thanh ghi (toán hạng nguồn thứ 1)

`opr2` – Thanh ghi hoặc hằng số (toán hạng nguồn thứ 2)



# Một số đặc điểm của toán hạng thanh ghi

- Đóng vai trò giống như biến trong các NNLT cấp cao (C, Java). Tuy nhiên, khác với biến chỉ có thể giữ giá trị theo kiểu dữ liệu được khai báo trước khi sử dụng, thanh ghi không có kiểu, thao tác trên thanh ghi sẽ xác định dữ liệu trong thanh ghi sẽ được đối xử như thế nào.
- Ưu điểm: bộ xử lý truy xuất thanh ghi nhanh nhất (hơn 1 tỉ lần trong 1 giây) vì thanh ghi là một thành phần phần cứng thường nằm chung mạch với bộ xử lý.
- Khuyết điểm: do thanh ghi là một thành phần phần cứng nên số lượng cố định và hạn chế. Do đó, sử dụng phải khéo léo.
- 8 thanh ghi thường được sử dụng để lưu các biến là \$16 – \$23, được đặt tên gọi nhớ như sau

\$16 – \$23      ~      \$s0 – \$s7 (saved register)



# Cộng, trừ số nguyên (1/4)

- **Lệnh cộng:**

add     \$s0 , \$s1 , \$s2 (cộng có dấu trong MIPS)

addu    \$s0 , \$s1 , \$s2 (cộng không dấu trong MIPS)

tương ứng với:  $a = b + c$  (trong C)

trong đó các thanh ghi \$s0 , \$s1 , \$s2 (trong MIPS)  
tương ứng với các biến a, b, c (trong C)

- **Lệnh trừ:**

sub     \$s3 , \$s4 , \$s5 (trừ có dấu trong MIPS)

subu    \$s3 , \$s4 , \$s5 (trừ không dấu trong MIPS)

tương ứng với:  $d = e - f$  (trong C)

trong đó các thanh ghi \$s3 , \$s4 , \$s5 (trong MIPS)  
tương ứng với các biến d, e, f (trong C)



## Cộng, trừ số nguyên (2/4)

- Lưu ý: toán hạng trong các lệnh trên phải là thanh ghi
- Trong MIPS, lệnh thao tác với số không dấu có ký tự cuối là “u” – *unsigned*. Các thao tác khác là thao tác với số có dấu. Số nguyên có dấu được biểu diễn dưới dạng bù 2.
- Làm sao biết được một phép toán (ví dụ  $a = b + c$  trong C) là thao tác trên số có dấu hay không dấu để biên dịch thành lệnh máy tương ứng (add hay addu) ?



# Cộng, trừ số nguyên (3/4)

- Làm thế nào để thực hiện câu lệnh C sau đây bằng lệnh máy MIPS?

$$a = b + c + d - e$$

- Chia nhỏ thành nhiều lệnh máy

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

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

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

- Chú ý: một lệnh trong C có thể gồm nhiều lệnh máy.
- Ghi chú: ký tự “#” dùng để chú thích trong hợp ngữ cho MIPS
- *Tại sao không xây dựng các lệnh MIPS có nhiều toán hạng nguồn hơn ?*



# Cộng, trừ số nguyên (4/4)

- Làm thế nào để thực hiện dãy tính sau?

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

- 8 thanh ghi thường được sử dụng để lưu tạm kết quả trung gian, đánh số \$8 - \$15

\$8 - \$15 ~ \$t0 - \$t7 (temporary register)

- Như vậy dãy tính trên có thể được thực hiện như sau:

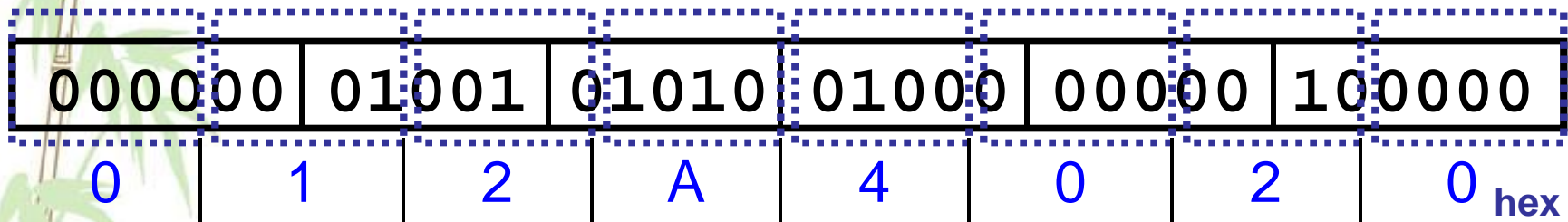
```
add $t0,$s1,$s2 # temp = g + h
add $t1,$s3,$s4 # temp = i + j
sub $s0,$t0,$t1 # f=(g+h)-(i+j)
```



# Ví dụ mã máy của lệnh add

add \$t0, \$t1, \$t2

Mã máy (dưới dạng nhị phân)



Giá trị thập phân tương ứng của từng trường

0	9	10	8	0	32
---	---	----	---	---	----

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





# Thanh ghi Zero

- Làm sao để thực hiện phép gán trong MIPS ?
- MIPS định nghĩa thanh ghi zero ( \$0 hay \$zero) luôn mang giá trị 0 nhằm hỗ trợ thực hiện phép gán và các thao với 0.

Ví dụ:

`add $s0, $s1, $zero` (trong MIPS)

tương ứng với `f = g` (trong C)

Trong đó các thanh ghi \$s0, \$s1 (trong MIPS) tương ứng với các biến f, g (trong C)

Lệnh `add $zero, $zero, $s0` Hợp lệ ? Ý nghĩa ?

- *Tại sao không có lệnh gán trực tiếp giá trị của 1 thanh ghi vào 1 thanh ghi ?*



# Tính toán luận lý

- Các lệnh:

- and, or: toán hạng nguồn thứ 2 phải là thanh ghi

- and `$t0, $t0, $t1`

- or `$t0, $t0, $t1`

- nor: toán hạng nguồn thứ 2 phải là thanh ghi

- nor `$t0, $t1, $t3 # $t1 = ~( $t1 | $t3 )`

- ~~– not:~~

- $A \text{ nor } 0 = \text{not } (A \text{ or } 0) = \text{not } (A)$

- *Tại sao không có lệnh `not` mà lại sử dụng lệnh `nor` thay cho lệnh `not` ?*

- *Tại sao không có các lệnh tính toán luận lý còn lại như: `xor`, `nand`, ... ?*

# Dịch

- `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`  
( $85 \times 2^2$ )

- `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`  
( $85/2^2$ )

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

→ `$s2 = 1111 1111 1111 1111 1111 1111 1111 0000 = -16`

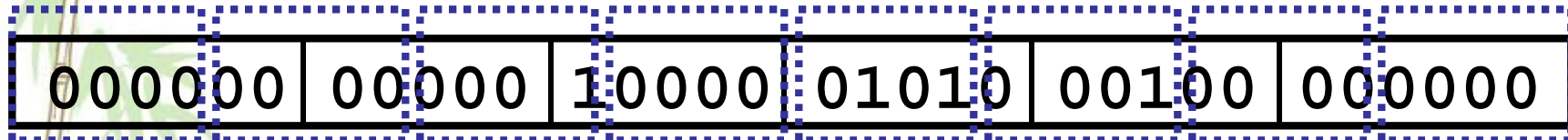
`$s1 = 1111 1111 1111 1111 1111 1111 1100 0000 = -4`  
( $-16/2^2$ )

Toán hạng nguồn thứ 2 phải là hằng số

## Ví dụ mã máy của lệnh sll

sll \$t2, \$s0, 4

Mã máy (dưới dạng nhị phân)



0 | 0 | 1 | 0 | 5 | 1 | 0 | 0<sub>hex</sub>

Giá trị thập phân tương ứng của từng trường

0	0	16	10	4	0
---	---	----	----	---	---

opcode = 0  
funct = 0 } Xác định thao tác dịch trái luận lý

rd = 10 (toán hạng đích là \$10 ~ \$t2)

rs = 0 (không dùng trong phép dịch)

rt = 16 (toán hạng nguồn là \$16 ~ \$s0)

shamt = 4 (số bit dịch là 4)



# Truy xuất bộ nhớ

- Bộ nhớ là mảng 1 chiều  
các ô nhớ có địa chỉ

6

5

5

5

5

6

Dữ liệu

1

101

10

100

. . .

Địa chỉ

0

1

2

3

. . .

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- Trong cấu trúc R-format hỗ trợ các lệnh số học và luận lý (đã tìm hiểu), các toán hạng `rs`, `rt`, `rd` giữ địa chỉ các thanh ghi
- Làm sao để truy xuất dữ liệu trong bộ nhớ?
  - Cần toán hạng giữ địa chỉ ô nhớ
- Có 2 hướng giải quyết
  - Cho phép `rt`, `rd` lưu địa chỉ bộ nhớ. Có khả thi ?
  - Tạo ra cấu trúc lệnh khác để thao tác với bộ nhớ
- Hỏi thêm: có cần phải lưu dữ liệu trong bộ nhớ rồi mới nạp vào thanh ghi không? Tại sao không nạp dữ liệu của chương trình trực tiếp vào các thanh ghi để xử lý ?



# Cấu trúc lệnh truy xuất bộ nhớ

- Lệnh thao tác với bộ nhớ cần ít nhất
  - 1 toán hạng nguồn và 1 toán hạng đích
- Cấu trúc R-Format

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

- Tạo cấu trúc lệnh mới thế nào để giảm thiểu thay đổi so với cấu trúc R-Format à Cấu trúc I-Format

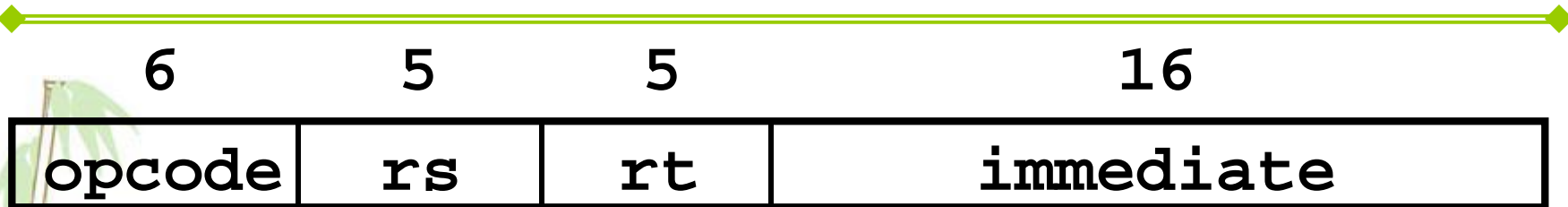
opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

opcode	rs	rt	immediate
--------	----	----	-----------

- Để xác định 1 vùng nhớ trong lệnh, cần 2 yếu tố:
  - Một thanh ghi chứa địa chỉ 1 vùng nhớ (xem như con trỏ tới vùng nhớ)
  - Một số nguyên (xem như độ dài (tính theo byte) từ địa chỉ trong thanh ghi trên). *Tại sao lại cần giá trị này?*
- Địa chỉ vùng nhớ sẽ được xác định bằng tổng 2 giá trị này.
- Ví dụ:  $8(\$t0)$ 
  - Xác định một vùng nhớ có địa chỉ bằng  $(\$t0 + 8)$  (byte)



# Cấu trúc I-Format



– opcode: mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-Format, chỉ khác là 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 một trường 12-bit: để nhất quán với các cấu trúc lệnh khác trong khi kích thước mỗi trường vẫn hợp lý.

– rs: chứa địa chỉ thanh ghi nguồn thứ 1

– rt (register target): chứa địa chỉ thanh ghi đích.

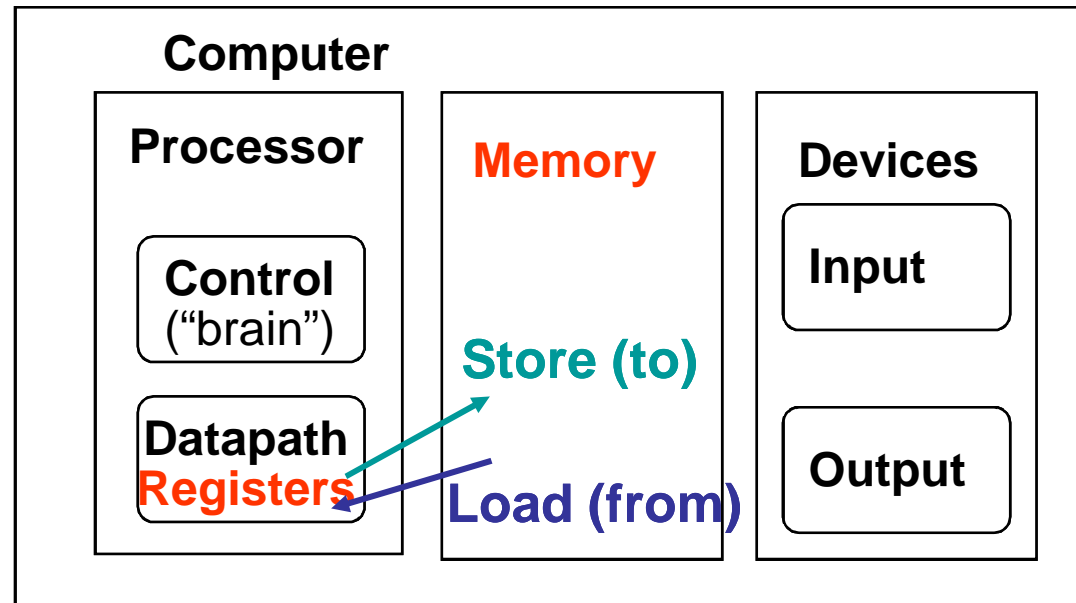
– immediate: 16 bit, có thể biểu diễn số nguyên từ  $-2^{15}$  tới  $(2^{15}-1)$

• Đủ lớn để chứa giá trị độ dời (offset) từ địa chỉ trong thanh ghi cơ sở rs nhằm phục vụ việc truy xuất bộ nhớ trong lệnh lw và sw.



# Lệnh thao tác với bộ nhớ

- MIPS hỗ trợ các lệnh di chuyển dữ liệu (Data transfer instructions) để chuyển dữ liệu giữa thanh ghi và vùng nhớ:
  - Vùng nhớ vào thanh ghi (nạp - load)
  - Thanh ghi vào vùng nhớ (lưu - store)



- Như vậy, bộ xử lý nạp các dữ liệu (và lệnh) vào các thanh ghi để xử lý rồi lưu kết quả ngược trở lại bộ nhớ





# Lệnh di chuyển dữ liệu (1/2)

- Cú pháp:

opt opr, opr1(opr2)

– trong đó:

opt - Tên thao tác

opr - Thanh ghi lưu trữ nhớ

opr1 - Hằng số nguyên

opr2 - Thanh ghi chứa địa chỉ vùng nhớ

## Lệnh di chuyển dữ liệu (2/2)

- Nạp 1 từ dữ liệu bộ nhớ (Load Word –  $lw$ ) vào thanh ghi



$lw \ \$t0, 12(\$s0)$

Lệnh này nạp từ nhớ có địa chỉ ( $\$s0 + 12$ ) vào thanh ghi  $\$t0$

- Lưu 1 từ dữ liệu thanh ghi (Store Word –  $sw$ ) vào bộ nhớ



$sw \ \$t0, 12(\$s0)$

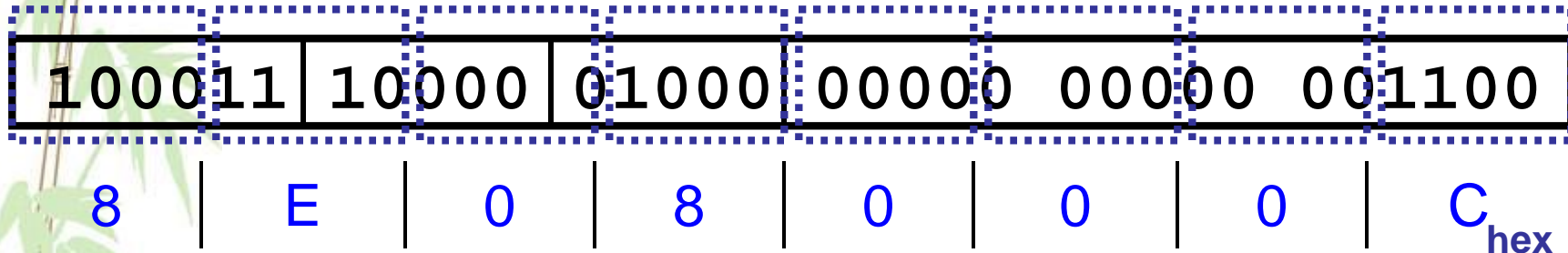
Lệnh này lưu giá trị trong thanh ghi  $\$t0$  vào vùng nhớ có địa chỉ ( $\$s0 + 12$ )



## Ví dụ mã máy của lệnh lw

lw \$t0, 12(\$s0)

Mã máy (dưới dạng nhị phân)



Giá trị thập phân tương ứng của từng trường

35	16	8	12
----	----	---	----

opcode = 35 (Xác định thao tác nạp từ nhớ)  
rs = 16 (toán hạng đích là \$16 ~ \$s0)  
rt = 18 (toán hạng nguồn là \$8 ~ \$t0)  
immediate = 12 (hằng số là 12)



# Ý nghĩa của toán hạng truy xuất bộ nhớ

- Chú ý:

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



# Ví dụ truy xuất mảng

- Giả sử

- A là mảng các từ nhớ
- địa chỉ bắt đầu của A:  $\$s3$
- g:  $\$s1$
- h:  $\$s2$

- Câu lệnh C :

$g = h + A[5];$

được biên dịch thành lệnh MIPS như sau:

```
lw      $t0, 20($s3)      # $t0 = A[5]
add     $s1, $s2, $t0      # $s1 = h + A[5]
```

- Chú ý:

- $A[5]$  là phần tử thứ 5 của mảng A, mỗi phần tử là một từ nhớ (word). Do đó, sẽ tương đương với từ nhớ bắt đầu tại địa chỉ  $\$s3 + 20$



# Nguyên tắc lưu trữ và truy xuất dữ liệu trong bộ nhớ (1/2)

- Nguyên tắc Alignment Restriction: các đối tượng lưu trong bộ nhớ phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng
- MIPS lưu và cho phép truy xuất dữ liệu trong bộ nhớ theo nguyên tắc Alignment Restriction, nghĩa là từ nhớ phải bắt đầu tại địa chỉ là bội số của 4
- Lệnh `lw $t0, 18($s3)` có hợp lệ không ?

**Aligned**

**Not  
Aligned**

0	1	2	3
Aligned	Aligned	Aligned	Aligned
Not Aligned	Aligned	Aligned	Aligned
Not Aligned	Not Aligned	Aligned	Aligned
Not Aligned	Aligned	Not Aligned	Aligned
Not Aligned	Not Aligned	Not Aligned	Aligned
Not Aligned	Aligned	Aligned	Not Aligned

Ký số hex cuối  
trong địa chỉ:

**0, 4, 8, or  $C_{hex}$**

**1, 5, 9, or  $D_{hex}$**

**2, 6, A, or  $E_{hex}$**

**3, 7, B, or  $F_{hex}$**



## Nguyên tắc lưu trữ và truy xuất dữ liệu trong bộ nhớ (2/2)

- MIPS lưu trữ dữ liệu trong bộ nhớ theo nguyên tắc Big Endian, nghĩa là đối với giá trị có kích thước lớn hơn 1 byte thì byte cao sẽ lưu tại địa chỉ thấp, (vs. Little Endian trong kiến trúc x86)
- Ví dụ: lưu trữ giá trị 4 byte 12345678h trong bộ nhớ

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



## Con trỏ vs. giá trị

- Lưu ý phân biệt 2 trường hợp sau (giả sử x: \$t0, y: \$t1, z: \$t2)
  - Nếu ghi thì tương đương `add $t2, $t1, $t0`  
\$t0 và \$t1 lưu giá trị  
 $z = x + y$  (trong C)
  - Nếu ghi thì tương đương `lw $t2, 0($t0)`  
\$t0 chứa một địa chỉ (vai trò như một con trỏ)  
 $z = *x$  (trong C)





## Lệnh nạp, lưu 1 byte nhớ

- Ngoài các lệnh nạp, lưu từ nhớ (`lw`, `sw`), MIPS còn cho phép nạp, lưu từng byte nhớ nhằm hỗ trợ các thao tác với ký tự 1 byte (ASCII).
  - load byte: `lb`
  - store byte: `sb`
- Cú pháp 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`.
  - 24 bit còn lại sẽ có giá trị theo bit dấu của giá trị 1 byte (sign-extended)
  - Nếu không muốn các bit còn lại có giá trị theo bit dấu, sử dụng lệnh `lbu` (load byte unsigned)



## Lệnh nạp, lưu $\frac{1}{2}$ từ nhớ (2 byte)

- MIPS còn hỗ trợ các lệnh nạp, lưu  $\frac{1}{2}$  từ nhớ (2 byte) nhớ nhằm hỗ trợ các thao tác với ký tự 2 byte (Unicode).
  - load half: `lh` (lưu  $\frac{1}{2}$  từ nhớ (2 byte) vào 2 byte thấp của thanh ghi)
  - store half: `sh`
- Cú pháp tương tự `lw`, `sw`
- *Tại sao lại hỗ trợ loại lệnh này trong khi vẫn có thể sử dụng các lệnh nạp byte nhớ để thực hiện thay*



## Toán hạng thanh ghi và vùng nhớ

- Trong MIPS, chỉ có các lệnh nạp, lưu mới sử dụng toán hạng vùng nhớ
  - *Tại sao không sử dụng toán hạng vùng nhớ trong các lệnh khác như số học, luận lý,...?*
- Một nhiệm vụ của trình biên dịch là ánh xạ các biến được sử dụng trong chương trình thành các thanh ghi
  - Điều gì xảy ra nếu biến sử dụng trong các chương trình nhiều hơn số lượng thanh ghi?
  - Nhiệm vụ của trình biên dịch: spilling

# Thao tác với hằng số

- Các hằng số xuất hiện trong các lệnh dịch và lệnh di chuyển được gọi là các toán hạng hằng số
- Các thao tác với hằng số xuất hiện rất thường xuyên, do đó, MIPS hỗ trợ một lớp các lệnh thao tác với hằng số (tên lệnh kết thúc bằng ký tự i - immediate): `addi`, `andi`, `ori`, ...
- Các lệnh thao tác với hằng số có cấu trúc I-Format

<b>opcode</b>	<b>rs</b>	<b>rt</b>	<b>immediate</b>
---------------	-----------	-----------	------------------

- *Tại sao lại cần các lệnh thao tác với hằng số trong khi các lệnh này đều có thể được thực hiện bằng cách kết hợp các lệnh nạp, lưu với các thao tác trên thanh ghi ?*



# Ví dụ lệnh thao tác hằng số

- Lệnh cộng với hằng số (tương tự như lệnh `add`, chỉ khác ở toán hạng cuối cùng là một hằng số thay vì là thanh ghi):

`addi $s0,$s1,10` (cộng hằng số có dấu)

`addiu $s0,$s1,10` (cộng hằng số không dấu)

Biểu diễn lệnh dưới dạng nhị phân

001000	10001	10000	000000000000001010
--------	-------	-------	--------------------

Giá trị thập phân tương ứng của từng trường

8	17	16	10
---	----	----	----

`opcode = 8`: xác định thao tác cộng hằng số có dấu

`rs = 17` (toán hạng nguồn thứ 1 là `$17 ~ $s1`)

`rt = 16` (toán hạng đích là `$16 ~ $s0`)

`immediate = 10` (hằng số là 10)

- Muốn thực hiện phép trừ một hằng số thì sao?

`addi $s0,$s1,-10`

- Tại sao không có lệnh trừ hằng số, chẳng hạn `subi`?



# Vấn đề của I-Format (1/3)

- Vấn đề:

- Các lệnh thao tác với hằng số (`addi`, `lw`, `sw`, ...) có cấu trúc I-Format, nghĩa là trường hằng số (`immediate`) chỉ có 16 bit.

<code>opcode</code>	<code>rs</code>	<code>rt</code>	<code>immediate</code>
---------------------	-----------------	-----------------	------------------------

- Nếu muốn thao tác với các hằng số 32 bit thì sao ?
- Tăng kích thước `immediate` thành 32 bit?  
à tăng kích thước các lệnh thao tác với hằng số có cấu trúc I-Format



## Vấn đề của I-Format (2/3)

- Giải pháp:

- Hỗ trợ thêm lệnh mới nhưng không phá vỡ các cấu trúc lệnh đã có

- Lệnh mới:

lui register, immediate

- Load Upper Immediate
- Đưa hằng số 16 bit vào 2 byte cao của một thanh ghi
- Giá trị các bit 2 byte thấp được gán 0
- Lệnh này có cấu trúc I-Format





## Vấn đề của I-Format (3/3)

- Giải pháp (tt):

- Lệnh `lui` giải quyết vấn đề như thế nào?
- Ví dụ: muốn cộng giá trị 32 bit `0xABABCD CD` vào thanh ghi `$t0`

không thể thực hiện:

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

mà thực hiện như sau:

```
lui     $at, 0xABAB  
ori     $at, $at, 0xCD CD  
add     $t0, $t0, $at
```





# Tràn số trong phép tính số học

- Nhắc lại: tràn số xảy ra khi kết quả phép tính vượt quá độ chính xác giới hạn cho phép (của máy tính).
- MIPS cung cấp 2 loại lệnh số học:
  - Cộng (add), cộng hằng số (addi) và trừ (sub) phát hiện tràn số
  - Cộng không dấu (addu), cộng hằng số không dấu (addiu) và trừ không dấu (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



# Trắc nghiệm

- A. Kiểu cần được xác định khi khai báo biến trong C và khi sử dụng lệnh trong MIPS.
- B. Do chỉ có 8 thanh ghi lưu trữ (\$s) và 8 thanh ghi tạm (\$t), nên không thể chuyển từ chương trình C có nhiều hơn 16 biến thành chương trình MIPS.
- C. Nếu p (lưu trong \$s0) là một con trỏ trỏ tới mảng ints, thì `p++;` sẽ tương ứng với `addi $s0 $s0 1`

	ABC
1 :	FFF
2 :	FFT
3 :	FTF
4 :	FTT
5 :	TFF
6 :	TFT
7 :	TF
8 :	TTT



# Trắc nghiệm

Hãy chuyển lệnh  $*x = *y$  (trong C) thành lệnh tương ứng trong MIPS

(các con trỏ  $x, y$  được lưu trong  $\$s0$   $\$s1$ )

A: add  $\$s0, \$s1, \text{zero}$   
B: add  $\$s1, \$s0, \text{zero}$   
C: lw  $\$s0, 0(\$s1)$   
D: lw  $\$s1, 0(\$s0)$   
E: lw  $\$t0, 0(\$s1)$   
F: sw  $\$t0, 0(\$s0)$   
G: lw  $\$s0, 0(\$t0)$   
H: sw  $\$s1, 0(\$t0)$

0:	A
1:	B
2:	C
3:	D
4:	E→F
5:	E→G
6:	F→E
7:	F→H
8:	H→G
9:	G→H

# Trắc nghiệm

- Lệnh nào sau đây có biểu diễn tương ứng với  $35_{10}$ ?

1. `add $0, $0, $0`
2. `subu $s0, $s0, $s0`
3. `lw $0, 0($0)`
4. `addi $0, $0, 35`
5. `subu $0, $0, $0`
6. Lệnh không phải là dãy bit

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Số hiệu và tên của các thanh ghi:

0: \$0, .. 8: \$t0, 9: \$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Mã thao tác và mã chức năng (nếu có)

`add:`      pcode = 0,      funct = 32  
`subu:`     opcode = 0,      funct = 35  
`addi:`     opcode = 8  
`lw:`       opcode = 35



# Đáp án

- Lệnh nào sau đây có biểu diễn tương ứng với  $35_{10}$ ?

1. add \$0, \$0, \$0

0	0	0	0	0	32
---	---	---	---	---	----

2. subu \$s0, \$s0, \$s0

0	16	16	16	0	35
---	----	----	----	---	----

3. lw \$0, 0(\$0)

35	0	0			0
----	---	---	--	--	---

4. addi \$0, \$0, 35

8	0	0			35
---	---	---	--	--	----

5. subu \$0, \$0, \$0

0	0	0	0	0	35
---	---	---	---	---	----

6. Lệnh không phải là dãy bit

Số hiệu và tên của các thanh ghi:

0: \$0, .. 8: \$t0, 9: \$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Mã thao tác và mã chức năng (nếu có)

add: pcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

lw: opcode = 35

- MIPS đã hỗ trợ các nhóm lệnh xử lý dữ liệu:
  - Lệnh số học
  - Lệnh luận lý
  - Lệnh nạp lưu dữ liệu
- Ngoài các lệnh xử lý dữ liệu, máy tính (**computer**) còn phải hỗ trợ các lệnh điều khiển quá trình thực thi các lệnh.
- Trong NNLT C, bạn đã bao giờ sử dụng lệnh `goto` để nhảy tới một nhãn (**labels**) chưa ?



# Lệnh `if` trong C

- 2 loại lệnh `if` trong C

`if (condition) clause`

`if (condition) clause1 else clause2`

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

`if (condition) goto L1;`

`clause2;`

`goto L2;`

`L1: clause1;`

`L2:`





## Cấu trúc lệnh rẽ nhánh có điều kiện

- Lệnh rẽ nhánh có điều kiện cần
  - 2 toán hạng nguồn để so sánh và
  - 1 toán hạng cho biết địa chỉ cần nhảy tới
- Không cần tạo cấu trúc lệnh mới à Sử dụng cấu trúc I-Format

opcode	rs	rt	immediate
--------	----	----	-----------

- opcode mã thao tác, cho biết lệnh làm gì
- rs và rt chứa các giá trị cần so sánh
- *immediate chứa địa chỉ (nhãn) cần nhảy tới ?*
- *immediate chỉ có 16 bit, nghĩa là chỉ có thể nhảy tới địa chỉ từ  $0 - 2^{16}$  (65,535) ?*



# Lệnh rẽ nhánh có điều kiện

- Cú pháp

`beq register1, register2, L1`

`beq` nghĩa là “Branch if (registers are) equal”

tương ứng với lệnh `if` trong C như sau:

`if (register1 == register2) goto L1`

`bne register1, register2, L1`

`bne` nghĩa là “Branch if (registers are) not equal”

tương ứng với lệnh `if` trong C như sau:

`if (register1 != register2) goto L1`

- Ví dụ:

`if (b == 0)  
    a = a + 1`

`else  
    a = a + 2`

`beq $t1,$0,hit  
addi $t0,$t0,1`

`hit:  
addi $t0,$t0,1`



## Lệnh rẽ nhánh có điều kiện: Định vị theo thanh ghi PC (1/3)

- `immediate` chứa khoảng cách so với địa chỉ nằm trong thanh ghi PC (Program Counter), thanh ghi chứa địa chỉ lệnh đang được thực hiện
- Cách xác định địa chỉ này gọi là: **PC-Relative Addressing** (định vị theo thanh ghi PC)
- Lúc này trường `immediate` được xem như 1 số có dấu cộng với địa chỉ trong thanh ghi PC tạo thành địa chỉ cần nhảy tới.
- Như vậy, có thể nhảy tới, lui 1 khoảng  $2^{15}$  (byte ?) từ lệnh sẽ được thực hiện, đủ đáp ứng hầu hết các yêu cầu nhảy lặp của chương trình (thường tối đa 50 lệnh).



## Lệnh rẽ nhánh có điều kiện: Định vị theo thanh ghi PC (2/3)

- Chú ý: mỗi lệnh có kích thước 1 từ nhớ (32 bit) và MIPS truy xuất bộ nhớ theo nguyên tắc nguyên tắc Alignment Restriction, do đó đơn vị của `immediate`, khoảng cách so với PC, là từ nhớ
- Như vậy, các lệnh rẽ nhánh có thể nhảy tới các địa chỉ có khoảng cách  $\pm 2^{15}$  từ nhớ từ PC ( $\pm 2^{17}$  bytes).



## Lệnh rẽ nhánh có điều kiện: Định vị theo thanh ghi PC (3/3)

- Cách tính địa chỉ rẽ nhánh:

- Nếu không thực hiện rẽ nhánh:

$$PC = PC + 4$$

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

- Nếu thực hiện rẽ nhánh:

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

- Tại sao cộng `immediate` với (PC+4), thay vì với PC ?

- Nhận xét: trường `immediate` cho biết số lệnh cần nhảy qua để tới được nhãn.



## Ví dụ cấu trúc I-Format của lệnh rẽ nhánh có điều kiện

beq \$t1,\$0,hit

addi \$t0,\$t0,1

hit:

addi \$t0,\$t0,1

Biểu diễn lệnh dưới dạng nhị phân

000100	01001	00000	000000000000000001
--------	-------	-------	--------------------

Giá trị thập phân tương ứng của từng trường

4	9	0	1
---	---	---	---

opcode = 4 (mã thao tác của lệnh beq)

rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)

rt = 0 (toán hạng nguồn thứ 2 là \$0)

immediate = 1



## Một số vấn đề của định vị theo thanh ghi PC

- Giá trị các trường của lệnh rẽ nhánh có thay đổi không nếu di chuyển mã nguồn ?
- Nếu phải nhảy ra ngoài khoảng  $2^{15}$  lệnh từ lệnh rẽ nhánh thì sao ?
- Tăng kích thước trường `immediate` à tăng kích thước lệnh rẽ nhánh ?





# Cấu trúc lệnh rẽ nhánh không điều kiện J-Format

- Lệnh rẽ nhánh không điều kiện cần
  - 1 toán hạng cho biết địa chỉ cần nhảy tới
- Cấu trúc lệnh R-Format và I-Format

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

opcode	rs	rt	immediate
--------	----	----	-----------

- Tạo cấu trúc lệnh mới thế nào để giảm thiểu thay đổi so với cấu trúc R-Format và I-Format → Cấu trúc J-Format

opcode	target address
--------	----------------

- opcode mã thao tác, cho biết lệnh làm gì
- target address chứa địa chỉ (từ nhớ) cần nhảy tới



## Vấn đề của Cấu trúc J-Format

- Như vậy, với cấu trúc J-Format, có thể nhảy trong khoảng  $2^{26}$  từ nhớ ( $\sim 2^{28}$  byte)
- Có nghĩa là không thể nhảy tới các từ nhớ có địa chỉ từ  $2^{27}$  tới  $2^{32}$  ?
  - Tuy nhiên, nhu cầu này là không cần thiết vì chương trình thường không quá lớn như vậy (thường trong giới hạn 256 MB)
  - Nếu cần nhảy tới các địa chỉ này, MIPS hỗ trợ lệnh `jr` (sẽ được học sau).

# Lệnh rẽ nhánh không điều kiện

- Cú pháp

`j label`

nghĩa là “jump to label”

tương ứng với lệnh trong C sau: `goto label`

Có thể viết dưới dạng lệnh rẽ nhánh có điều kiện như sau:

```
beq    $0,$0,label
```

- Ví dụ

```
if (b == 0)
    a = 0
else
    a = 1
```

```
beq    $t1,$0,hit
addi   $t0,$0,1
j      end

hit:
add    $t0,$0,$0

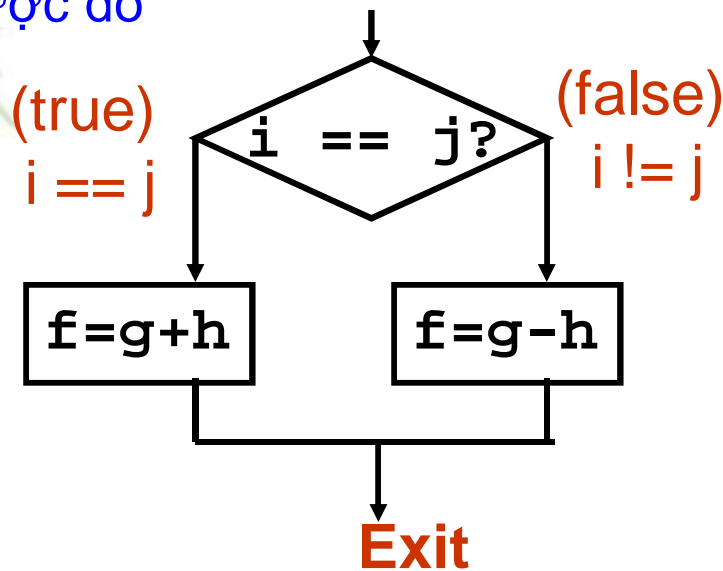
end:
```



# Biên dịch lệnh `if` thành lệnh máy MIPS

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

- Vẽ lược đồ



- Ánh xạ biến vào thanh ghi:

f: \$s0

g: \$s1

h: \$s2

i: \$s3

j: \$s4

- Chuyển thành lệnh máy MIPS:

```
beq $s3,$s4,True    # branch i==j
sub $s0,$s1,$s2      # f=g-h(false)
j    Exit            # goto Exit
True: add $s0,$s1,$s2 # f=g+h(true)
Exit:
```



# Lặp trong MIPS (1/2)

- Lặp trong C; A[ ] là một mảng các số nguyên int

```
do {  
    g = g + A[i];  
    i = i + j;  
} while (i != h);
```

- Có thể viết lại như sau:

```
Loop:  g = g + A[i];  
       i = i + j;  
       if (i != h) goto Loop;
```

- Ánh xạ biến vào thanh ghi như sau:

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

- Chuyển thành lệnh MIPS như sau:

```
Loop:  sll $t1,$s3,2      # $t1= 4*i  
       add $t1,$t1,$s5    # $t1=addr A  
       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  # goto Loop if i!=h
```



## Lặp trong MIPS (2/2)

- 3 kiểu lặp trong C:
  - while
  - do... while
  - for
- Viết lại dưới dạng goto, chuyển thành các lệnh MIPS sử dụng các lệnh rẽ nhánh có điều kiện



# So sánh không bằng trong MIPS (1/3)

- beq và bne được sử dụng trong trường hợp so sánh bằng (== và != trong C). Còn những trường hợp so sánh không bằng < và > thì sao?

- Hướng tiếp cận

- Thêm tất cả các lệnh so sánh không bằng: bgt, blt, ble, bge ?
- Chỉ cần thêm 1 lệnh mà có thể thực hiện các phép so sánh không bằng

- MIPS hỗ trợ lệnh:

- “Set on Less Than”
- Cú pháp: `slt reg1, reg2, reg3` (Cấu trúc R-Format)
- Ý nghĩa

```
if (reg2 < reg3)
    reg1 = 1;
else reg1 = 0;
```

“set” nghĩa là “set to 1”

“reset” nghĩa là “set to 0”





# So sánh không bằng trong MIPS (2/3)

- Câu lệnh sau:

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

- Được chuyển thành lệnh MIPS như sau...

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h  
bne $t0,$0,Less # goto Less  
# if $t0!=0(if (g<h))  
Less: # Less:
```

- Thanh ghi \$0 luôn chứa giá trị 0, nên lệnh bne và beq thường được dùng để so sánh sau lệnh slt.
- Cặp slt à bne tương đương if(... < ...)goto...
- Các phép so sánh còn lại >, ≤ and ≥ thì sao?
  - Có thể thực hiện cách kết hợp lệnh slt và các lệnh rẽ nhánh có điều kiện beq và bne ?



# So sánh không bằng trong MIPS (3/3)

- ```
# a:$s0, b:$s1
slt $t0,$s0,$s1
beq $t0,$0,skip
<stuff>
skip:
```
- ```
# a:$s0, b:$s1
slt $t0,$s0,$s1
bne $t0,$0,skip
<stuff>
skip:
```
- ```
# a:$s0, b:$s1
slt $t0,$s1,$s0
beq $t0,$0,skip
<stuff>
skip:
```
- ```
# a:$s0, b:$s1
slt $t0,$s1,$s0
bne $t0,$0,skip
<stuff>
skip:
```

```
# $t0 = 1 if a<b
# skip if a >= b
# do if a<b
```

```
# $t0 = 1 if a<b
# skip if a<b
# do if a>=b
```

```
# $t0 = 1 if a>b
# skip if a<=b
# do if a>b
```

```
# $t0 = 1 if a>b
# skip if a>b
# do if a<=b
```



## Hằng số trong so sánh không bằng

- MIPS hỗ trợ lệnh `slti` để thực hiện so sánh không bằng với hằng số (cấu trúc I-Format).
  - Hữu ích đối với vòng lặp `for`

**C** `if (g >= 1) goto Loop`

`Loop: . . .`

**MIPS**

```
    slti $t0,$s0,1          # $t0 = 1 if
                           # $s0<1 (g<1)
    beq  $t0,$0,Loop        # goto Loop
                           # if $t0==0
                           # (if (g>=1))
```

- Cặp `slt` và `beq` tương ứng với `if(... ≥ ...)goto...`
- Có thể sử dụng cặp lệnh `add/or` và `slt` thay cho `slti`. Tại sao phải tạo ra 1 lệnh mới ?
- Ngoài ra, còn có các lệnh: `sltu`, `sltiu`
- Giá trị của `$t0`, `$t1` với (`$s0 = FFFF FFFAhex`, `$s1 = 0000 FFFAhex`) ?

```
    slt $t0, $s0, $s1
```

```
    sltu $t1, $s0, $s1
```




## Ví dụ: lệnh switch trong C (1/2)

- `switch (k) {  
    case 0: f=i+j; break;     /* k=0 */  
    case 1: f=g+h; break;     /* k=1 */  
    case 2: f=g-h; break;     /* k=2 */  
    case 3: f=i-j; break;     /* k=3 */  
}`
- Viết lại dưới dạng các lệnh if như sau:  
`if (k==0) f=i+j;  
else if (k==1) f=g+h;  
    else if (k==2) f=g-h;  
        else if (k==3) f=i-j;`
- Ánh xạ biến vào thanh ghi:  
`f: $s0, g: $s1, h: $s2,  
i: $s3, j: $s4, k: $s5`

## Ví dụ: lệnh switch trong C (1/2)

- Chuyển thành lệnh MIPS như sau:



```
bne $s5,$0,L1      # branch k!=0
add  $s0,$s3,$s4   # k==0 so f=i+j
j    Exit          # end of case so Exit
L1: addi $t0,$s5,-1  # $t0=k-1
bne  $t0,$0,L2     # branch k!=1
add  $s0,$s1,$s2   # k==1 so f=g+h
j    Exit          # end of case so Exit
L2: addi $t0,$s5,-2  # $t0=k-2
bne  $t0,$0,L3     # branch k!=2
sub  $s0,$s1,$s2   # k==2 so f=g-h
j    Exit          # end of case so Exit
L3: addi $t0,$s5,-3  # $t0=k-3
bne  $t0,$0,Exit   # branch k!=3
sub  $s0,$s3,$s4   # k==3 so f=i-j
Exit:
```

# Trắc nghiệm

```

Loop: addi $s0, $s0, -1    # i = i - 1
      slti $t0, $s1, 2     # $t0 = (j < 2)
      beq  $t0, $0, Loop   # goto Loop if $t0 == 0
      slt  $t0, $s1, $s0   # $t0 = (j < i)
      bne  $t0, $0, Loop   # goto Loop if $t0 != 0

      ( $s0=i, $s1=j )
  
```

Biểu thức điều kiện (C) nào trong câu lệnh while (bên dưới) tương ứng với đoạn lệnh MIPS ở trên?

do {i--;} while(\_\_\_\_);

0:	j	<	2	&&	j	<	i
1:	j	>=	2	&&	j	<=	i
2:	j	>	2	&&	j	>=	i
3:	j	>=	2	&&	j	>	i
4:	j	<	2	&&	j	<	i
5:	j	<	2	&&	j	<	i
6:	j	>=	2	&&	j	<=	i
7:	j	<	2	&&	j	>=	i
8:	j	>=	2	&&	j	>	i
9:	j	<	2	&&	j	<	i



## Tóm tắt một số lệnh MIPS đã tìm hiểu

Name	Example	Comments
32 registers	<code>\$s0, \$s1, ..., \$s7</code> <code>\$t0, \$t1, ..., \$t7,</code> <code>\$zero</code>	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers <code>\$s0–\$s7</code> map to 16–23 and <code>\$t0–\$t7</code> map to 8–15. MIPS register <code>\$zero</code> always equals 0.
$2^{30}$ memory words	<code>Memory[0],</code> <code>Memory[4], ...,</code> <code>Memory[4294967292]</code>	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add <code>\$s1,\$s2,\$s3</code>
sub	R	0	18	19	17	0	34	sub <code>\$s1,\$s2,\$s3</code>
lw	I	35	18	17	100			lw <code>\$s1,100(\$s2)</code>
sw	I	43	18	17	100			sw <code>\$s1,100(\$s2)</code>
and	R	0	18	19	17	0	36	and <code>\$s1,\$s2,\$s3</code>
or	R	0	18	19	17	0	37	or <code>\$s1,\$s2,\$s3</code>
nor	R	0	18	19	17	0	39	nor <code>\$s1,\$s2,\$s3</code>
andi	I	12	18	17	100			andi <code>\$s1,\$s2,100</code>
ori	I	13	18	17	100			ori <code>\$s1,\$s2,100</code>
sll	R	0	0	18	17	10	0	sll <code>\$s1,\$s2,10</code>
srl	R	0	0	18	17	10	2	srl <code>\$s1,\$s2,10</code>
beq	I	4	17	18	25			beq <code>\$s1,\$s2,100</code>
bne	I	5	17	18	25			bne <code>\$s1,\$s2,100</code>
slt	R	0	18	19	17	0	42	slt <code>\$s1,\$s2,\$s3</code>
j	J	2	2500					j 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Hình  
2.13  
trang  
78,  
P&H



# Thủ tục trong C

```
main() {  
    int a,b,c;  
    ...  
    c = sum(a,b);  
    ...  
}  
/* khai báo hàm sum */  
int sum (int x, int y) {  
    return x+y;  
}
```

- Lời gọi thủ tục và khai báo thủ tục được chuyển thành lệnh máy như thế nào ?
- Đối số được truyền vào thủ tục như thế nào ?
- Kết quả trả về của thủ tục được truyền ra ngoài như thế nào ?

# Nhận xét

- Khi gọi thủ tục thì lệnh tiếp theo được thực hiện là lệnh đầu tiên của thủ tục
- à Có thể xem tên thủ tục là một nhãn và lời gọi thủ tục là một lệnh nhảy tới nhãn này

## C

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

```
...
```

```
int sum (...)
```

## MIPS

```
j sum # nhảy tới  
# nhãn sum
```

```
...
```

```
sum:
```

- Sau khi thực hiện xong thủ tục phải quay về thực hiện tiếp lệnh ngay sau lời gọi thủ tục

```
return ...
```

```
j ?
```



# Ví dụ

**C**

```
int main() {  
    ...  
    c=sum(a,b); /* a,b:$s0,$s1 */  
    ...  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

- Hỏi: Tại sao lại dùng jr ? Mà không đơn giản dùng j?
- Trả lời: thủ tục sum có thể được gọi ở nhiều chỗ khác nhau, do đó vị trí quay về mỗi lần gọi khác nhau sẽ khác nhau.

địa chỉ

**MIPS**

1000	add	\$a0,\$s0,\$zero	# x = a
1004	add	\$a1,\$s1,\$zero	# y = b
1008	addi	\$ra,\$zero,1016	# lưu địa chỉ quay về # vào \$ra=1016
1012	j	sum	# nhảy tới nhãn sum
1016	...		
2000	sum:	add \$v0,\$a0,\$a1	# khai báo thủ tục sum
2004	jr	\$ra	# nhảy tới địa chỉ # trong \$ra



# Nhận xét

- Thay vì phải dùng 2 lệnh để lưu địa chỉ quay về vào `$ra` và nhảy tới thủ tục:

```
1008 addi $ra,$zero,1016 # $ra=1016
1012 j sum                # goto sum
```

- MIPS còn hỗ trợ 1 lệnh `j al` (jump and link) để thực hiện 2 công việc trên:

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

- *Tại sao lại thêm lệnh `jal`?*
  - không cần phải xác định tường minh địa chỉ quay về trong `$ra`
- *Lý do nào khác ?*



# Ví dụ

```
int main() {  
    ...  
    c=sum(a,b); /* a,b:$s0,$s1 */  
    ...  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

địa chỉ

1000	add	\$a0,\$s0,\$zero	# x = a
1004	add	\$a1,\$s1,\$zero	# y = b
1008	jal	sum	# lưu địa chỉ quay về # vào \$ra=1012 và nhảy # tới sum
1012	...		
2000	sum:	add \$v0,\$a0,\$a1	# khai báo thủ tục sum
2004	jr	\$ra	# nhảy tới địa chỉ # trong \$ra

# Các lệnh mới

- **jal (jump and link): J-Format**
  - Cú pháp: `jal label`
  - 1 (link): Lưu địa chỉ của lệnh kế tiếp vào thanh ghi `$ra`
  - 2 (jump): nhảy tới nhãn `label`
- **Lệnh jr (jump register): R-Format**
  - Cú pháp: `jr register`
  - Nhảy tới địa chỉ nằm trong thanh ghi `register`
- **2 lệnh này được sử dụng hiệu quả trong thủ tục:**
  - `jal` lưu địa chỉ quay về vào thanh ghi `$ra` và nhảy tới thủ tục
  - `jr $ra` Nhảy tới địa chỉ quay về đã được lưu trong `$ra`



# Các thanh ghi mới

- MIPS hỗ trợ thêm một số thanh ghi để lưu trữ các dữ liệu phục vụ cho thủ tục:
  - Đối số  $\$a0, \$a1, \$a2, \$a3$
  - Kết quả trả về  $\$v0, \$v1$
  - Địa chỉ quay về  $\$ra$
- Nếu thủ tục sử dụng nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên ? Sử dụng thêm nhiều thanh ghi hơn...  
Bao nhiêu thanh ghi cho đủ ?  
à Sử dụng ngăn xếp (stack)





# Bài tập

```
main() {  
    int i,j,k,m;  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}  
/* khai báo hàm mult */  
int mult (int mcand, int mlier){  
    int product;  
    product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier -1; }  
    return product;  
}
```



# Thủ tục lồng nhau

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

- Thủ tục `sumSquare` gọi thủ tục `mult`.
  - Vấn đề
    - Địa chỉ quay về của thủ tục `sumSquare` trong thanh ghi `$ra` sẽ bị ghi đè bởi địa chỉ trả về của thủ tục `mult` khi thủ tục này được gọi
    - Như vậy cần phải lưu lại địa chỉ quay về của thủ tục `sumSquare` (trong thanh ghi `$ra`) trước khi gọi thủ tục `mult`.
- à Sử dụng thanh ghi... Bao nhiêu cho đủ?
- à Sử dụng ngăn xếp (stack).

# Mô hình cấp phát bộ nhớ của C

- Một chương trình C thực thi sẽ được cấp phát các vùng nhớ sau:

Địa chỉ ¥

\$sp  
Con trỏ  
ngăn xếp

Stack

Vùng nhớ được sử dụng trong quá trình thực thi thủ tục như lưu các biến cục bộ, lưu địa chỉ trả về,...

Heap

Vùng nhớ chứa các biến cấp phát động. Ví dụ: con trỏ C được cấp phát động bởi hàm malloc()

Static

Vùng nhớ chứa các biến cấp phát tĩnh của mỗi chương trình. Ví dụ: biến toàn cục của C

0  
Code

Mã nguồn chương trình



# Sử dụng ngăn xếp (1/2)

- Con trỏ ngăn xếp, thanh ghi \$sp, được sử dụng để định vị vùng ngăn xếp.
- Để sử dụng ngăn xếp, cần khai báo kích thước vùng ngăn xếp bằng cách tăng giá trị con trỏ ngăn xếp.
- Lệnh MIPS tương ứng với

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

## Sử dụng ngăn xếp (2/2)



```
sumSquare:      # x,y : $a0,$a1
    addi $sp,$sp,-8      # khai báo kích thước
                        # ngăn xếp cần dùng
    sw $ra, 4($sp)       # cất địa chỉ quay về
                        # của thủ tục sumSquare
                        # vào ngăn xếp
    "push" sw $a1, 0($sp) # cất y vào ngăn xếp
    add $a1,$a0,$zero    # gán x vào $a1
    jal mult            # gọi thủ tục mult
    lw $a1, 0($sp)       # sau khi thực thi xong
                        # thủ tục mult, khôi
                        # phục y từ ngăn xếp
    "pop" add $v0,$v0,$a1 # mult()+y
    lw $ra, 4($sp)       # lấy lại địa chỉ quay về
                        # của thủ tục sumSquare
                        # đã lưu vào ngăn xếp,
                        # đưa vào thanh ghi $ra
    addi $sp,$sp,8       # kết thúc dùng ngăn xếp
    jr      $ra

mult: ...
```



# Cấu trúc cơ bản của thủ tục

## Đầu thủ tục

```
entry_label:  
addi $sp,$sp, -framesize  
sw $ra, framesize-4($sp)      # cất địa chỉ trả  
                              # về của thủ tục  
                              # trong $ra vào  
                              # ngăn xếp
```

Lưu tạm các thanh ghi khác nếu cần

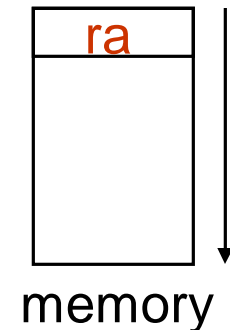
## Thân thủ tục ...

(có thể gọi các thủ tục khác...)

## Cuối thủ tục

Phục hồi các thanh ghi khác nếu cần

```
lw $ra, framesize-4($sp)      # khôi phục $ra  
addi $sp,$sp, framesize  
jr $ra
```





## Một số nguyên tắc sử dụng thủ tục

- Thủ tục R (caller) gọi thủ tục E (callee)

### Trong thủ tục R

1. Lưu địa chỉ trả về (trong  $\$ra$ ) của R vào ngăn xếp
2. Gán các đối số (nếu có) R truyền vào E
3. Gọi lệnh `jal`

### Trong thủ tục E

3. Khởi tạo ngăn xếp
4. Lưu vào ngăn xếp các thanh ghi trong R có thể bị thay đổi trong E.
5. ...
6. Khôi phục các dữ liệu đã lưu tạm trong ngăn xếp
7. Phục hồi ngăn xếp
8. Gọi lệnh `jr $ra` để trở lại thủ tục R





# Trắc nghiệm

```
int fact(int n){  
    if(n == 0) return 1; else return(n*fact(n-1));}
```

Khi chuyển sang MIPS...

- A. CÓ THỂ sao lưu \$a0 vào \$a1 (và sau đó không lưu lại \$a0 hay \$a1 vào ngăn xếp) để lưu lại n qua những lời gọi đệ qui.
- B. PHẢI lưu \$a0 vào ngăn xếp vì nó sẽ thay đổi.
- C. PHẢI lưu \$ra vào ngăn xếp do cần để biết địa chỉ quay về...

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT



# Trắc nghiệm

```
r: ...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
...      ### cất các thanh ghi vào ngăn xếp?
jal e     # gọi thủ tục e
...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra    # quay về thủ tục gọi r

e: ...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra     # quay về thủ tục r
```

Thủ tục r cần cất các thanh ghi nào vào ngăn xếp trước khi gọi “jal e”?

- 0: 0 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 1: 1 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 2: 2 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 3: 3 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 4: 4 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 5: 5 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)
- 6: 6 of (\$s0,\$sp,\$v0,\$t0,\$a0,\$ra)

# Đáp án

```

r: ...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
...      ### cất các thanh ghi vào ngăn xếp?
jal e      # gọi thủ tục e
...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra     # quay về thủ tục gọi r

e: ...      # đọc ghi $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra     # quay về thủ tục r

```

Thủ tục `r` cần cất các thanh ghi nào vào ngăn xếp trước khi gọi “jal e”?

0:	0	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
1:	1	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
2:	2	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
3:	3	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
4:	4	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
5:	5	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )
6:	6	of	( \$s0, \$sp, \$v0, \$t0, \$a0, \$ra )




Không cần cất  
vào ngăn xếp

Cần cất vào ngăn xếp



# Vai trò 32 thanh ghi của MIPS



The constant 0  
Reserved for Assembler  
Return Values  
Arguments  
Temporary  
Saved  
More Temporary  
Used by Kernel  
Global Pointer  
Stack Pointer  
Frame Pointer  
Return Address

\$0	\$zero
\$1	\$at
\$2-\$3	\$v0-\$v1
\$4-\$7	\$a0-\$a3
\$8-\$15	\$t0-\$t7
\$16-\$23	\$s0-\$s7
\$24-\$25	\$t8-\$t9
\$26-27	\$k0-\$k1
\$28	\$gp
\$29	\$sp
\$30	\$fp
\$31	\$ra



# Nguyên tắc sử dụng thanh ghi (1/2)

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



# Nguyên tắc sử dụng thanh ghi (2/2)

- $\$ra$ : **Có thể thay đổi**. Lời gọi lệnh jal sẽ làm thay đổi giá trị thanh ghi này. Thủ tục gọi lưu lại thanh ghi này vào ngăn xếp nếu cần.
- $\$v0 - \$v1$ : **Có thể thay đổi**. Các thanh ghi này chứa các kết quả trả về.
- $\$a0 - \$a3$ : **Có thể thay đổi**. Đây là các thanh ghi chứa đối số. Thủ tục gọi cần lưu lại giá trị nếu nó cần sau khi gọi thủ tục.
- $\$t0 - \$t9$ : **Có thể thay đổi**. Đây là các thanh ghi tạm nên có thể bị thay đổi bất kỳ lúc nào. Thủ tục gọi cần lưu lại giá trị nếu nó cần sau các lời gọi thủ tục.



# Tóm tắt các cấu trúc lệnh MIPS

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>i mm.</i> format
J-format	op	target address					Jump instruction format

Hình 2.26 trang 104, P&H





## Tóm tắt một số lệnh MIPS đã tìm hiểu

MIPS instructions	Name	Format
add	add	R
subtract	sub	R
add immediate	addi	I
load word	lw	I
store word	sw	I
load half	lh	I
store half	sh	I
load byte	lb	I
store byte	sb	I
load upper immediate	lui	I
and	and	R
or	or	R
nor	nor	R
and immediate	andi	I
or immediate	ori	I
shift left logical	sll	R
shift right logical	srl	R
branch on equal	beq	I
branch on not equal	bne	I
set less than	slt	R
set less than immediate	slti	I
jump	j	J
jump register	jr	R
jump and link	jal	J

Hình 2.47 trang 146, P&H


# Lệnh giả

- Lệnh giả (Pseudo Instruction) là các lệnh hợp ngữ không có cài đặt lệnh máy tương ứng, nhằm mục đích giúp cho việc lập trình hợp ngữ dễ dàng hơn

Pseudo MIPS	Name	Format
move	move	R
multiply	mult	R
multiply immediate	mult i	I
load immediate	l i	I
branch less than	blt	I
branch less than or equal	ble	I
branch greater than	bgt	I
branch greater than or equal	bge	I

Hình 2.47 trang 146, P&H

## Một số Syscall thực hiện nhập xuất



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)



# Tìm hiểu thêm...

- Quá trình biên dịch và thực thi chương trình (phần 2.10, trang 106-115, P&H) + các khái niệm
  - Symbol table
  - Compiler, Linker, Loader
  - Dynamically Linked Library (DLL)
  - Java bytecode, Java Virtual Machine (JVM), Just In Time Compiler (JIT)
- Bộ lệnh Intel IA-32 (phần 2.16, trang 134-143, P&H) + các khái niệm
  - General Purpose Register (GPR)
  - Addressing Modes



# Tham khảo

- Chương 2, trang 28, P&H





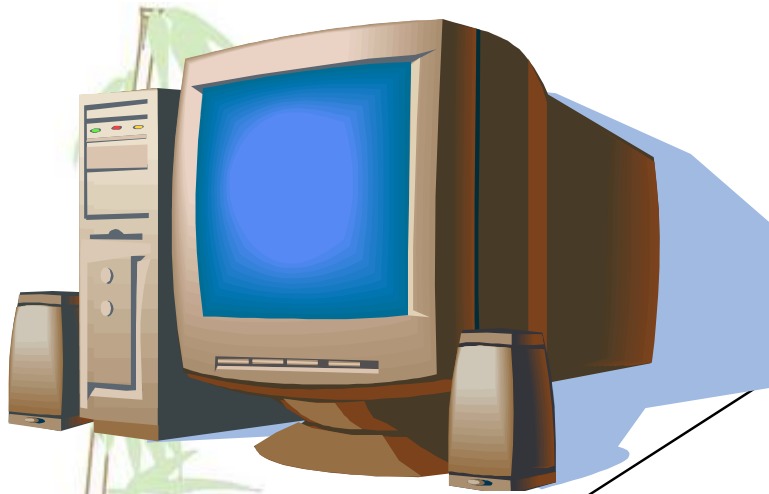
Khoa  
**CÔNG NGHỆ THÔNG TIN**  
ĐH Khoa học Tự nhiên TP HCM

# **Bài 04: Cấu trúc và hoạt động của Bộ xử lý**

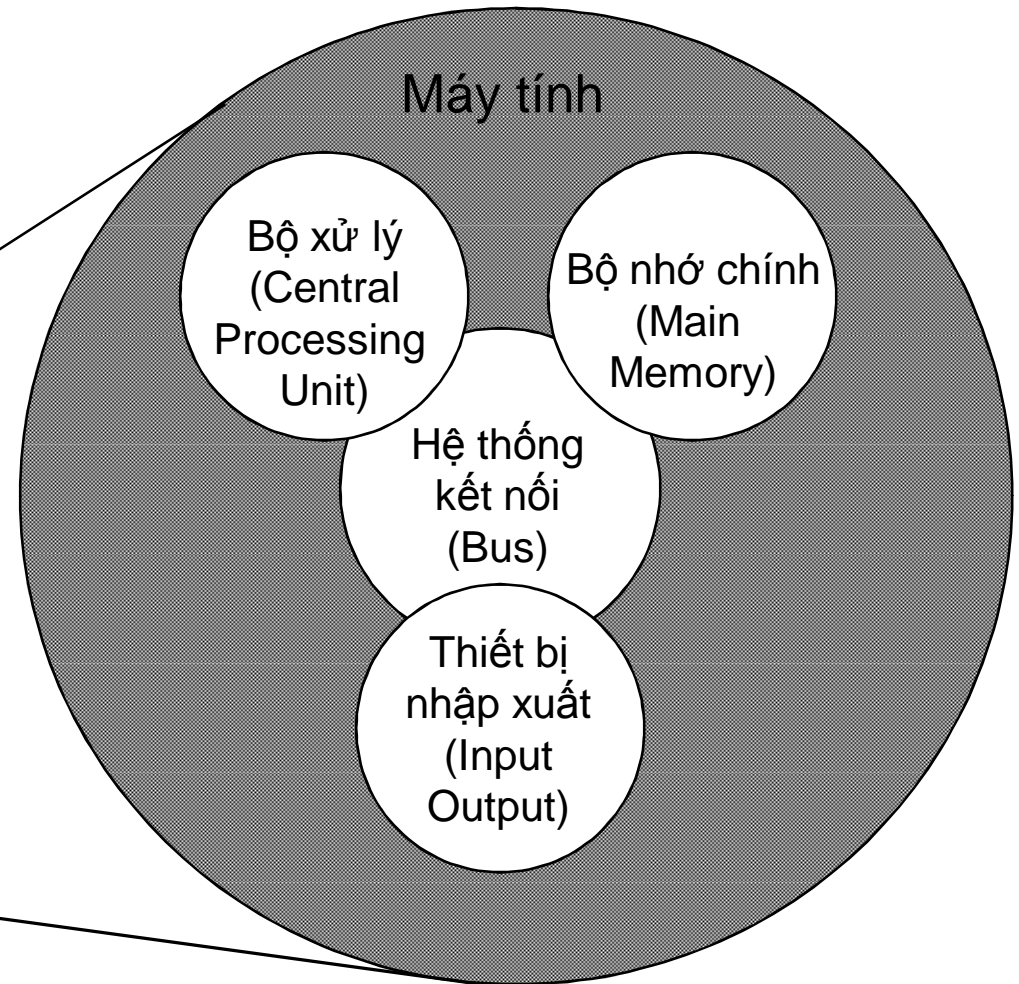
**Phạm Tuấn Sơn**

**[ptson@fit.hcmus.edu.vn](mailto:ptson@fit.hcmus.edu.vn)**

# Cấu trúc máy tính

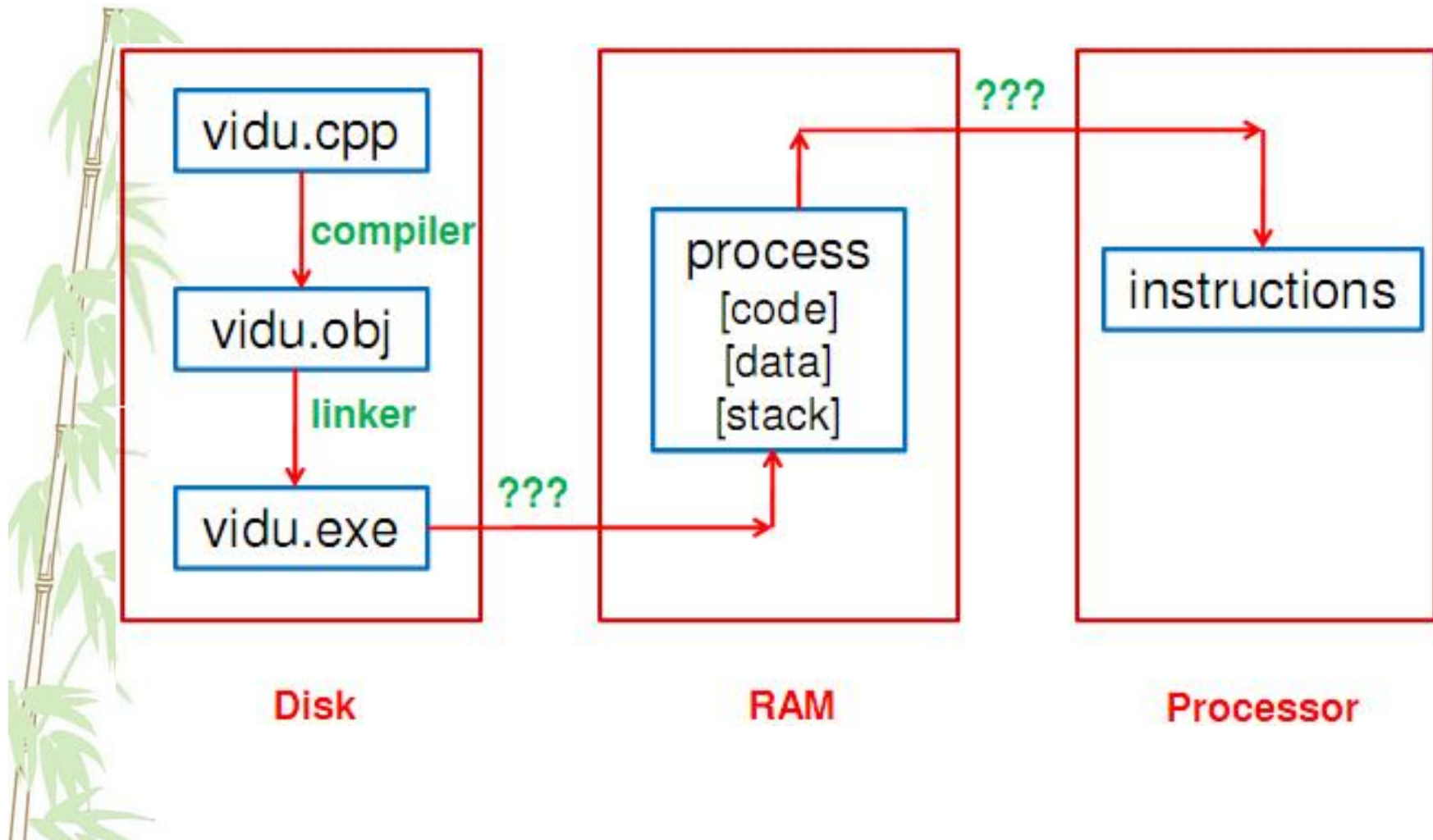


Máy tính  
(Computer)

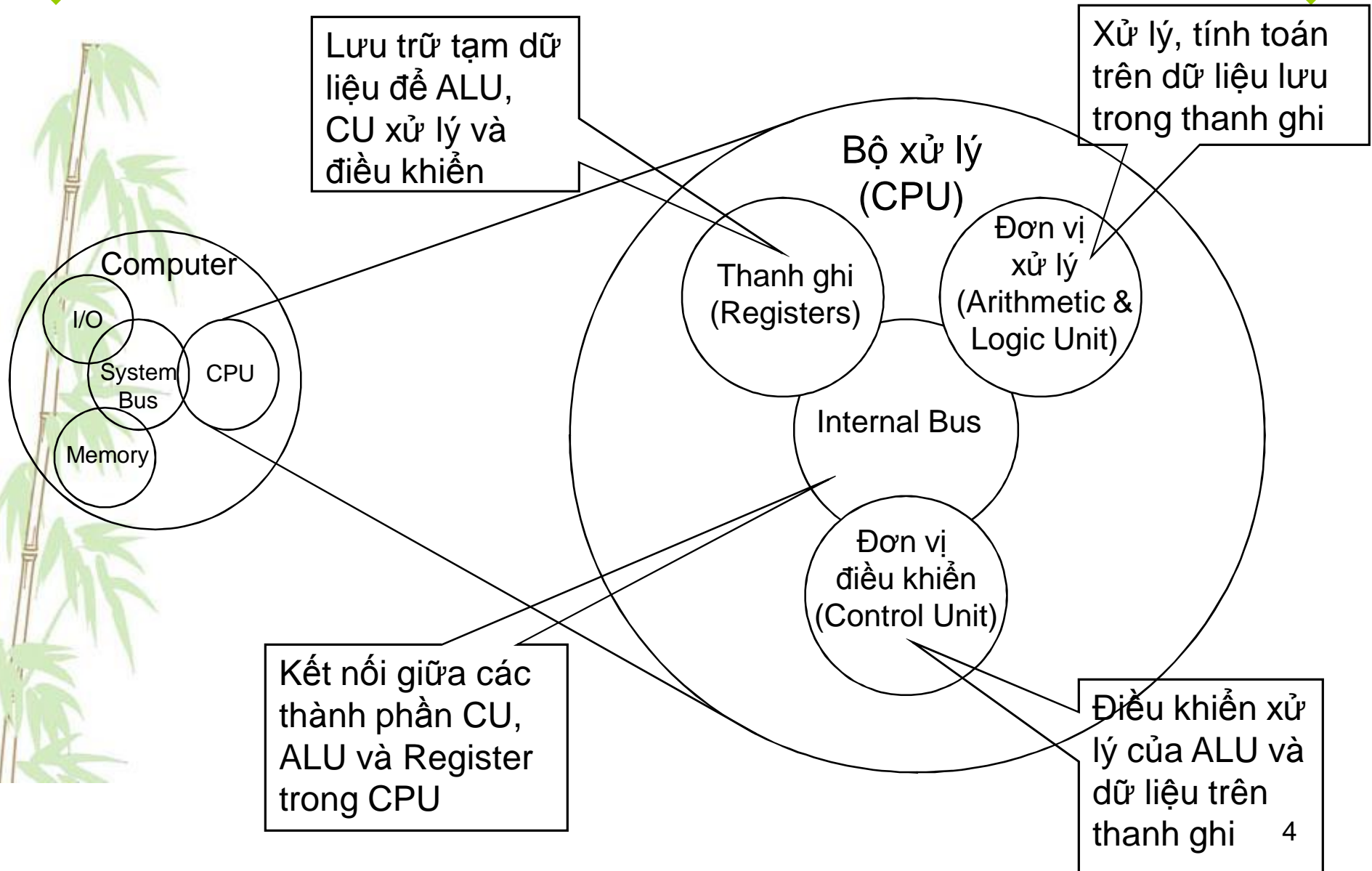




# Thực thi chương trình



# Cấu trúc bộ xử lý



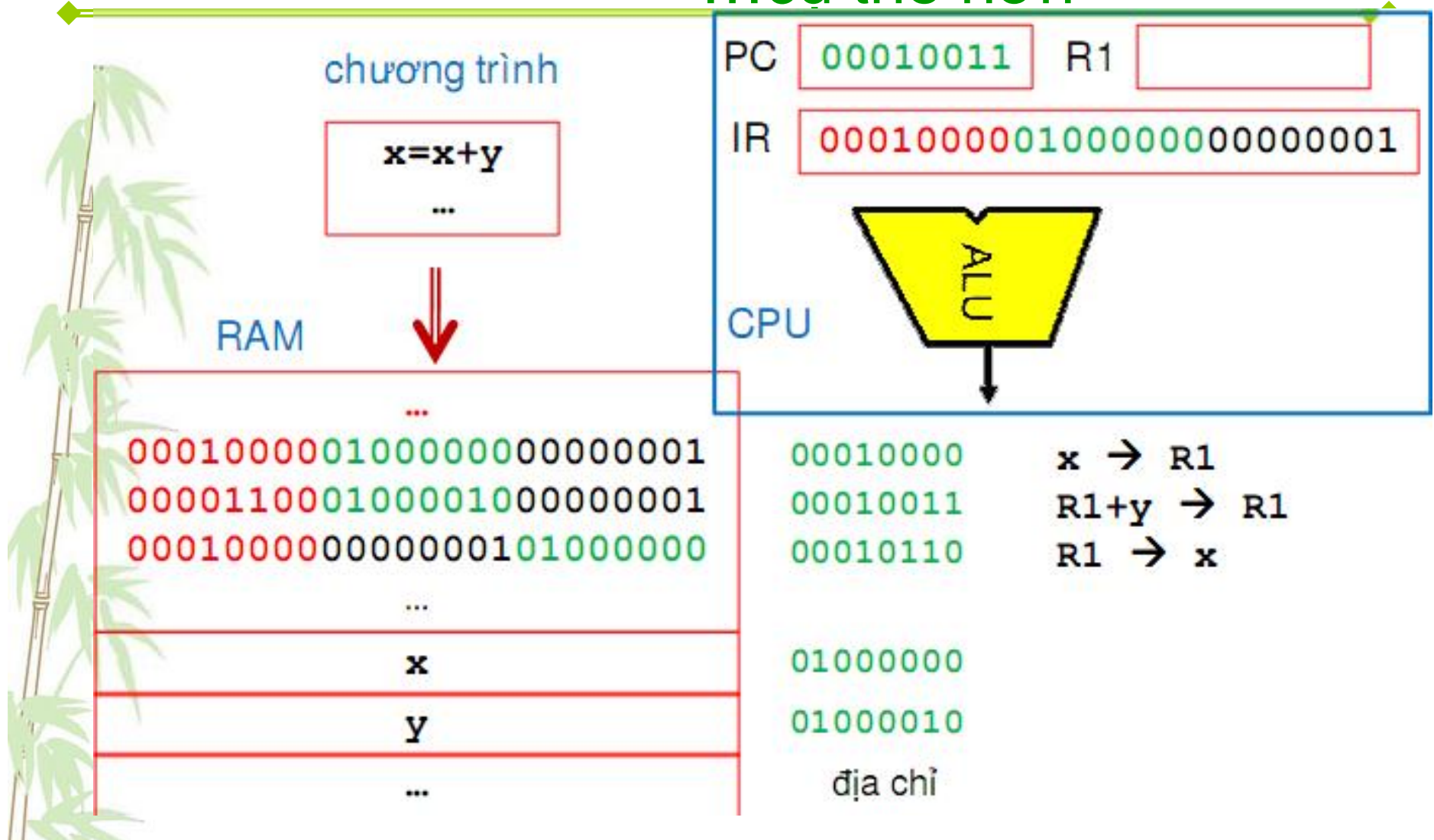


# Lệnh máy

- Lệnh máy (machine instruction/ instruction/ machine code) là dãy bit chứa yêu cầu mà bộ xử lý phải thực hiện
- Cấu trúc của một lệnh máy thường gồm:
  - Mã thao tác (opcode): cho biết lệnh thực hiện thao tác gì (+, -, and, or, ...)
  - Các toán hạng (operand): cho biết các đối tượng bị tác động bởi thao tác trong mã thao tác (thanh ghi, vùng nhớ, hằng số, ...)
- Mỗi bộ xử lý chỉ hiểu được một số lệnh với một vài cấu trúc nhất định

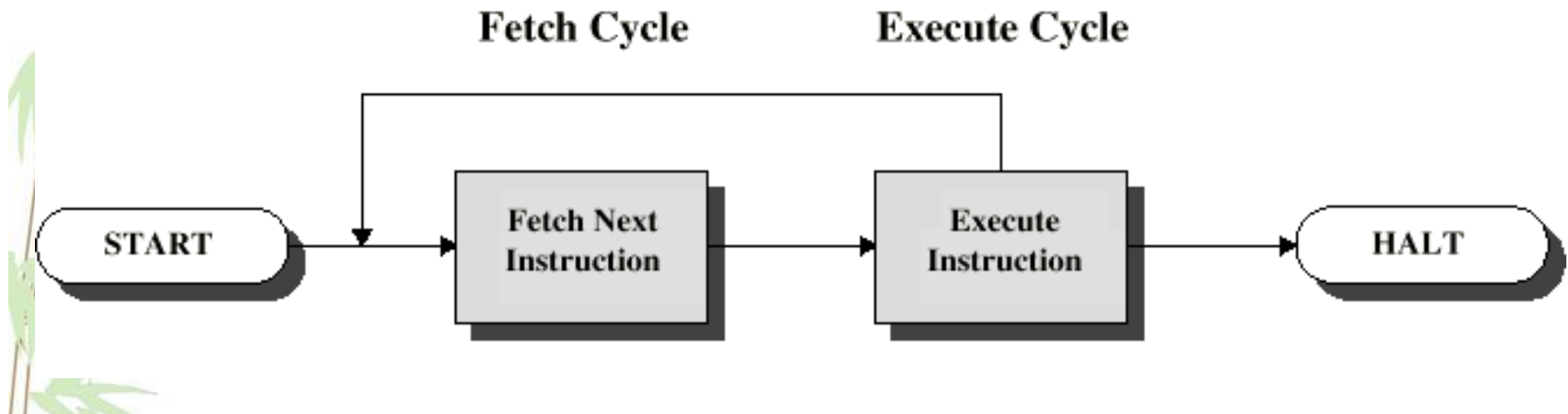


# Thực thi chương trình... ...cụ thể hơn



# Hoạt động của CPU

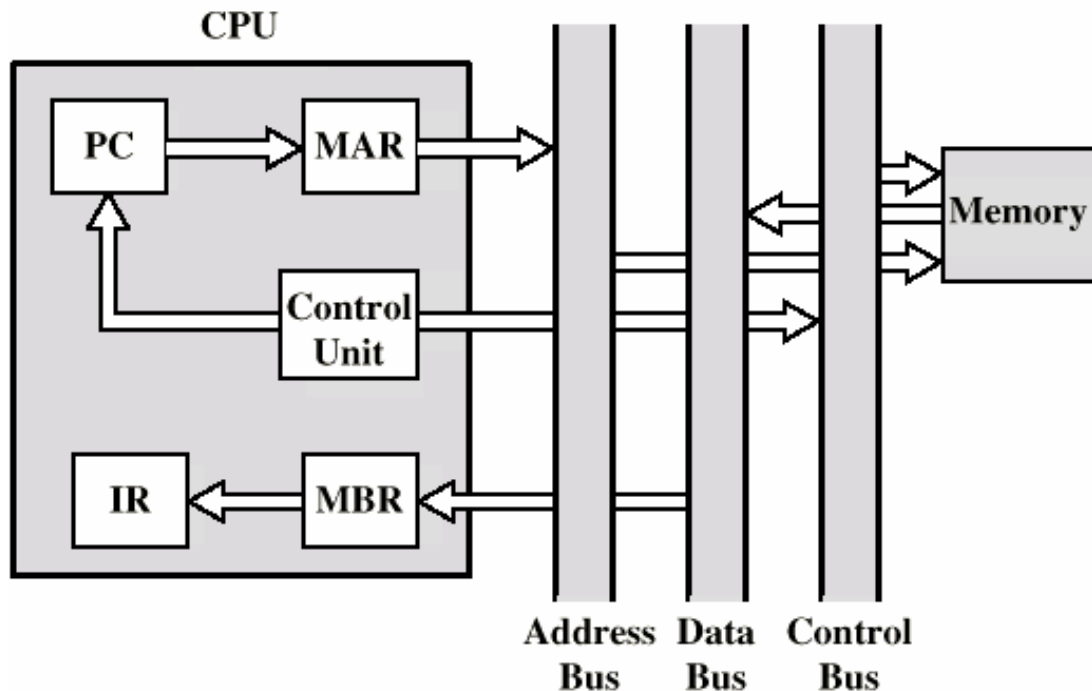
- Xử lý lệnh máy qua 2 bước, gọi là chu kỳ lệnh (instruction cycle)
  - Nạp lệnh (Fetch)
    - Di chuyển lệnh từ bộ nhớ vào thanh ghi
  - Thực thi lệnh
    - Giải mã lệnh và thực hiện thao tác yêu cầu



# Quá trình nạp lệnh



- MAR  $\rightarrow$  (PC)
- MBR  $\rightarrow$  Memory
- IR  $\rightarrow$  (MBR)
- PC  $\rightarrow$  (PC) + 1



- Thanh ghi MAR (Memory Address Register)
  - Lưu địa chỉ được gửi ra/ nhận vào từ bus địa chỉ.
- Thanh ghi MBR (Memory Buffer Register)
  - Lưu giá trị được gửi ra/ nhận vào từ bus dữ liệu.
- Thanh ghi PC (Program Counter)
  - Lưu địa chỉ của lệnh sẽ được nạp.
- Thanh ghi IR (Instruction Register)
  - Lưu lệnh sẽ được xử lý.
- Bộ xử lý di chuyển lệnh từ vùng nhớ có địa chỉ trong thanh ghi PC vào thanh ghi IR.
- Mặc định, giá trị thanh ghi PC được tăng 1 lượng bằng chiều dài của lệnh được nạp.





# Quá trình thực thi lệnh

- Bộ xử lý giải mã lệnh trong thanh ghi IR và thực hiện thao tác yêu cầu như:
  - Thực hiện các phép tính số học và luận lý
  - Thực hiện di chuyển dữ liệu giữa thanh ghi và bộ nhớ
  - Thực hiện di chuyển dữ liệu giữa thanh ghi và thiết bị nhập xuất
  - Thực hiện các thao tác điều khiển như rẽ nhánh





# Ví dụ quá trình xử lý lệnh của CPU

## Cấu trúc lệnh

4 bit	12 bit
Mã thao tác	Địa chỉ

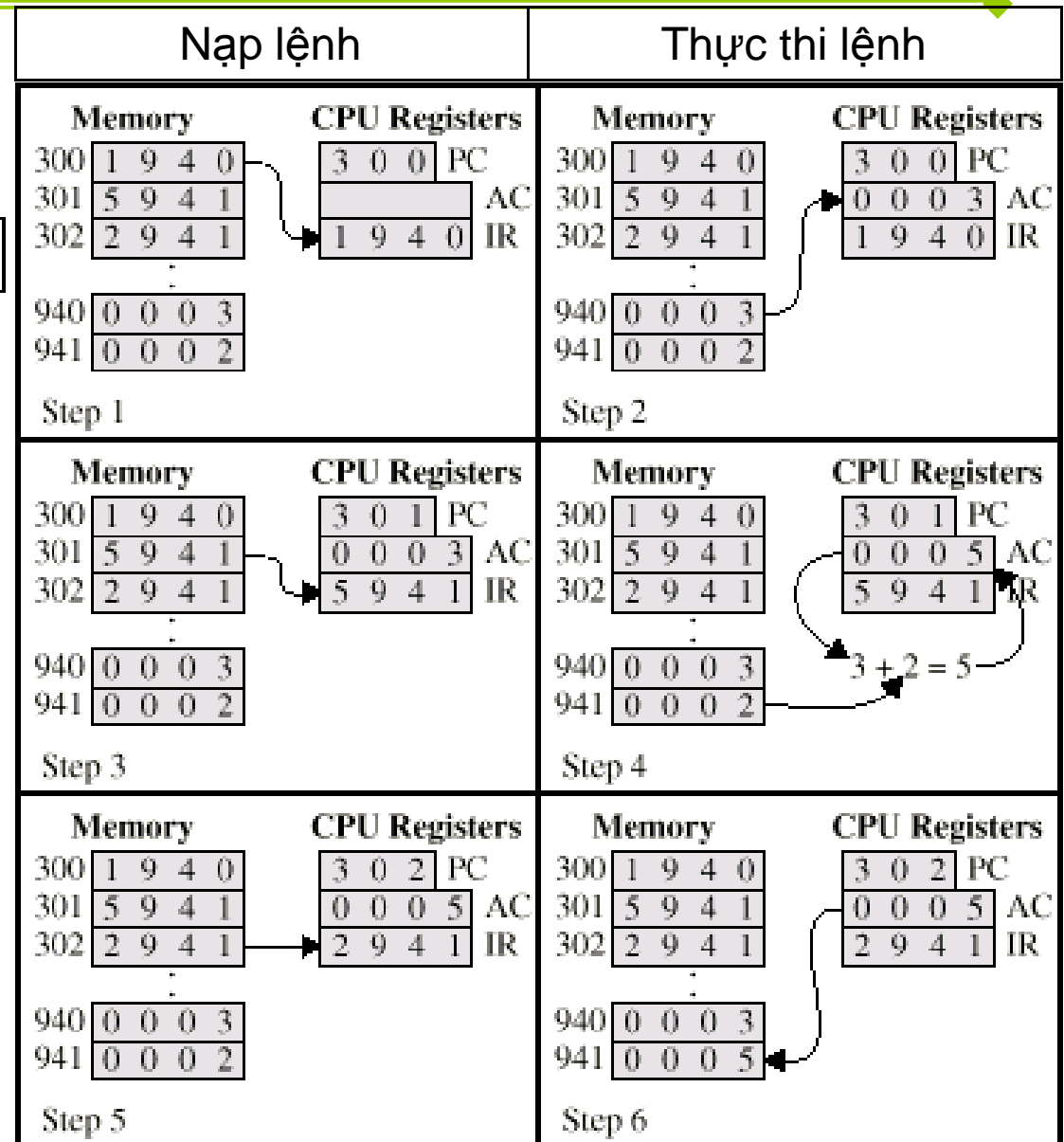
Các thanh ghi: PC, IR, AC

## Mã thao tác

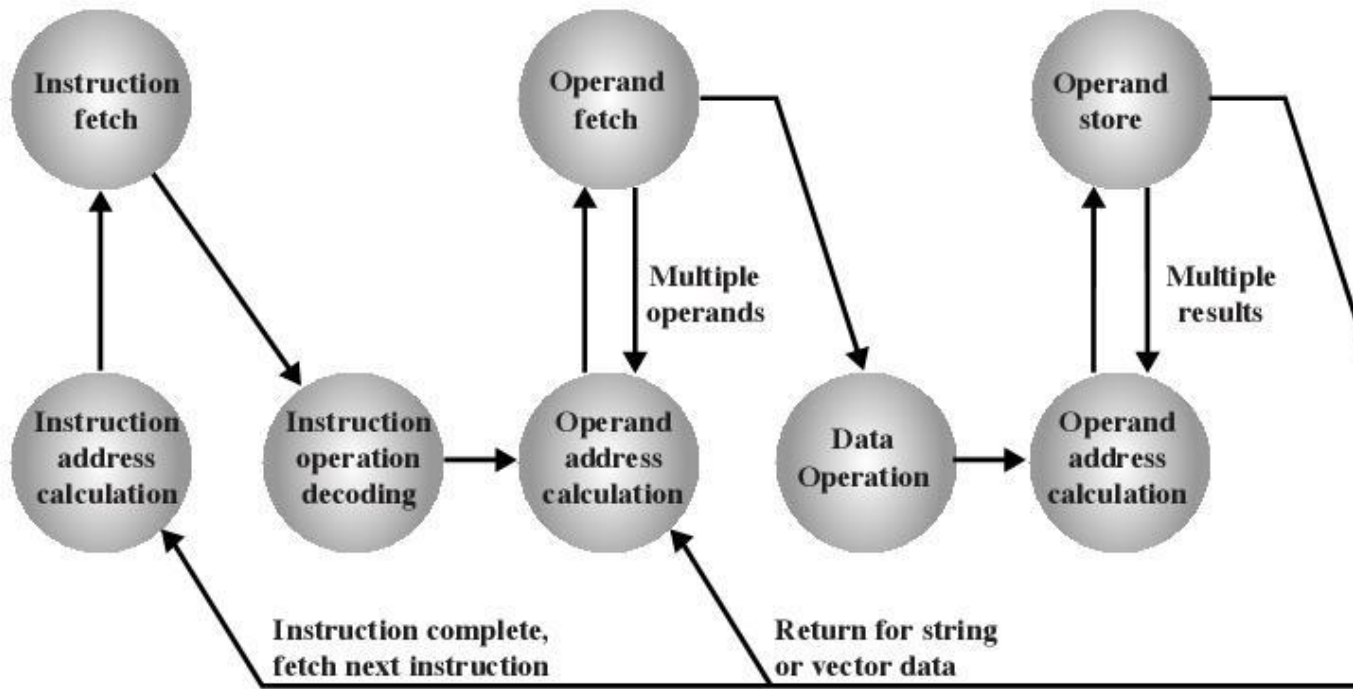
0001 = Nạp dữ liệu từ “địa chỉ”  
vào thanh ghi AC

0010 = Lưu dữ liệu từ thanh ghi AC  
vào bộ nhớ tại “địa chỉ”

0101 = Cộng dồn giá trị tại “địa chỉ”  
vào thanh ghi AC



# Chu kỳ lệnh tổng quát



1. Tính địa chỉ của lệnh
2. Nạp lệnh
3. Giải mã lệnh
4. Tính địa chỉ của toán hạng
5. Nạp toán hạng
6. Thực thi lệnh
7. Tính địa chỉ của toán hạng chứa kết quả
8. Ghi kết quả

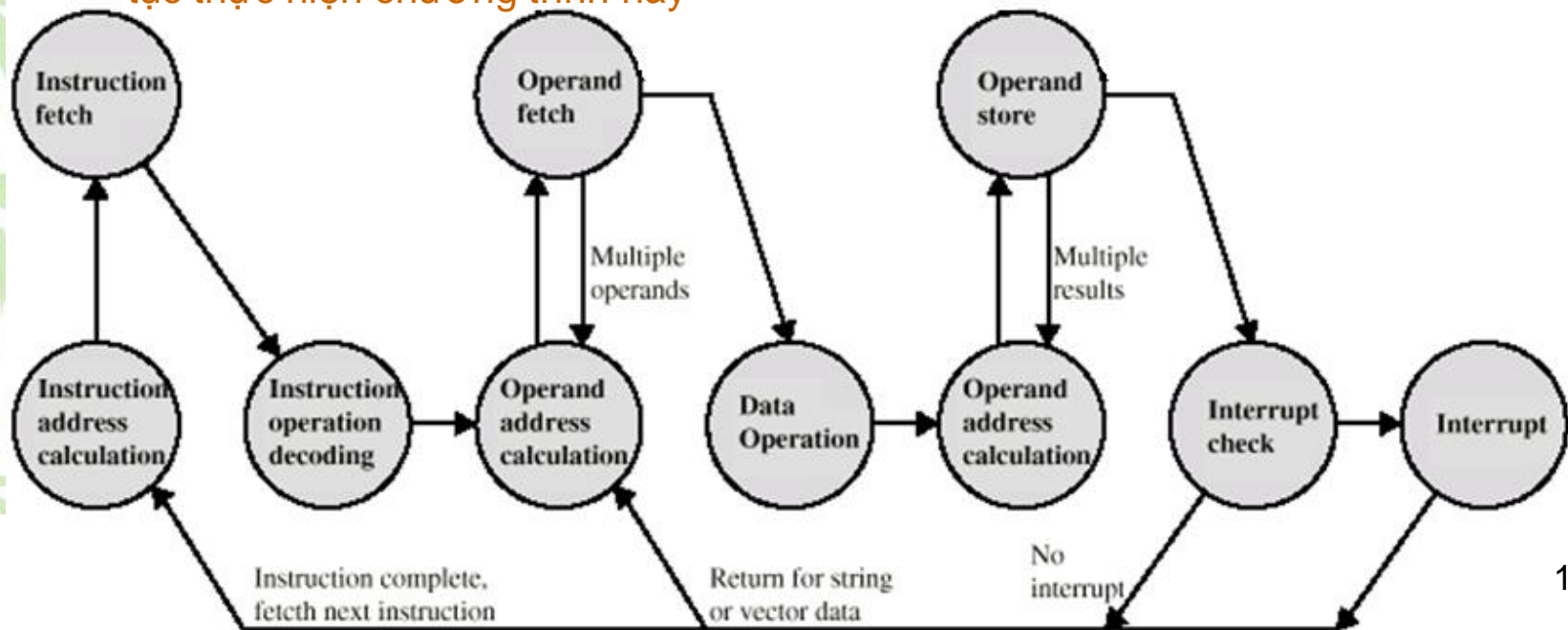


# Ngắt

- Ngắt (Interrupt) là cơ chế cho phép ngắt quá trình thực thi tuần tự thông thường từng lệnh của bộ xử lý để phục vụ công việc khác như nhập xuất.
- Một số loại ngắt
  - Ngắt chương trình
    - Debug chương trình
    - Trường hợp tràn số, chia cho 0,...
  - Ngắt đồng hồ
    - Được phát sinh bởi bộ định giờ bên trong bộ xử lý
    - Được sử dụng trong các môi trường đa nhiệm
  - Nhập xuất
    - Ví dụ: nhập ký tự,...
  - Lỗi phần cứng
    - Ví dụ: lỗi truyền dữ liệu,...

# Quá trình phục vụ ngắt

- Bộ xử lý kiểm tra ngắt mỗi khi thực thi xong 1 lệnh dựa vào tín hiệu ngắt
- Nếu không có ngắt, nạp lệnh kế tiếp có địa chỉ trong PC.
- Nếu có ngắt:
  - Tạm ngừng thực thi tiếp các lệnh của chương trình đang được thực hiện.
  - Lưu lại các dữ liệu đang thực hiện dang dở của chương trình.
  - Đặt địa chỉ bắt đầu thủ tục xử lý ngắt vào thanh ghi PC.
  - Xử lý ngắt
  - Khôi phục các dữ liệu đang thực hiện dang dở của chương trình bị ngắt và tiếp tục thực hiện chương trình này





# Tham khảo

- Chương 12, William Stallings

