



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

# **TH028 – Kiến trúc máy tính và hợp ngữ**

## ***Bài 6. Kiến trúc bộ lệnh MIPS***

**Phạm Tuấn Sơn**

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



# Mục tiêu

- Sau bài này, SV có khả năng:
  - Có khả năng lập trình hợp ngữ MIPS
  - Giải thích quan điểm thiết kế bộ lệnh MIPS
  - Trình bày các vấn đề cần quan tâm khi thiết kế một bộ lệnh
  - Có khả năng tự thiết kế một bộ lệnh theo một quan điểm nào đó



# Nội dung

- Kiến trúc bộ lệnh MIPS
- Các vấn đề khi thiết kế bộ lệnh
- Quan điểm thiết kế bộ lệnh MIPS



# Kiến trúc bộ lệnh

- Công việc cơ bản nhất của CPU là thực thi các lệnh (*instruction*).
- Các CPU khác nhau cài đặt các tập lệnh khác nhau. Tập hợp các lệnh mà một CPU nào đó cài đặt gọi là kiến trúc bộ lệnh (*Instruction Set Architecture – ISA*).
  - Ví dụ: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...
- Môn học sẽ sử dụng kiến trúc MIPS để minh họa.
- Tại sao sử dụng MIPS để giảng dạy thay vì Intel 80x86?



# Lệnh số học trong MIPS

- Cú pháp:

opt opr,opr1,opr2

trong đó:

opt – Tên thao tác (toán tử, tác tử)

opr – Thanh ghi (toán hạng, tác tố đích)  
chứa kết quả

opr1 – Thanh ghi (toán hạng nguồn thứ 1)

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



# Toán hạng thanh ghi

- MIPS hỗ trợ 32 thanh ghi đánh số từ \$0 – \$31.  
*Tại sao là 32 ?* Để dễ sử dụng, các thanh ghi còn có thể được truy xuất thông qua tên của nó.
- Mỗi thanh ghi có kích thước 32 bit. *Tại sao là 32 ?*
  - Trong MIPS, nhóm 32 bit được gọi là một từ (**word**)
- Trong đó, 8 thanh ghi thường được sử dụng để thực hiện các phép tính được đánh số \$16 – \$23.  
$$\$16 - \$23 \sim \$s0 - \$s7 \text{ (saved register)}$$

(tương ứng với biến C)



# 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.



# 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  $a = b + c$ ) là thao tác trên số có dấu hay không dấu ?
- Có thể sử dụng 1 toán hạng đóng 2 vai trò vừa là toán hạng nguồn, vừa là toán hạng đích → lệnh chỉ cần 2 toán hạng. *Tại sao không ?*



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



# Cộng, trừ số nguyên (4/4)

- Làm thế nào để thực hiện dãy tính sau?

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

- MIPS hỗ trợ thêm 8 thanh ghi tạm đánh số \$8 – \$15 để lưu các kết quả trung gian

\$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)
```



# 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 ?*



# Thao tác luận lý

- Các thao tác số học xem dữ liệu trong thanh ghi như một giá trị đơn (số nguyên có dấu/ không dấu)
- Cần có các thao tác trên từng bit dữ liệu → thao tác luận lý
- Các thao tác luận lý xem dữ liệu trong thanh ghi là dãy 32 bit thay vì một giá trị đơn.
- 2 loại thao tác luận lý:
  - Phép toán luận lý
  - Phép dịch luận lý



# Lệnh luận lý

- Cú pháp:

opt opr,opr1,opr2

Trong đó:

opt – Tên thao tác

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)

- Tại sao toán hạng nguồn thứ 1 không thể là hằng số ?

- *Tại sao các lệnh luận lý (và hầu hết các lệnh của MIPS sẽ học) đều có 1 thao tác và 3 toán hạng (như các lệnh số học) ?*

# Phép toán luận lý

- 2 phép toán luận lý cơ bản: AND và OR
- Bảng chân trị:

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

- AND: kết quả là 1 chỉ khi tất cả đầu vào đều bằng 1
- OR: kết quả là 0 chỉ khi tất cả đầu vào đều bằng 0





# Lệnh tính toán luận lý

- Các lệnh:

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

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

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

- ~~– not:~~

- `A nor 0 = not (A or 0) = 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`, ...?*





# Sử dụng các phép toán luận lý (1/3)

- Nhận xét: bit nào `and` với 0 sẽ ra 0, `and` với 1 sẽ ra chính nó.
- Phép `and` được sử dụng để giữ lại giá trị 1 số bit, trong khi xóa tất cả các bit còn lại. Bit nào cần giữ giá trị thì `and` với 1, bit nào không quan tâm thì `and` với 0. Dãy bit có vai trò này gọi là mặt nạ (**mask**).

– Ví dụ:

**mask:**

1011 0110 1010 0100 0011	1101 1001 1010
0000 0000 0000 0000 0000	1111 1111 1111

– Kết quả sau khi thực hiện `and`:

0000 0000 0000 0000 0000	1101 1001 1010
--------------------------	----------------

**mask 12 bit cuối**



## Sử dụng các phép toán luận lý (2/3)

- Giả sử  $\$t0$  giữ giá trị của dãy bit đầu và  $\$t1$  chứa giá trị mask trong ví dụ trên, ta có lệnh sau:  
 $\text{and } \$t0, \$t0, \$t1$
- Sử dụng phép  $\text{and}$  để chuyển từ ký tự thường thành ký tự hoa; từ ký tự số thành số



# Sử dụng các phép toán luận lý (3/3)

- Nhận xét: bit nào `or` với 1 sẽ ra 1, `or` với 0 sẽ ra chính nó.
- Phép `or` được sử dụng để bật lên 1 số bit, trong khi giữ nguyên giá trị tất cả các bit còn lại. Bit nào cần bật lên thì `or` với 1, bit nào không quan tâm thì `or` với 0.
  - Ví dụ, nếu `$t0` có giá trị `0x12345678`, và `$t1` có giá trị `0xFFFF` thì sau lệnh:  
`or $t0, $t0, $t1`
  - ... `$t0` sẽ có giá trị `0x1234FFFF` (nghĩa là giữ lại 16 bit cao và bật tất cả 16 bit thấp).

# Lệnh dịch

- Cú pháp:

opt opr,opr1,opr2

Trong đó

opt – Tên thao tác

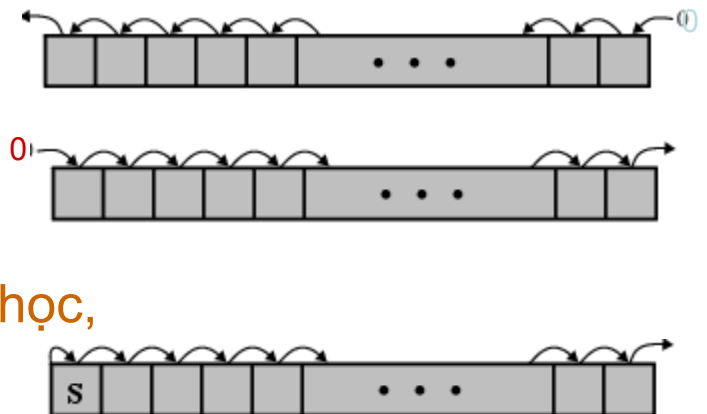
opr – Toán hạng đích chứa kết quả (thanh ghi)

opr1 – Toán hạng nguồn thứ 1 (thanh ghi)

opr2 – Số bit dịch (hằng số < 32)

- Các lệnh:

1. `sll` (shift left logical): dịch trái luận lý, thêm vào các bit 0 bên phải
2. `srl` (shift right logical): dịch phải luận lý<sup>0</sup> và thêm vào các bit 0 bên trái
3. `sra` (shift right arithmetic): dịch phải số học, thêm vào các bit dấu bên trái





# Ví dụ

- `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`  
( $-16 / 2^2$ )



## Nhận xét

- Có thể sử dụng các phép dịch trái để thực hiện phép nhân cho 2 mũ:

`a *= 8;` (trong C)

Tương ứng với lệnh:

`sll $s0, $s0, 3` (trong MIPS)

- Tương tự, sử dụng phép dịch phải để thực hiện phép chia cho 2 mũ
- Đối với số có dấu, sử dụng phép dịch số học `sra`



# Biểu diễn lệnh

- Máy tính, hay nói chính xác là CPU, hiểu được các lệnh như “add \$t0, \$0, \$0” ? Không.
- Các lệnh như “add \$t0, \$0, \$0” là một cách thể hiện dễ hiểu, gọi là hợp ngữ (Assembly)
- Máy tính (CPU) chỉ hiểu được các bit 0 và 1. Dãy bit mà máy tính hiểu được để thực hiện 1 công việc gọi là lệnh máy (machine language instruction).
- Mỗi lệnh máy MIPS có kích thước 32 bit (*Tại sao ?*), được chia làm các nhóm bit, gọi là trường (field), mỗi nhóm bit có một vai trò trong lệnh máy.



# Cấu trúc lệnh MIPS

- Các lệnh đã học (add, addu, sub, subu, and, or, nor, sll, srl, sra) đều có cấu trúc như sau:  $6 + 5 + 5 + 5 + 5 + 6 = 32$  (bit)

6	5	5	5	5	6
---	---	---	---	---	---

- Để dễ hiểu, mỗi trường được đặt tên như sau:

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

- Cấu trúc trên được gọi là R-Format
- Tại sao mỗi trường có kích thước như vậy ?*





# Cấu trúc R-Format (1/3)

- opcode: mã thao tác, cho biết lệnh làm 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 mỗi trường có kích thước 6 bit?*
- *Tại sao không kết hợp 2 trường `opcode` và `funct` thành 1 trường duy nhất 12-bit ?*



## Cấu trúc R-Format (2/3)

- rs (**S**ource **R**egister): thanh ghi nguồn, thường được dùng để chứa toán hạng nguồn thứ 1
- rt (**T**arget **R**egister): thanh ghi nguồn, thường được dùng để chứa toán hạng nguồn thứ 2 (misnamed)
- rd (**D**estination **R**egister): thanh ghi đích, thường được dùng để chứa kết quả của lệnh.
- 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)



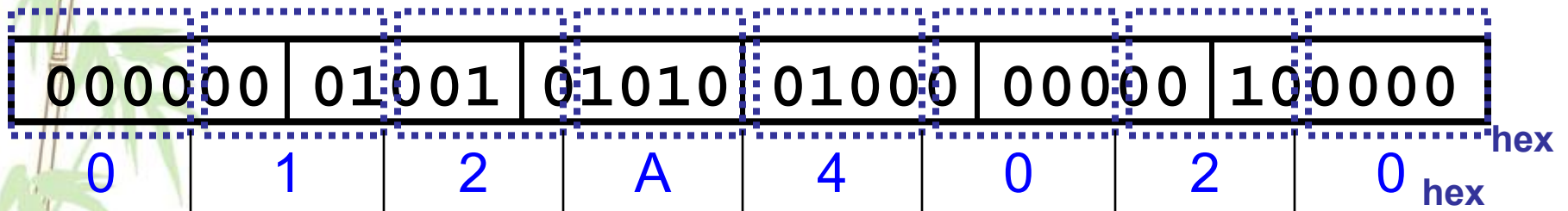
# Cấu trúc R-Format (3/3)

- 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.

# Ví dụ cấu trúc R-Format (1/2)

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

Biểu diễn lệnh dưới dạng nhị phân



Giá trị thập phân tương ứng của từng trường

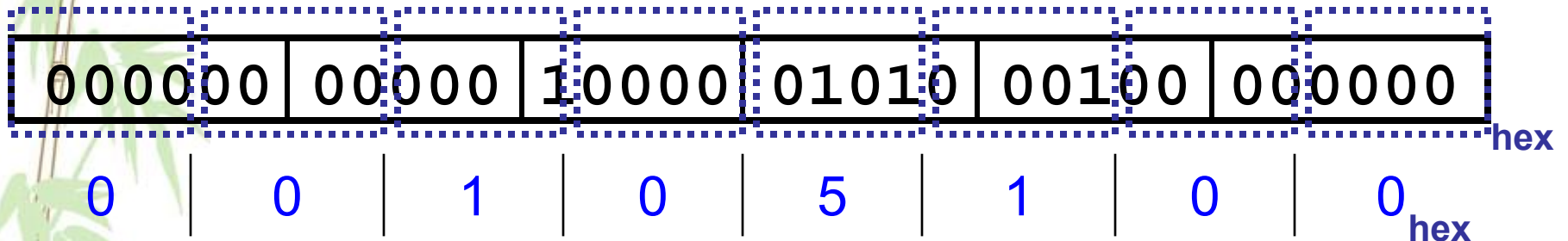
0	9	10	8	0	32
---	---	----	---	---	----

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)

## Ví dụ cấu trúc R-Format (2/2)

sll \$t2, \$s0, 4

Biểu diễn lệnh dưới dạng nhị phân



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)

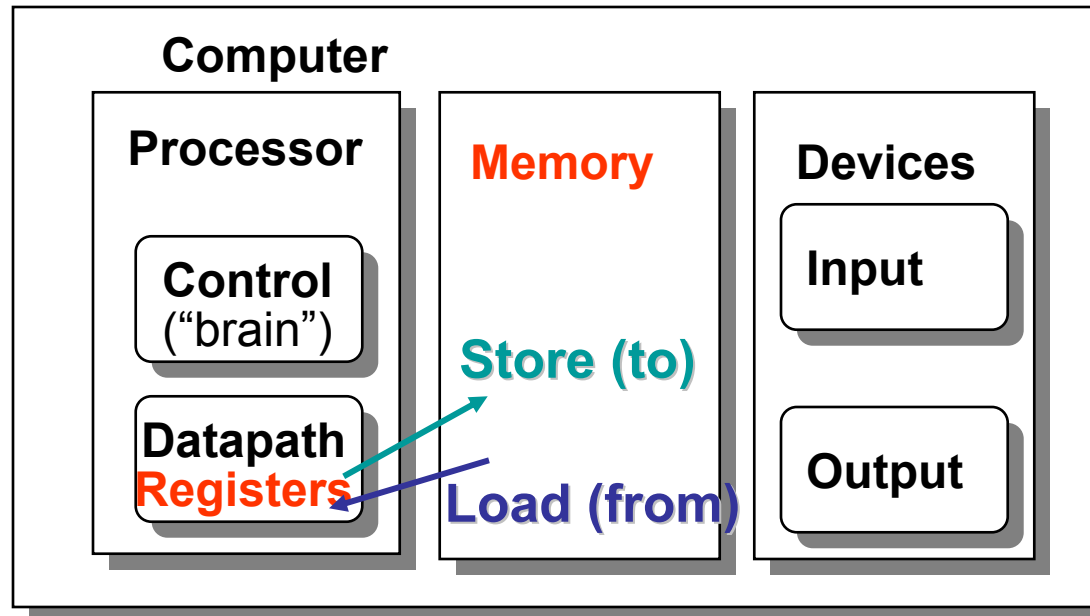


# Toán hạng vùng nhớ (1/2)

- Các lệnh đã học chỉ thao tác trên dữ liệu là số nguyên và dãy bit nằm trong các thanh ghi
- Dữ liệu thực tế không đơn giản như vậy. Làm sao để thao tác trên các kiểu dữ liệu phức tạp hơn như mảng hay cấu trúc?
- Cần bộ nhớ để lưu mọi dữ liệu và lệnh
- Bộ xử lý nạp các dữ liệu và lệnh này vào các thanh ghi để xử lý rồi lưu kết quả ngược trở lại bộ nhớ

# Toán hạng vùng nhớ (2/2)

- 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)



- Tại sao lại không hỗ trợ các lệnh thao tác trực tiếp trên vùng nhớ ?





# Thao tác trên vùng nhớ

- Bộ nhớ là mảng 1 chiều  
các ô nhớ có địa chỉ
- Lệnh nạp, lưu dữ liệu cần ít nhất  
1 toán hạng nguồn và 1 toán hạng đích
- Cấu trúc R-Format

.	.
.	.
.	.
3	100
2	10
1	101
0	1

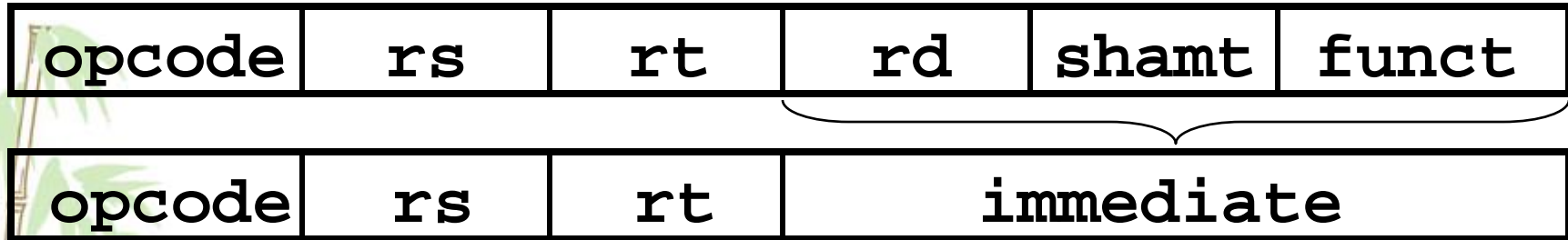
Địa chỉ    Dữ liệu

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

- Sử dụng cấu trúc R-Format cho các lệnh nạp, lưu dữ liệu ?
- Nếu không sử dụng cấu trúc R-Format, hướng giải quyết nhằm giảm thiểu thay đổi so với cấu trúc này ?



# Xác định địa chỉ vùng nhớ



- 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 có 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 giá trị trong thanh ghi \$t0 cộng thêm 8 (byte)



# Lệnh di chuyển dữ liệu (1/2)

- Cú pháp:

opt opr,opr1(opr2)

trong đó:

opt - Tên thao tác

opr - Thanh ghi lưu từ 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  $(\$s0 + 12)$



# Di chuyển dữ liệu: Thanh ghi vào 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

## Con trỏ vs. giá trị

- **Nguyên tắc:** Một thanh ghi có thể lưu bất kỳ giá trị 32 bit nào, có thể là số nguyên (có dấu/ không dấu), có thể là địa chỉ của một vùng nhớ
- Nếu ghi thì  
`add $t2, $t1, $t0`  
`$t0` và `$t1` lưu giá trị
- Nếu ghi thì  
`lw $t2, 0($t0)`  
`$t0` chứa một địa chỉ (vai trò như một con trỏ)

# Nguyên tắc lưu dữ liệu trong bộ nhớ (1/2)

- MIPS lưu dữ liệu trong bộ nhớ theo nguyên tắc Alignment Restriction, nghĩa là các đối tượng lưu trong bộ nhớ phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng
- Như vậy, từ nhớ phải bắt đầu tại địa chỉ là bội số của 4

**Aligned**

**Not  
Aligned**

0	1	2	3

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 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 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



## Một số lưu ý về định vị dữ liệu trong bộ nhớ

- MIPS truy xuất dữ liệu trong bộ nhớ theo nguyên tắc Alignment Restriction
- Tuy nhiên, bộ nhớ ngày nay lại được đánh địa chỉ theo từng byte (8 bit).
- Lưu ý: để truy xuất vào một từ nhớ sau một từ nhớ thì cần tăng 1 lượng 4 byte chứ không phải 1 byte
- Do đó, luôn nhớ rằng đối với các lệnh  $lw$  và  $sw$  thì độ dời (offset) phải là bội số của 4





## Ví dụ

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

- Câu lệnh C :

$g = h + A[5];$

được biên dịch thành lệnh máy MIPS như sau:

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



## Lệnh nạp, lưu 1 byte nhớ (1/2)

- 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). *Tại sao?*
  - 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  $\$s0$ .

## Lệnh nạp, lưu 1 byte nhớ (2/2)

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). *Tại sao ?*
  - 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`



# Vai trò của thanh ghi vs. vùng nhớ

- *Tại sao không sử dụng toàn bộ toán hạng vùng nhớ ?*
- *Tại sao không xây dựng thật nhiều thanh ghi để không phải dùng toán hạng vùng nhớ ?*
- Một chương trình trên máy tính cho dù được viết bằng bất cứ NNLT nào, để thực thi được trên máy tính, thì đều phải biên dịch thành các lệnh máy
- 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

## Cấu trúc I-Format (1/2)

- Như vậy MIPS hỗ trợ thêm 1 cấu trúc lệnh mới:  $6 + 5 + 5 + 16 = 32$  bits

6	5	5	16
---	---	---	----

- Tên dễ hiểu như sau:

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

- Chú ý:** chỉ có trường “immediate” là không nhất quán với cấu trúc R-format. Tuy nhiên, quan trọng nhất là trường `opcode` không thay đổi so với cấu trúc R-Format

# Cấu trúc I-Format (2/2)

- 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 (I-Format) trong khi kích thước mỗi trường vẫn hợp lý.
- rs: thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- rt (**register target**): thanh ghi đích, thường được dùng để chứa kết quả của lệnh.
- 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`.





# Toán hạng Hằng số (1/2)

- 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 ?*





## Toán hạng Hằng số (2/2)

- 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

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

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

`immediate = 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?  
→ 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 (1/2)

- Nhắc lại: tràn số xảy ra khi kết quả phép tính vượt quá độ chính xác giới hạn cho phép (của máy tính).
- Ví dụ (số nguyên không dấu 4-bit):

+15	1111
<u>+3</u>	<u>0011</u>
+18	10010

- Nhưng không có chỗ để chứa cả 5 bit nên chỉ chứa kết quả 4 bit 0010, là +2 → sai.



# Tràn số trong phép tính số học (2/2)

- Một số ngôn ngữ có khả năng phát hiện tràn số (Ada), một số không (C)
- 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, zero
B: add $s1, $s0, 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

- Nếu chỉ với các lệnh xử lý dữ liệu (thao tác số học, thao tác luận lý, nạp, lưu dữ liệu) thì chỉ dừng lại ở việc xây dựng 1 **calculator**.
- 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:`



# Lệnh rẽ nhánh của MIPS

- Rẽ nhánh có điều kiện

`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`

- Rẽ nhánh không điều kiện

`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:

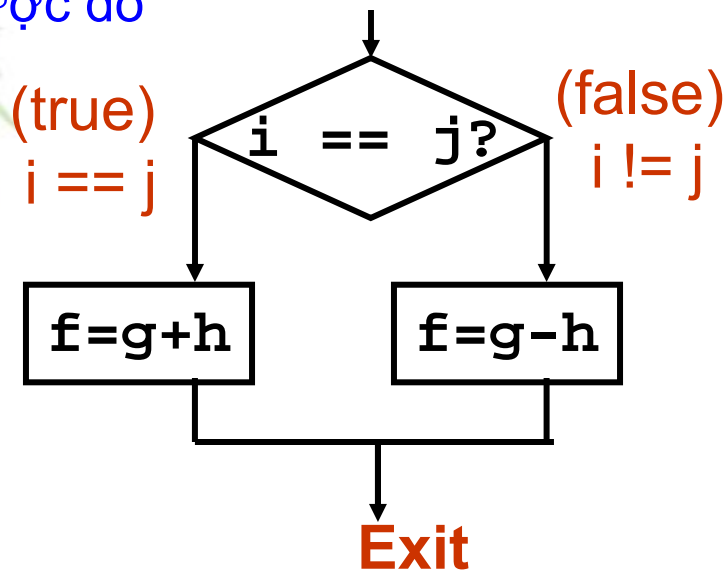
`beq $0, $0, label`



# 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    Fin             # goto Fin
```

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

```
Fin:
```



## Lệnh rẽ nhánh: Định vị theo thanh ghi PC (1/4)

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

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

- opcode xác định beq hay bne
- 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: Định vị theo thanh ghi PC (2/4)

- `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: Định vị theo thanh ghi PC (3/4)

- 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: Định vị theo thanh ghi PC (4/4)

- 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

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

End:

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 = 3 ???

Biểu diễn lệnh dưới dạng nhị phân

000100	01001	00000	000000000000000011
--------	-------	-------	--------------------

Giá trị thập phân tương ứng của từng trường

4	9	0	3
---	---	---	---



## 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` → tăng kích thước lệnh rẽ nhánh ?

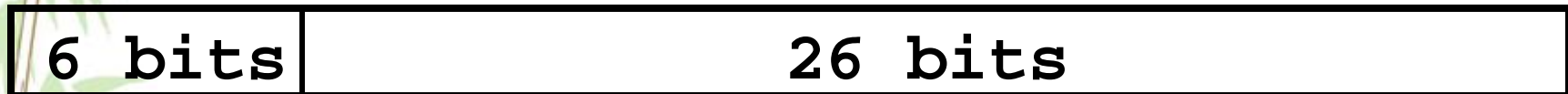
# Cấu trúc J-Format

- MIPS hỗ trợ lệnh  $j$  (và lệnh  $j_{a1}$  sẽ học sau) cho phép nhảy tới bất kỳ nơi nào trong bộ nhớ.
- Mỗi lệnh 32 bit nên lý tưởng nhất là có thể nhảy trong khoảng  $2^{32}$  (4 Gi) bộ nhớ.
- Tuy nhiên, mỗi lệnh cần có trường opcode (6-bit ?) để xác định chức năng của lệnh.
- Hình thành một cấu trúc lệnh mới J-Format, nhưng vẫn nhất quán với các cấu trúc lệnh đã tồn tại R-Format và I-Format



# Cấu trúc J-Format

- Cấu trúc lệnh J-Format như sau:  $6 + 26 = 32$  bit



- Tên dễ nhớ như sau



- Chú ý:
  - Giữ trường `opcode` giống như cấu trúc R-format và I-format.
  - Kết hợp tất cả các trường còn lại thành trường địa chỉ đích (target address) có kích thước lớn hơn.
  - Tương tự lệnh rẽ nhánh, địa chỉ đích của lệnh nhảy được tính theo đơn vị từ nhớ



# Cấu trúc J-Format

- Như vậy, với cấu trúc J-Format, có thể nhảy trong khoảng  $2^{26}$
- Có nghĩa là không thể nhảy tới các 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).





# 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 (1/4)

- 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?
- MIPS hỗ trợ lệnh:
  - “Set on Less Than”
  - Cú pháp: `slt reg1, reg2, reg3`
  - Nghĩa là: `reg1 = (reg2 < reg3);`

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

Same thing...

“set” nghĩa là “set to 1”,  
“reset” nghĩa là “set to 0”.



# So sánh không bằng trong MIPS (2/4)

- 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:
```

- Thanh ghi \$0 luôn chứa giá trị 0, nên lệnh bne và beq thường được dùng để so sánh sau lệnh slt.
- Cặp `slt`  $\rightarrow$  `bne` tương đương `if (... < ...)` goto...



# So sánh không bằng trong MIPS (3/4)

- Các phép so sánh còn lại  $>$ ,  $\leq$  and  $\geq$  thì sao?
- Có thể thực hiện phép  $>$ ,  $\geq$ ,  $\leq$  bằng cách kết hợp lệnh `slt` và các lệnh rẽ nhánh?
- Tại sao MIPS không có 3 lệnh so sánh tương ứng, chẳng hạn *sgt*, *sle*, *sge* ?



# So sánh không bằng trong MIPS (4/4)

- `# 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. *Tại sao?*
  - Hữu ích đối với vòng lặp `for`

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

`Loop: . . .`

**M  
I  
P  
S**

`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` → `beq` tương ứng với `if(... ≥ ...)goto...`**



## 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



# Thủ tục trong C

```
main() {  
    int a,b;  
    ...  
    ... sum(a,b) ;  
    ...  
}  
/* really dumb mult function */  
int sum (int x, int y){  
    return x+y;  
}
```

- Thủ tục được chuyển thành lệnh máy như thế nào ?
- Dữ liệu nào được lưu trữ như thế nào ?

# Nhận xét

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

<code>sum(a, b) ;</code>	<code>␣</code>	<code>sum</code>	<code># nhảy tới</code>
<code>...</code>	<code>␣</code>	<code>...</code>	<code># nhãn sum</code>
<code>int sum (...)</code>	<code>␣</code>	<code>sum:</code>	

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

<code>return ...</code>	<code>␣</code>	<code>?</code>
-------------------------	----------------	----------------



# Ví dụ

```
... 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	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	
2004	jr	\$ra	# nhảy tới địa chỉ
			# trong \$ra

- Ghi chú: tất cả các lệnh MIPS đều có kích thước 4 byte (32 bit). Tại sao ?



# Lưu trữ dữ liệu

- MIPS hỗ trợ thêm một số thanh ghi để lưu trữ các dữ liệu phục vụ cho thủ tục:

– Đối số	<code>\$a0, \$a1, \$a2, \$a3</code>
– Kết quả trả về	<code>\$v0, \$v1</code>
– Biến cục bộ	<code>\$s0, \$s1, ... , \$s7</code>
– Địa chỉ quay về	<code>\$ra</code>

- Nếu có 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 ?  
→ Sử dụng ngăn xếp (stack).



# Nhận xét

```
... sum(a,b); ... /* a,b:$s0,$s1 */  
}
```

**C**

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

- M  
I  
P  
S**
- 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.

2000 sum: add \$v0,\$a0,\$a1  
2004 jr \$ra # *lệnh mới*





## 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 `jal` (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?*

# Các lệnh mới

- **jal (jump and link):**
  - Cú pháp: `jal label`
  - 1 (link): Lưu địa chỉ của lệnh kế tiếp vào thanh ghi `$ra` (tại sao là lệnh kế tiếp mà không phải là lệnh hiện tại?)
  - 2 (jump): nhảy tới nhãn `label`
- **Lệnh jr (jump register)**
  - Cú pháp: `jr register`
  - Nhảy tới địa chỉ nằm trong thanh ghi `register` thay vì nhảy tới 1 nhãn.
- **2 lệnh này được sử dụng hiệu quả trong thủ tục:**
  - `jal` lưu địa chỉ quay về vào thanh ghi `$ra` và nhảy tới thủ tục
  - `jr $ra` Nhảy tới địa chỉ quay về đã được lưu trong `$ra`



# Bài tập

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

- Thủ tục được chuyển thành lệnh máy như thế nào ?
- Dữ liệu nào được lưu trữ như thế nào ?



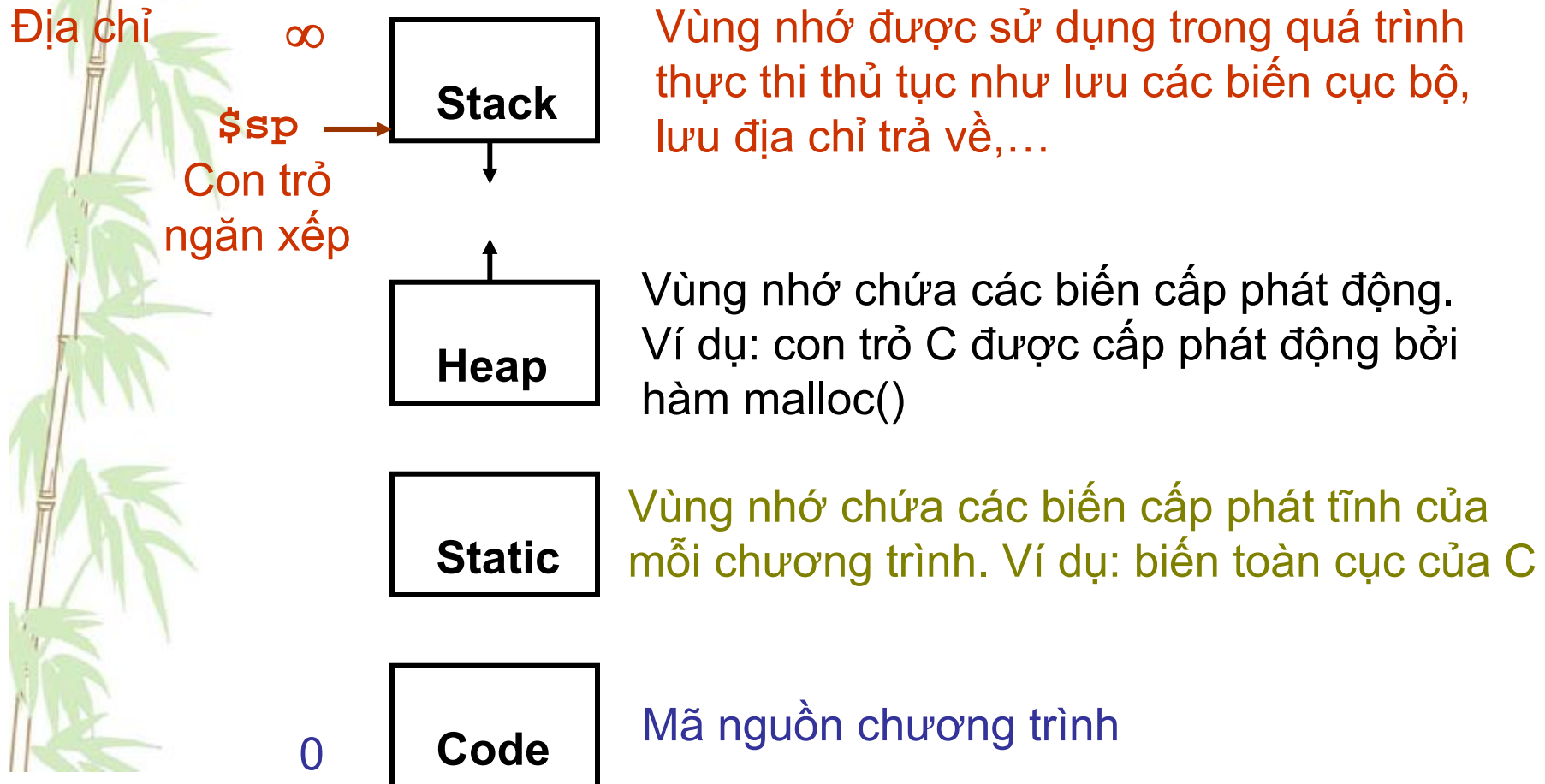
# Thủ tục lồng nhau

```
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`.
- Trình biên dịch giải quyết vấn đề này

# Mô hình cấp phát bộ nhớ của C

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




# Sử dụng ngăn xếp

- 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



```
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
                        # cửa 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
                        # cửa 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: ...
```





## Các bước thực thi một thủ tục

- 1) Lưu tạm các dữ liệu cần thiết vào ngăn xếp.
- 2) Gán các đối số (nếu có).
- 3) Gọi lệnh `jal`
- 4) Khôi phục các dữ liệu đã lưu tạm vào ngăn xếp.



# Cấu trúc cơ bản của việc thực thi thủ tục

## Đầu thủ tục

```
entry_label:  
addi $sp,$sp, -framesize  
sw $ra, framesize-4($sp)
```

# cất địa chỉ trả  
# về cửa thủ tục  
# trong \$ra vào  
# ngăn xếp

Lưu tạm các thanh ghi khác nếu cần

## Thân thủ tục ...

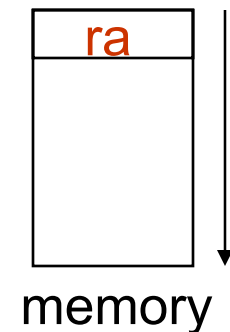
(có thể gọi các thủ tục khác...)

## Cuối thủ tục

Phục hồi các thanh ghi khác nếu cần

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

# khôi phục \$ra






# Một số nguyên tắc khi thực thi thủ tục

- Gọi thủ tục bằng lệnh `jal` và quay về từ lời gọi thủ tục bằng lệnh `jr $ra`
- 4 thanh ghi chứa đối số `$a0`, `$a1`, `$a2` và `$a3`
- Kết quả trả về chứa trong thanh ghi `$v0` (và `$v1` nếu cần)
- Phải tuân theo nguyên tắc sử dụng các thanh ghi (register conventions)



# Vai trò của 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 (1/3)

- $\$0$ : **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 (2/3)

- $\$ra$ : **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.



# Nguyên tắc sử dụng thanh ghi (3/3)

- Tóm tắt nguyên tắc sử dụng thanh ghi trong thủ tục
  - Nếu thủ tục R gọi thủ tục E, thì thủ tục R phải lưu vào ngăn xếp các thanh ghi tạm có thể bị sử dụng trước khi gọi lệnh  $j_{a1}$ .
  - Thủ tục E phải lưu lại các thanh ghi lưu trữ nếu nó dùng các thanh ghi này.
  - Nhớ: thủ tục gọi/ thủ tục được gọi chỉ cần lưu các thanh ghi tạm/ thanh ghi lưu trữ nó dùng, không phải tất cả thanh ghi.





# 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



# Một số vấn đề thiết kế bộ lệnh

- Thao tác

- Những loại thao tác nào?
- Bao nhiêu thao tác?
- Thực hiện công việc gì?
- Độ phức tạp của công việc?

- Thanh ghi

- Số lượng thanh ghi?
- Thao tác nào thực hiện trên thanh ghi nào?

- Định dạng lệnh

- Kích thước lệnh
- Kích thước các trường
- Số lượng toán hạng
- Các kiểu định vị toán hạng

# Kích thước lệnh

- Lệnh càng dài thì càng có thể có nhiều mã thao tác, nhiều toán hạng, truy xuất được địa chỉ vùng nhớ lớn hơn, nghĩa là càng có khả năng thực hiện được nhiều công việc hơn nhưng ...
- Kích thước lệnh ảnh hưởng và bị ảnh hưởng bởi:
  - **Cấu trúc đường truyền bus**
    - Chiều dài lệnh nên bằng hoặc là bội số của đường truyền dữ liệu từ bộ nhớ (độ rộng bus dữ liệu)
  - **Kích thước và tổ chức bộ nhớ**
    - Kích thước lệnh sao cho có thể truy xuất được toàn bộ nhớ.
    - Kích thước lệnh nên bằng, ít nhất bội số của một đơn vị bộ nhớ.
  - **Tốc độ CPU**
    - Chênh lệch giữa tốc độ xử lý của CPU và tốc độ đọc bộ nhớ có thể làm giảm hiệu năng của toàn bộ hệ thống (thực thi lệnh nhanh hơn nạp lệnh nhiều)
    - Giải pháp:
      - Dùng lệnh có kích thước ngắn
      - Dùng bộ nhớ cache
- Trong cùng bộ lệnh, các lệnh có kích thước khác nhau, mã thao tác có kích thước khác nhau ?

# RISC vs. CISC

- Xu hướng ban đầu là xây dựng bộ lệnh theo kiến trúc CISC, nghĩa là đưa vào CPU càng nhiều lệnh càng tốt và có thể thực hiện các thao tác phức tạp
  - Kiến trúc VAX có các lệnh thực hiện nhân các đa thức !
- RISC - Reduced Instruction Set Computing (Cocke IBM, Patterson, Hennessy, 1980s)
  - Giữ bộ lệnh nhỏ và đơn giản để dễ dàng xây dựng nhanh chóng phần cứng.
  - Phần mềm sẽ thực hiện các thao tác phức tạp từ các thao tác đơn giản hơn của phần cứng.



## 4 nguyên tắc thiết kế bộ lệnh MIPS

- Bộ lệnh MIPS được xây dựng theo kiến trúc RISC với 4 nguyên tắc sau:
  - Đơn giản và có quy tắc  
(Simplicity favors regularity)
  - 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)