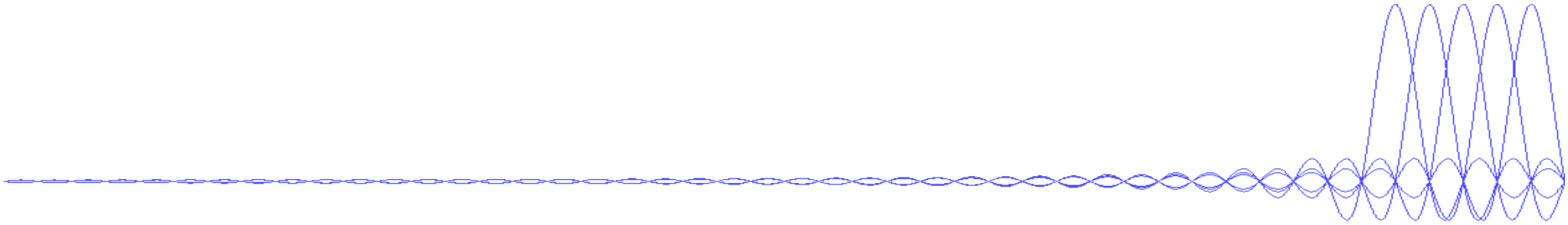
COMPUTER ENGINEERING

KIẾN TRÚC MÁY TÍNH

UIT
TRƯỜNG ĐẠI HỌC
CÔNG NGHỆ THÔNG TIN

KIẾN TRÚC BỘ LỆNH



Kiến trúc bộ lệnh

Mục tiêu:

1. **Hiểu cách biểu diễn và cách thực thi các lệnh trong máy tính**
2. **Chuyển đổi lệnh ngôn ngữ cấp cao sang assembly và mã máy**
3. **Chuyển đổi lệnh mã máy sang ngôn ngữ cấp cao hơn**
4. **Biết cách lập trình bằng ngôn ngữ assembly cho MIPS**

Slide được dịch và các hình được lấy từ sách tham khảo:

Computer Organization and Design: The Hardware/Software Interface, Patterson, D. A., and J. L. Hennessy, Morgan Kaufman, Revised Fourth Edition, 2011.

Kiến trúc bộ lệnh

1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

Giới thiệu

- Để ra lệnh cho máy tính ta phải nói với máy tính bằng ngôn ngữ của máy tính. Các từ của ngôn ngữ máy tính gọi là các lệnh ***(instructions)*** và tập hợp tất cả các từ gọi là bộ lệnh ***(instruction set)***
- Bộ lệnh trong chương này là MIPS, một bộ lệnh kiến trúc máy tính được thiết kế từ năm 1980. Cùng với hai bộ lệnh thông dụng nhất ngày nay:
  - ARM (rất giống MIPS)
  - The Intel x86

Kiến trúc bộ lệnh

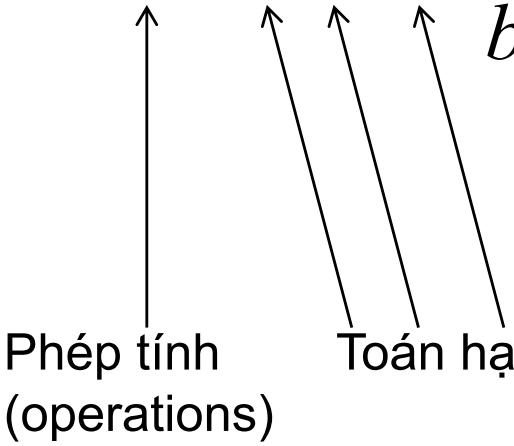
1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

Phép tính (Operations)

Ví dụ:

add *a, b, c* ➔ *Chỉ dẫn cho máy tính thực hiện cộng 2 biến b với c và ghi kết quả vào biến a,*

$$a = b + c.$$



Phép tính (Operations)

Ví dụ 1.

C/Java

```
a = b + c;
d = a – e;
```

MIPS

```
add a, b, c
sub d, a, e
```

Ví dụ 2.

C/Java

```
f = (g + h) – (i + j);
```

MIPS

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

COMPUTER ENGINEERING

Ví dụ một số lệnh trên MIPS

Category	Instruction	Example	Meaning
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10

Kiến trúc bộ lệnh

1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

Toán hạng

Có 3 loại toán hạng:

1. Toán hạng thanh ghi (Register Operands)
2. Toán hạng bộ nhớ (Memory Operands)
3. Toán hạng hằng (Constant or Immediate Operands)

Toán hạng

Toán hạng thanh ghi:

❖ Không giống như các chương trình trong ngôn ngữ cấp cao, các toán hạng của các lệnh số học bị hạn chế, chúng phải đặt trong các vị trí đặc biệt được xây dựng trực tiếp trong phần cứng được gọi là **thanh ghi** (số lượng thanh ghi có giới hạn: MIPS-32, ARM Cortex A8-40).

❖ Kích thước của một thanh ghi trong kiến trúc MIPS là 32 bit; nhóm 32 bit xuất hiện thường xuyên nên chúng được đặt tên là "từ" (**word**) trong kiến trúc MIPS.

(Lưu ý: một "từ" trong kiến trúc bộ lệnh khác có thể không là 32 bit)

❖ Một sự khác biệt lớn giữa các biến của một ngôn ngữ lập trình và các biến thanh ghi là số thanh ghi bị giới hạn (thường là 32 thanh ghi trên các máy tính hiện nay)

Toán hạng

Các thanh ghi trong MIPS:

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Toán hạng

Toán hạng bộ nhớ (1):

❖Vi xử lý chỉ có thể giữ một lượng nhỏ dữ liệu trong các thanh ghi, trong khi bộ nhớ máy tính chứa hàng triệu dữ liệu.

❖Với lệnh MIPS, phép tính số học chỉ xảy ra trên thanh ghi, do đó, MIPS phải có các lệnh chuyển dữ liệu giữa bộ nhớ và thanh ghi. Lệnh như vậy được gọi là **lệnh chuyển dữ liệu**.

Lệnh chuyển dữ liệu: *Một lệnh di chuyển dữ liệu giữa bộ nhớ và thanh ghi*

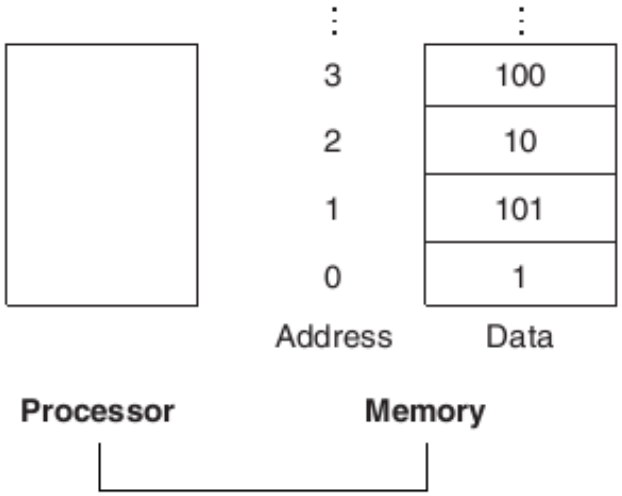
❖ Để truy cập vào một từ trong bộ nhớ, lệnh phải cung cấp **địa chỉ** bộ nhớ.

Địa chỉ*: Một giá trị sử dụng để phân định vị trí của một phần tử dữ liệu cụ thể trong một mảng bộ nhớ.*

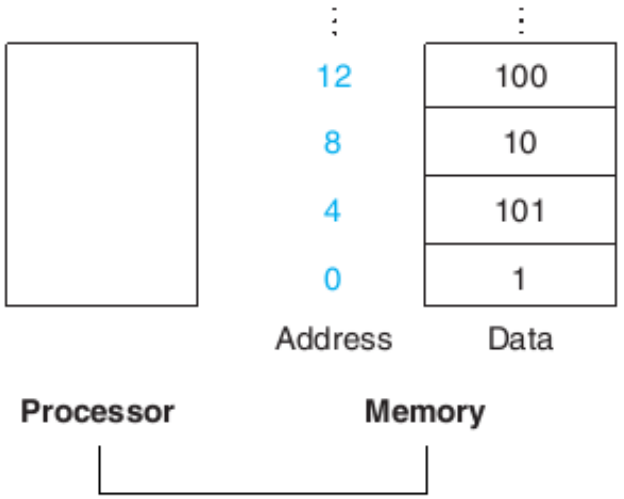
Toán hạng

Toán hạng bộ nhớ (2):

❖ Bộ nhớ chỉ là một mảng đơn chiều lớn, với địa chỉ đóng vai trò là chỉ số trong mảng đó, bắt đầu từ 0. Ví dụ, trong hình 1, địa chỉ của phần tử thứ ba là 2, và giá trị của bộ nhớ [2] là 10.



Hình 1: Địa chỉ và nội dung của bộ nhớ giả lập như mảng.



Hình 2: Địa chỉ và nội dung bộ nhớ MIPS thực tế.

Mỗi từ nhớ (word) của MIPS là 4 bytes. MIPS định địa chỉ theo byte, địa chỉ của mỗi word là địa chỉ của byte đầu tiên trong word đó. **Do đó, địa chỉ mỗi word trong MIPS phải là bội của 4.**

Toán hạng

Toán hạng bộ nhớ (3):

❖ Lệnh chuyển dữ liệu từ bộ nhớ vào thanh ghi gọi là nạp (load) **(viết tắt lw – load word)**. Định dạng của các lệnh nạp:

lw *$s1,20($s2)*

Độ dời (offset)

Địa chỉ nền/cơ sở (Base address)
Thanh ghi chứa địa chỉ nền/cơ sở gọi là thanh ghi nền/cơ sở (Base register)

- *$s1:* thanh ghi nạp dữ liệu vào.
- Một hằng số (20) và thanh ghi ($s2) được sử dụng để truy cập vào bộ nhớ. Tổng số của hằng số và nội dung của thanh ghi này là địa chỉ bộ nhớ của phần tử cần truy cập đến. Nội dung của từ nhớ này sẽ được đưa từ bộ nhớ vào thanh ghi *$s1*

Toán hạng

Toán hạng bộ nhớ (4):

Ví dụ về lệnh lw:

Giả sử rằng *A* là một mảng của 100 phần tử (mỗi phần tử cần 1 word lưu trữ) và trình biên dịch đã kết hợp các biến g và h với các thanh ghi *$s1* và *$s2*. Giả định rằng địa chỉ bắt đầu của mảng A (hay **địa chỉ cơ sở/nền)** chứa trong *$s3*. Hãy biên dịch đoạn lệnh bằng ngôn ngữ C sau sang MIPS:

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

Thực tế trong MIPS một word là 4 bytes, do đó lệnh đúng phải là:

lw $ t0, 32($s3)

➔ Biên dịch:

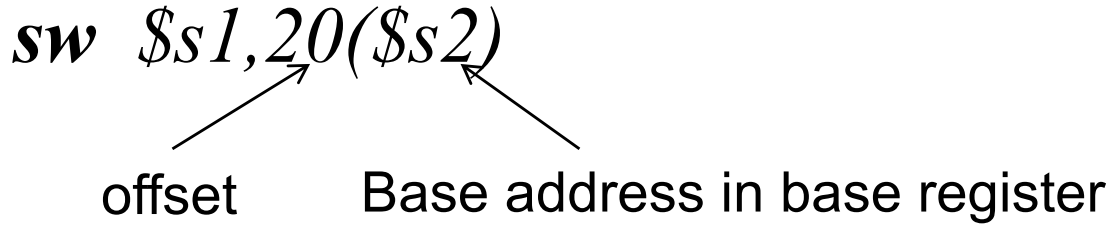
```
lw  $t0, 8($s3)          # $t0 nhận A[8]
add  $s1,$s2,$t0         # g = h + A[8]
```

❖ Hằng số trong một lệnh truyền dữ liệu (*8*) gọi là **offset**, và thanh ghi chứa địa chỉ bắt đầu của mảng (*$s3*) gọi là **thanh ghi cơ sở**.

Toán hạng

Toán hạng bộ nhớ (5):

❖ Lệnh chuyển dữ liệu từ thanh ghi ra bộ nhớ, gọi là lệnh lưu (store) **(viết tắt sw – store word)**. Định dạng của các lệnh lưu:



- $s1: thanh ghi chứa dữ liệu cần lưu.
- Một hằng số (20) và thanh ghi ($s2) được sử dụng để truy cập vào bộ nhớ. Tổng số của hằng số và nội dung của thanh ghi này là địa chỉ bộ nhớ, nơi mà nội dung đang chứa trong thanh ghi *$s1* sẽ được lưu vào đây.

Toán hạng

Toán hạng bộ nhớ (6):

Ví dụ lệnh *sw*:

Giả sử biến *h* được kết nối với thanh ghi *$s2* và địa chỉ cơ sở của mảng *A* là trong *$s3*. Biên dịch câu lệnh C thực hiện dưới đây sang MIPS?

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

➔Biên dịch:

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

Toán hạng

Toán hạng bộ nhớ (7):

- **Alignment Restriction**: Trong MIPS, các từ phải bắt đầu từ địa chỉ là bội số của 4. Yêu cầu này được gọi là một **"alignment restriction"** và nhiều kiến trúc hiện nay buộc tuân theo quy định này nhằm giúp việc truyền dữ liệu nhanh hơn. Tuy nhiên một số kiến trúc vẫn không bắt buộc quy định này.

  (Chú ý: Tại sao tuân theo điều này giúp truyền dữ liệu nhanh hơn → đọc chương 5 sách tham khảo chính)

- Leftmost - **"Big End", "Big Endian"**

  Rightmost - **"Little End", "Little Endian"**

➔ MIPS thuộc dạng *nào?*

Toán hạng

Toán hạng bộ nhớ (7):

`0xFF00AA11`.

Little Endian	
Address	**Contents**
4003	FF
4002	00
4001	AA
4000	11

Big Endian	
Address	**Contents**
4003	11
4002	AA
4001	00
4000	FF

Toán hạng

Toán hạng hằng:

Một hằng số/số tức thời (constant/immediate number) có thể được sử dụng trong trong một phép toán

Ví dụ:

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

Toán hạng hằng

Toán hạng

Tóm lại, chỉ có 3 loại toán hạng trong một lệnh của MIPS

1. Toán hạng thanh ghi (Register Operands)
2. Toán hạng bộ nhớ (Memory Operands)
3. Toán hạng hằng (Constant or Immediate Operands)

Lưu ý:

❖ Các hằng số trong MIPS có thể âm nên không cần phép trừ một thanh ghi và một số tức thời trong MIPS.

❖ Trong thực tế, có một phiên bản khác của MIPS làm việc với các thanh ghi 64 bits, gọi là MIPS-64. MIPS xem xét trong môn học này là MIPS làm việc với các thanh ghi chỉ 32 bit, gọi là MIPS-32.

⇒**Trong phạm vi môn học này, MIPS dùng chung sẽ hiểu là MIPS-32**

Kiến trúc bộ lệnh

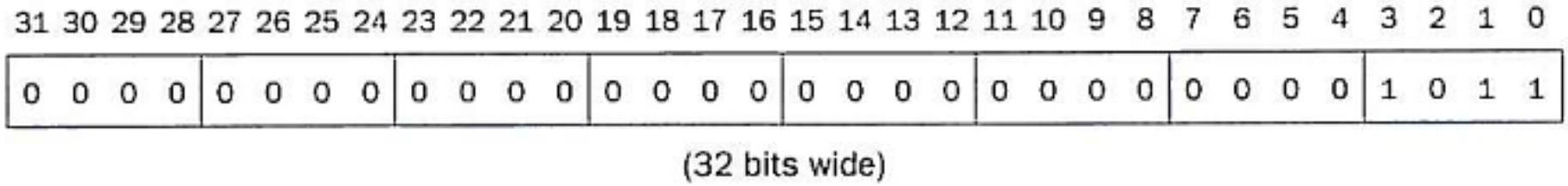
1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

COMPUTER ENGINEERING

Số có dấu và không dấu

- Con người được dạy để suy nghĩ trong hệ cơ số 10, nhưng con số có thể được biểu diễn trong bất kỳ cơ số nào. Ví dụ, 123 cơ số 10 = 1111011 cơ số 2.
- Số lưu trữ trong máy tính như một chuỗi các tín hiệu điện thế cao và thấp, do đó chúng được xem như hệ cơ số 2.

Ví dụ: Hình vẽ dưới đây cho thấy như thế nào một word của MIPS lưu trữ số 1011:



- Một word của MIPS có 32 bit, do đó có thể biểu diễn các số từ 0 đến $2^{32}-1$ (4.294.967.295)
- **Bit trọng số nhỏ nhất *(The least significant bit – LSB)*:** Bit ngoài cùng bên phải trong một từ nhớ (bit 0)
- **Bit trọng số lớn nhất *(The most significant bit – MSB)*:** Bit ngoài cùng bên trái trong một từ nhớ (bit 31)

COMPUTER ENGINEERING

Số có dấu và không dấu

❖ Số dương và âm trong máy tính:

Các máy tính hiện tại sử dụng **bù hai** để biểu diễn nhị phân cho **số có dấu**.

- Nếu MSB = 0: số dương
- Nếu MSB = 1: số âm.

➔Bit thứ 32 (MSB) còn được gọi là **bit dấu**.

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2,147,483,645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2,147,483,646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2,147,483,648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2,147,483,647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2,147,483,646_{ten}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

COMPUTER ENGINEERING

Số có dấu và không dấu

- Nửa phần dương của các con số, từ 0 đến $2{,}147{,}483{,}647_{ten}$ $(2^{31} - 1)$, biểu diễn như thường.
- Phần số âm biểu diễn:

$1000\ldots0000_{two} = -2{,}147{,}483{,}648_{ten}$

$1000\ldots0001_{two} = -2{,}147{,}483{,}647_{ten}$

$1111\ldots1111_{two} = -1_{ten}$

- Bù hai có một số âm $-2{,}147{,}483{,}648_{ten}$, mà không có số dương tương ứng.

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
. . . . . .

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
. . . . . .

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Số có dấu và không dấu

Công thức chuyển từ một số bù hai sang số hệ 10:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Lưu ý: Bit dấu được nhân với -2^{31}, và phần còn lại của các bit sau đó được nhân với các số dương của các giá trị cơ số nào tương ứng của chúng.

Ví dụ: đổi từ hệ 2 sang hệ 10

$$1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1111\ \ 1100_{two}$$

Trả lời:

$$\begin{aligned}
&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0}) \\
&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0 \\
&= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten} \\
&= -4_{ten}
\end{aligned}$$

Số có dấu và không dấu

Mở rộng số có dấu:

Làm thế nào để chuyển đổi một số nhị phân được biểu diễn trong *n* bit thành một số biểu diễn với nhiều hơn *n* bit?

Ví dụ:

Chuyển đổi số nhị phân 16 bit của số 2_{ten} và -2_{ten} thành số nhị phân 32 bit.

➔ 2_{ten}: $0000\ 0000\ 0000\ 0010_{two}$ ⟹ $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

➔ -2_{ten}: $1111\ 1111\ 1111\ 1110_{two}$ ⟹ $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$

Số có dấu và không dấu

Khi làm việc với các lệnh của MIPS, lưu ý:

- Mở rộng có dấu (Sign-extend)
- Mở rộng không dấu (Zero-extend)

Tổng kết:

- Giới thiệu lệnh máy tính, tập lệnh là gì

  (*Tập lệnh được sử dụng cụ thể trong môn học này là MIPS 32 bits)*

- Tập lệnh bao gồm các nhóm lệnh cơ bản: Nhóm lệnh logic, nhóm lệnh số học, nhóm lệnh trao đổi dữ liệu và nhóm lệnh nhảy

- Với MIPS, toán hạng cho các lệnh được chia thành ba nhóm: nhóm toán hạng thanh ghi, nhóm toán hạng bộ nhớ và nhóm toán hạng là số tức thời

- Nhắc lại số có dấu và số không dấu

Kiến trúc bộ lệnh

1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

COMPUTER ENGINEERING

Biểu diễn lệnh

❖ Làm thế nào một lệnh (**add** $t0, $s1, $s2) lưu giữ được trong máy tính?

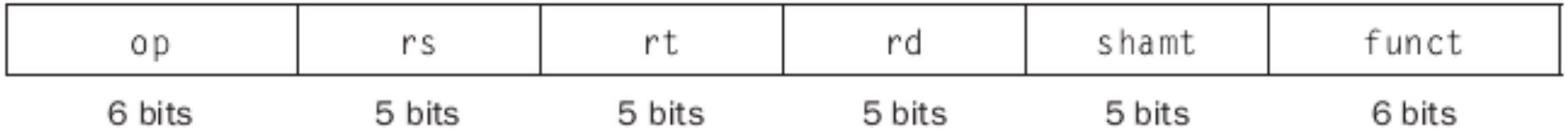
Máy tính chỉ có thể làm việc với các tín hiệu điện tử thấp và cao, do đó một lệnh lưu giữ trong máy tính phải được biểu diễn như là một chuỗi của "0" và "1", được gọi là **mã máy/lệnh máy**.

❖ **Ngôn ngữ máy (Machine language):** biểu diễn nhị phân được sử dụng để giao tiếp trong một hệ thống máy tính.

❖ Để chuyển đổi từ một lệnh sang **mã máy (machine code)** sử dụng **định dạng lệnh (instruction format).**

Định dạng lệnh*: Một hình thức biểu diễn của một lệnh bao gồm các trường của số nhị phân.*

Ví dụ một định dạng lệnh:



Biểu diễn lệnh

- **Ví dụ:** Chuyển đổi một lệnh cộng trong MIPS thành một lệnh máy:

add *$t0,$s1,$s2*

Với định dạng lệnh:

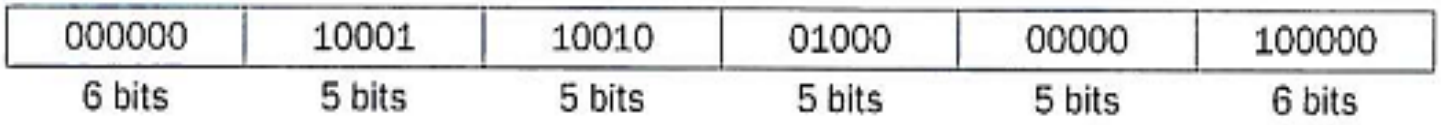
Biểu diễn lệnh

❖ **Trả lời:** Chuyển đổi một lệnh cộng trong MIPS thành một lệnh máy:

add *$t0, $s1, $s2*

Định dạng lệnh:

Mã máy:



- Mỗi phân đoạn của một định dạng lệnh được gọi là một **trường** (ví dụ trường op, rs, rt, rd, shamt, funct).
- Trong ngôn ngữ assembly MIPS, thanh ghi **$s0 đến $s7** có chỉ số tương ứng từ 16 đến 23, và thanh ghi **$t0 đến $t7** có chỉ số tương ứng từ 8 đến 15.
- Các trường rs, rt, rd chứa chỉ số của các thanh ghi tương ứng; trường op và funct có giá trị bao nhiêu cho từng loại lệnh do MIPS quy định

  → Trường 'shamt'?

Tra trong bảng "MIPS reference data" (trang 2 sách tham khảo chính) để có các giá trị cần thiết

Biểu diễn lệnh

❖ Từ một mã máy đang có, như thế nào máy tính hiểu?

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Trường đầu tiên (op, tức opcode có giá trị 0) và trường cuối cùng (funct, tức function có giá trị 20_{hex})** kết hợp báo cho máy tính biết rằng đây là **lệnh cộng (add)**.
- **Trường thứ hai (rs)** cho biết toán hạng thứ nhất của phép toán cộng (rs hiện có giá trị 17, tức toán hạng thứ nhất của phép công là thanh ghi $s1)
- **Trường thứ ba (rt)** cho biết toán hạng thứ hai của phép toán cộng (rt hiện có giá trị 18, tức toán hạng thứ hai của phép công là thanh ghi $s2)
- **Trường thứ tư (rd)** là thanh ghi đích chứa tổng của phép cộng (rd hiện có giá trị 8, tức thanh ghi đích chứa tổng là $t0).
- **Trường thứ năm (shamt)** không sử dụng trong lệnh add này

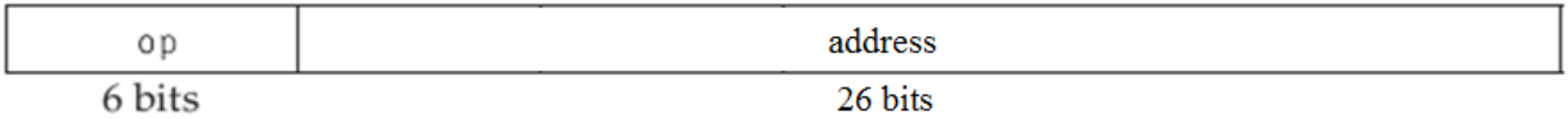
Biểu diễn lệnh

Các dạng khác nhau của định dạng lệnh MIPS :

- **R-type hoặc R-format (cho các lệnh chỉ làm việc với thanh ghi)**

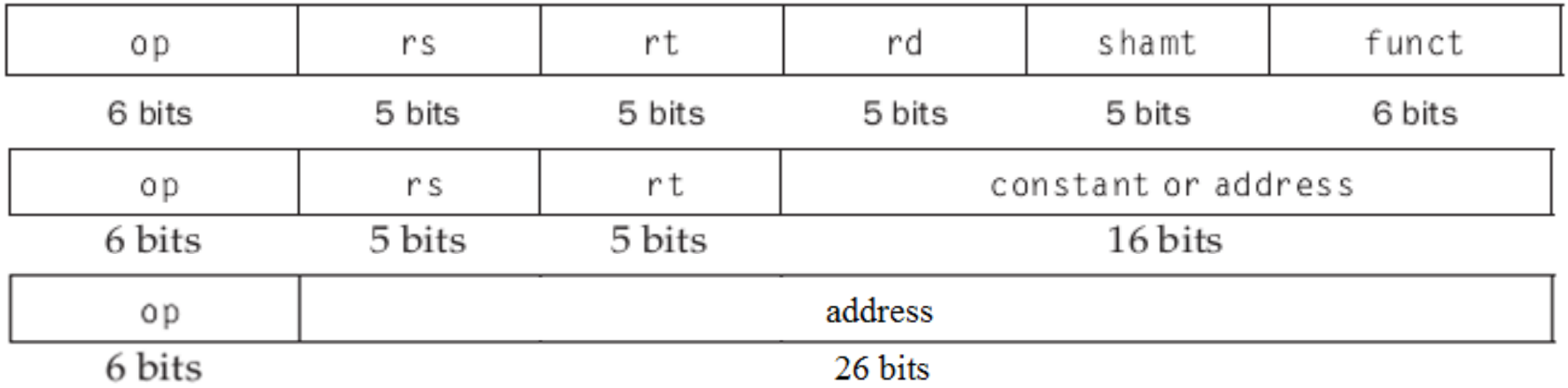
- **I-type hoặc I-format (cho các lệnh có liên quan đến số tức thời và truyền dữ liệu)**

- **J-type hoặc J-format (lệnh nhảy, lệnh ra quyết định)**



Biểu diễn lệnh

Các dạng khác nhau của định dạng lệnh MIPS :



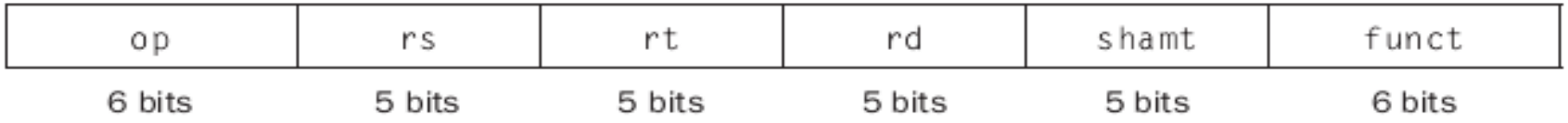
op (Hay còn gọi là opcode, mã tác vụ)**:** Trong cả ba định dạng của lệnh, trường op luôn chiếm 6 bits.

Khi máy tính nhận được mã máy, phân tích op sẽ cho máy tính biết được đây là lệnh gì (*), từ đó cũng biết được mã máy thuộc loại định dạng nào, sau đó các trường tiếp theo sẽ được phân tích.

(*)***Lưu ý:*** *MIPS quy định nhóm các lệnh làm việc với 3 thanh ghi (R-format) đều có op là 0. Vì vậy, với R-format, cần dùng thêm trường 'funct' để biết chính xác lệnh cần thực hiện là lệnh nào.*

Biểu diễn lệnh

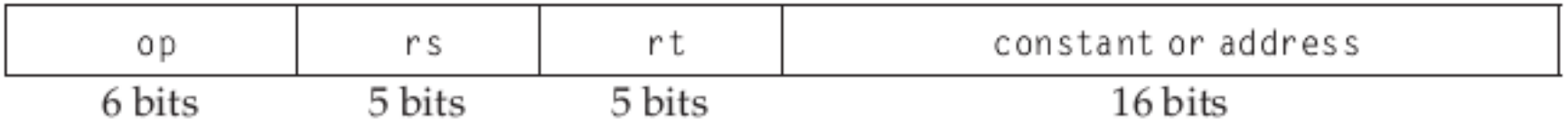
Các trường của R-format:



- **rs:** Thanh ghi chứa toán hạng nguồn thứ nhất
- **rt:** Thanh ghi chứa toán hạng nguồn thứ hai
- **rd:** Thanh ghi toán hạng đích, nhận kết quả của các phép toán.
- **shamt:** Chỉ dùng trong các câu lệnh dịch bit (shift) - chứa số lượng bit cần dịch (không được sử dụng sẽ chứa 0)
- **funct:** Kết hợp với op (khi op bằng 0) để cho biết mã máy là lệnh gì

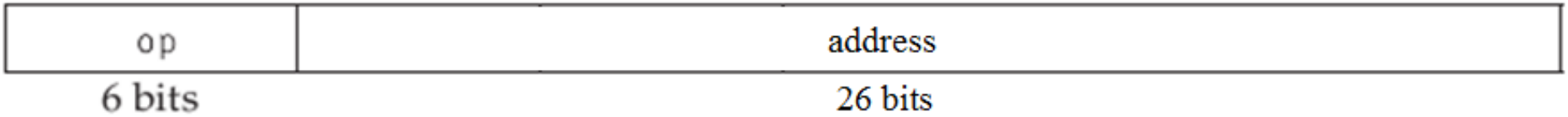
Biểu diễn lệnh

Các trường của I-format và J-format:



Vùng "constant or address" (thỉnh thoảng gọi là vùng immediate) là vùng chứa số16 bit.

- ✓ Với lệnh liên quan đến memory (như lw, sw): giá trị trong thanh ghi rs cộng với số 16 bits này sẽ là địa chỉ của vùng nhớ mà lệnh này truy cập đến.
- ✓ Với lệnh khác (như addi): 16 bits này chứa số tức thời



Vùng "address" là vùng chứa số 26 bit (dùng cho lệnh 'j')

Biểu diễn lệnh

- Ví dụ một số lệnh MIPS và các trường tương ứng

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

  - "reg" nghĩa là chỉ số thanh ghi (giữa 0 và 31)
  - "address" nghĩa là 1 địa chỉ 16 bit.
  - "n.a." (không áp dụng) nghĩa là trường này không xuất hiện trong định dạng này.
  - Lưu ý rằng lệnh 'add' và 'sub' có cùng giá trị trong trường "op"; do đó phần cứng sẽ sử dụng thêm trường "funct" để quyết định đây là lệnh gì
    - $Funct = 32_{ten} = 20_{hex}$ → *lệnh 'add'*
    - $Funct = 34_{ten} = 22_{hex}$ → *lệnh 'sub'*

Biểu diễn lệnh

Ví dụ: Chuyển ngôn ngữ cấp cao → Assembly MIPS → mã máy

Chuyển câu lệnh sau sang assembly MIPS và sau đó chuyển thành mã máy:

$$A[300] = h + A[300]$$

Biết *A* là một mảng nguyên, mỗi phần tử của *A* cần một từ nhớ để lưu trữ; *$t1* chứa địa chỉ nền/cơ sở của mảng *A* và *$s2* tương ứng với biến nguyên *h*.

Đáp án: Assembly MIPS:

```
lw $t0,1200($t1)    # Dùng thanh ghi tạm $t0 nhận A[300]
add $t0,$s2,$t0     # Dùng thanh ghi tạm $t0 nhận h + A[300]
sw $t0,1200($t1)    # Lưu h + A[300] trở lại vào A[300]
```

Mã máy cho ba lệnh trên:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Biểu diễn lệnh

Kết luận:

1. Các lệnh được biểu diễn như là các con số.
2. Chương trình được lưu trữ trong bộ nhớ được đọc hay viết giống như các con số.
   - Xem lệnh như là dữ liệu là cách tốt nhất để đơn giản hóa cả bộ nhớ và phần mềm của máy tính.
   - Để chạy/thực thi một chương trình, đơn giản chỉ cần nạp chương trình và dữ liệu vào bộ nhớ; sau đó báo với máy tính để bắt đầu thực thi chương trình tại vị trí mà nó đã được cấp phát.

Kiến trúc bộ lệnh

1. Giới thiệu
2. Các phép tính
3. Toán hạng
4. Số có dấu và không dấu
5. Biểu diễn lệnh
6. Các phép tính Logic
7. Các lệnh điều kiện và nhảy

Các phép tính Logic

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Hình 7: C và Java các phép tính logic và lệnh MIPS tương ứng.

- **Shift:** Lệnh dịch chuyển bit.
- **AND:** là phép toán logic *"VÀ"*.
- **OR:** là một phép toán logic *"HOẶC"*
- **NOT:** kết quả là 1 nếu bit đó là 0 và ngược lại.
- **NOR**: NOT OR.
- Hằng số rất hữu ích trong các phép toán logic AND và OR cũng như trong phép tính số học, vì vậy MIPS cung cấp các lệnh trực tiếp **andi** và **ori**.

Kiến trúc bộ lệnh

1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

COMPUTER ENGINEERING

Các lệnh điều kiện và nhảy

- Một máy tính (PC) khác với các máy tính tay (calculator) chính là dựa trên khả năng đưa ra quyết định.
- Trong ngôn ngữ lập trình, đưa ra quyết định thường được biểu diễn bằng cách sử dụng câu lệnh *"if"*, đôi khi kết hợp với câu lệnh *"go to"*.
- Ngôn ngữ Assembly MIPS cũng chứa các lệnh hỗ trợ ra quyết định, tương tự với câu lệnh *"if"* và *"go to"*.

  Ví dụ: ***beq*** *register1, register2, L1*

  Lệnh này có nghĩa là đi đến câu lệnh có nhãn *L1* nếu giá trị của thanh ghi *register1* bằng giá trị thanh ghi *register2*.

  Từ *'beq'* là viết tắt của "branch if equal" (rẽ nhánh nếu bằng)

➔ Các lệnh như *'beq'* được gọi là **lệnh rẽ nhánh có điều kiện**.

Các lệnh điều kiện và nhảy

Các lệnh rẽ nhánh có điều kiện (conditional branch) của MIPS:

Conditional branch	branch on equal	beq $s1, $s2 25	if ($s1 == $s2) goto PC + 4 + 100
	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) goto PC + 4 + 100
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0
	set on less than unsigned	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0
	set on less than immediate	slt $s1, $s2, 20	if ($s2 < 20) $s1 = 1; else $s1 = 0
	set on less than immediate unsigned	slt $s1, $s2, 20	if ($s2 < 20) $s1 = 1; else $s1 = 0

Các lệnh điều kiện và nhảy

Ngoài ra còn có các lệnh rẽ nhánh có điều kiện khác, nhưng là nhóm **lệnh giả (pseudo instructions)**

Conditional branch (pseudo instruction)	branch on less than	blt
	branch greater than	bgt
	branch less than or equal	ble
	branch greater than or equal	bge

(Tham khảo trang số 2, sách tham khảo chính)

Các lệnh điều kiện và nhảy

Cặp (slt → beq) tương đương với if(… ≥ …) goto…

Cặp (slt → bne) tương đương với if(… < …) goto…

```
slt $t0,$s0,$s1          # $t0 = 1 if $s0 <$s1
beq $t0,$zero,skip             # if $s0 >= $s1, goto
skip
<stuff>                  # do if $s0 < $s1
```

```
slt $t0,$s0,$s1          # $t0 = 1 if $s0 <$s1
bne $t0,$zero,skip             # if $s0 < $s1, goto skip
<stuff>                  # do if $s0 >= $s1
```

Các lệnh điều kiện và nhảy

Các lệnh rẽ nhánh không điều kiện (unconditional branch) của MIPS:

Unconditional jump	jump	j label
	jump register	jr $ra
	jump and link	jal label

COMPUTER ENGINEERING

Các lệnh điều kiện và nhảy

❖ Biên dịch ***if-then-else*** từ ngôn ngữ cấp cao sang assembly MIPS:

Cho đoạn mã sau:

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

Biết *f, g, h, i* và *j* là các biến. Nếu năm biến *f* đến *j* tương ứng với 5 thanh ghi $s0 đến $s4, mã MIPS cho câu lệnh *if* này là gì?

Trả lời:

```
        bne $s3,$s4,Else      # go to Else if i != j
        add $s0, $s1, $s2     # f = g + h (skipped if i != j)
        j exit                # go to Exit
Else:   sub  $s0, $s1, $s2    # f = g – h (skipped if i = j)
exit:
```

COMPUTER ENGINEERING

Các lệnh điều kiện và nhảy

❖ Biên dịch 1 vòng lặp ***while*** từ ngôn ngữ cấp cao sang assembly MIPS

Cho đoạn mã sau:

while (save[i] == k)
i += 1;

Giả định rằng *i* và *k* tương ứng với thanh ghi *$s3* và *$s5;* và địa chỉ nền/cơ sở của mảng ***save*** lưu trong *$s6*. Mã assembly MIPS tương ứng với đoạn mã C trên là gì?

❖ **Trả lời**:

```
Loop:   sll $t1,$s3,2      # Temp reg $t1 = 4 * i
        add $t1,$t1,$s6    # $t1 = address of save[i]
        lw $t0,0($t1)      # Temp reg $t0 = save[i]
        bne $t0,$s5, Exit  # go to Exit if save[i] != k
        addi $s3,$s3,1     # i = i + 1
        j Loop             # go to Loop
Exit:
```

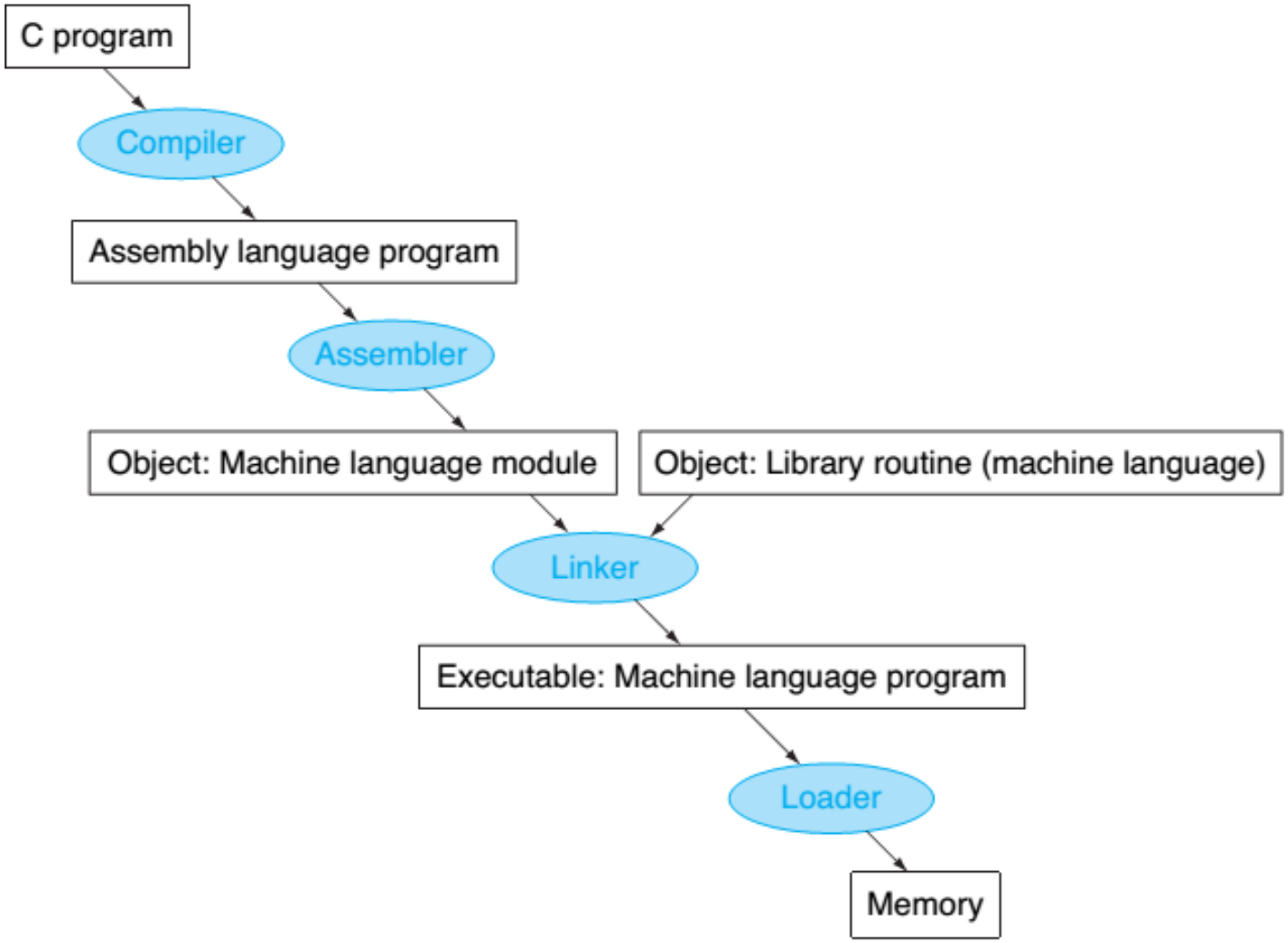
COMPUTER ENGINEERING

Kiến trúc bộ lệnh

1. **Giới thiệu**
2. **Các phép tính**
3. **Toán hạng**
4. **Số có dấu và không dấu**
5. **Biểu diễn lệnh**
6. **Các phép tính Logic**
7. **Các lệnh điều kiện và nhảy**

Chuyển đổi và bắt đầu một chương trình

Bốn bước trong việc chuyển đổi một chương trình C trong một tập tin trên đĩa vào một chương trình đang chạy trên máy tính.



Kiến trúc bộ lệnh

Tổng kết:

- MIPS có ba định dạng lệnh: R-format, I-format, J-format. Từ đó, hiểu cách một lệnh từ ngôn ngữ cấp cao chuyển thành assembly của MIPS, và từ assembly của MIPS chuyển thành mã máy dựa theo ba định dạng trên
- Biết quy tắc hoạt động của nhóm lệnh logic của MIPS
- Biết quy tắc hoạt động của nhóm lệnh nhảy (nhảy có điều kiện và không điều kiện) của MIPS

Thủ tục (Procedure) cho assembly MIPS

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖ Một thủ tục (procedure) hay một hàm (function) là một công cụ mà lập trình viên sử dụng để xây dựng cấu trúc của những chương trình, với mục đích vừa làm cho các chương trình đó dễ hiểu hơn vừa làm cho mã nguồn của các chương trình này có thể được tái sử dụng.

- ✓ Một chương trình có nhiều chức năng, mỗi chức năng sẽ được đưa vào một hàm, hoặc một thủ tục
- ✓ Các thủ tục hoặc hàm con này cho phép lập trình viên tại một thời điểm chỉ cần tập trung vào một phần của công việc, dễ dàng quản lý việc lập trình hơn

❖ Assembly cũng giống như các ngôn ngữ cấp cao, một chương trình với nhiều chức năng thì mỗi chức năng có thể đưa vào một thủ tục khác nhau.

Chú ý: Các thuật ngữ Routine/Procedure/Function có thể gặp trong một số môi trường khác nhau; trong assembly và phạm vi môn học này, tất cả đều được dịch là hàm hoặc thủ tục

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

- Để thực thi một thủ tục, chương trình phải tuân theo sáu bước sau:

1. *Đặt các tham số ở một nơi mà thủ tục có thể truy xuất được.*
2. *Chuyển quyền điều khiển cho thủ tục.*
3. *Yêu cầu tài nguyên lưu trữ cần thiết cho thủ tục đó.*
4. *Thực hiện công việc (task).*
5. *Lưu kết quả ở một nơi mà chương trình có thể truy xuất được.*
6. *Trả điều khiển về vị trị mà thủ tục được gọi. Vì một thủ tục có thể được gọi từ nhiều vị trí trong một chương trình.*

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

Một số quy ước của MIPS với thanh ghi cần lưu ý:

- Thanh ghi $at ($1), $k0 ($26), $k1 ($27) được dành cho hệ điều hành và assembler; không nên sử dụng trong lúc lập trình thông thường.
- Thanh ghi $a0-$a3 ($4-$7) được sử dụng để truyền bốn tham số đến một *thủ tục* (nếu có hơn 4 tham số truyền vào, các tham số còn lại được lưu vào stack). Thanh ghi $v0-$v1 ($2-$3) được sử dụng để lưu giá trị trả về của hàm.
- Các thanh ghi $t0-$t9 ($9-$15, 24, 25) được sử dụng như các thanh ghi tạm. Các thanh ghi $s0-$s7 ($16-$23) được sử dụng như các thanh ghi lưu giá trị bền vững.

Trong MIPS, việc tính toán có thể cần một số thanh ghi trung gian, tạm thời, các thanh ghi $t nên được dùng cho mục đích này (nên thanh ghi nhóm $t thường gọi là thanh ghi tạm); còn kết quả cuối của phép toán nên lưu vào các thanh ghi $s (nên thanh ghi nhóm $s thường được gọi là nhóm thanh ghi lưu giá trị bền vững)

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

Một số quy ước của MIPS với thanh ghi cần lưu ý:

- Theo quy ước của MIPS, một thủ tục nếu sử dụng bất kỳ thanh ghi loại $s nào sẽ phải lưu lại giá trị của thanh ghi $s đó trước khi thực thi hàm. Và trước khi thoát ra khỏi hàm, các giá trị cũ của các thanh ghi $s này cũng phải được trả về lại. Các thanh ghi $s này được xem như biến cục bộ của hàm.

  Thanh ghi loại $t được xem là các thanh ghi tạm, nên việc sử dụng thanh ghi loại $t trong thủ tục không cần lưu lại như các thanh ghi loại $s

Vì thanh ghi là loại bộ nhớ có tốc độ truy xuất nhanh nhất, được dùng để lưu trữ dữ liệu trong một máy tính và số thanh ghi là hạn chế nên lập trình hợp ngữ phải hướng đến tận dụng thanh ghi một cách tối đa nhất.

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Các khái niệm và tên gọi:

- Địa chỉ trả về (**return address**): là một liên kết tới vùng đang gọi cho phép một thủ tục trả về đúng địa chỉ; trong MIPS, nó được lưu trữ ở thanh ghi $ra.

  Cụ thể: khi chương trình đang thực thi và một thủ tục được gọi, sau khi thủ tục thực thi xong, luồng thực thi lệnh phải quay về lại chương trình và thực hiện tiếp lệnh ngay phía sau thủ tục được gọi. Địa chỉ lệnh được thực thi sau khi thủ tục hoàn thành chính là địa chỉ trả về.

- **Caller:** Là chương trình gọi một thủ tục và cung cấp những giá trị tham số cần thiết.
- **Callee**: Là một thủ tục thực thi một chuỗi những lệnh được lưu trữ dựa trên những tham số được cung cấp bởi caller và sau đó trả điều khiều về cho caller.

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Các khái niệm và tên gọi:

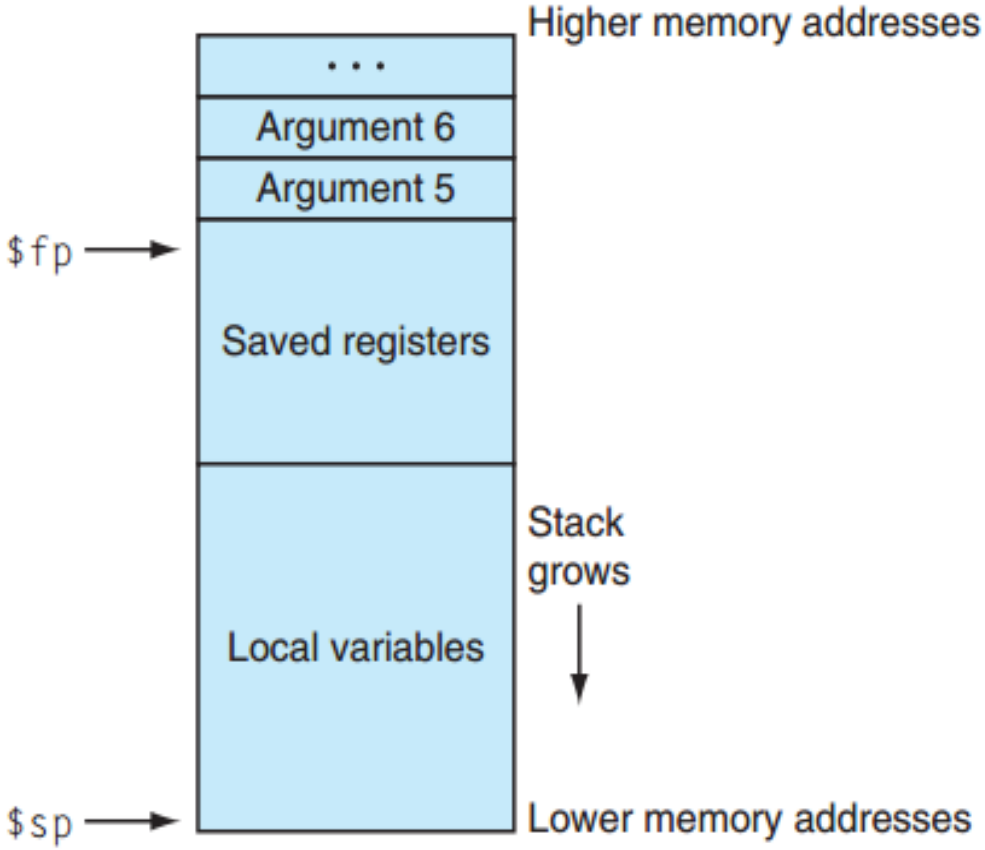
- **Program counter (PC):** Là thanh ghi chứa địa chỉ của lệnh kế tiếp được thực thi trong chương trình.
  - ✓ Thanh ghi PC còn được gọi là con trỏ PC hay con trỏ lệnh
  - ✓ Trong MIPS 32 bits, mỗi lệnh được lưu trong một word 4 bytes, do đó, để chỉ tới lệnh kế tiếp được thực thi trong chương trình, PC tăng lên 4 (ngoài trừ lệnh nhảy)

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Các khái niệm và tên gọi:

- Stack (ngăn xếp): Là một cấu trúc dữ liệu trong bộ nhớ cho việc lưu dữ liệu theo hàng đợi dạng vào-sau ra-trước (last-in first-out)

Mỗi thủ tục luôn cần một vùng nhớ đi cùng với nó (vì mỗi thủ tục đều cần không gian để lưu lại các biến cục bộ của nó (là các thanh ghi $s được sử dụng trong thân hàm hoặc các biến cục bộ khác), hoặc để chứa các tham số input nếu số tham số lớn hơn 4 hoặc chứa giá trị trả về output nếu số giá trị trả về lớn hơn 2). Vùng nhớ này được quy ước hoạt động như một stack.

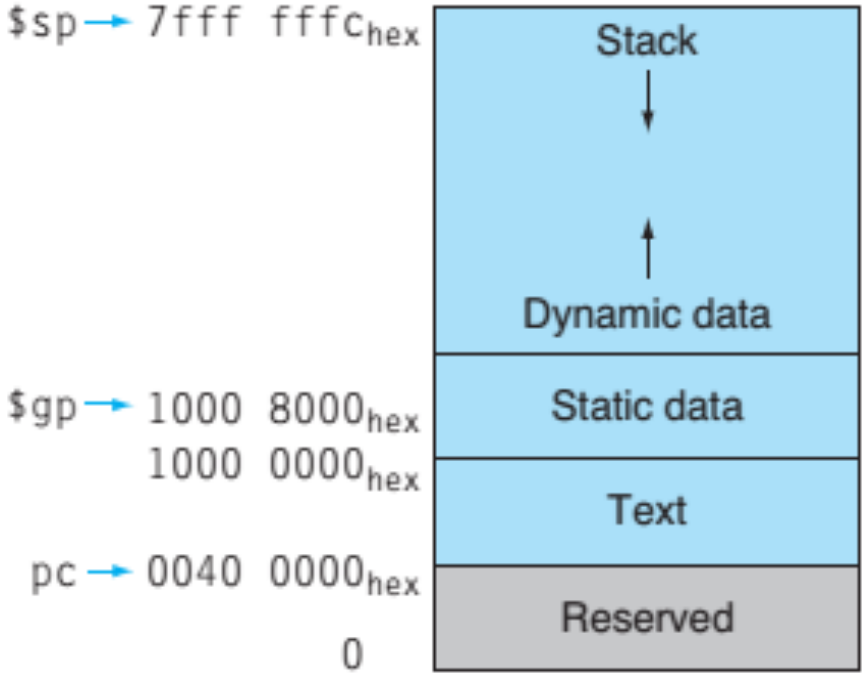


Hình ảnh ví dụ của một stack với các frame (stack này phục vụ cho thủ tục mà bốn tham số đầu vào đã được truyền thông qua thanh ghi, tham số thứ 5 và 6 được lưu trong stack như hình)

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

Xem thêm về cách bố trí vùng **Stack, Heap, Data** và **Text Segment** trong bộ nhớ của một chương trình

- **Heap:** Là khu vực bộ nhớ cấp phát động, có thể được cấp phát thêm khi đạt đến giới hạn.
- **Text segment:** Đoạn mã chương trình.



Cấp phát bộ nhớ cho chương trình và dữ liệu kiến trúc MIPS

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Các khái niệm và tên gọi:

- **Stack pointer** (SP): Địa chỉ của phần tử được cấp phát gần nhất (tức phần tử được cấp phát cuối cùng cho đến thời điểm hiện tại). Trong MIPS, giá trị này lưu trong thanh ghi $sp (29).
- Trong khi đó, thanh ghi $fp ($30) dùng làm con trỏ frame (**Frame pointer**).

  Stack gồm nhiều frame; frame cuối cùng trong stack được trỏ tới bởi $sp, frame đầu tiên (cho vùng lưu các thanh ghi cũng như biến cục bộ) được trỏ tới bởi $fp (xem hình slide trước)
- Stack được xây dựng theo kiểu **từ địa chỉ cao giảm dần xuống thấp**, vì thế frame pointer luôn ở trên stack pointer.

  Đây là quy ước của MIPS. Tất nhiên stack có thể được xây dựng theo cách khác, tùy vào mỗi môi trường làm việc mà ta tuân thủ các quy ước.

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Các khái niệm và tên gọi:

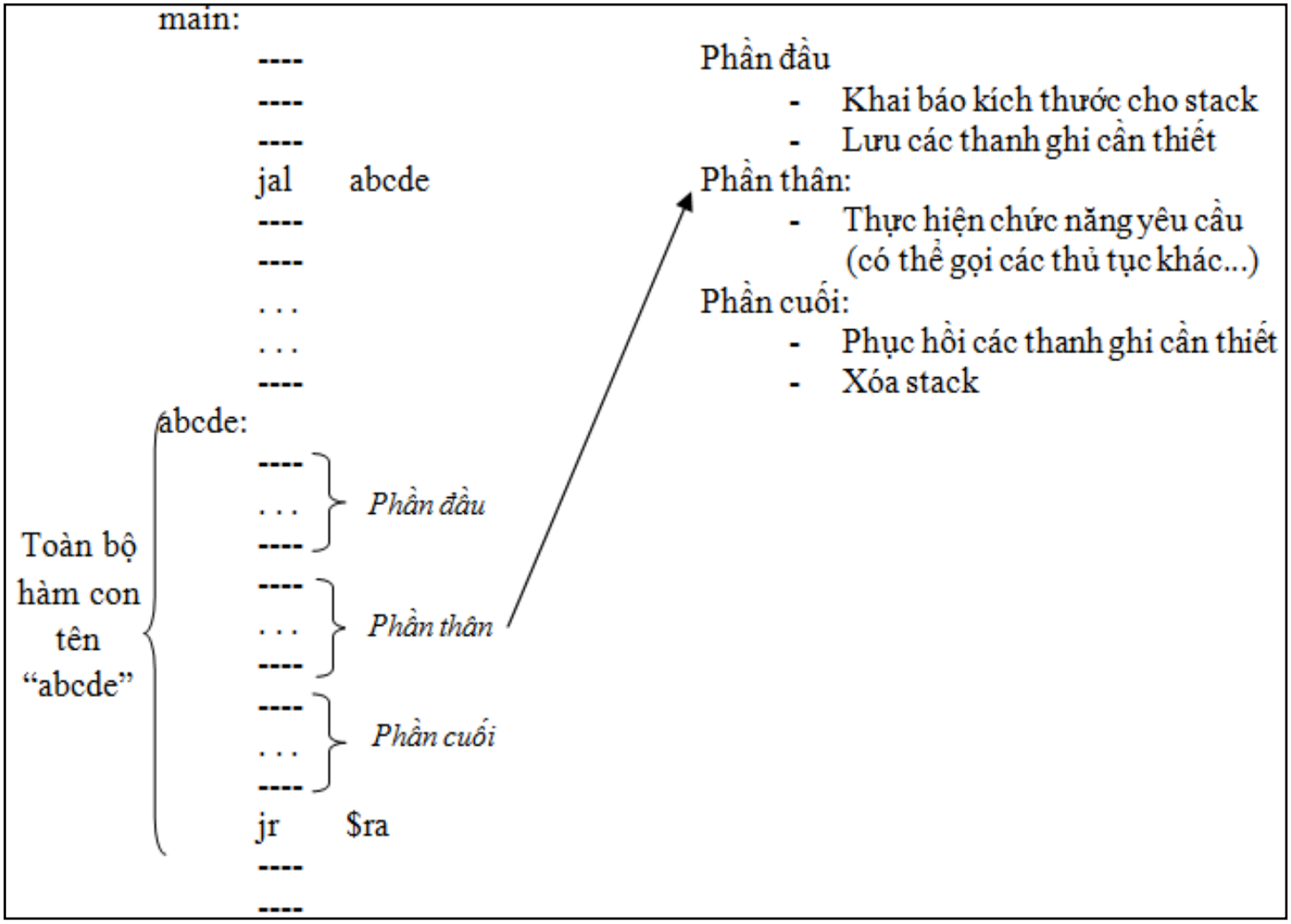
- **Push**: Việc lưu một phần tử vào stack gọi là Push
- **Pop**: Việc lấy một phần tử ra khỏi stack gọi là Pop

Trong tập lệnh của MIPS chuẩn, không có lệnh Pop và Push. Việc thực hiện này phải do người lập trình thực hiện:

- ✓ Theo quy ước của MIPS, khi PUSH một phần tử vào stack, đầu tiên $sp nên trừ đi 4 byte (1 word) rồi sau đó lưu dữ liệu vào word nhớ có địa chỉ đang chứa trong $sp.
- ✓ Ngược lại, khi POP một phần tử từ stack, dữ liệu được load ra từ word nhớ có địa chỉ đang chứa trong $sp rồi sau đó $sp được cộng thêm 4

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Cấu trúc một thủ tục/hàm con



Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Cấu trúc một thủ tục/hàm con

Trong chương trình chính, khi thủ tục được gọi, con trỏ PC sẽ chuyển quyền điều khiển xuống vị trí của thủ tục; sau khi thủ tục được thực hiện xong, con trỏ PC sẽ chuyển về thực hiện lệnh ngay sau lệnh gọi thủ tục.

Việc này được thực hiện nhờ vào cặp lệnh:

jal ten_thu_tuc

jr $ra

(ten_thu_tuc là nhãn của lệnh đầu tiên trong thủ tục, cũng được xem là tên của thủ tục)

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Cấu trúc một thủ tục/hàm con

- Để gọi thủ tục, dùng lệnh ***jal*** *(Lệnh nhảy-và-liên kết: jump-and-link)*
  - ✓ Đầu tiên, địa chỉ của lệnh ngay sau lệnh *jal* phải được lưu lại để sau khi thủ tục thực hiện xong, con trỏ PC có thể chuyển về lại đây (thanh ghi dùng để lưu địa chỉ trả về là *$ra*); sau khi địa chỉ trả về đã được lưu lại, con trỏ PC chuyển đến địa chỉ của lệnh đầu tiên của thủ tục để thực hiện.
  - ✓ Vậy có hai công việc được thực hiện trong ***jal*** theo thứ tự:

```
jal <address>          #   $ra = PC + 4
                       #    PC = <address>
```

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Cấu trúc một thủ tục/hàm con

- Sau khi thủ tục thực hiện xong, để trả về lại chương trình chính, dùng lệnh ***jr*** *(jump regiser)*
  - ✓ Lệnh "jr $ra" đặt ở cuối thủ tục thực hiện việc lấy giá trị đang chứa trong thanh ghi $ra gán vào PC, giúp thanh ghi quay trở về thực hiện lệnh ngay sau lệnh gọi thủ tục này.

```
jr    $ra          #    PC = $ra
```

❖Cấu trúc một thủ tục/hàm con

Như vậy, mỗi hàm con/thủ tục được chia thành ba phần. Phần thân dùng để tính toán chức năng của thủ tục này, bắt buộc phải có. Phần đầu và phần cuối có thể có hoặc không, tùy từng yêu cầu của chương trình con.

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

❖Cấu trúc một thủ tục/hàm con

Ví dụ 1: Một thủ tục chỉ có thân hàm (không cần phần đầu và cuối)

Đoạn code sau thực hiện việc gọi hàm con (thủ tục) tên ABC

```
        add $t0, $t1, $t2
        jal ABC
        or $t3, $t4, $s5
        j EXIT
ABC:
        add $t0, $t3, $t5
        sub $t2, $t3, $t0
        jr $ra
EXIT:
```

❖Cấu trúc một thủ tục/hàm con

Ví dụ 2: Một thủ tục chỉ có đủ ba phần: thân hàm, phần đầu và cuối

Viết đoạn code thực hiện

- **Chương trình chính có:**
  - ✓ Bốn giá trị a, b, c, d lần lượt chứa trong $s0, $1, $s2, $s3
  - ✓ Chương trình truyền bốn tham số này vào thủ tục tên "proc_example" để lấy kết quả của phép toán trên. Sau đó đưa kết quả vào thanh ghi $s6
- **Thủ tục proc_example có:**
  - ✓ Bốn input (a, b, c, d)
  - ✓ Một output, là kết quả của phép toán: (a + b) – (c + d)
  - ✓ Quá trình tính toán sử dụng 2 thanh ghi tạm $t1, $t2 và một thanh ghi $s0

```
        add $a0, $s0, $zero
        add $a1, $s1, $zero
        add $a2, $s2, $zero
        add $a3, $s3, $zero
        jal proc_example

        add $s6, $v0, $zero
        ......
proc_example:
        addi $sp, $sp, -4
        sw $s0, 0($sp)
        add $t0, $a0, $a1
        add $t1, $a2, $a3
        sub $s0, $t0, $t1
        add $v0, $s0, $zero
        lw $s0, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

Bốn tham số truyền cho thủ tục proc_example trước khi gọi hàm. (Theo quy ước, tham số truyền cho thủ tục phải truyền vào các thanh ghi $a0-$a3)

Sau khi hàm proc_example thực hiện xong, giá trị trả về của hàm theo quy ước chứa trong $v0. Lệnh này chuyển dữ liệu từ $v0 vào thanh ghi $s6

Phần đầu: lưu dữ liệu vào stack cho thủ tục. Do trong phần thân hàm có sử dụng thanh ghi $s0 nên trước khi tính toán, giá trị cũ của $s0 nên được lưu trong stack. Nếu có n thanh ghi cần lưu, mỗi thanh ghi một word, ta cần 4n byte cho stack. Trong trường hợp này, ta chỉ cần 4 byte. Vì stack phát triển theo kiểu từ cao xuống thấp, nên cấp phát 1 frame (1 word cho stack) thì $sp trừ 4. Hai lệnh này còn tương ứng với PUSH một phần tử vào stack

Phần thân: Sau khi tính toán, kết quả trả về phải lưu vào $v0 như quy ước

Phần cuối: Trước khi ra khỏi hàm, giá trị đang lưu của $s0 phải phục hồi lại; và 1 frame của stack phải được xóa đi bằng cách lấy $sp + 4. Hai lệnh này tương ứng với POP một phần tử ra khỏi stack

❖Cấu trúc một thủ tục/hàm con

Trong ví dụ 1, thân thủ tục không sử dụng bất kỳ thanh ghi $s nào, nên không cần khởi tạo stack để lưu các thanh ghi này lại. Trong ví dụ 2, thân thủ tục có sử dụng thanh ghi $s, nên stack phải được khởi tạo để lưu; dẫn đến trước khi kết thúc hàm phải giá trị đã lưu phải trả về lại các thanh ghi và xóa stack.

(Lưu ý: Code trong ví dụ 2 có thể viết lại mà không cần dùng thanh ghi $s0; nhưng ví dụ này cố tình dùng $s0 để thấy được việc lưu trên stack phải thực hiện như thế nào)

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

Tóm lại:

- Với một thủ tục/hàm con:
  - ✓ Input: $a0, $a1, $a2, $a3
  - ✓ Output: $v0, $v1
  - ✓ Địa chỉ quay về được lưu trong $ra
- Stack được sử dụng trong các trường hợp sau
  - ✓ Nếu thủ tục (hàm con) cần nhiều hơn bốn thanh ghi để làm tham số hoặc nhiều hơn hai thanh ghi làm giá trị trả về thì không gian stack sẽ được dùng hỗ trợ trong trường hợp này.
  - ✓ Nếu thủ tục có sử dụng một số thanh ghi loại $s, thì theo quy ước của MIPS, trước khi thực hiện thủ tục, giá trị của các thanh ghi này phải được lưu trữ vào stack và được hồi phục lại khi kết thúc thủ tục *(Các thanh ghi $t không cần lưu)*.

Các Thủ Tục Hỗ Trợ Trong Phần Cứng Máy Tính

Nested procedure:

- Các thủ tục có thể gọi lồng vào nhau.
- Các thủ tục mà không gọi các thủ tục khác là các thủ tục lá (leaf procedures). Ngược lại là nested procedures.
- Chúng ta cần cẩn thận khi sử dụng các thanh ghi trong các thủ tục, càng cần phải cẩn thận hơn khi gọi một nested procedure.

Tổng kết:

- Tuần 3 và 4 đã trình bày các nội dung liên quan đến hợp ngữ của MIPS, biết như thế nào để lập trình một chương trình dùng assembly.
- Với một chương trình assembly dài, có nhiều chức năng, như lập trình ngôn ngữ cấp cao, mỗi chức năng có thể đưa vào một thủ tục (procedure) (như các hàm hoặc hàm con trong ngôn ngữ cấp cao)
- Khi lập trình với procedure cho MIPS, các điểm cần chú ý: stack, $sp, $fp và cách truyền tham số.