Kiến trúc máy tính
Computer architecture

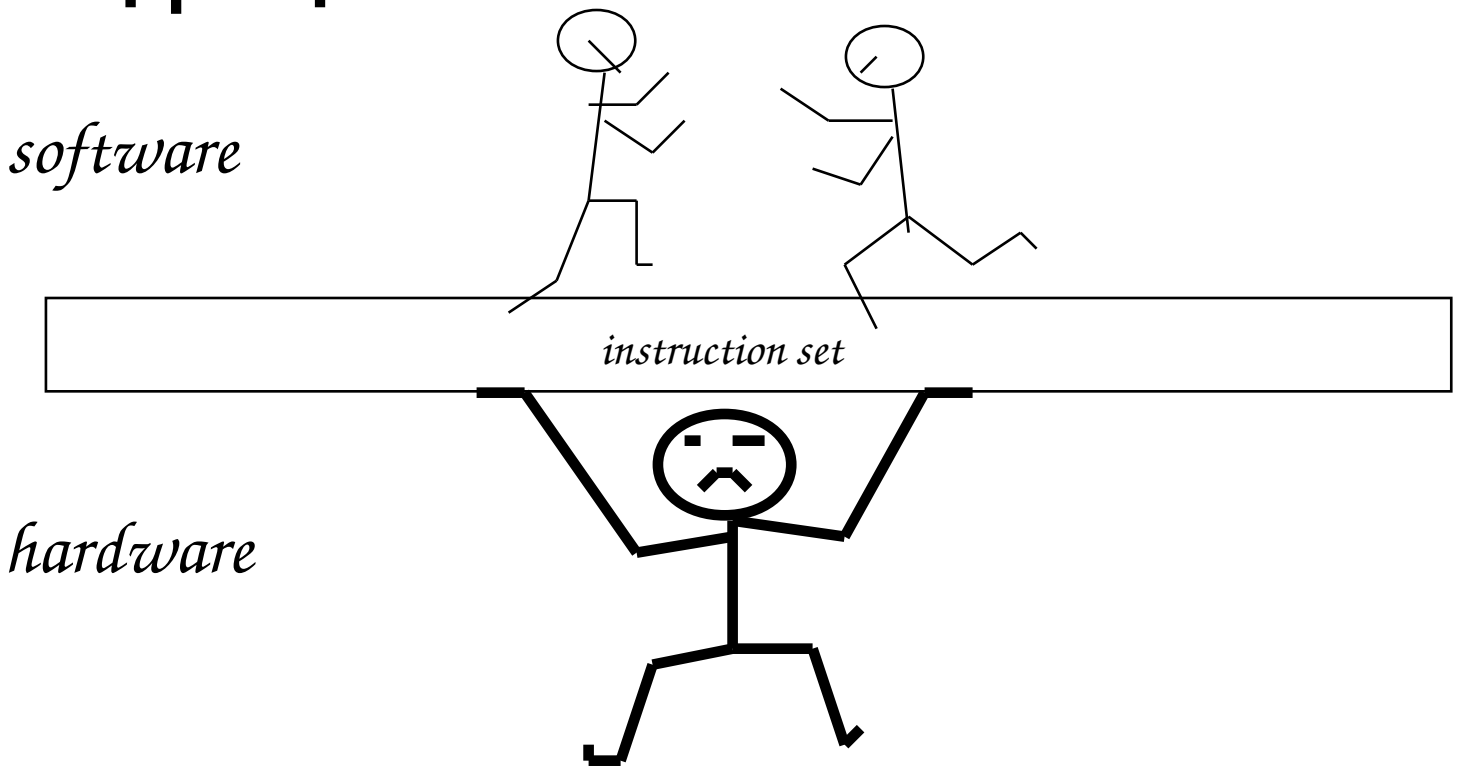
To improve is to change; to be perfect is to change often.

Winston Churchill

Chương 2: Kiến trúc tập lệnh

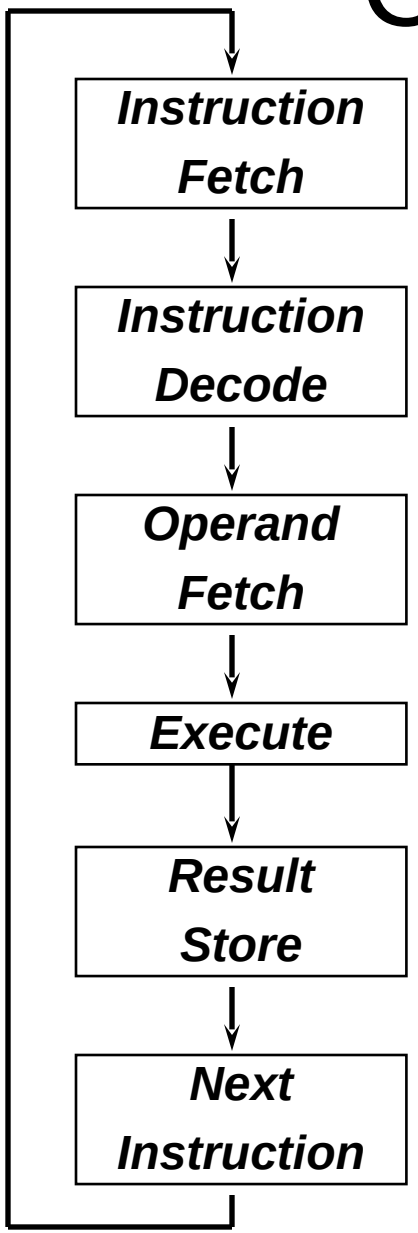
1. Tập lệnh MIPS
2. Biên dịch mã máy

Kiến trúc tập lệnh



- Multiple Implementations: 8086 → Pentium 4
- ISAs evolve: MIPS I

Chu kỳ thực thi lệnh cơ bản



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in register or storage for later use

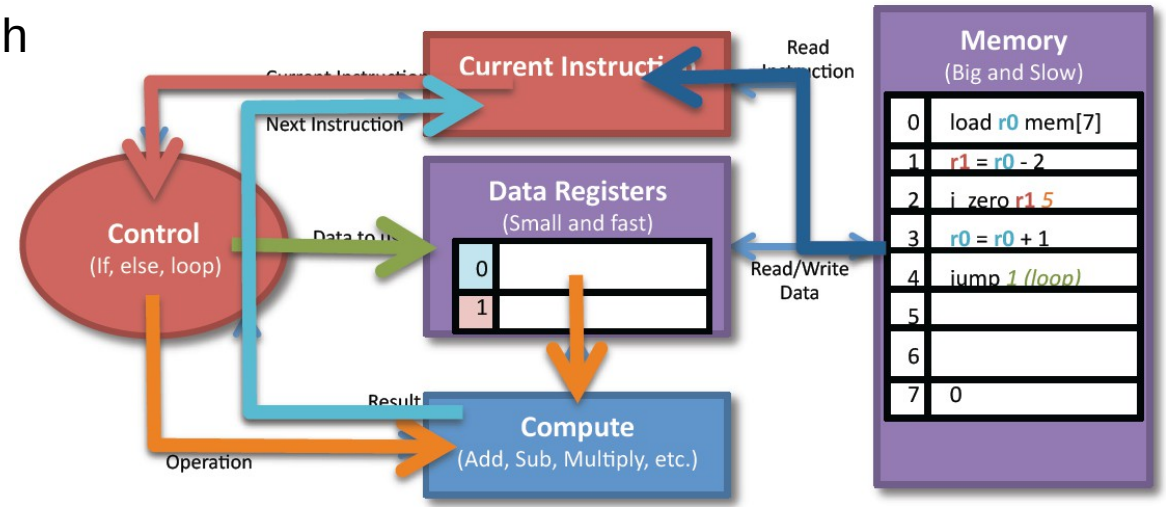
Determine successor instruction

Thực thi chương trình

Bộ xử lý thực thi chương trình như thế nào?

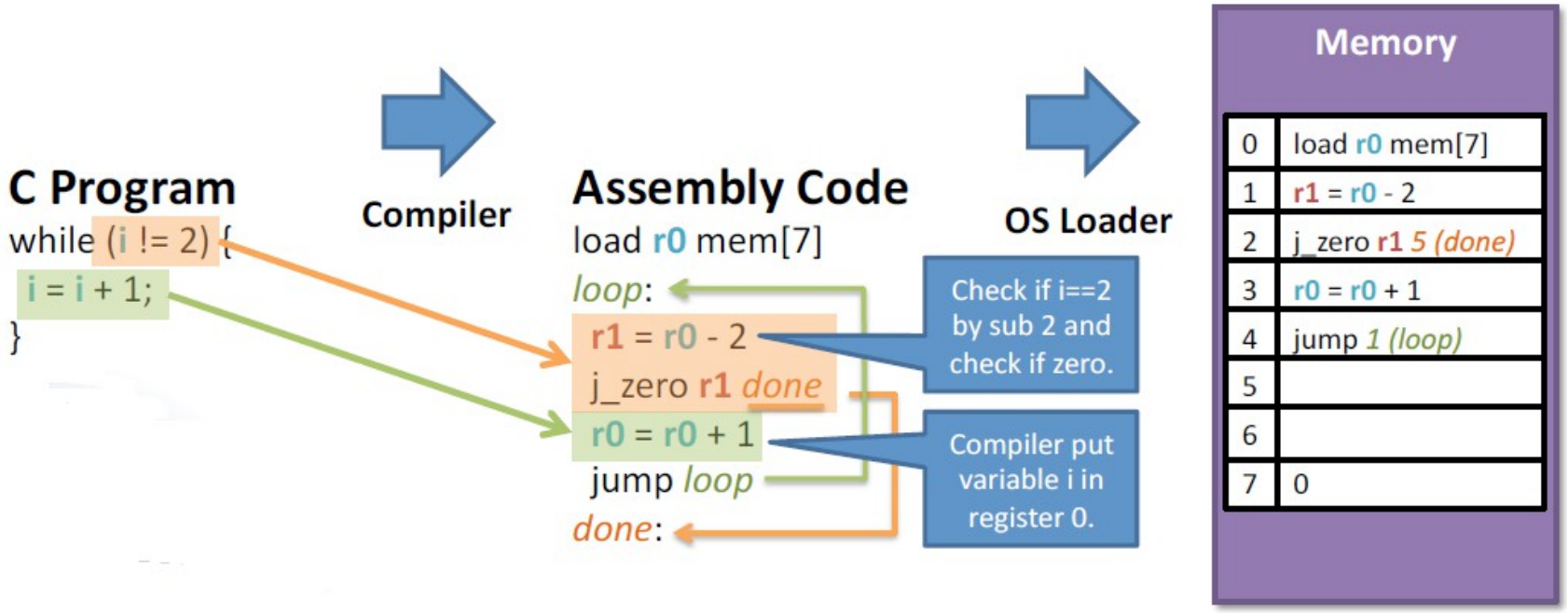
1. Tải lệnh
2. Tìm ra toán tử được sử dụng
3. Tìm ra dữ liệu nào được sử dụng
4. Thực hiện tính toán
5. Tìm lệnh tiếp theo

Lặp lại quá trình …



Bài giảng nhấn mạnh sự thực thi trong bộ xử lý MIPS

Thực thi lệnh



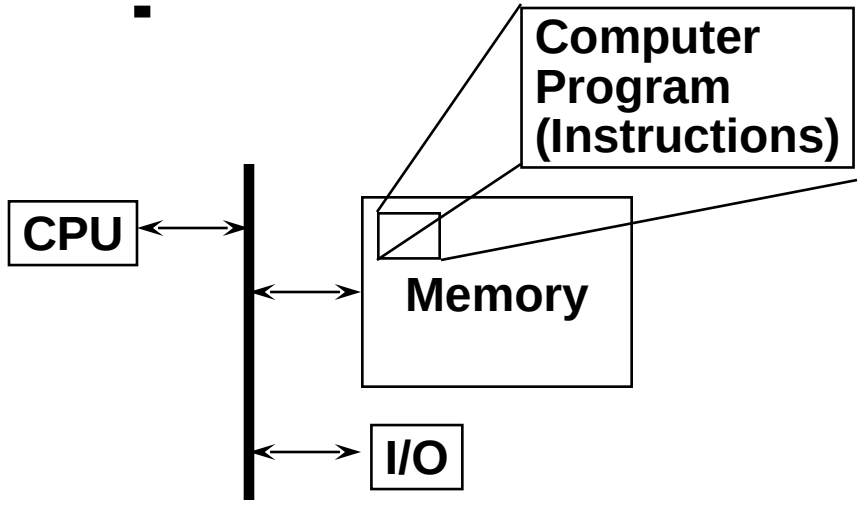
Today we're going to learn the details!

Thực thi lệnh

Programmer's View

ADD	01010
SUBTRACT	01110
AND	10011
OR	10001
COMPARE	11010
.	.
.	.
.	.

Computer's View



Kiến trúc Princeton (Von Neumann)

--- Data and Instructions mixed in same unified memory

--- Program as data

--- Storage utilization

--- Single memory interface

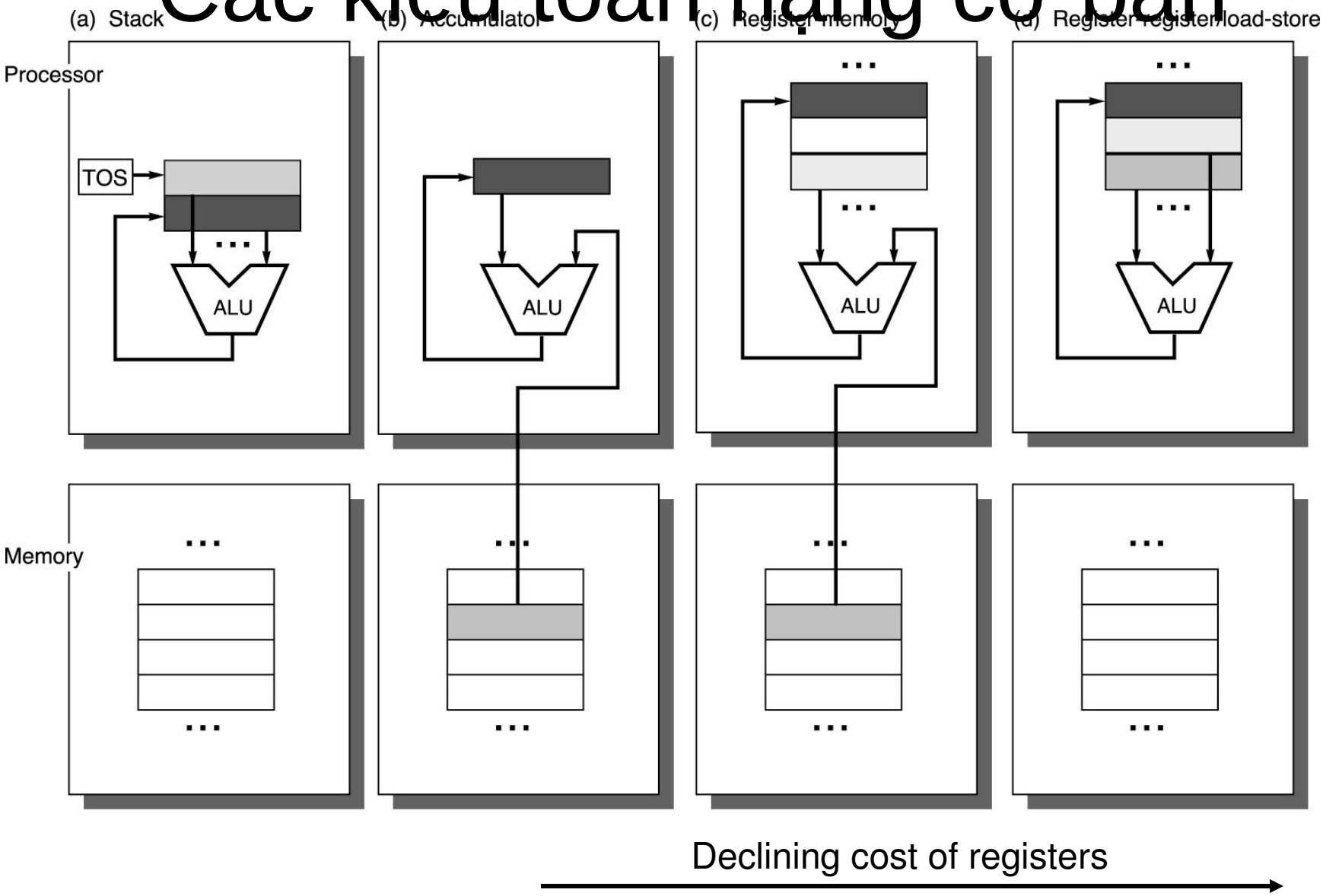
Kiến trúc Harvard

--- Data & Instructions in separate memories

--- Has advantages in certain high performance implementations

--- Can optimize each memory

Các kiểu toán hạng cơ bản



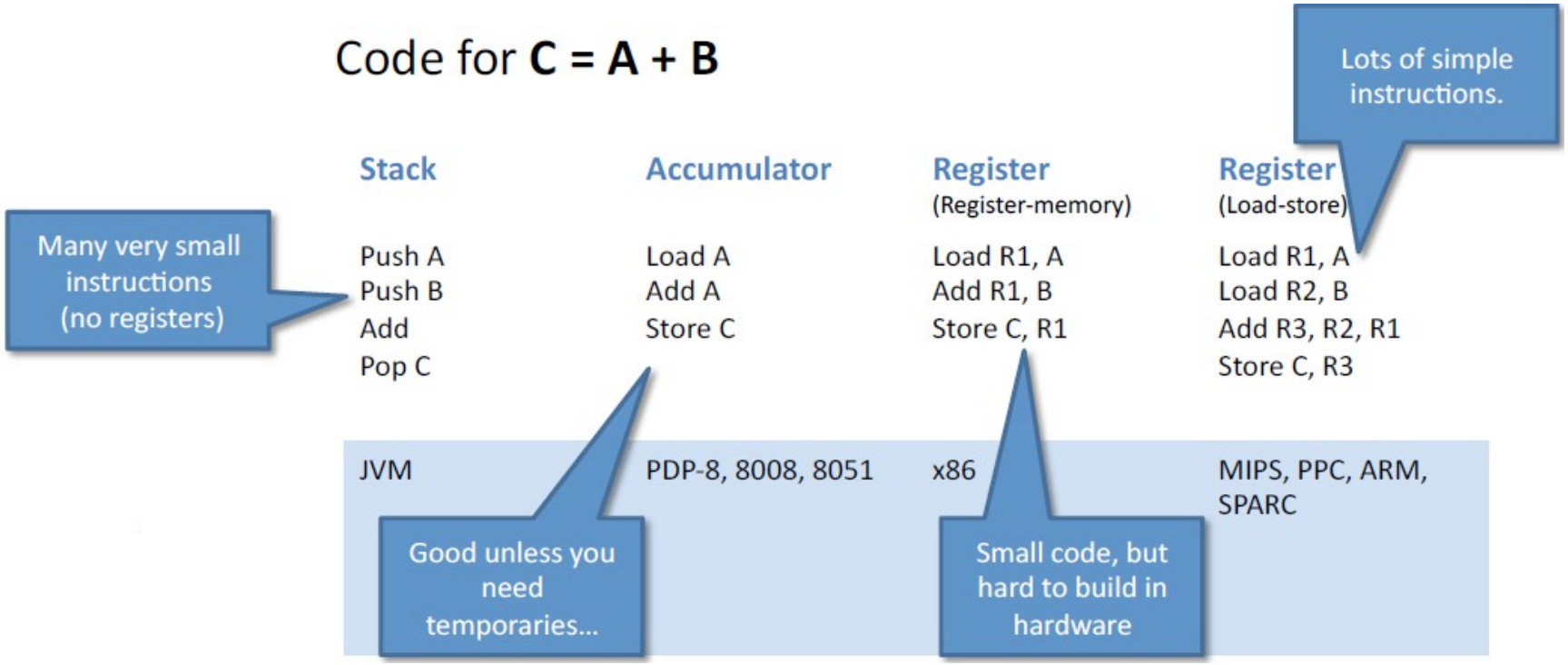
So sánh số lượng toán hạng

Thực thi phép toán (C = A + B) với các kiểu toán hạng khác nhau

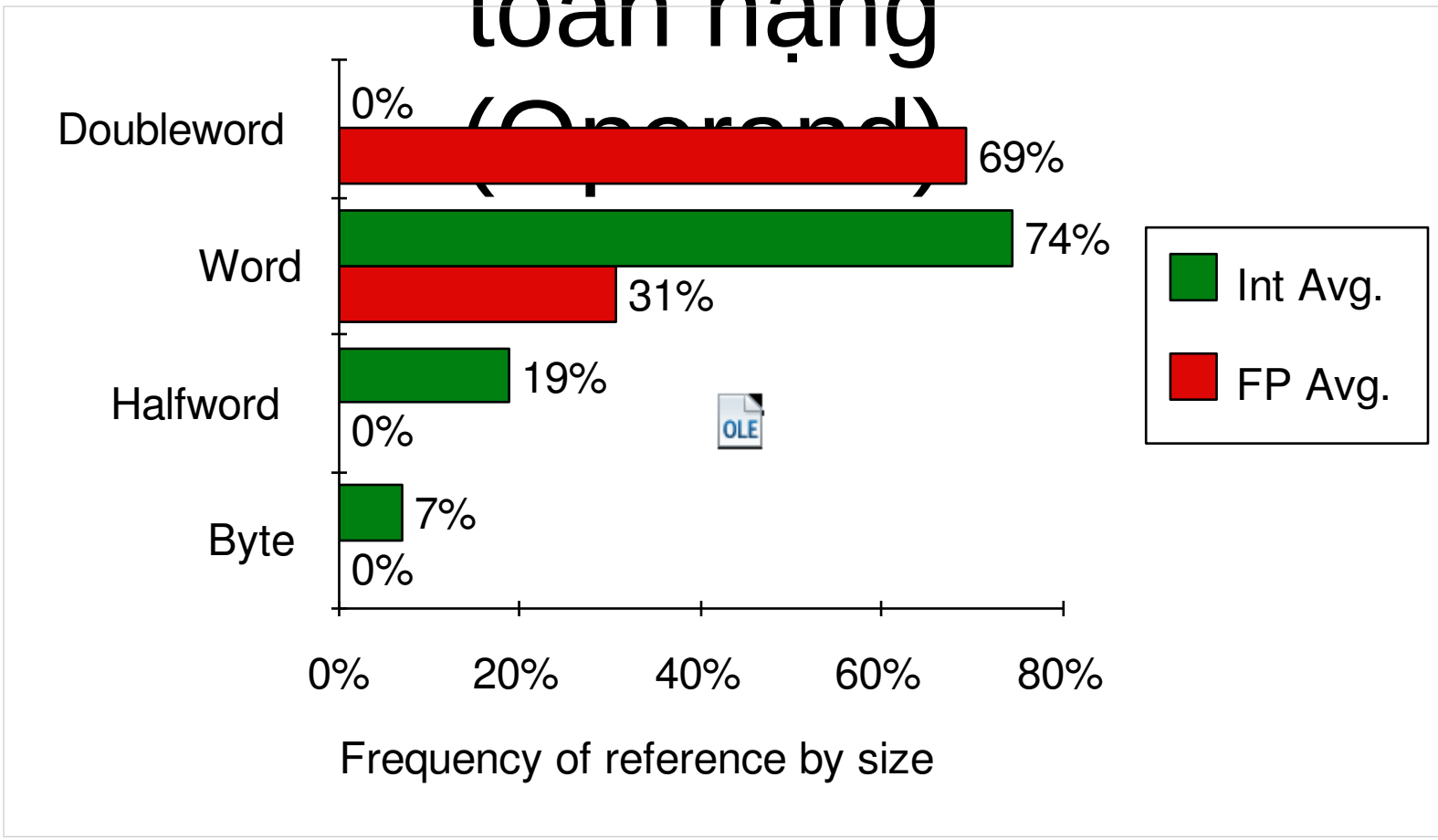
Stack	*Accumulator*	*Register (register-memory)*	*Register (load-store)*
Push A	*Load A*	*Load R1,A*	*Load R1,A*
Push B	*Add B*	*Add R1,B*	*Load R2,B*
Add	*Store C*	*Store C, R1*	*Add R3,R1,R2*
Pop C			*Store C,R3*

$$ExecutionTime \quad \frac{1}{Performance} \quad Instructions \quad \frac{Cycles}{Instruction} \quad \frac{Seconds}{Cycle}$$

So sánh số lượng các chỉ thị



Kích thước toán hạng (Operand)



Tập thanh ghi trong MIPS

32 Thanh ghi đa dụng.

– R0…R31 or $0…$31

– Các biến phải lưu trên thanh ghi.

Một số trường hợp đặc biệt.

– **R0** luôn có giá trị **zero (0)**

– **R29** là thanh ghi con trỏ ngăn xếp

– **R31 được sử dụng cho thủ tục quay lại địa chỉ**

Một số thanh ghi đặc biệt.

– **PC (Program Counter)**: Thanh ghi lệnh hiện tại

– HI & LO kết quả của phép nhân

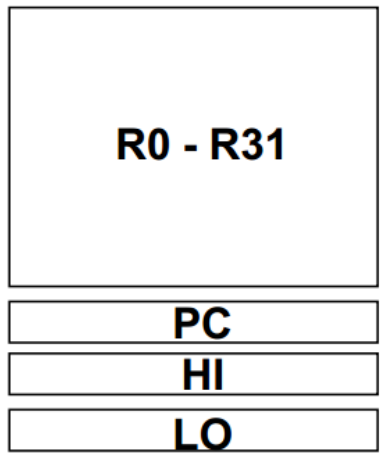
– Thanh ghi dấu phẩy động

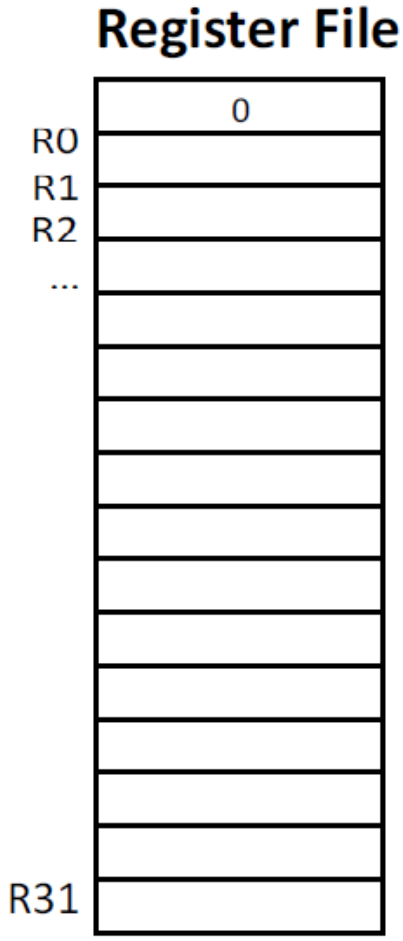
– Một số thanh ghi điều khiển (kiểm soát lỗi hoặc trạng thái)

Câu hỏi: Tại sao giá trị thanh ghi R0 luôn bằng 0?

Trả lời: Luôn cần giá trị 0 trong một chương trình

Move:
add dest, src, R0





Tổ chức bộ nhớ

Bộ nhớ là một mảng 1 chiều lưu trữ dữ liệu

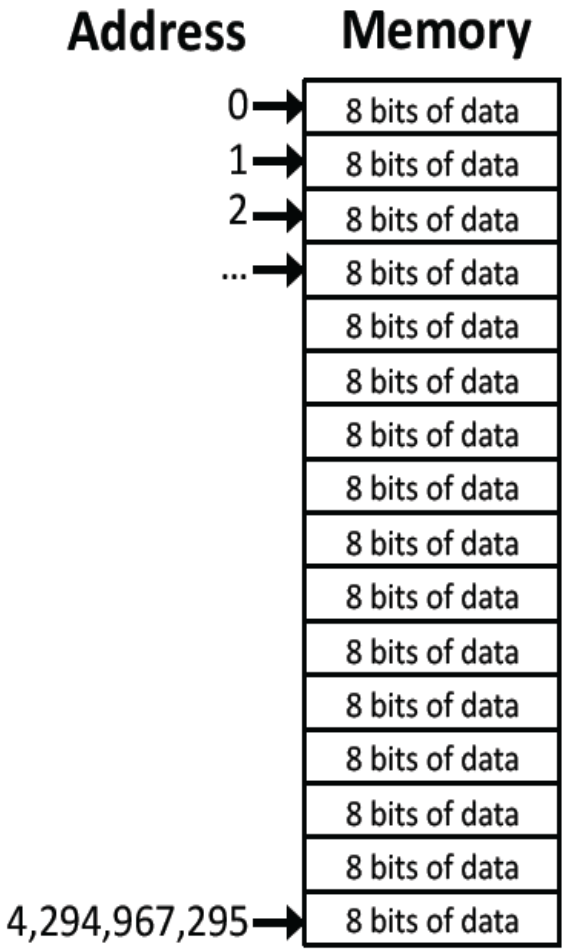
- Mỗi ô nhớ có kích thước 1 byte (8 bits)
- Địa chỉ ô nhớ được xác định theo địa chỉ byte.

Địa chỉ bộ nhớ được đánh chỉ số trên mảng

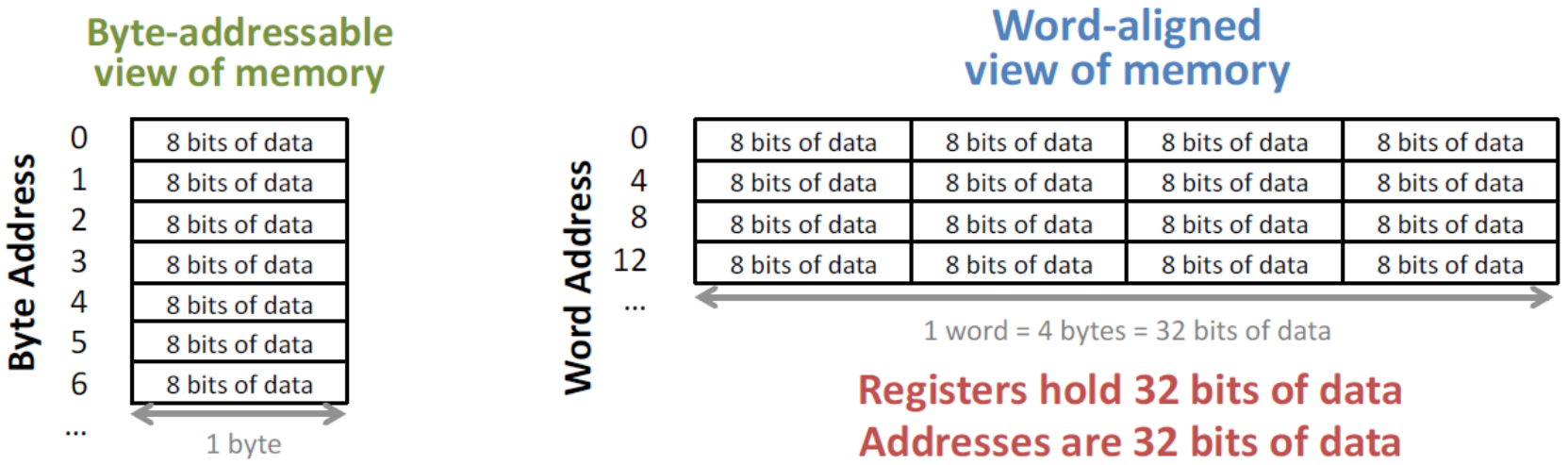
- Máy tính 32 bits, có 232 ô nhớ (4GB)
- Máy tính 64-bit, có 264 ô nhớ (16ExaByte)

Ví dụ:

– 64-bit x86 lên tới ~48-bits một ô nhớ (4PetaByte).



Tổ chức bộ nhớ

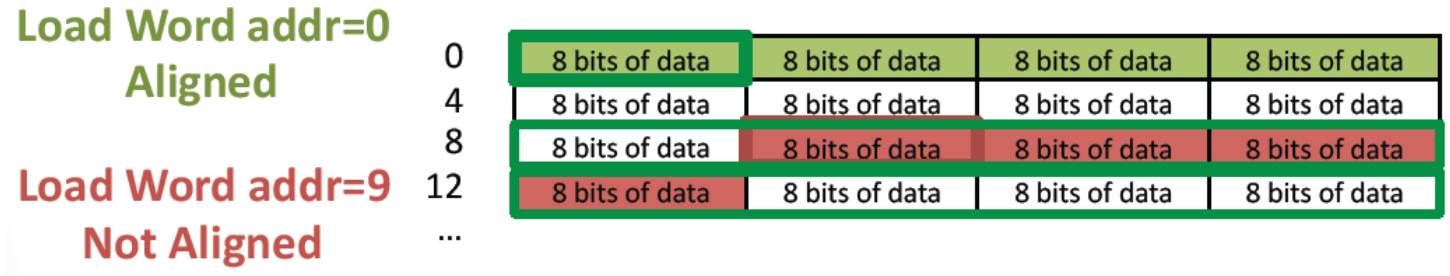


Các kiểu dữ liệu trong MIPS được định nghĩa là 1-byte hoặc 1-word

– Một từ gồm 32 bits = 4 bytes

– 232 bytes = 230 words: addresses 0, 4, 8, …

Câu hỏi: địa chỉ của một từ được xác định như thế nào ?



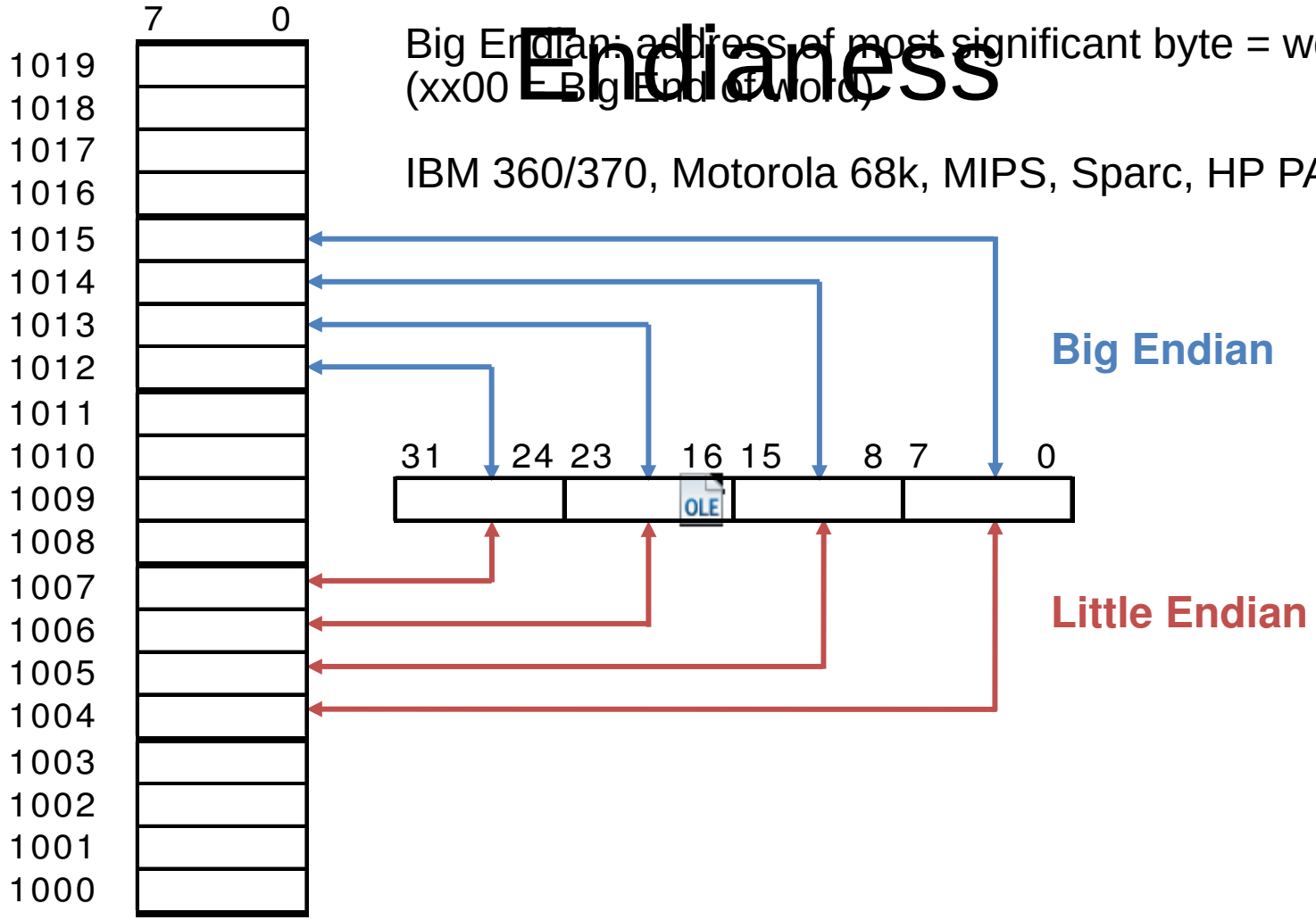
Địa chỉ Alignment : Tạo trên 4 byte (word) ở đường biên (e.g., 0, 4, 8, 12…)

– Dữ liệu được lưu trữ ở địa chỉ byte chia hết cho kích thước

Ánh xạ địa chỉ theo byte: Endianess

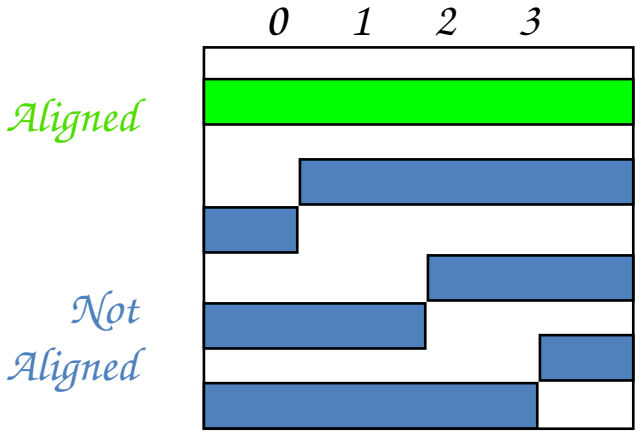
Big Endian: address of most significant byte = word address (xx00 = Big End of word)

IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA



Little Endian: address of least significant byte = word address

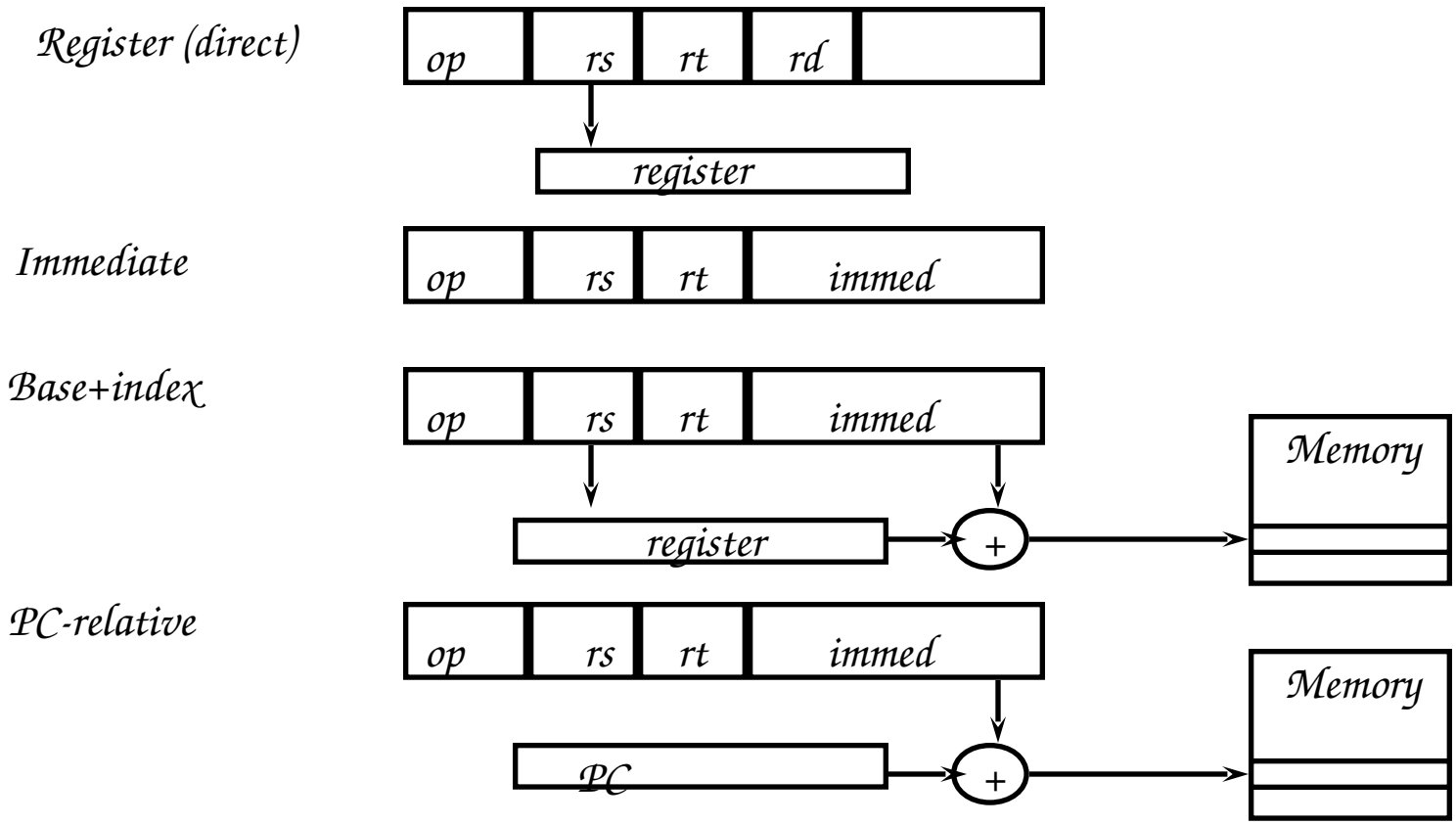
Ánh xạ địa chỉ theo tuyến : Alignment



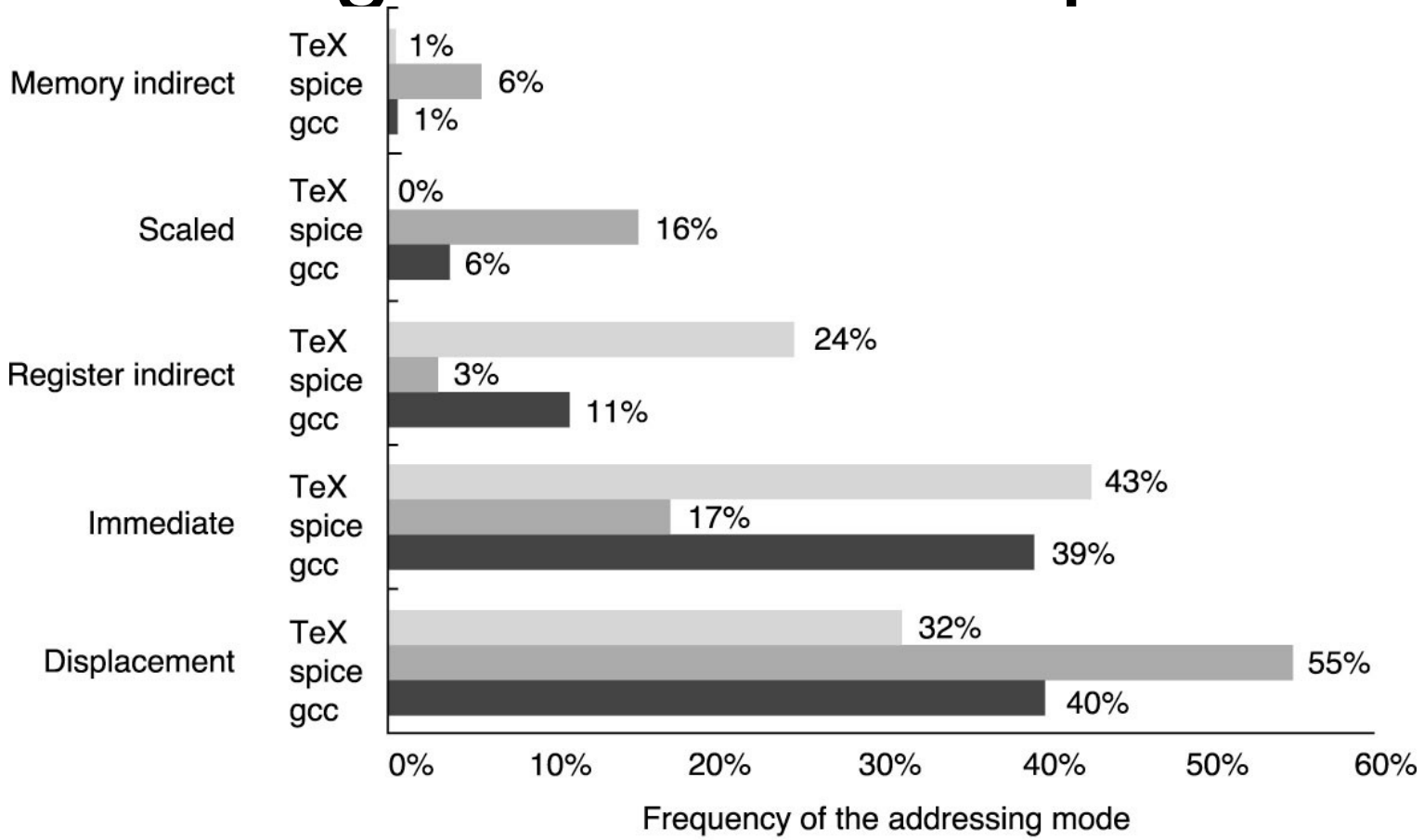
Alignment: require that objects fall on address that is multiple of their size.

Dạng chỉ thị MIPS và các chế độ đánh địa chỉ

- *Các chỉ thị lệnh dài 32 bits*



Thống kê các kiểu địa chỉ



for 51% of all references

Ví dụ: Tập lệnh MIPS

Định dạng trường lệnh 3 toán hạng :

- *op* dest, src1, src2 — dest ← src1 *op* src2

- Addition
  - add a, b, c — a ← b + c
  - addi a, b, 12 — a ← b + 12

"i" is for immediate

- Subtraction
  - sub a, b, c — a ← b - c

- Complex: f = (g + h) - (i + j)
  - add t0, g, h — t0 ← g +
  - add t1, i, j — t1 ← I +
  - sub f, t0, t1 — f ← t0 - t1

Complex operation many instructions with temporary values.

Phân loại tập lệnh

Toán tử
– Số học (add, multiply, subtract, divide, …)
– Logic (and, or, not, xor, …)

Dịch chuyển dữ liệu
– Move (register to register)
– Load (memory to register)
– Store (register to memory)

Điều khiển dữ liệu
– Branch (có điều kiện, e.g., <, >, ==)
– Jump (không điều kiện, e.g., goto)

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Bảng tham khảo định dạng lệnh

add	add $1, $2, $3	$1 = $2+$3
sub	sub $1,$2, $3	$1 = $2 - $3
add immediate	addi $1, $2, 100	$1 = $2 + 100
add unsigned	addu $1, $2, $3	$1 = $2 + $3
subtract unsigned	subu $1, $2, $3	$1 = $2 - $3
add imm. unsigned	addiu $1, $2, 100	$1 = $2 + 100
multiply	mult $2, $3	hi, lo = $2 * $3
multiply unsigned	multu $2, $3	hi, lo = $2 * $3
divide	div $2, $3	lo = $2/$3, hi = $2 mod $3
divide unsigned	divu $2, $3	lo = $2/$3, hi = $2 mod $3
move from hi	mfhi $1	$1 = hi
move from low	mflo $1	$1 = lo
and	and $1, $2, $3	$1 = $2 & $3
or	or $1, $2, $3	$1 = $2 | $3
and immediate	andi $1, $2, 100	$1 = $2 & 100
or immediate	ori $1, $2, 100	$1 = $2 | 100
shift left logical	sll $1, $2, 10	$1 = $2 << 10
shift right logical	srl $1, $2, 10	$1 = $2 >> 10
load word	lw $1, 100($2)	$1 = memory[$2+100]
store word	sw $1, 100($2)	memory[$2 + 100] = $1
load upper immediate	lui $1, 100	$1 = 100 * 2^16
branch on equal	beq $1, $2, 100	if ($1 == $2) go to PC + 4 + 100*4
branch on not equal	bne $1, $2, 100	if ($1 != $2) go to PC + 4 + 100*4
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set less than immediate	slti $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
set less than unsigned	sltui $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
set less than immediate unsigned	sltui $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
jump	j 10000	goto 10000
jump register	jr $31	goto $31
jump and link	jal 100000	$31 = PC + 4; goto 10000

Các lệnh thực thi

Phép toán (Data operations)
– Số học (add, multiply, subtra
…)
– Logic (and, or, not, xor, …)
Dịch chuyển dữ liệu (Data transfer)
– Move (register to register)
– Load (memory to register)
– Store (register to memory)
Rẽ nhánh (Sequencing)
– Branch (conditional, e.g., <,
– Jump (unconditional, e.g., g

1. Data operations: add/sub

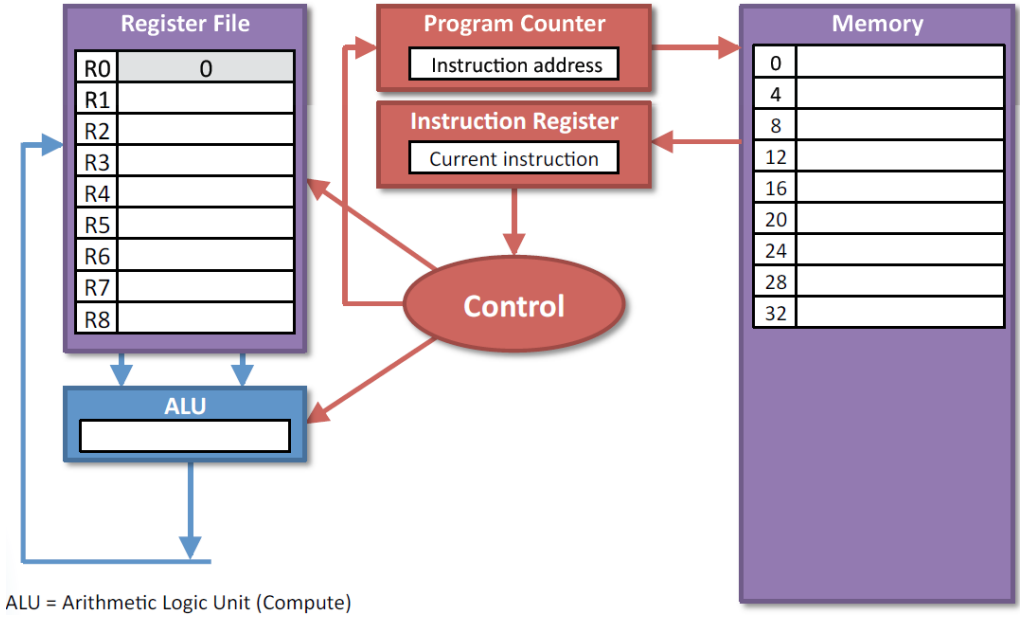
2. Data transfers: load word/store word

3. Sequencing: Branch/jump

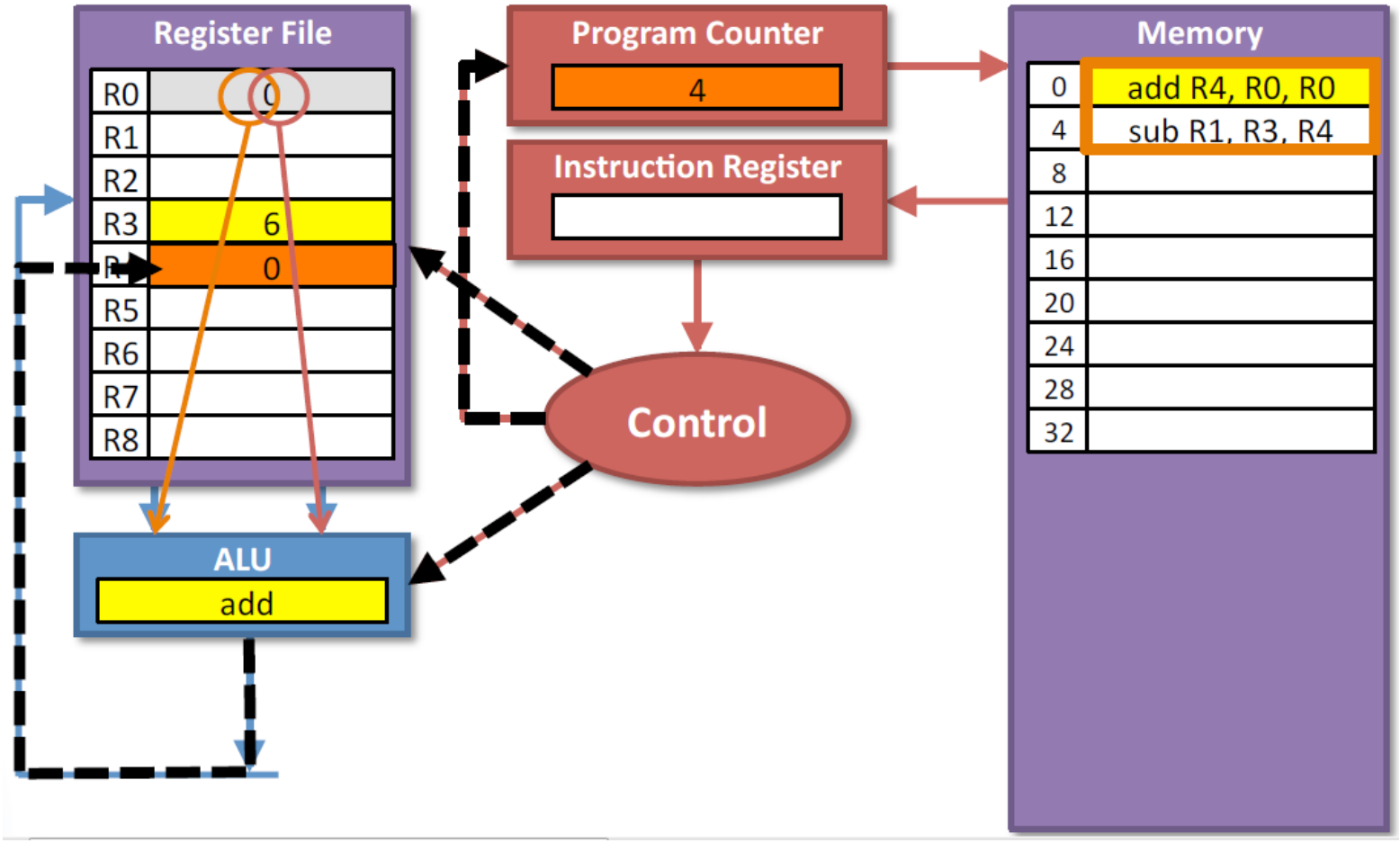
add	add $1, $2, $3	$1 = $2+$3
sub	sub $1,$2, $3	$1 = $2 - $3
add immediate	addi $1, $2, 100	$1 = $2 + 100
	addu $1, $2, $3	$1 = $2 + $3
	subu $1, $2, $3	$1 = $2 - $3
	addiu $1, $2, 100	$1 = $2 + 100
	mult $2, $3	hi, lo = $2 * $3
	multu $2, $3	hi, lo = $2 * $3
	div $2, $3	lo = $2/$3, hi = $2 mod $3
	divu $2, $3	lo = $2/$3, hi = $2 mod $3
	mfhi $1	$1 = hi
	mflo $1	$1 = lo
and	and $1, $2, $3	$1 = $2 & $3
	or $1, $2, $3	$1 = $2 | $3
	andi $1, $2, 100	$1 = $2 & 100
	ori $1, $2, 100	$1 = $2 | 100
	sll $1, $2, 10	$1 = $2 << 10
	srl $1, $2, 10	$1 = $2 >> 10
	lw $1, 100($2)	$1 = memory[$2+100]
	sw $1, 100($2)	memory[$2 + 100] = $1
	lui $1, 100	$1 = 100 * 2^16
branch on equal	beq $1, $2, 100	if ($1 == $2) go to PC + 4 + 100*4
	bne $1, $2, 100	if ($1 != $2) go to PC + 4 + 100*4
	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
	slti $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
	sltui $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
ned	sltui $1, $2, 100	if ($2 < 100) $1 = 1 else $1 = 0
	j 10000	goto 10000
	jr $31	goto $31
jump and link	jal 100000	$31 = PC + 4; goto 10000

Các phép toán

1. **Program Counter (PC)** lưu trữ địa chỉ lệnh.
2. **Các lệnh được nạp từ bộ nhớ vào thanh ghi lệnh.**
3. **Bộ điều khiển giải mã lệnh và báo cho ALU và tệp thanh ghi phải làm gì.**
4. **ALU thực thi lệnh và kết quả được chuyển lại thanh ghi dữ liệu**
5. **Bộ điều khiển cập nhật lại giá trị của PC cho lệnh tiếp theo.**



Ví dụ về lệnh: Add/sub (1 of 2)

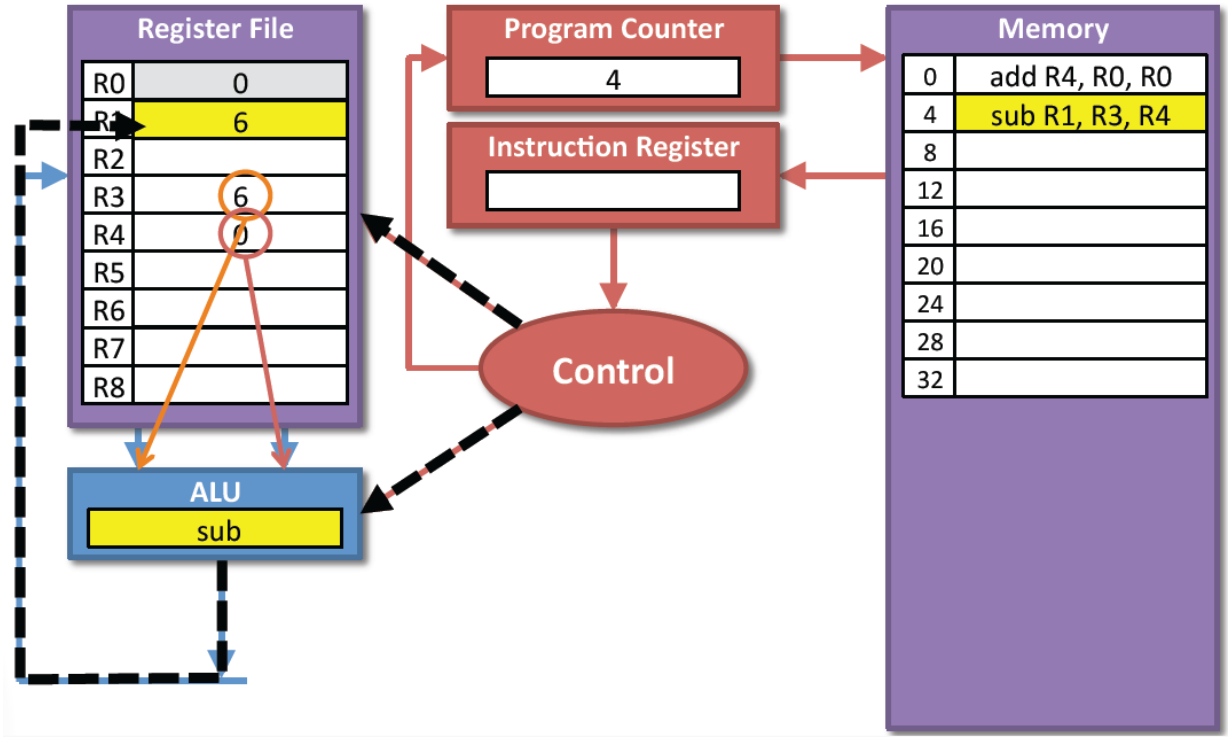


Ví dụ về lệnh: Add/sub (2 of 2)

Bộ đếm chương trình (PC) nạp lệnh từ thanh ghi lệnh

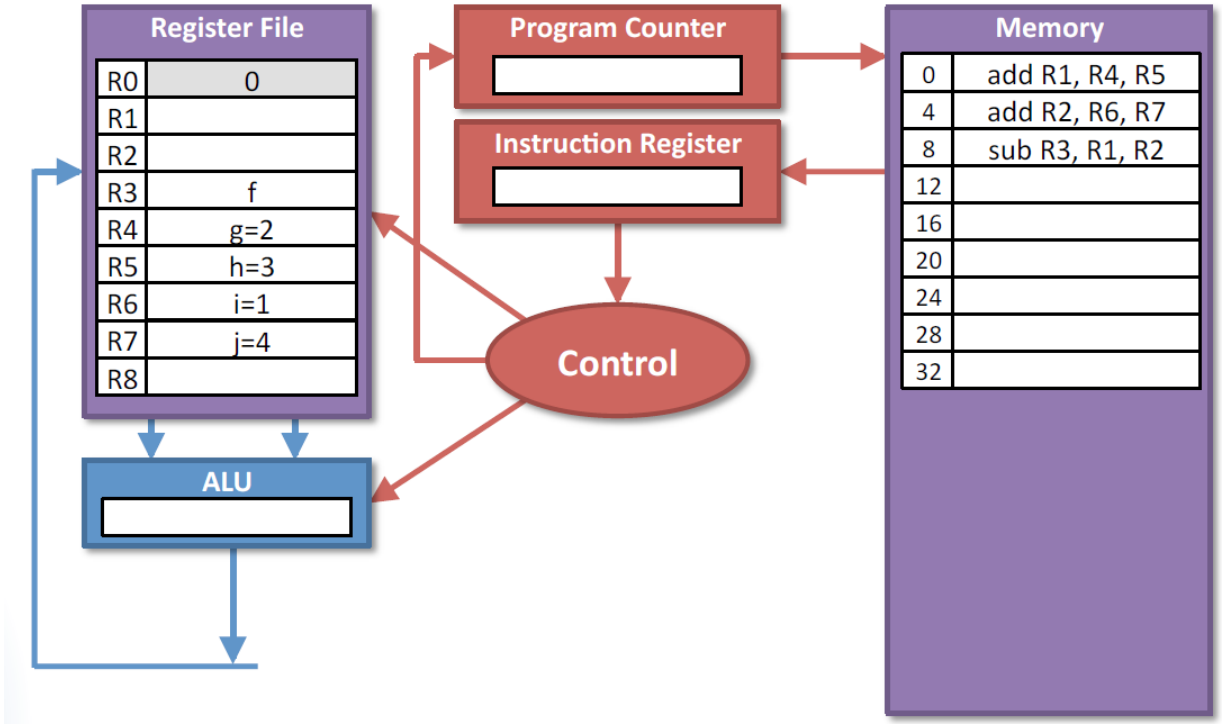
Control báo cho **ALU và tệp thanh ghi (Register File) phải làm gì**.

ALU ghi kết quả vào **Register File.**



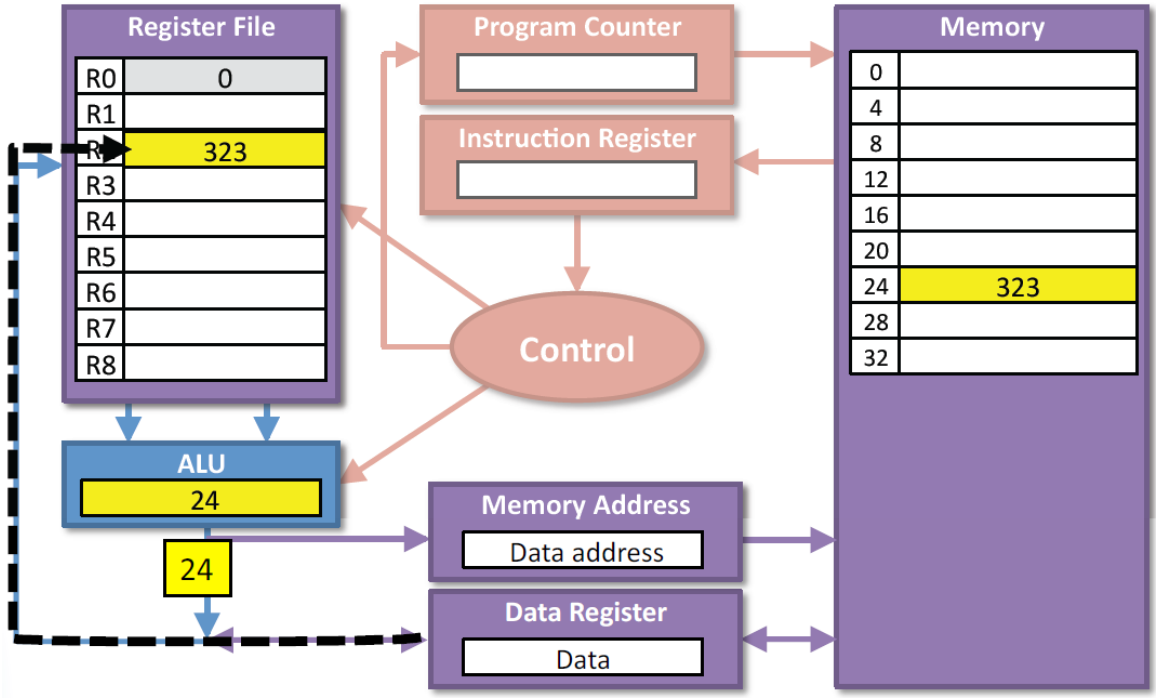
Thực hành: add/sub

f=(g+h) -(i+j)
R3=f
R4=g
R5=h
R6=I
R7=j

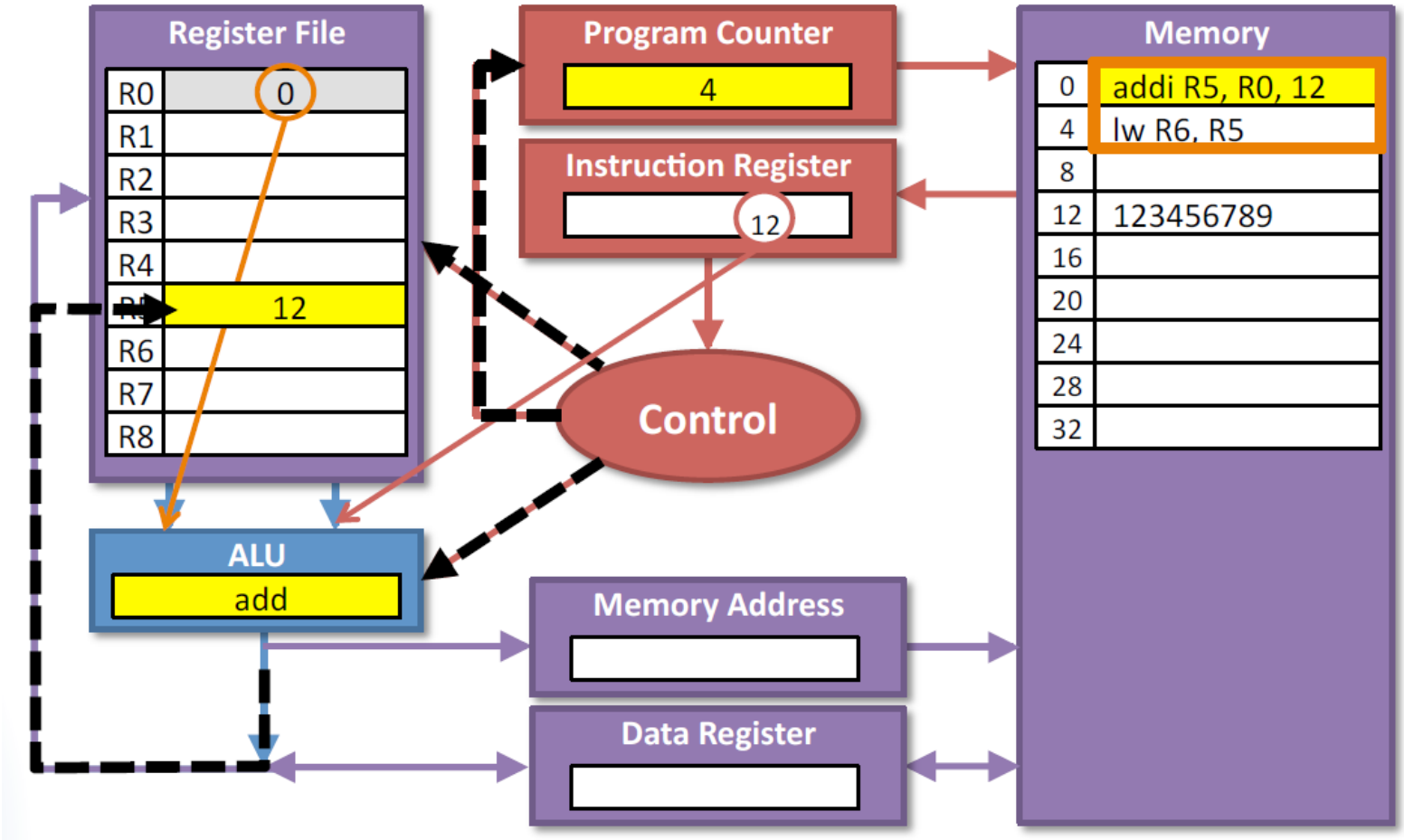


Dịch chuyển dữ liệu

1. ALU tính toán ra *địa chỉ*
2. Địa chỉ gửi tới thanh ghi địa chỉ (**Memory Address Register)**
3. Kết quả xác định hướng dịch chuyển đi/ đến được lưu trữ trên thanh ghi dữ liệu bộ nhớ (**Memory Data Register)**
4. Dữ liệu từ bộ nhớ có thể được ghi lại trên tệp thanh ghi (**Register File) hoặc ghi vào bộ nhớ.**

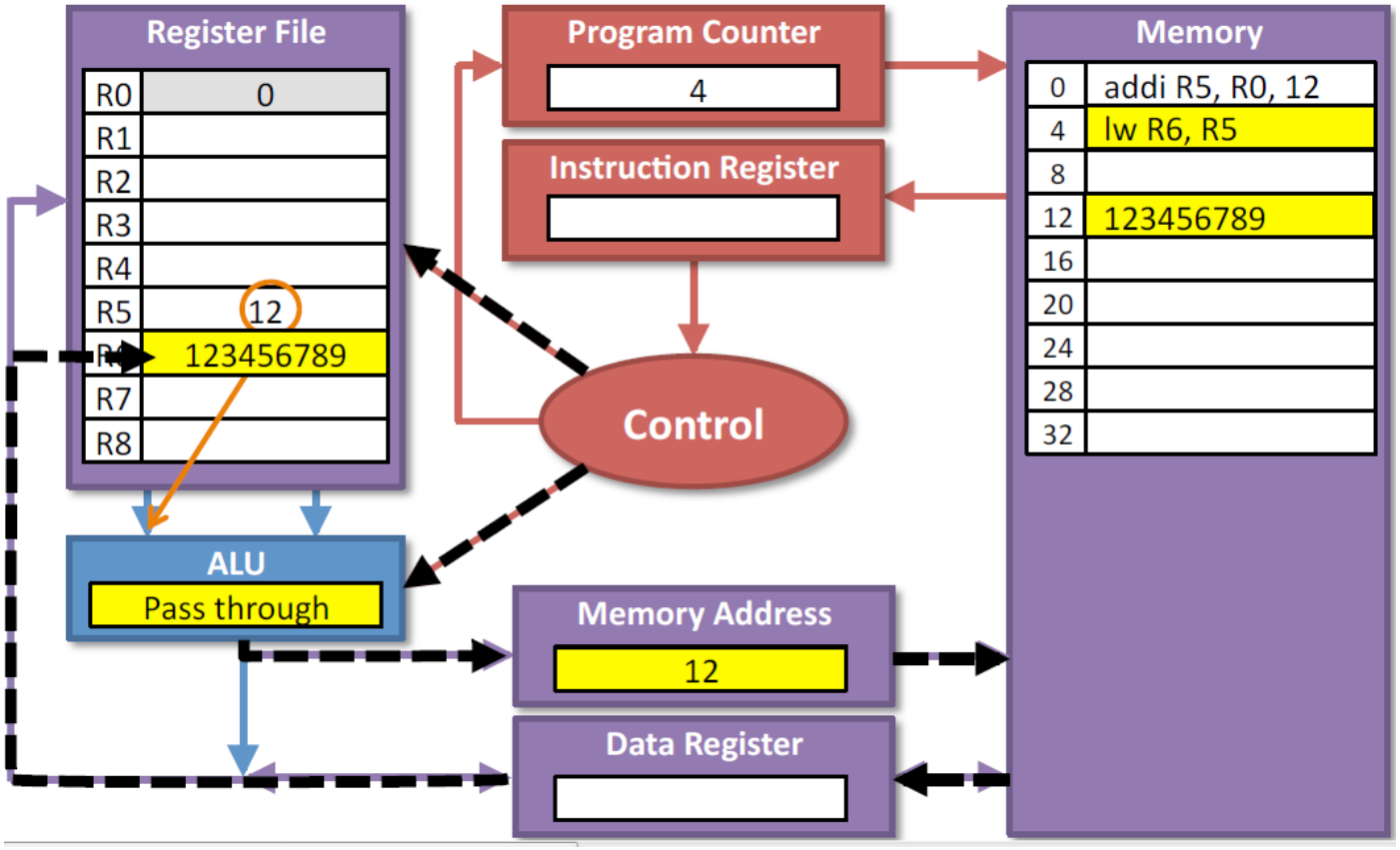


Lệnh tải từ: Load word (1 of 2)



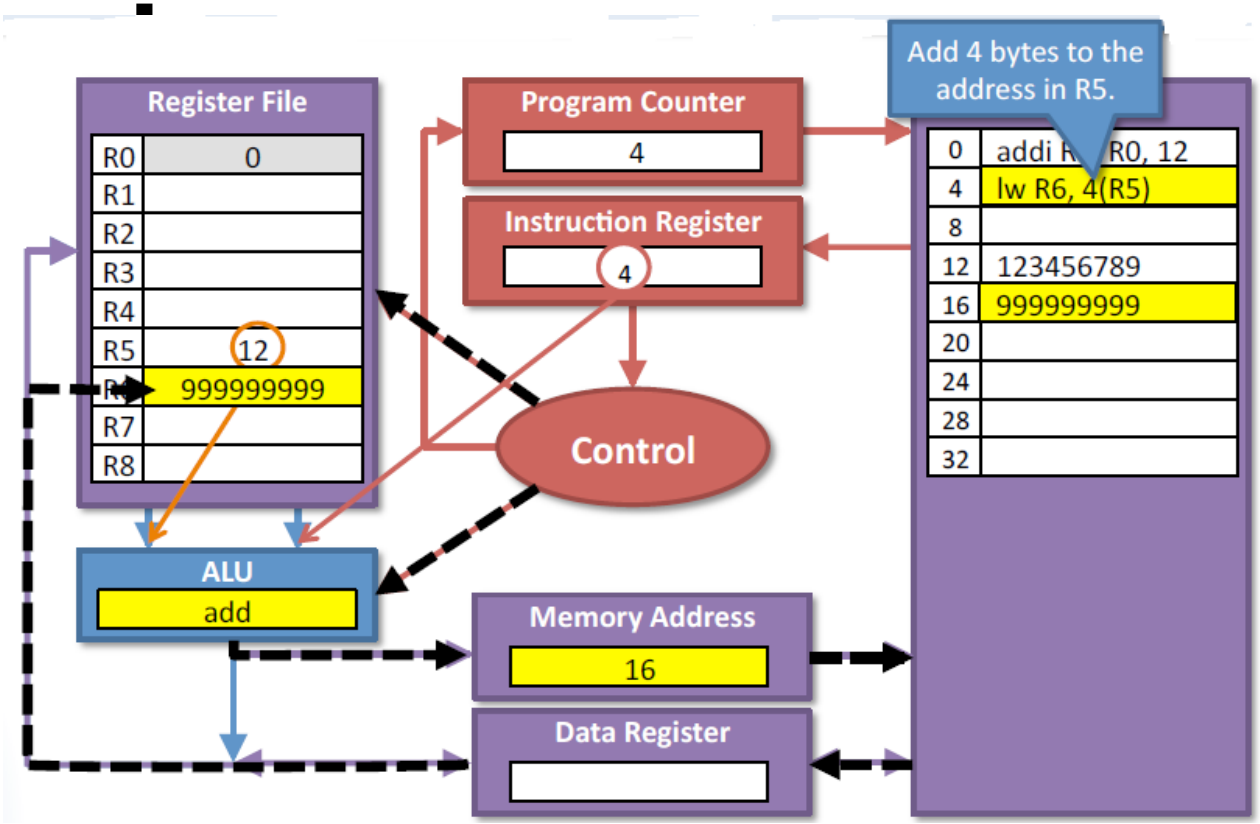
Lệnh tải từ: Load word (2 of 2)

Sử dụng địa chỉ từ tệp thanh ghi để tải dữ liệu từ bộ nhớ



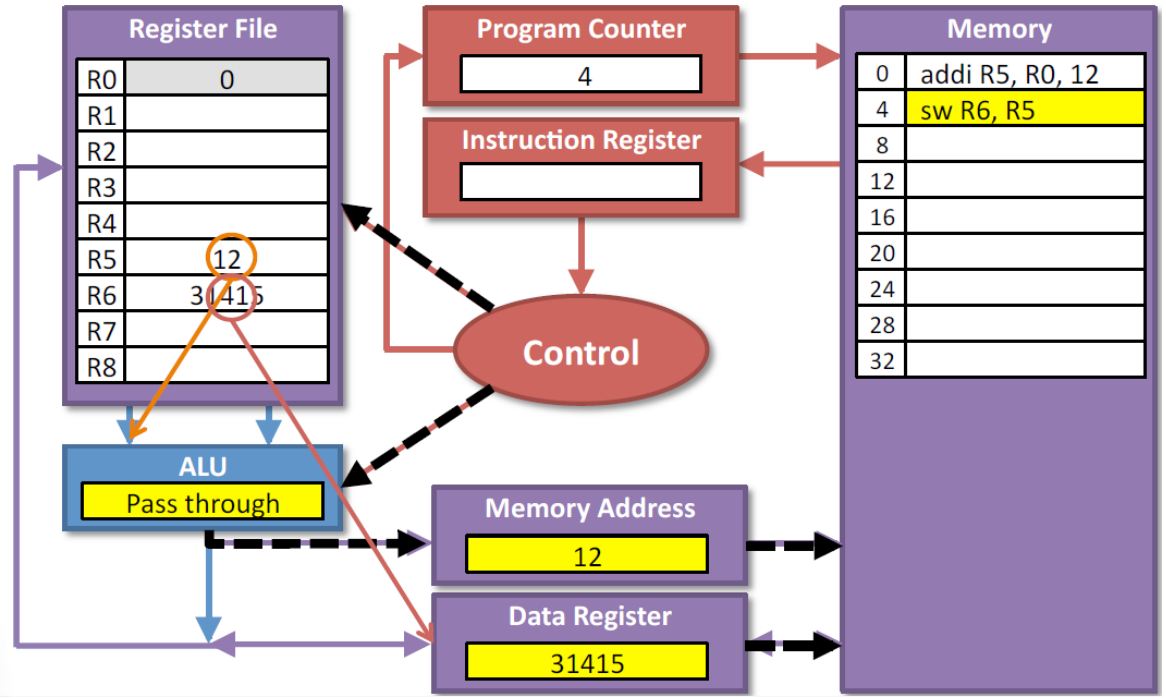
Lệnh tải từ với độ lệch địa chỉ.

Độ lệch được thêm vào địa chỉ như là một phần của các câu lệnh lw/sw



Lệnh lưu từ: Store word

Để thực hiện lưu trữ cần thông tin: Địa chỉ (từ ALU), **dữ liệu** (từ thanh ghi)

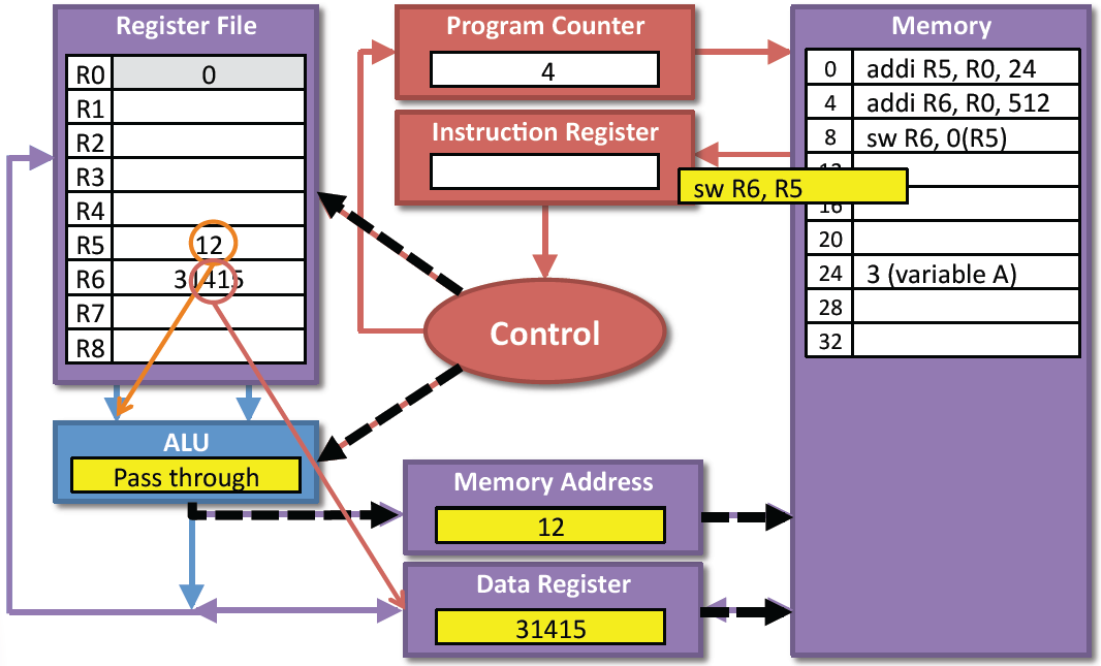


Ví dụ:

Biến A = 3
Địa chỉ của A = 24

Thực hiện chương trình:
Ghi giá trị A bằng 512

Địa chỉ của A vào R5
Lưu giá trị mới của A vào R6
Store : Mem[R5] ← R6



Các lệnh điều khiển rẽ nhánh

Lệnh điều khiển rẽ nhánh

– Câu lệnh nào được thực thi tiếp theo?

– Thay đổi luồng điều khiển chương trình “control flow”

Câu lệnh điều kiện trong MIPS

– **bne R0, R1, Label** branch if *not equal* to label

– **beq R3, R4, Label** branch if *equal* to label

Example:

```
R1 = i; R2 = j; R3 = h
```

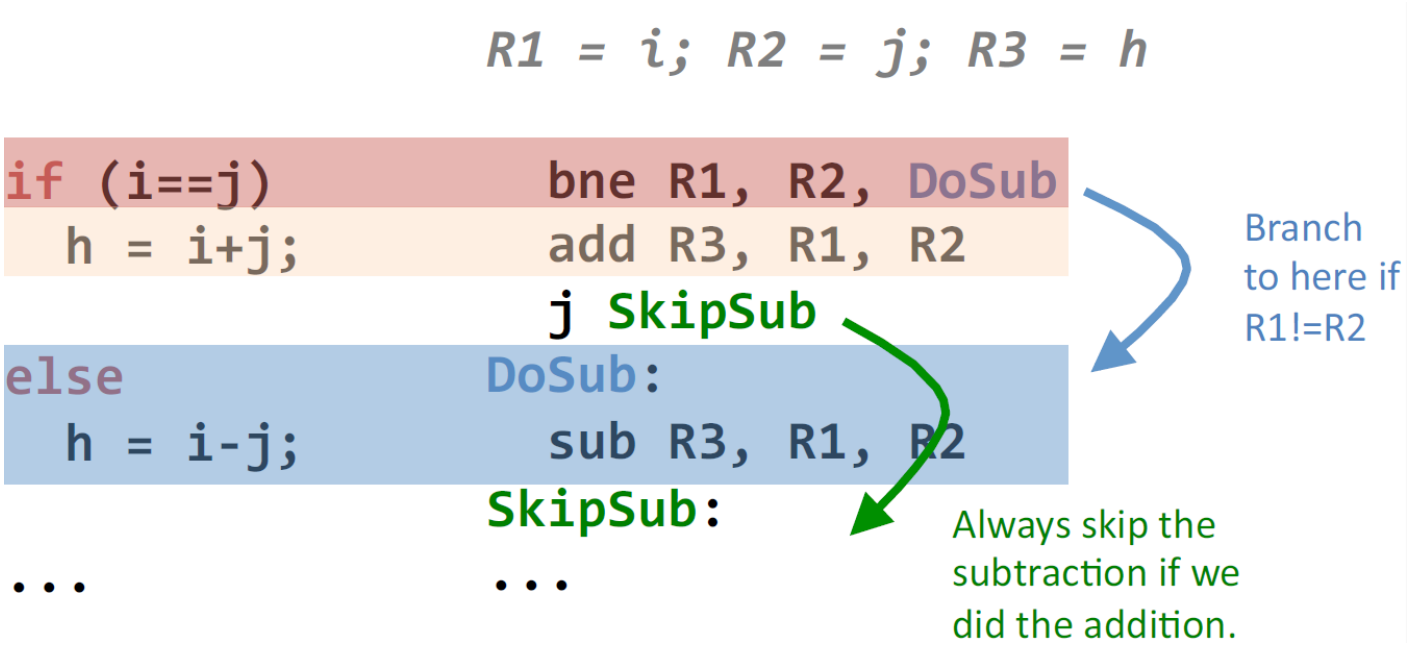
```
if (i==j)          bne R1, R2, Skip
  h = i+j;         add R3, R1, R2
...              Skip:
                   ...
```

Branch to here R1!=R2

Lệnh nhảy không điều kiện

Lệnh nhảy không điều kiện: jump
– **j Label** jump to label

Example:



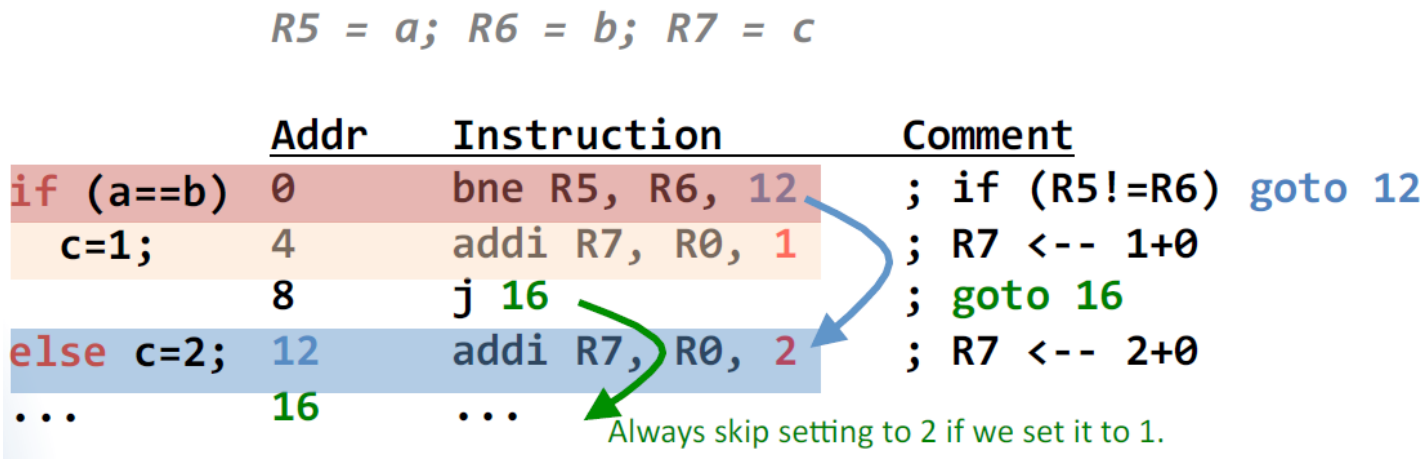
Các chỉ thị rẽ nhánh

Thay đổi đường dữ liệu chương trình → Thay đổi bộ đếm chương trình PC

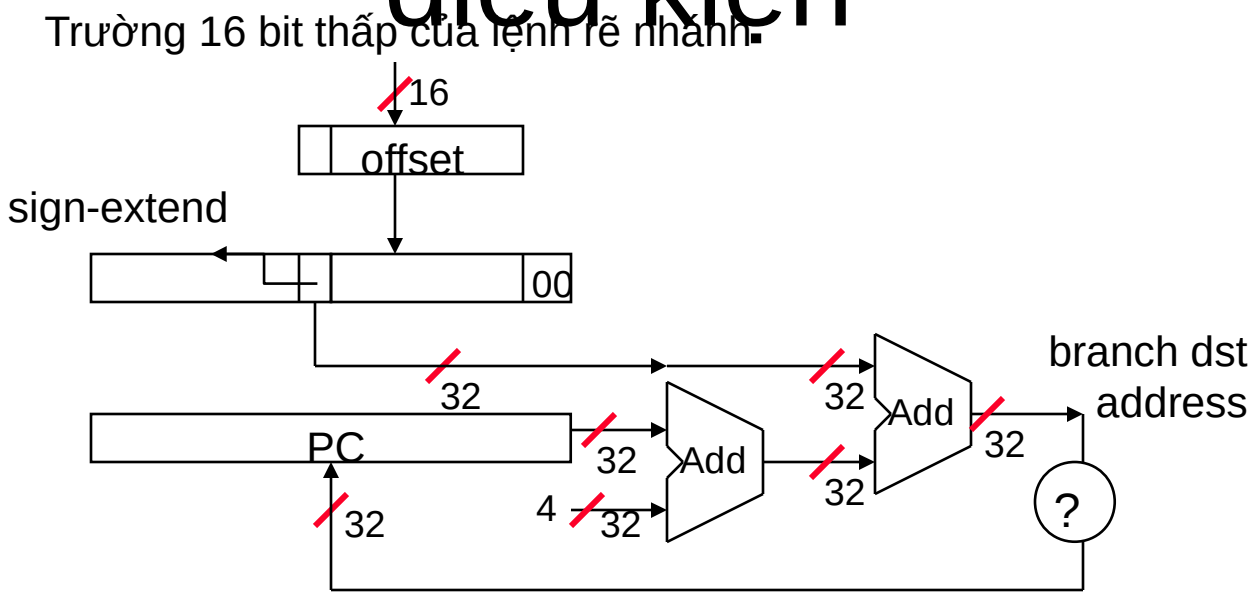
– **j** jump — Nhảy tới nhãn không điều kiện

– **bne** branch not equal — Nhảy tới nhãn nếu giá trị các thanh ghi không bằng nhau

Ví dụ: **if (a==b) c=1; else c=2;**



Xác định địa chỉ rẽ nhánh có điều kiện

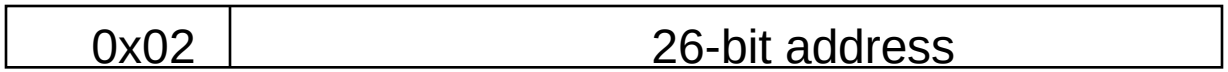


Lệnh nhảy không điều kiện

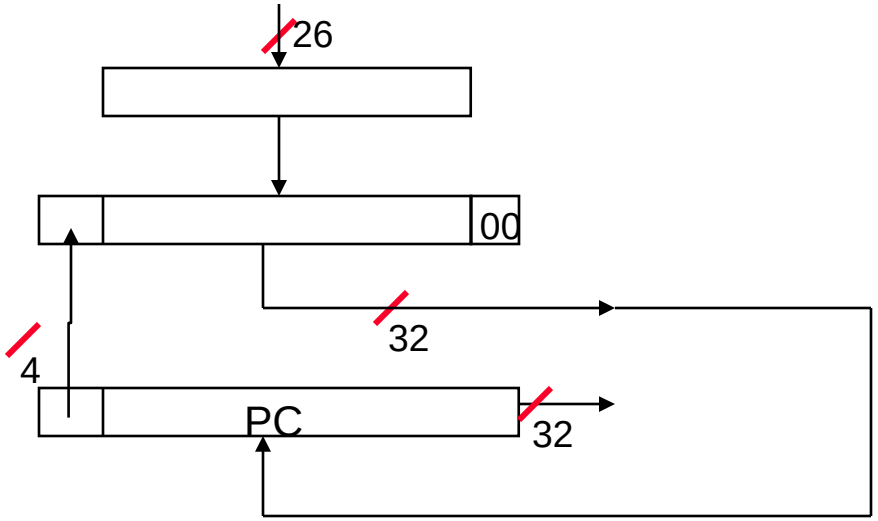
- Lệnh nhảy không điều kiện:

```
j  label    #go to label
```

- Định dạng lệnh (J Format):



từ trường 26 bits thấp của lệnh nhảy



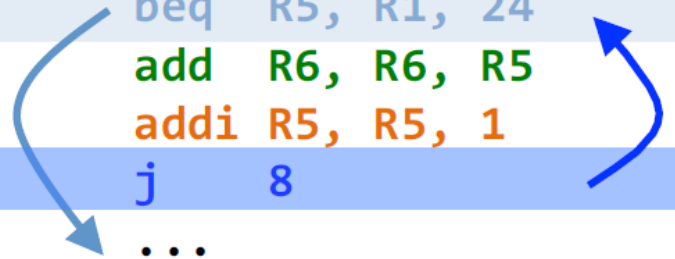
Vòng lặp

```
for (j=0; j<10; j++) {
  b = b + j;
}
...
```

R5 = j; R6 = b;

We need the constant 10 for the loop comparison, so put it in R1.

Addr	Instruction	Comment
0	addi R5, R0, 0	; j ← 0 + 0
4	addi R1, R0, 10	; R1 ← 0 + 10
8	beq R5, R1, 24	; if (j == 10) goto 24
12	add R6, R6, R5	; b ← b + j
16	addi R5, R5, 1	; j ← j + 1
20	j 8	; goto 8
24	...	; pop out of loop, continue

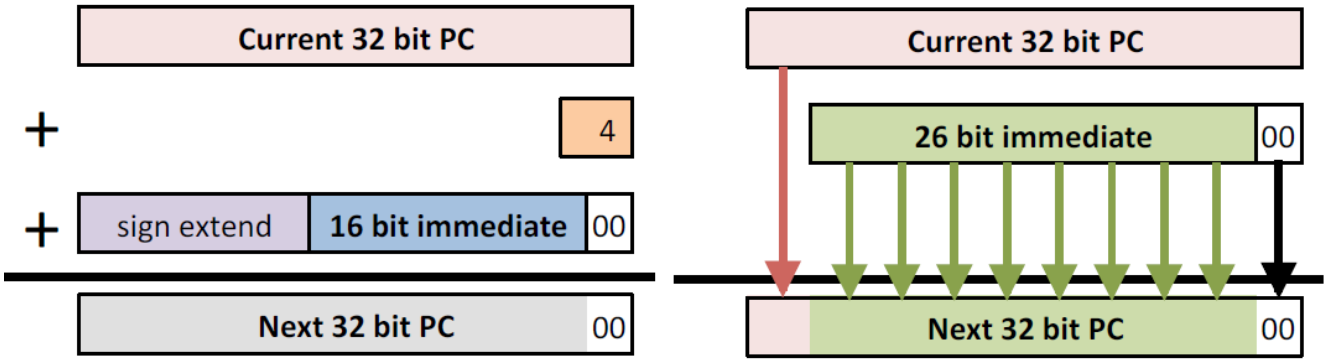


Địa chỉ trong lệnh rẽ nhánh và lệnh nhảy

Question:
Sử dụng lệnh nhảy bne/beq với khoảng cách bao nhiêu?
Answer:
Từ -32,767 đến +32,768 lệnh từ chỉ thị lệnh hiện tại.

Các lệnh rẽ nhánh
– **bne/beq** I-format **16 bit immediate**
– **j** J-format **26 bit immediate**

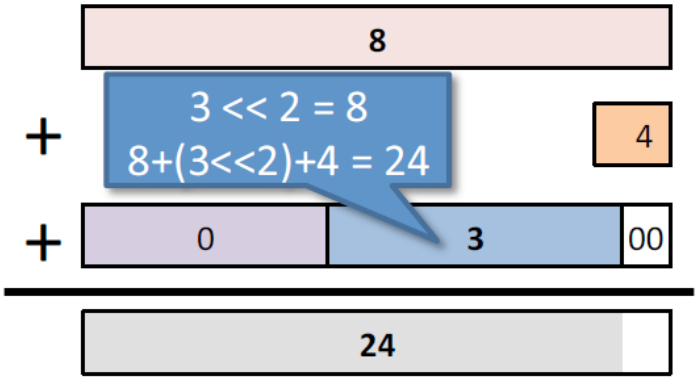
Địa chỉ là 32 bits! Điều khiển bằng cách nào?
– Xem xét **bne/beq** như là độ lệch tương đối (**relative offsets)** (**cộng với giá trị** PC hiện thời)
– Xem xét **j** như là một giá trị tuyệt đối (**absolute value)** (**thay thế** 26 bits của PC)



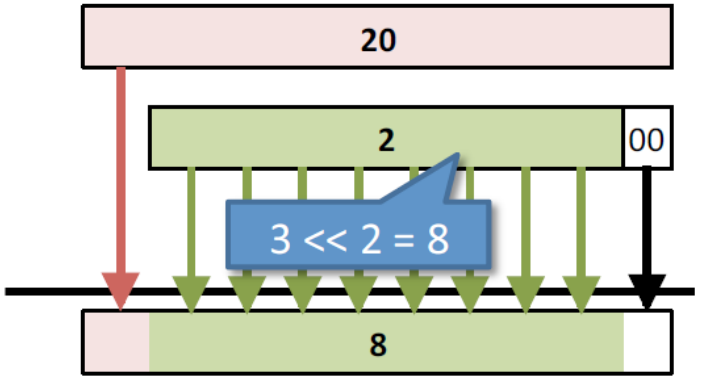
Ví dụ nhảy địa chỉ: loops

```
for (j=0; j<10; j++)
{
  b = b + j;
}
```

```
Addr    Instruction          Comment
0       addi R5, R0, 0       ; j ← 0 + 0
4       addi R1, R0, 10      ; R1 ← 0 + 10
8       beq  R5, R1, 24      ; if ( j == 10) goto 24
12      add  R6, R6, R5      ; b ← b + j
16      addi R5, R5, 1       ; j ← j + 1
20      j    8               ; goto 8
24      ...                  ; done with loop
```



beq: PC=PC+4+(3<<2)



j: PC=[PC(31:28):2]<<2

Biên dịch thành mã máy

Mã hóa và các định dạng

Định dạng lệnh (mã máy)

Ngôn ngữ máy

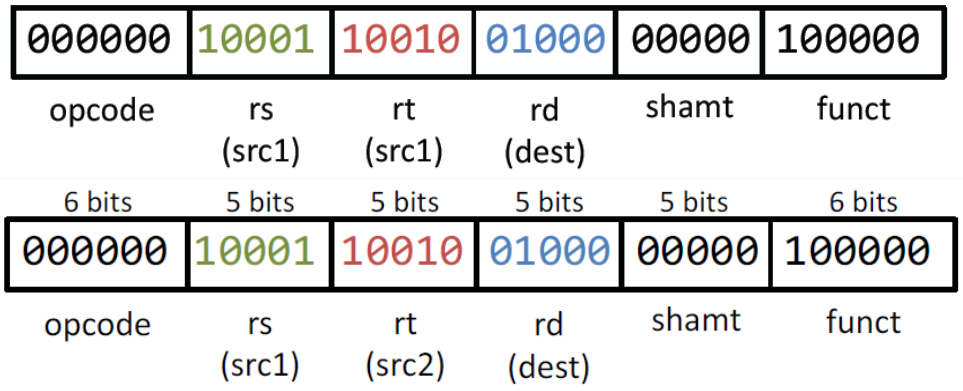
– Máy tính không hiểu được chuỗi ký tự sau “add R8, R17, R18”

– Các lệnh phải được chuyển đổi thành ngôn ngữ máy(1s and 0s)

Ví dụ:

add R8, R17, R18 → 000000 10001 10010 01000 00000 100000

Các trường lệnh MIPS



• **opcode** mã lệnh xác định phép toán (e.g., “add” “lw”)

• **rs** chỉ số thanh ghi chứa toán hạng nguồn 1 trong tệp thanh ghi

• **rt** chỉ số thanh ghi chưa toán hạng nguồn 2 trong tệp thanh ghi

• **rd** chỉ số thanh ghi lưu kết quả

• **shamt** Số lượng dịch(cho chỉ thị dịch)

• **funct** mã chức năng thêm cho phần mã lệnh (add = 32, sub =34)

Định dạng lệnh MIPS

Câu hỏi: Lệnh cộng tức thời (addi) cần bao nhiêu bit để lưu giá trị hằng số?
Trả lời:
I-format: 5+5+6 bits = 16 bits.
Giá trị nằm trong khoảng
Từ -32,768 đến +32,767

- **MIPS có 3 dạng chỉ thị :**
  - R: operation 3 registers — no immediate
  - I: operation 2 registers — short immediate
  - J: jump 0 registers — long immediate

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	shmt	funct	Arithmetic instruction format
I-format	op	rs	rt	address /immediate			Transfer (load/store), branch, immediate format
J- format	op	target address					Jump instruction format

Tổng kết các lệnh MIPS

Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 4	$s1 = $s2 + 4
	shift left logical	0 & 00	sll $s1, $s2, 4	$s1 = $s2 << 4
	shift right logical	0 & 02	srl $s1, $s2, 4	$s1 = $s2 >> 4 (fill with zeros)
	shift right arithmetic	0 & 03	sra $s1, $s2, 4	$s1 = $s2 >> 4 (fill with sign bit)
	and	0 & 24	and $s1, $s2, $s3	$s1 = $s2 & $s3
	or	0 & 25	or $s1, $s2, $s3	$s1 = $s2 | $s3
	nor	0 & 27	nor $s1, $s2, $s3	$s1 = not ($s2 | $s3)
	and immediate	c	and $s1, $s2, ff00	$s1 = $s2 & 0xff00
	or immediate	d	or $s1, $s2, ff00	$s1 = $s2 | 0xff00
	load upper immediate	f	lui $s1, 0xffff	$s1 = 0xffff0000

Tổng kết các lệnh MIPS

Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	2b	sw $s1, 100($s2)	Memory($s2+100) = $s1
	load byte	20	lb $s1, 101($s2)	$s1 = Memory($s2+101)
	store byte	28	sb $s1, 101($s2)	Memory($s2+101) = $s1
	load half	21	lh $s1, 101($s2)	$s1 = Memory($s2+102)
	store half	29	sh $s1, 101($s2)	Memory($s2+102) = $s1
Cond. branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than immediate	a	slti $s1, $s2, 100	if ($s2<100) $s1=1; else $s1=0
	set on less than	0 & 2a	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1; else $s1=0
Uncond. jump	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

Hằng số (lệnh trực tiếp)

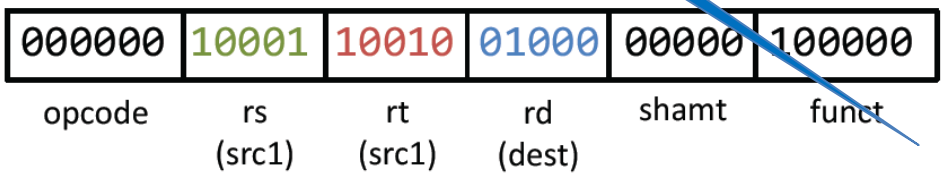
Cần bao nhiêu bit khi sử dụng các thanh ghi để lưu giá trị, số lượng than ghi là hữu hạn?

Lưu trữ dữ liệu về hằng số trên chỉ thị, không sử dụng tệp thanh ghi.

Các hằng số nhỏ (tức thì) được sử dụng ở hầu hết các đoạn mã (~50%)

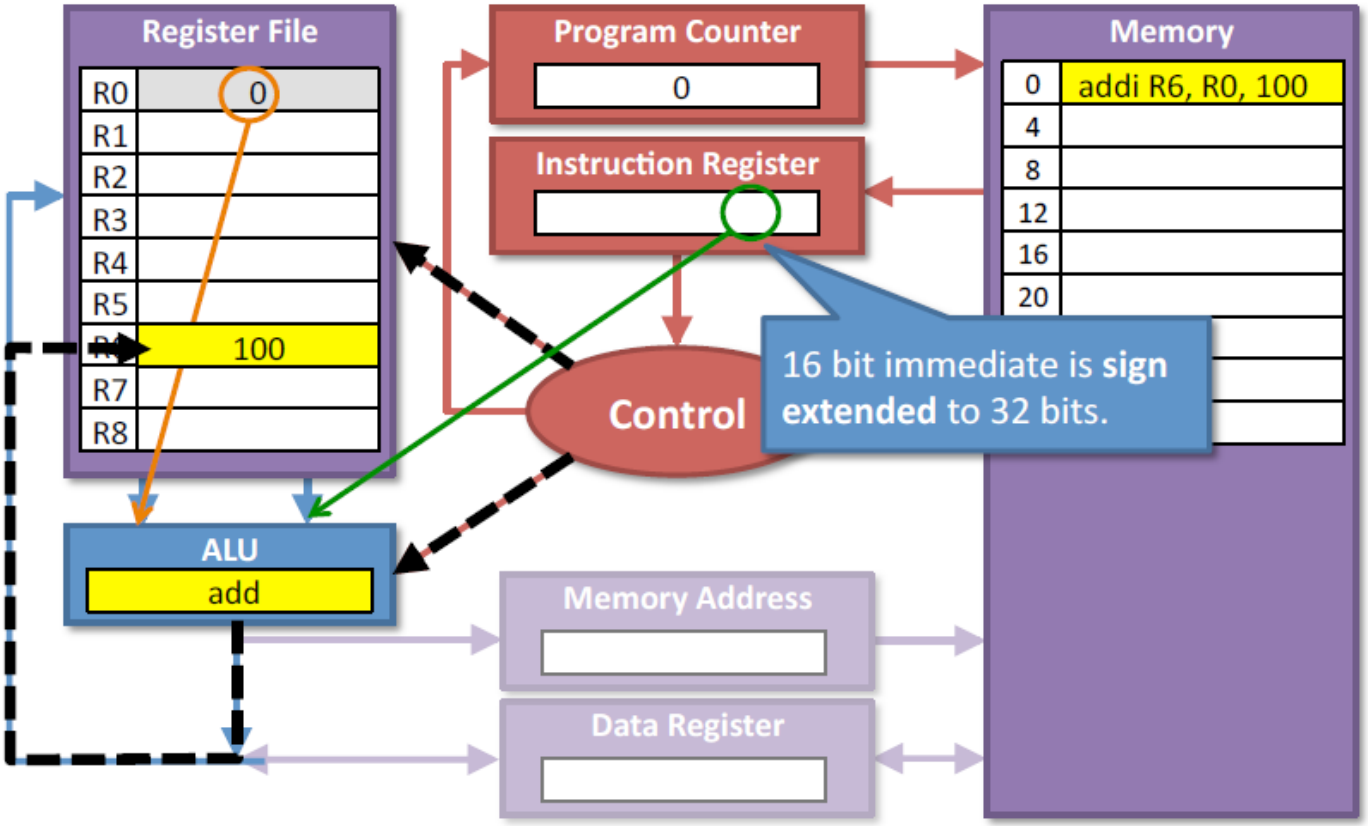
If (a==b) c=1;
else c=2;

- Làm thế nào để thực thi trong bộ xử lý ?
  - Đặt các hằng số vào bộ nhớ và tải chúng (chậm)
  - Gán cứng giá trị trên thanh ghi(giống R0) (Bao nhiêu?)
- **Nó thực thi như thế nào:**
  - Các chỉ thị có thể chứa hằng số bên trong.
  - Bộ điều khiển sẽ gửi giá trị hằng số đến bộ ALU
  - **addi R29, R30, 4** ← giá trị 4 nằm trong câu lệnh
- **Nhưng xảy ra một vấn đề**
  - Số bit mã hóa của một trường lệnh là 32 bits. Cần dùng cho trường mã lệnh và các thanh ghi. Làm cách nào để cân đối được không gian cho các hằng số và chỉ thi lệnh?



Tải các giá trị tức thì (hằng số)

Bộ điều khiển (Control) báo cho ALU nhận toán hạng từ tệp thanh ghi và từ chỉ thị lệnh.

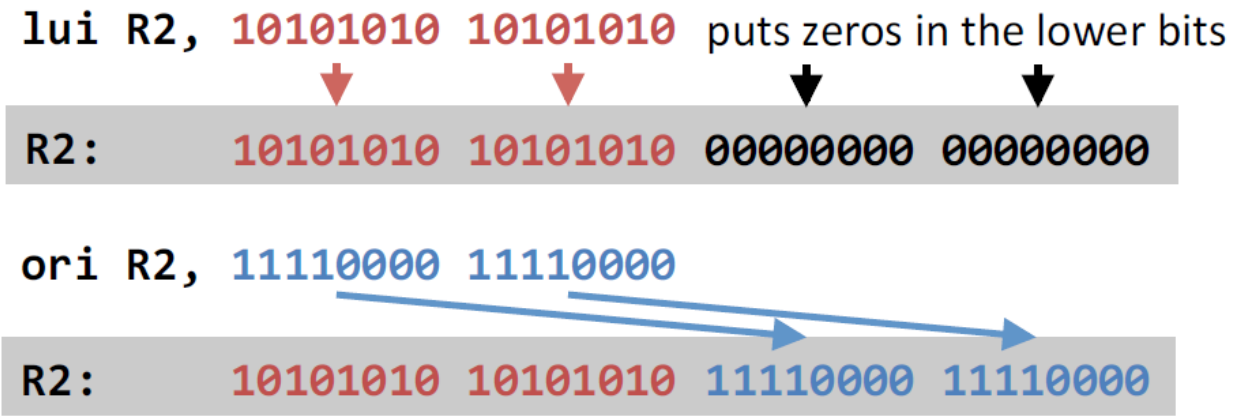


Tải các giá trị lớn

Question: ori có sử dụng cho các số có dấu?
Answer: không.

- Trường lệnh trực tiếp giới hạn trong 16 bits (-32,768 to +32,767)
  - **Làm thế nào để tải được các giá trị lớn?**
- Sử dụng 2 lệnh để tải
  - **Load Upper Immediate** (**lui**): Loads **upper** 16 bits
  - **Or Immediate** (**ori**): Loads **lower** 16bits

- Ví dụ: **10101010 10101010 11110000 11110000**



Thủ tục gọi hàm

- Lời gọi thủ tục và khai báo thủ tục được chuyển thành lệnh máy như thế nào?
- Đối số (biến số) được truyền vào thủ tục như thế nào?
- Kết quả trả về của thủ tục được truyền ra ngoài như thế nào?
- Thủ tục được gọi: **Callee**
- Thủ tục gọi: **Caller**

Các thủ tục gọi hàm

Các thủ tục (hàm/chương trình con) sử dụng cho chương trình có cấu trúc.

```
main( ) { for ( j=0; j<10; j++ )
If (a[ j ] == 0)
a[ j ] = update(a[ j ], j);
}
```

Thực thi các thủ tục cần các điều kiện sau:

- Đưa dữ liệu vào nơi thủ tục cần truy cập vào
- Bắt đầu thực thi
- Làm việc/ sử dụng thanh ghi
- Quay lại thủ tục gọi (**caller)**
- Nhận kết quả và trả kết quả về

Các thủ tục gọi hàm

```
main( ) { for ( j=0; j<10; j++ )
If (a[ j ] == 0)
a[ j ] = update(a[ j ], j);
}
```

Caller: ***main***()
Callee: ***update***()
Parameters: **a[j], j**
Results: (stored in) **a[j]**

```
……
Int update (int x, int y)  {
return x+y;
}
```

1. Đưa các tham số (**parameters) vào thủ tục được gọi (callee)**
2. Chuyển quyền điều khiển tới thủ tục được gọi
3. Cấp các thanh ghi cần thiết cho thủ tục được gọi
4. Thực thi đoạn mã
5. Trả kết quả (**results) vào vị trí hàm gọi có thể truy cập**
6. Trả điều khiển đến vị trí trước khi gọi thủ tục

…without messing up the caller's registers!

Nguyên tắc sử dụng thủ tục

Ví dụ: f(g,h,i,j)=(g+h) – (i+j)

```
add R1, R4, R5 ;        g=R4, h=R5
add R2, R6, R7 ;        i=R6, j=R7
sub  R3, R1, R2
```

Nếu thủ tục gọi(e.g., **main()**) sử dụng R1, R2 hoặc R3 sẽ phải được lưu lại, bởi vì thủ tục được gọi sẽ ghi đè lên khi thực thi.

Một số vấn đề:

• Thủ tục được gọi không biết về các thanh ghi thủ tục gọi sử dụng (Bao gồm nhiều thủ tục gọi khác nhau)

• Thủ tục gọi không biết thanh ghi nào mà thủ tục được gọi sẽ sử dụng! (Có thể gọi nhiều thủ tục con)

• Phân chia giữa callee và caller
• Theo quy ước này cho phép bất kỳ thủ tục gọi nào đều gọi tới bất kỳ thủ tục được gọi
• Callee và caller đều biết cái gì cần được lưu trữ

Các thanh ghi lưu trữ: Quy ước trong MIPS

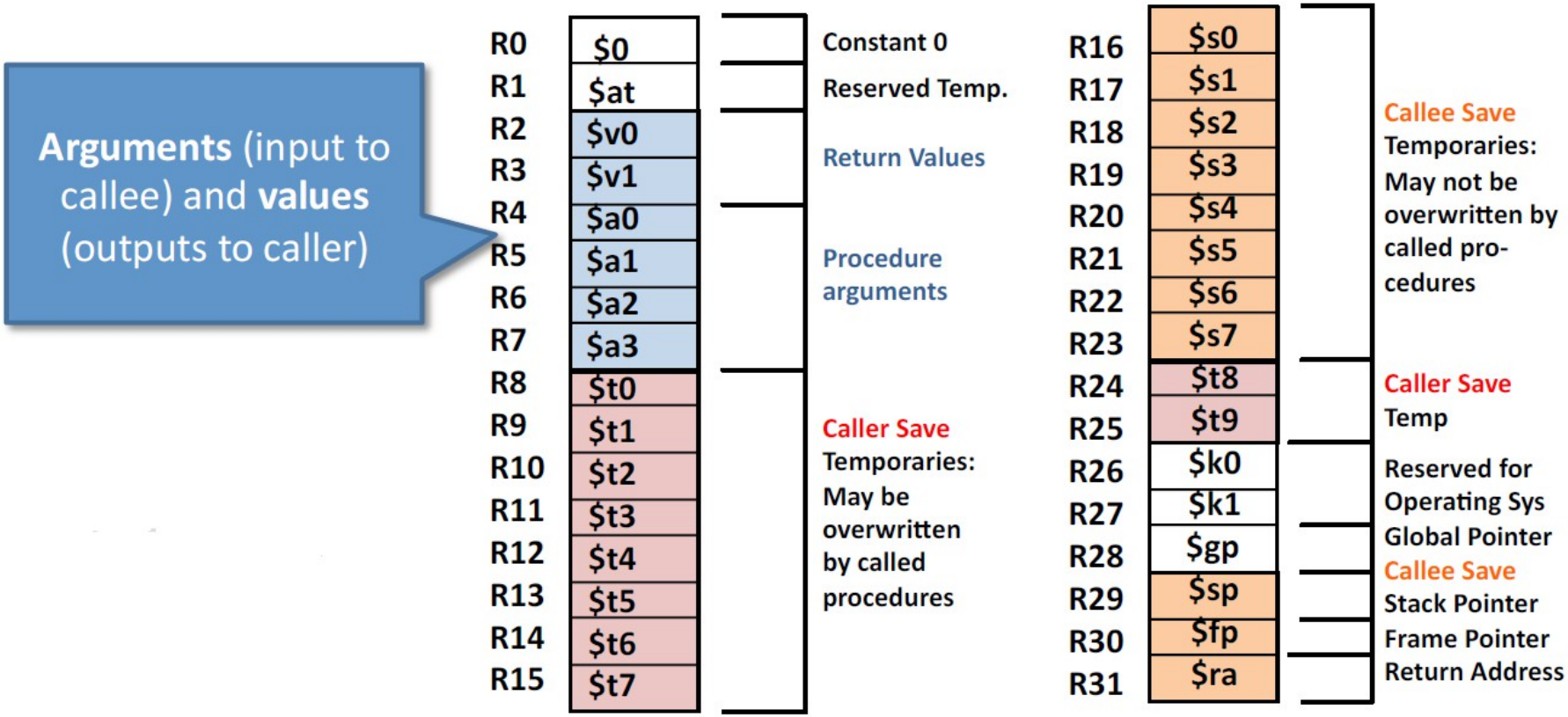
Question: Các thanh ghi **$s0 - $s8** và **$sp, $fp, $ra** là gì ?
Answer:
Là tên chuẩn cho các thanh ghi **R16** - **R23** và **R29**-**R31**.

• Quy ước trong **MIPS**
- Thống nhất theo “điều kiện" hoặc "giao thức” sau đó
- Xác định chính xác việc sử dụng và một số quy ước đặt tên
- Được thiết lập như là một phần của kiến trúc
- Được sử dụng bởi tất cả các trình biên dịch, chương trình, và các thư viện
- Đảm bảo khả năng tương thích

• **Callee lưu vào các thanh ghi sau đây, nếu sử dụng chúng**
– **$s0** - **$s7** (s=saved)
– **$sp, $fp,$ra**

• **Caller phải lưu vào bất kỳ thanh ghi nào cần sử dụng đến.**

Tên các thanh ghi MIPS và các quy ước



Làm thế nào để thực thi một thủ tục gọi

Chuyển đổi quyền điều khiển (transfer control) tới callee

jal Procedure Address ; nhảy và kết nối thủ tục (jump-and-link to the procedure)

– Địa chỉ trả về PC+4 được lưu trong **$ra**

Trả điều khiển (return control) tới caller:

jr $ra ; nhảy và trả về địa chỉ lưu trong $ra (jump-return to the address in $ra)

– Phải lưu lại địa chỉ quay về!

Quy ước thanh ghi cho thủ tục gọi:

– **$a0** - **$a3**: Các thanh ghi đối số (4)

– **$v0** - **$v1**: Các thanh ghi biến (2) cho kết quả trả về

– **$ra**: **Địa chỉ quay về**

Ví dụ

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

Địa chỉ	Chỉ thị lệnh
1000	add $a0, $s0, $zero # x = a
1004	add $a1, $s1, $zero # y = b
1008	addi $ra, $zero, 1016 # Lưu địa chỉ quay về
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 lệnh trong $ra

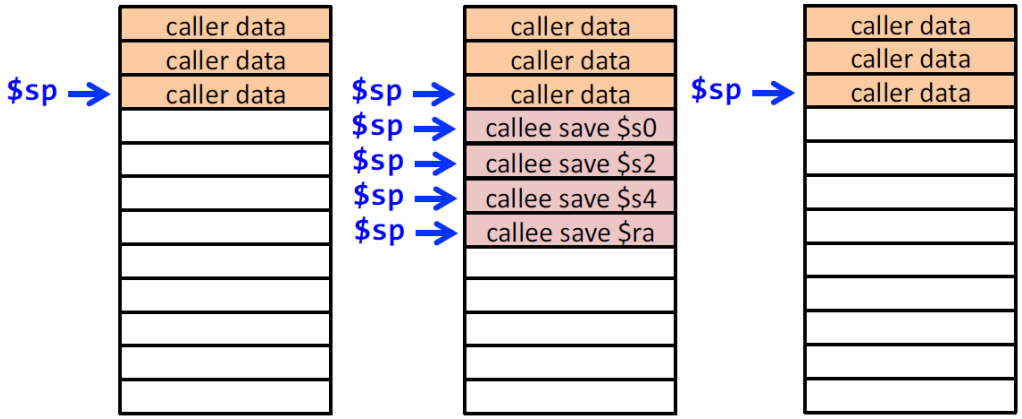
Địa chỉ	Chỉ thị lệnh
1000	add $a0, $s0, $zero # x = a
1004	add $a1, $s1, $zero # y = b
1008	jal sum # Lưu địa chỉ quay về vào $ra
1012	……..
2000	sum: add $v0, $a0,$a1 # Khai báo thủ tục sum
2004	jr $ra #nhảy tới địa chỉ lệnh trong $ra

Các ví dụ về thủ tục gọi hàm và ngăn xếp

Lưu trữ vào thanh ghi (trong ngăn xếp)

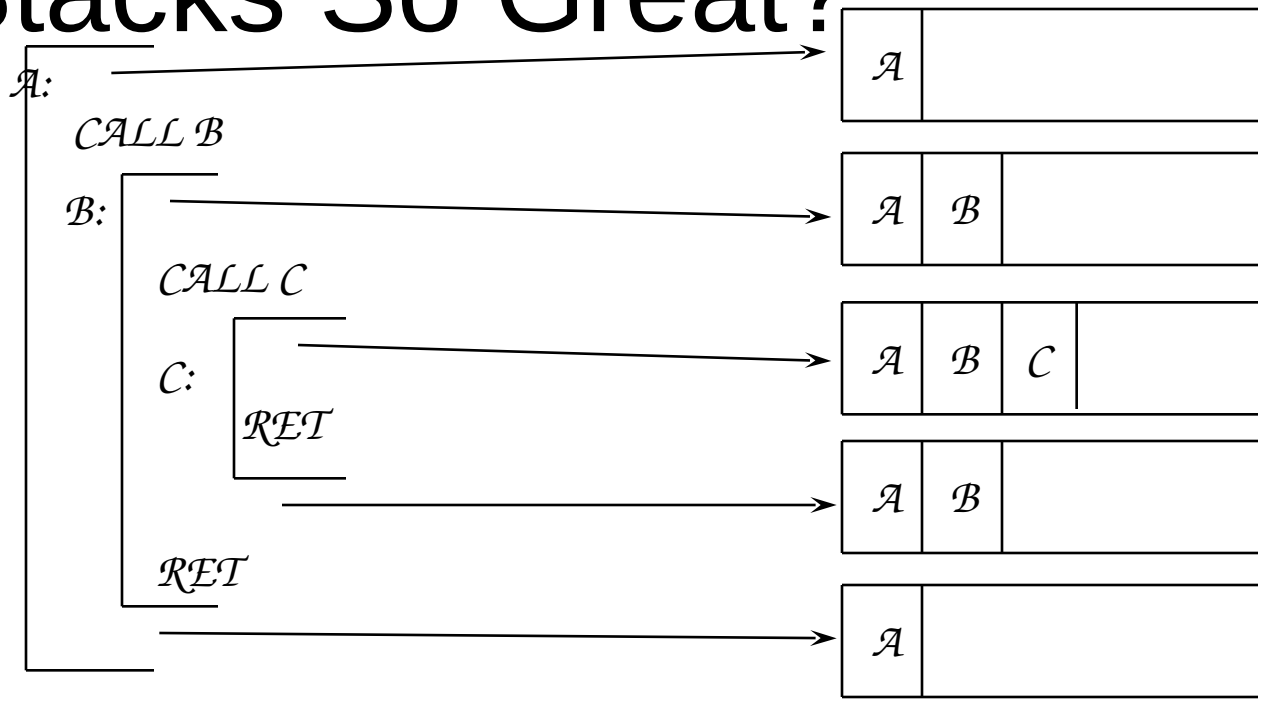
- Ngăn xếp là một phần của bộ nhớ lưu trữ dữ liệu tạm thời.
- Con trỏ ngăn xếp (Lưu trong $sp) trỏ tới điểm cuối cùng của ngăn xếp trong bộ nhớ
- Trong MIPS ngăn xếp đi từ trên xuống.
- Các thủ tục di chuyển con trỏ ngăn xếp khi chúng lưu dữ liệu trong ngăn xếp.
- Mỗi thủ tục quay lại ngăn xếp đến trạng thái trước khi được gọi.

Nếu thủ tục sử dụng nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi lưu trữ
1/ Sử dụng thêm nhiều thanh ghi hơn? Bao nhiêu thì đủ?
2/ Sử dụng ngăn xếp (stack)



Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



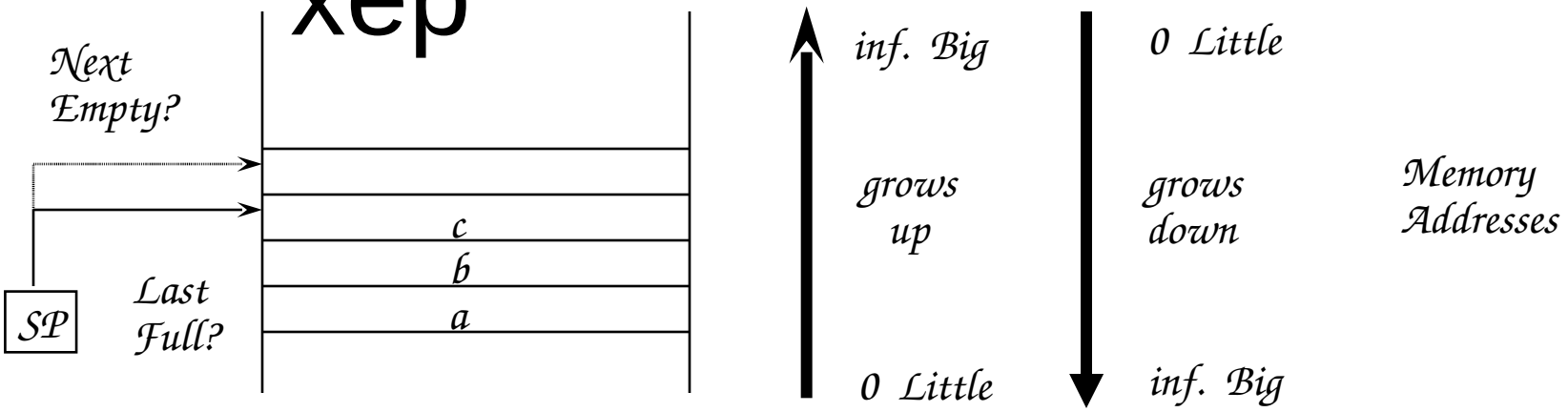
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Các bộ nhớ ngăn xếp

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

Little --> Big/Last Full

POP: *Read from Mem(SP)*
Decrement SP

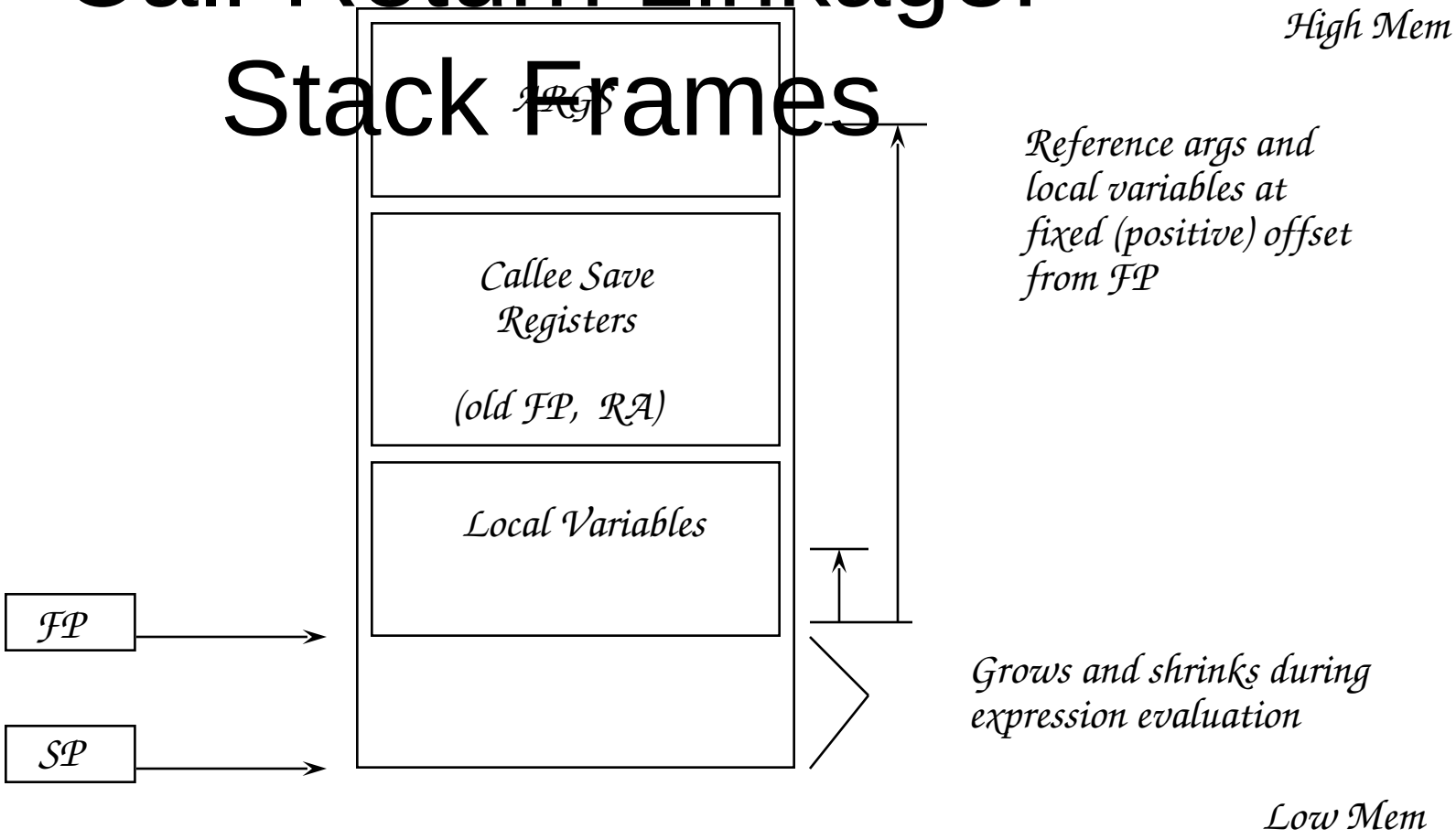
PUSH: *Increment SP*
Write to Mem(SP)

Little --> Big/Next Empty

POP: *Decrement SP*
Read from Mem(SP)

PUSH: *Write to Mem(SP)*
Increment SP

Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed /next)
- Compilers normally keep scalar variables

Các kiến trúc tập lệnh khác (ISAs)

Có rất nhiều kiến trúc tập lệnh khác nhau (ISAs):

- x86 (Intel/AMD)
- ARM (ARM)
- JVM (Java)
- PPC (IBM, Motorola)
- SPARC (Oracle, Fujitsu)
- PTX (Nvidia)
- etc.

Chú ý đến một số vấn đề chính :

- Các kiểu mã máy (Machine types)
- Phân loại các kiểu tập lệnh (ISA classes)
- Các chế độ đánh địa chỉ (Addressing modes)
- Độ lớn của chỉ thị (Instruction width)
- Phân biệt kiến trúc CISC vs. RISC

Phân loại tập lệnh cơ bản

- **Accumulator** (1 register)
  – 1 address **add A** **acc ← acc + mem[A]**

- **General purpose register file** (load/store)
  – 3 addresses **add Ra Rb Rc** **Ra ← Rb + Rc**
  load Ra Rb **Ra ← Mem[Rb]**

- **General purpose register file** (Register - Memory)
  – 2 address **add Ra B** **Ra ← Mem[B]**

- **Stack** (not a register file but an operand stack)
  – 0 address **add** **tos ← tos + next**
  tos = top of stack

- **Comparison**:
  – Bytes per instruction? Number of instructions? Cycles per instruction?

Các chế độ đánh địa chỉ

Question:

Why auto-increment/ decrement? Why scaled?

Answer:

Helpful for walking through arrays.

Addressing mode	Example	Meaning
Register	add R4, R3	R4←R4+R3
Immediate	add R4, 23	R4←R4+23
Displacement	add R4, 100(R1)	R4←R4+Mem[100+R1]
Register indirect	add R4, (R1)	R4←R4+Mem[R1]
Indexed/Base	add R3,(R1+R2)	R3←R3+Mem[R1+R2]
Direct or absolute	add R1,(1001)	R1←R1+Mem[1001]
Memory indirect	add R1,@(R3)	R1←R1+Mem[Mem[R3]]
Auto-increment	add R1,(R2)+	R1←R1+Mem[R2]; R2←R2+d
Auto-decrement	add R1,-(R2)	R2←R2-d; R1←R1+Mem[R2]
Scaled	add R1, 100(R2)[R3]	R1←R1+Mem[100+R2+R3*d]

MIPS

Tổng kết: Các kiến trúc tập lệnh

• **Architecture** = what's visible to the program about the machine
– Not everything in the implementation is "visible"
– The implementation may not follow the architecture
– The invisible stuff is the "microarchitecture" and it's very messy, but very fun (huge engineering challenges; lots of money)
• A big piece of the ISA is the **assembly language** structure
– Primitive instructions (appear to) execute **sequentially** and **atomically**
– Formats, computations, addressing modes, etc.
• **CISC**: lots of complicated instructions
• **RISC**: a few basic instructions
• All recent machines are RISC, but x86 is still CISC (although they do RISC tricks on the inside)