



KIẾN TRÚC MÁY TÍNH

KHOA HỌC & KỸ THUẬT MÁY TÍNH



Võ Tấn Phương

<http://www.cse.hcmut.edu.vn/~vtphuong>

Chapter 3

MIPS Instruction Set Architecture

Nội dung trình bày

❖ Kiến trúc tập lệnh (Instruction Set Architecture)

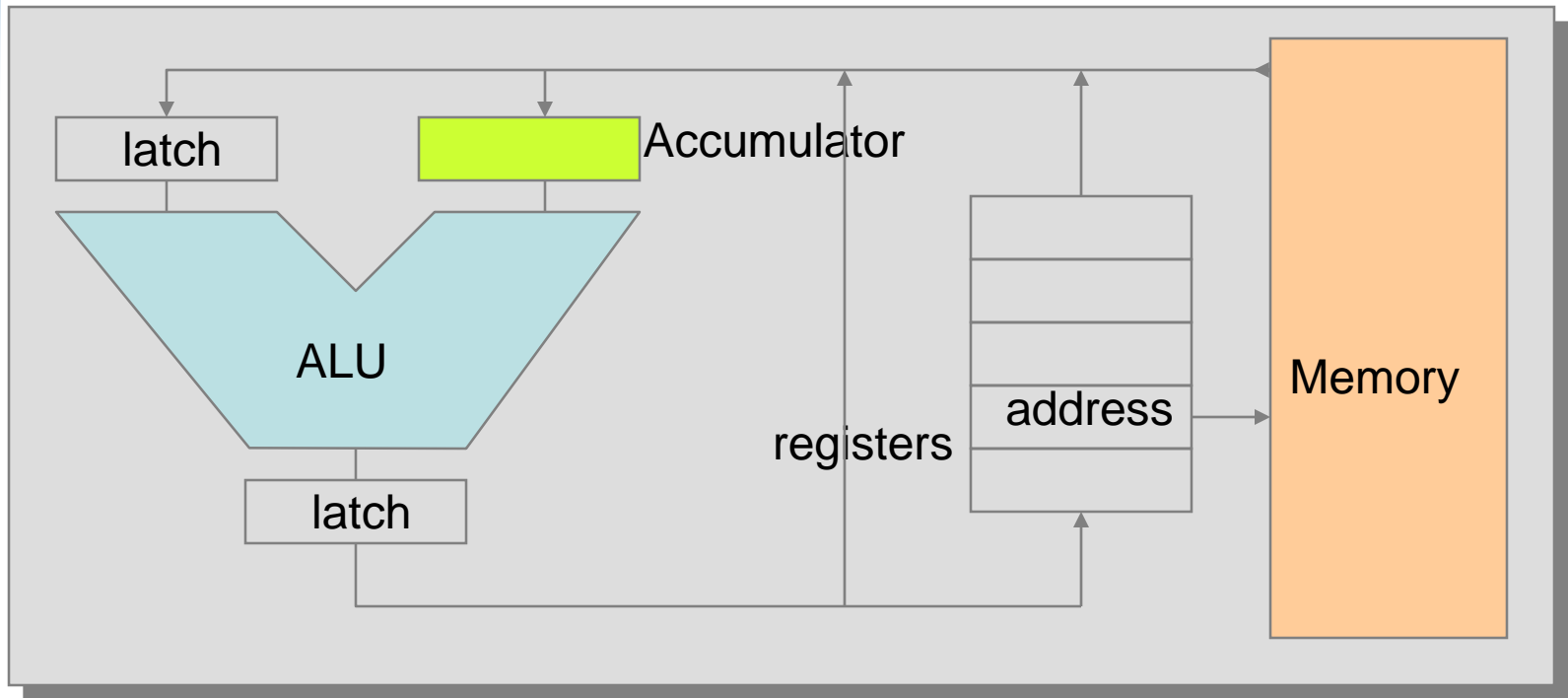
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Kiến trúc tập lệnh (ISA)

- ❖ Là **dao diện chính** giữa phần cứng và phần mềm, là **cái nhìn trừu tượng** của phần cứng trên quan điểm phần mềm
- ❖ Kiến trúc tập lệnh bao gồm...
 - ✧ Tập lệnh và định dạng lệnh
 - ✧ Kiểu dữ liệu, cách mã hóa và biểu diễn
 - ✧ Đối tượng lưu trữ: Thanh ghi (Registers) và bộ nhớ (Memory)
 - ✧ Các chế độ định địa chỉ để truy xuất lệnh và dữ liệu
 - ✧ Xử lý các điều kiện ngoại lệ (vd: chia cho 0)

❖ Ví dụ	(Phiên bản)	Năm giới thiệu
✧ Intel	(8086, 80386, Pentium, ...)	1978
✧ MIPS	(MIPS I, II, III, IV, V)	1986
✧ PowerPC	(601, 604, ...)	1993

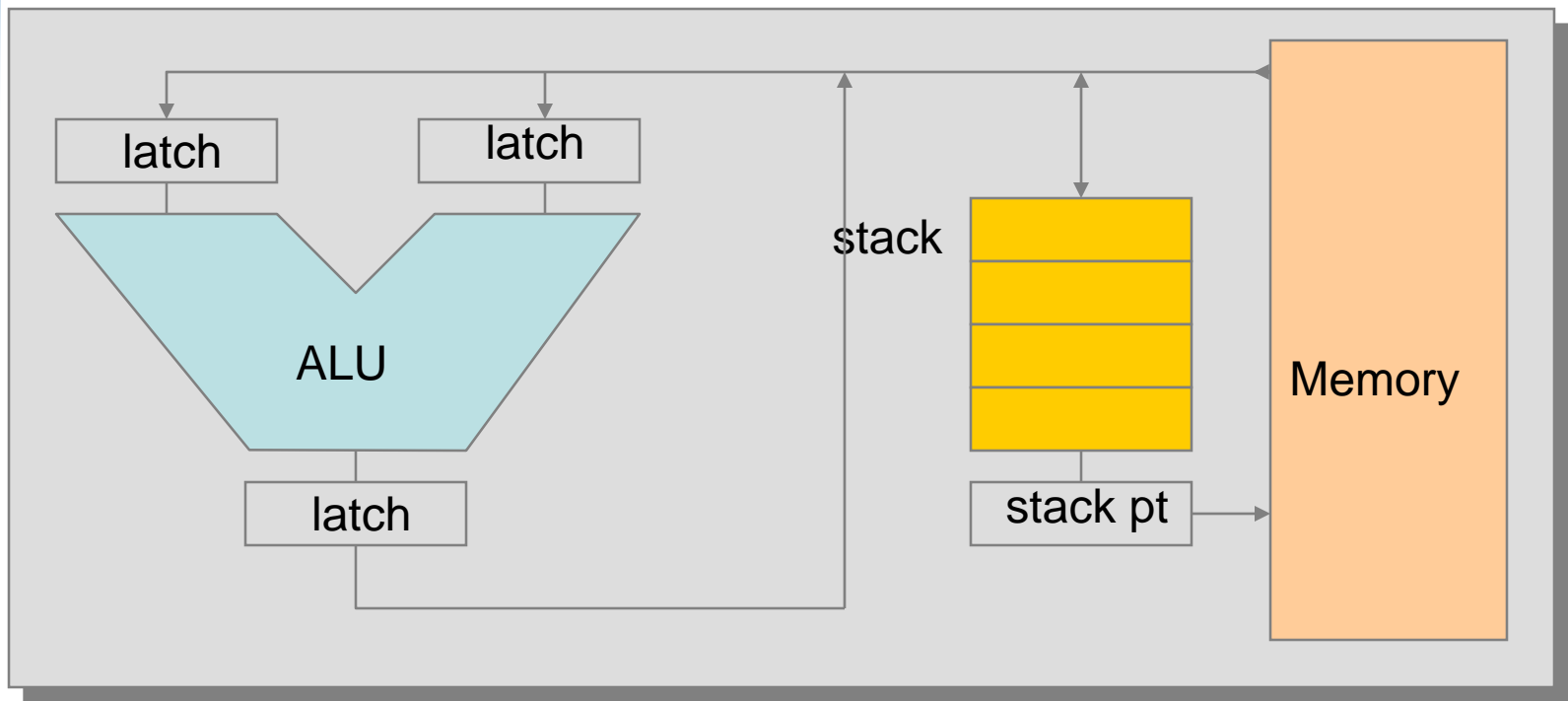
Kiến trúc thanh ghi tích lũy



Ví dụ lệnh: $a = b + c;$

```
load  b;    // accumulator is implicit operand
add   c;
store a;
```

Kiến trúc Stack



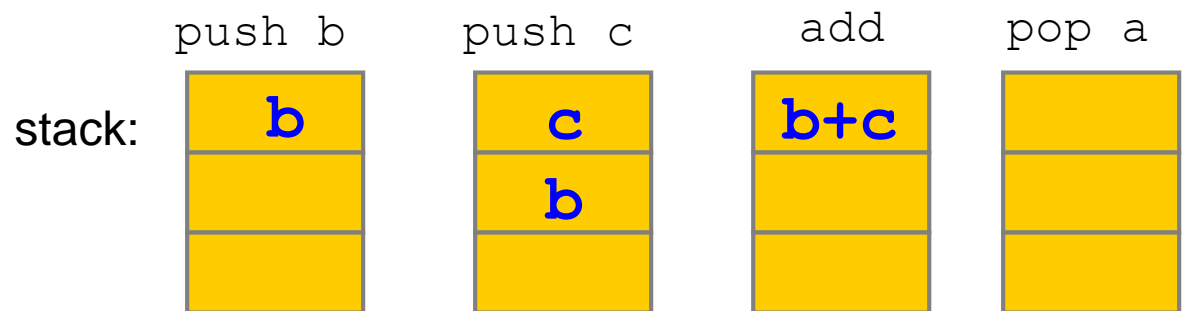
Ví dụ lệnh: $a = b + c;$

push b;

push c;

add;

pop a;



Các kiến trúc khác

Ví dụ lệnh $C = A + B$

Stack Architecture	Accumulator Architecture	Register-Memory	Memory-Memory	Register (load-store)
Push A	Load A	Load r1, A	Add C, B, 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

Bài tập: $C = A + B + 5$ chuyển sang lệnh dung kiến trúc Stack và thanh ghi tích lũy?

So sánh giữa các kiến trúc

❖ Thanh ghi tích lũy

- ✧ Một toán hạng (có thể là thanh ghi hoặc memory), thanh ghi tích lũy được sử dụng ngầm định

❖ Stack

- ✧ Không toán hạng: các toán hạng ngầm định trên đỉnh Stack (TOS)

❖ Register (load store)

- ✧ Ba toán hạng đều là thanh ghi
- ✧ Load & Store là các lệnh dành riêng cho việc truy xuất memory (truy xuất gián tiếp thông qua thanh ghi)

❖ Register-Memory

- ✧ Hai toán hạng, một là memory

❖ Memory-Memory

- ✧ Ba toán hạng, có thể tất cả là memory

Tập lệnh

- ❖ Tập lệnh là ngôn ngữ của bộ xử lý
- ❖ Kiến trúc tập lệnh MIPS được dùng trong môn học này
 - ✧ Loại: **Reduced Instruction Set Computer (RISC)**
 - ✧ Thiết kế đơn giản và tinh tế
 - ✧ Giống với kiến trúc RISC được phát triển giữa thập niên 80 đến thập niên 90
 - ✧ Rất phổ biến, được dùng bởi
 - Silicon Graphics, ATI, Cisco, Sony, etc.
 - ✧ Phổ biến sau bộ xử lý Intel IA-32
 - Gần 100 triệu bộ xử lý MIPS được bán trong năm 2002
- ❖ Ví dụ kiến trúc khác: Intel IA-32
 - ✧ Loại: **Complex Instruction Set Computer (CISC)**

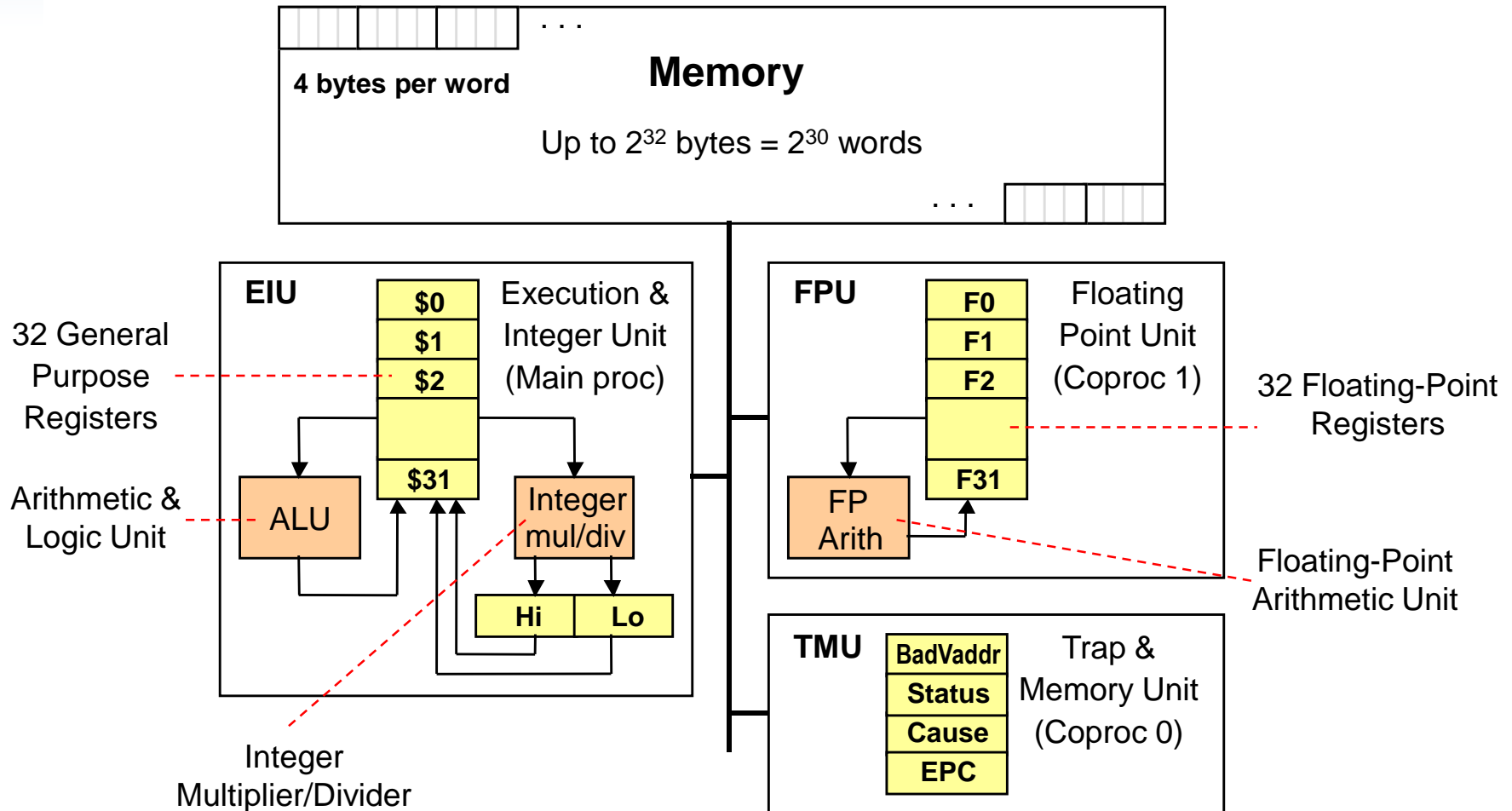
Ví dụ chương trình hợp ngữ MIPS

```
// In what follows R1,R2,R3 are registers, PC is program counter,  
// and addr is some value.  
  
ADD R1,R2,R3      //  $R1 \leftarrow R2 + R3$   
  
ADDI R1,R2,addr   //  $R1 \leftarrow R2 + \text{addr}$   
  
AND R1,R1,R2      //  $R1 \leftarrow R1 \text{ and } R2 \text{ (bit-wise)}$   
  
JMP addr          //  $PC \leftarrow \text{addr}$   
  
JEQ R1,R2,addr    // IF  $R1 == R2$  THEN  $PC \leftarrow \text{addr}$  ELSE  $PC++$   
  
LOAD R1, addr     //  $R1 \leftarrow \text{RAM}[\text{addr}]$   
  
STORE R1, addr    //  $\text{RAM}[\text{addr}] \leftarrow R1$   
  
NOP              // Do nothing  
  
// Etc. - some 50-300 command variants
```

Nội dung trình bày

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Sơ bộ kiến trúc MIPS



Bộ 32 thanh ghi đa mục đích MIPS

❖ 32 Thanh ghi đa dụng (General Purpose Registers)

✧ Sử dụng dấu \$ để biểu diễn thanh ghi

- \$0 là thanh ghi 0, \$31 là thanh ghi 31

✧ Tất cả thanh ghi là 32 bit MIPS32

✧ Thanh ghi \$0 luôn bằng 0

- Giá trị ghi vào thanh ghi \$0 được bỏ qua

❖ Quy ước tên tương ứng

✧ Mỗi thanh ghi có tên tương ứng

- Để chuẩn hóa mục đích sử dụng trong phần mềm

✧ Ví dụ: \$8 - \$15 tương ứng \$t0 - \$t7

- Được dùng cho các giá trị tạm (**temporary**)

\$0 = \$zero	\$16 = \$s0
\$1 = \$at	\$17 = \$s1
\$2 = \$v0	\$18 = \$s2
\$3 = \$v1	\$19 = \$s3
\$4 = \$a0	\$20 = \$s4
\$5 = \$a1	\$21 = \$s5
\$6 = \$a2	\$22 = \$s6
\$7 = \$a3	\$23 = \$s7
\$8 = \$t0	\$24 = \$t8
\$9 = \$t1	\$25 = \$t9
\$10 = \$t2	\$26 = \$k0
\$11 = \$t3	\$27 = \$k1
\$12 = \$t4	\$28 = \$gp
\$13 = \$t5	\$29 = \$sp
\$14 = \$t6	\$30 = \$fp
\$15 = \$t7	\$31 = \$ra

Quy ước tên gọi bộ thanh ghi MIPS

- ❖ **Assembler tham khảo thanh ghi bằng tên hoặc số**
 - ✧ Lập trình viên thường dùng thanh ghi theo tên
 - ✧ Assembler chuyển tham khảo từ tên sang số

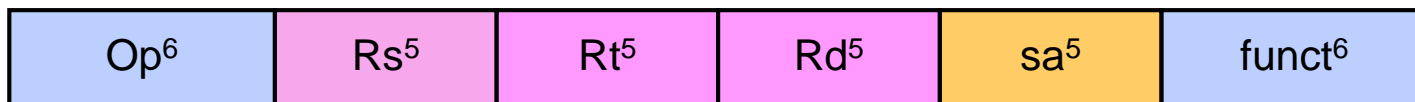
Name	Register	Usage
\$zero	\$0	Always 0 (forced by hardware)
\$at	\$1	Reserved for assembler use
\$v0 – \$v1	\$2 – \$3	Result values of a function
\$a0 – \$a3	\$4 – \$7	Arguments of a function
\$t0 – \$t7	\$8 – \$15	Temporary Values
\$s0 – \$s7	\$16 – \$23	Saved registers (preserved across call)
\$t8 – \$t9	\$24 – \$25	More temporaries
\$k0 – \$k1	\$26 – \$27	Reserved for OS kernel
\$gp	\$28	Global pointer (points to global data)
\$sp	\$29	Stack pointer (points to top of stack)
\$fp	\$30	Frame pointer (points to stack frame)
\$ra	\$31	Return address (used by jal for function call)

Ba định dạng lệnh của MIPS ISA

❖ Tất cả các lệnh đều có độ dài 32-bit, có 3 loại:

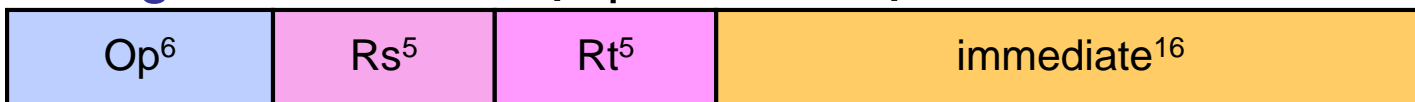
❖ Register (R-Type)

- ✧ Các lệnh sử dụng 2 toán hạng là giá trị 2 thanh ghi và lưu kết quả vào thanh ghi
- ✧ Op: mã phép toán quy định lệnh



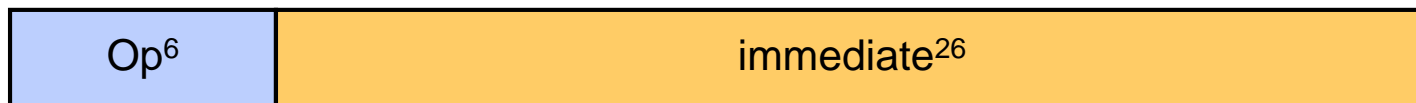
❖ Immediate (I-Type)

- ✧ Hằng số 16-bit là một phần của lệnh



❖ Jump (J-Type)

- ✧ Các lệnh nhảy có định dạng J-Type



Phân loại lệnh trong tập lệnh – nhóm lệnh

❖ Các lệnh số học nguyên (Integer Arithmetic)

- ✧ Các lệnh cộng/trừ, lệnh luận lý (and, or, nor, xor) và lệnh dịch (shift left, shift right)

❖ Các lệnh truy xuất dữ liệu từ bộ nhớ (Data Transfer)

- ✧ Lệnh Load&Store tương ứng thao tác Đọc/Ghi
- ✧ Hỗ trợ dữ liệu byte (1byte), half word (2byte), word (4byte)

❖ Nhảy và rẽ nhánh (Jump and Branch)

- ✧ Các lệnh điều khiển dòng thực thi khác cách tuần tự

❖ Các lệnh số học số thực (Floating Point Arithmetic)

- ✧ Các lệnh thao tác trên các thanh ghi số thực

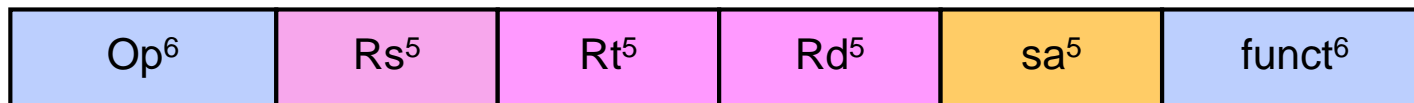
❖ Các lệnh phụ

- ✧ Các lệnh hỗ trợ xử lý ngoại lệ (exceptions)
- ✧ Các lệnh quản lý bộ nhớ

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

R-Type Format



❖ **Op**: mã phép toán (opcode)

✧ Cho biết lệnh làm phép toán gì

❖ **funct**: function code – mở rộng opcode

✧ Có thể có $2^6 = 64$ functions có thể mở rộng cho một opcode

✧ MIPS sử dụng **opcode 0** để định nghĩa lệnh **loại R-type**

❖ Ba thanh ghi toán hạn (Register Operand)

✧ **Rs, Rt**: Hai toán hạn nguồn

✧ **Rd**: Toán hạn đích chứa kết quả

✧ **sa**: Quy định số bit dịch trong các lệnh dịch

Các lệnh Cộng/Trừ số nguyên

Instruction	Meaning	R-Type Format						
add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x20	
addu \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x21	
sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x22	
subu \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x23	

- ❖ **add & sub**: “tràn” (overflow) sinh ra **arithmetic exception**
 - ✧ Trong trường hợp “tràn”, kết quả không được ghi vào thanh ghi đích
- ❖ **addu & subu**: hoạt động giống **add & sub**
 - ✧ Tuy nhiên các toán hạn được hiểu là số nguyên không dấu => không bị “tràn” (không xảy ra arithmetic exception)
 - ✧ **Cờ tràn không xét đến**
- ❖ Nhiều ngôn ngữ lập trình bỏ qua “tràn”:
 - ✧ Phép toán **+** được dịch thành **addu**
 - ✧ Phép toán **–** được dịch thành **subu**

Số nguyên nhị phân không dấu

❖ Cho 1 số n-bit, có dạng

$$X = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Tầm vực giá trị sẽ là: 0 đến $+2^n - 1$
- Ví dụ:
 - 0000 0000 0000 0000 0000 0000 0000 1011₂
= $0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
= $0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$
- Giá trị 1 số nhị phân không dấu 32-bit sẽ là:
 - 0 đến +4,294,967,295 (giá trị thập phân)

Số nguyên có dấu dạng bù 2

❖ Cho 1 số n-bit như sau:

$$X = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

■ Tầm giá trị: $-2^{(n-1)}$ đến $+2^{(n-1)} - 1$

■ Ví dụ:

$$\begin{aligned} & \blacksquare 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ & \quad = -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ & \quad = -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

■ Giá trị 1 số nhị phân có dấu 32-bit sẽ là

$$\blacksquare -2,147,483,648 \text{ đến } +2,147,483,647$$

Số âm có dấu

❖ Đảo giá trị bit và cộng 1

✧ Đảo giá trị bit: $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

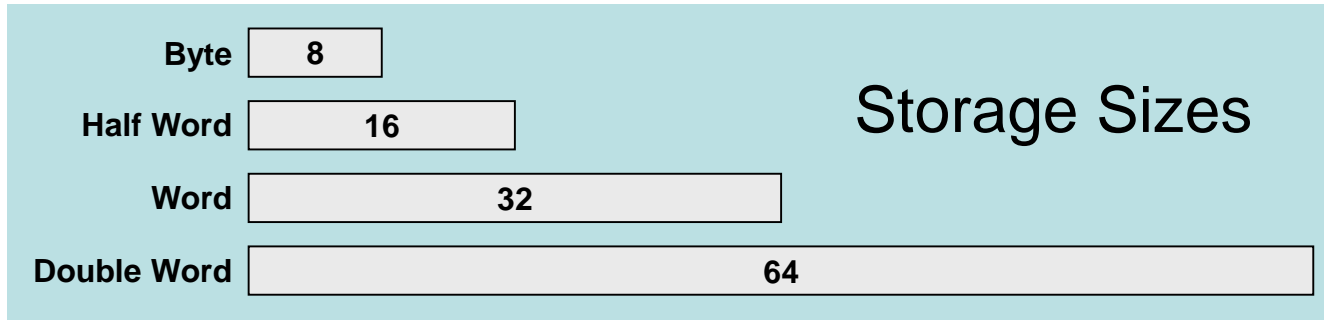
$$\bar{x} + 1 = -x$$

■ Ví dụ: giá trị (-) 2

$$\blacksquare +2 = 0000 \ 0000 \ \dots \ 0010_2$$

$$\begin{aligned} \blacksquare -2 &= 1111 \ 1111 \ \dots \ 1101_2 + 1 \\ &= 1111 \ 1111 \ \dots \ 1110_2 \end{aligned}$$

Tầm biểu diễn của số nguyên không dấu



Storage Type	Unsigned Range	Powers of 2
Byte	0 to 255	0 to $(2^8 - 1)$
Half Word	0 to 65,535	0 to $(2^{16} - 1)$
Word	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Double Word	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Số nguyên không dấu 20-bit lớn nhất?

Lời giải: $2^{20} - 1 = 1,048,575$

Tầm biểu diễn của số nguyên có dấu

Số nguyên có dấu n -bit: Tầm biểu diễn từ -2^{n-1} đến $(2^{n-1} - 1)$

Các số dương: 0 đến $2^{n-1} - 1$

Các số âm: -2^{n-1} đến -1

Storage Type	Unsigned Range	Powers of 2
Byte	-128 to +127	-2^7 to $(2^7 - 1)$
Half Word	-32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Word	-2,147,483,648 to +2,147,483,647	-2^{31} to $(2^{31} - 1)$
Double Word	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Câu hỏi: Cho biết tầm biểu diễn của số nguyên có dấu 20 bits?

Nhớ và tràn (Carry vs Overflow)

- ❖ Giá trị “nhớ” quan trọng khi...
 - ✧ Cộng hoặc trừ số nguyên không dấu
 - ✧ Báo tổng dạng không dấu bị ngoài tầm biểu diễn
 - ✧ Xảy ra khi < 0 hoặc $> \text{maximum}$ giá trị không dấu n -bit
- ❖ Giá trị “tràn” quan trọng khi ...
 - ✧ Cộng hoặc trừ số nguyên có dấu
 - ✧ Báo tổng dạng có dấu bị ngoài tầm biểu diễn
- ❖ Tràn số xảy ra khi
 - ✧ Cộng hai số dương được tổng là số âm
 - ✧ Cộng hai số âm được tổng là số dương

Ví dụ về Nhớ/Tràn

- ❖ “Nhớ” và “Tràn” là độc lập với nhau
- ❖ Có bốn trường hợp (Ví dụ số 8-bit)

				1				
	0	0	0	0	1	1	1	1
15								
+	0	0	0	0	1	0	0	0
8								
<hr/>								
	0	0	0	1	0	1	1	1
23								
Carry = 0 Overflow = 0								

1	1	1	1	1				
	0	0	0	0	1	1	1	1
								15
+	1	1	1	1	1	0	0	0
248 (-8)								
<hr/>								
	0	0	0	0	0	1	1	1
7								
Carry = 1 Overflow = 0								

				1				
	0	1	0	0	1	1	1	1
79								
+	0	1	0	0	0	0	0	0
64								
<hr/>								
	1	0	0	0	1	1	1	1
143 (-113)								
Carry = 0 Overflow = 1								

1				1	1			
	1	1	0	1	1	0	1	0
								218 (-38)
+	1	0	0	1	1	1	0	1
157 (-99)								
<hr/>								
	0	1	1	1	0	1	1	1
119								
Carry = 1 Overflow = 1								

Ví dụ về phép Cộng/Trừ

- ❖ Chuyển biểu thức sau sang hợp ngữ MIPS: $f = (g+h) - (i+j)$
- ❖ Các biến được biểu diễn bằng các thanh ghi
 - ✧ Giả sử f, g, h, i và j là các thanh ghi từ $\$s0$ đến $\$s4$
- ❖ Kết quả dịch: $f = (g+h) - (i+j)$

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

 - ✧ Kết quả tạm sử dụng thanh ghi $\$t0 = \8 và $\$t1 = \9
- ❖ Dịch: **addu \$t0, \$s1, \$s2** sang mã máy

- ❖ Lời giải:

op	rs = \$s1	rt = \$s2	rd = \$t0	sa	func
000000	10001	10010	01000	00000	100001

Các phép toán luận lý

❖ Xét 4 phép toán: **and**, **or**, **xor**, **nor**

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x or y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

❖ AND có tính chất xóa: **$x \text{ and } 0 = 0$**

❖ OR có tính chất tạo: **$x \text{ or } 1 = 1$**

❖ XOR có tính chất đảo: **$x \text{ xor } 1 = \text{not } x$**

❖ NOR có thể dùng như NOT:

✧ **$x \text{ nor } x$** tương ứng **$\text{not } x$**

Instruction	Meaning	R-Type Format					
and \$s1, \$s2, \$s3	$\$s1 = \$s2 \& \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x24
or \$s1, \$s2, \$s3	$\$s1 = \$s2 \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x25
xor \$s1, \$s2, \$s3	$\$s1 = \$s2 \wedge \$s3$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x26
nor \$s1, \$s2, \$s3	$\$s1 = \sim(\$s2 \$s3)$	op = 0	rs = \$s2	rt = \$s3	rd = \$s1	sa = 0	f = 0x27

❖ Ví dụ:

Giả sử $\$s1 = 0xabcd1234$ và $\$s2 = 0xffff0000$

and \$s0, \$s1, \$s2 # \$s0 = 0xabcd0000

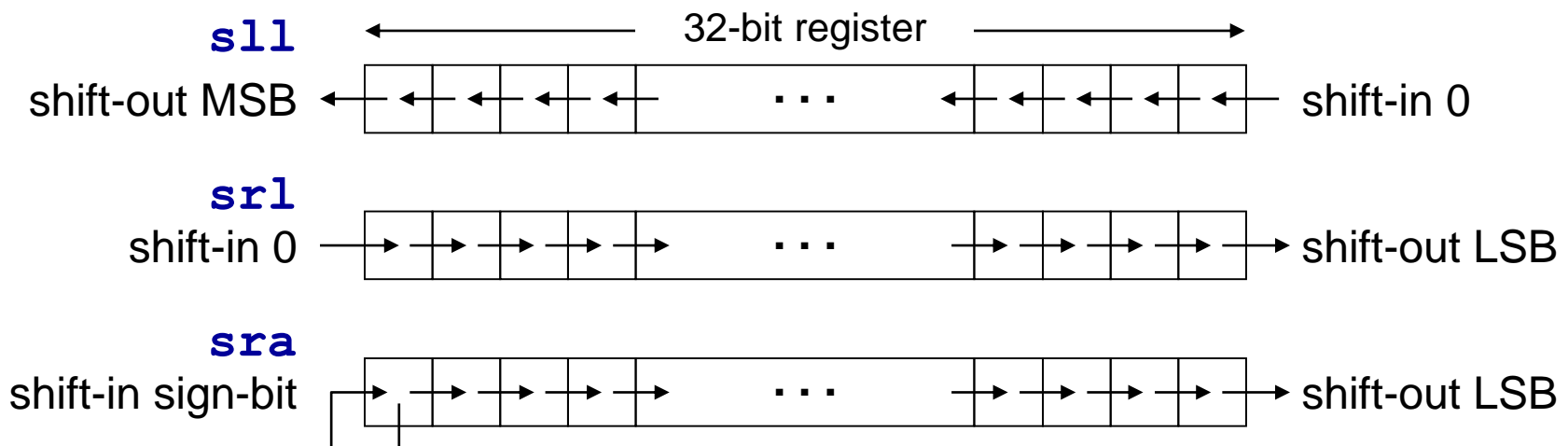
or \$s0, \$s1, \$s2 # \$s0 = 0xffff1234

xor \$s0, \$s1, \$s2 # \$s0 = 0x54321234

nor \$s0, \$s1, \$s2 # \$s0 = 0x0000edcb

Phép dịch (Shift Operation)

- ❖ Phép dịch thực hiện di chuyển tất cả các bit của một thanh ghi sang trái hoặc phải
- ❖ Có 3 lệnh dịch số lượng bit cố định: **sll**, **srl**, **sra**
 - ✧ **sll/srl** tương ứng **shift left/right logical** (số không dấu)
 - ✧ Trường “sa” (**5-bit shift amount**) chỉ số lượng bit được dịch
 - ✧ **sra** tương ứng **shift right arithmetic** (bảo toàn dấu)
 - ✧ Bit dấu (**sign-bit**) được thêm vào từ bên trái



Instruction		Meaning	R-Type Format					
sll	\$s1,\$s2,10	$\$s1 = \$s2 \ll 10$	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 0
srl	\$s1,\$s2,10	$\$s1 = \$s2 \gg 10$	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 2
sra	\$s1, \$s2, 10	$\$s1 = \$s2 \gg 10$	op = 0	rs = 0	rt = \$s2	rd = \$s1	sa = 10	f = 3
sllv	\$s1,\$s2,\$s3	$\$s1 = \$s2 \ll \$s3$	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 4
srlv	\$s1,\$s2,\$s3	$\$s1 = \$s2 \gg \$s3$	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 6
srav	\$s1,\$s2,\$s3	$\$s1 = \$s2 \gg \$s3$	op = 0	rs = \$s3	rt = \$s2	rd = \$s1	sa = 0	f = 7

❖ Dịch với số lượng bit thay đổi (**variable**): **sllv**, **srlv**, **srav**

✧ Giống **sll**, **srl**, **sra**, nhưng số lượng bit dịch chứa trong thanh ghi

❖ Ví dụ: giả sử $\$s2 = 0xabcd1234$, $\$s3 = 16$

sll $\$s1, \$s2, 8$ $\$s1 = \$s2 \ll 8$ $\$s1 = 0xcd123400$

sra $\$s1, \$s2, 4$ $\$s1 = \$s2 \gg 4$ $\$s1 = 0xfabcd123$

srlv $\$s1, \$s2, \$s3$ $\$s1 = \$s2 \gg \$s3$ $\$s1 = 0x0000abcd$

op=000000	rs=\$s3=10011	rt=\$s2=10010	rd=\$s1=10001	sa=00000	f=000110
-----------	---------------	---------------	---------------	----------	----------

Phép nhân dựa trên phép dịch

- ❖ Lệnh dịch trái (**sll**) có thể thực hiện phép nhân
 - ✧ Khi số nhân là mũ của 2
- ❖ Một số nguyên bất kỳ có thể biểu diễn thành tổng của các số là mũ của 2
 - ✧ Ví dụ: nhân **\$s1** với 36
 - Phân tích 36 thành $(4 + 32)$ và sử dụng tính chất phân phối của phép nhân
 - ✧ $\text{\$s2} = \text{\$s1} * 36 = \text{\$s1} * (4 + 32) = \text{\$s1} * 4 + \text{\$s1} * 32$

```
sll  $t0, $s1, 2      ; $t0 = $s1 * 4
sll  $t1, $s1, 5      ; $t1 = $s1 * 32
addu $s2, $t0, $t1    ; $s2 = $s1 * 36
```


Câu hỏi. . .

Nhân \$s1 với 26, sử dụng lệnh dịch và cộng

Gợi ý: $26 = 2 + 8 + 16$

```
sll    $t0, $s1, 1        ; $t0 = $s1 * 2
sll    $t1, $s1, 3        ; $t1 = $s1 * 8
addu   $s2, $t0, $t1      ; $s2 = $s1 * 10
sll    $t0, $s1, 4        ; $t0 = $s1 * 16
addu   $s2, $s2, $t0      ; $s2 = $s1 * 26
```

Nhân \$s1 với 31, Gợi ý: $31 = 32 - 1$

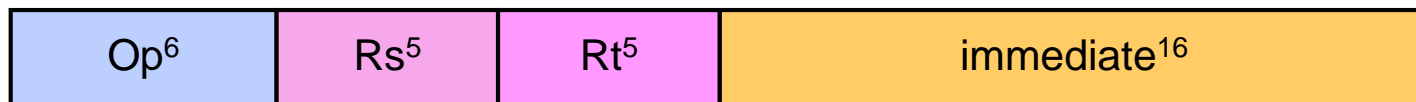
```
sll    $s2, $s1, 5        ; $s2 = $s1 * 32
subu   $s2, $s2, $s1      ; $s2 = $s1 * 31
```

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Định dạng lệnh I-Type

- ❖ Hằng số được sử dụng thường xuyên trong chương trình
 - ✧ Lệnh dịch thuộc R-type chứa sẵn **5-bit dịch trong lệnh (sa)**
 - ✧ Vậy các lệnh khác, ví dụ: $i = i + 1$; ?
- ❖ I-Type: Lệnh chứa hằng số (immediate)



- ❖ Hằng số **16-bit** được lưu trong lệnh
 - ✧ Rs thanh ghi toán hạn nguồn
 - ✧ Rt thanh ghi đích (**destination**)
- ❖ Ví dụ lệnh ALU kiểu I-Type:
 - ✧ Add immediate: **addi \$s1, \$s2, 5 # \$s1 = \$s2 + 5**
 - ✧ OR immediate: **ori \$s1, \$s2, 5 # \$s1 = \$s2 | 5**

Các lệnh số học luận lý (ALU) kiểu I-Type

Instruction	Meaning	I-Type Format				
addi \$s1, \$s2, 10	$\$s1 = \$s2 + 10$	op = 0x8	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
addiu \$s1, \$s2, 10	$\$s1 = \$s2 + 10$	op = 0x9	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
andi \$s1, \$s2, 10	$\$s1 = \$s2 \& 10$	op = 0xc	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
ori \$s1, \$s2, 10	$\$s1 = \$s2 10$	op = 0xd	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
xori \$s1, \$s2, 10	$\$s1 = \$s2 \wedge 10$	op = 0xe	rs = \$s2	rt = \$s1	imm ¹⁶ = 10	
lui \$s1, 10	$\$s1 = 10 \ll 16$	op = 0xf	0	rt = \$s1	imm ¹⁶ = 10	

❖ **addi**: overflow sinh ra **arithmetic exception**

✧ Trong trường hợp overflow, kết quả không được ghi vào thanh ghi đích

❖ **addiu**: giống lệnh **addi** nhưng **trần số được bỏ qua**

❖ Hằng số 16 bit trong lệnh **addi** và **addiu** là **số có dấu**

✧ Không có lệnh **subi** hoặc **subiu**

❖ Hằng số 16 bit trong lệnh **andi**, **ori**, **xori** là **số không dấu**

Ví dụ: Các lệnh ALU kiểu I-Type

❖ Ví dụ: giả sử **A, B, C** được ánh xạ vào **\$s0, \$s1, \$s2**

A = B+5; chuyển thành **addiu \$s0,\$s1,5**

C = B-1; chuyển thành **addiu \$s2,\$s1,-1**



op=001001	rs=\$s1=10001	rt=\$s2=10010	imm = -1 = 1111111111111111
-----------	---------------	---------------	-----------------------------

A = B&0xf; chuyển thành **andi \$s0,\$s1,0xf**

C = B|0xf; chuyển thành **ori \$s2,\$s1,0xf**

C = 5; chuyển thành **ori \$s2,\$zero,5**

A = B; chuyển thành **ori \$s0,\$s1,0**

❖ Tại sao không có lệnh **subi**? (lệnh **addi** dùng hằng số 16 bit **số có dấu**)

❖ Thanh ghi 0 (**\$zero**) giá trị luôn bằng 0

Khởi tạo hằng số 32-bit???

- ❖ Lệnh I-Type chỉ chứa hằng số 16-bit



- ❖ Làm thế nào để khởi tạo giá trị 32-bit cho một thanh ghi?
- ❖ Không thể có giá trị 32-bit trong lệnh I-Type ☹
- ❖ Gợi ý: dùng nhiều lệnh thay vì một lệnh ☺

✧ Ví dụ: khởi tạo `$s1=0xAC5165D9` (hằng số 32-bit)

```
addiu $s1,$0,0xAC51
sll   $s1,$s1,16
ori   $s1,$s1,0x65D9
```

lui: load upper immediate

```
lui $s1,0xAC51
```

```
ori $s1,$s1,0x65D9
```

	load upper 16 bits	clear lower 16 bits
<code>\$s1=\$17</code>	0xAC51	0x0000
<code>\$s1=\$17</code>	0xAC51	0x65D9

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lặp và duyệt mảng
- ❖ Các chế độ định địa chỉ

Định dạng lệnh J-Type



- ❖ Định dạng J-type áp dụng cho các lệnh nhảy không điều kiện (unconditional jump, giống như lệnh goto):

```
j    label    # jump to label
```

```
    . . .
```

```
label:
```

- ❖ Hằng số 26-bit được gắn vào trong lệnh
 - ✧ Hằng số này cho biết địa chỉ nhảy đến
- ❖ Thanh ghi Program Counter (PC) được thay đổi như sau:

✧ Next PC =

PC ⁴	immediate ²⁶	00
-----------------	-------------------------	----

least-significant
2 bits are 00

✧ 4 bit cao nhất của PC không đổi

✧ 2 bit cuối luôn bằng 00

Các lệnh rẽ nhánh có điều kiện

- ❖ Các lệnh **so sánh và rẽ nhánh (branch)** của MIPS:

beq Rs,Rt,label nhảy tới **label** if (**Rs == Rt**)

bne Rs,Rt,label nhảy tới **label** if (**Rs != Rt**)

- ❖ Các lệnh **so sánh với zero và rẽ nhánh (branch)** của MIPS

Các lệnh so sánh với zero sử dụng rất nhiều trong chương trình

bltz Rs,label nhảy tới **label** if (**Rs < 0**)

bgtz Rs,label nhảy tới **label** if (**Rs > 0**)

blez Rs,label nhảy tới **label** if (**Rs <= 0**)

bgez Rs,label nhảy tới **label** if (**Rs >= 0**)

- ❖ Không cần lệnh **beqz** & **bnez**. Tại sao?

Các lệnh Set on Less Than

- ❖ MIPS cung cấp lệnh **gán bằng 1 khi nhỏ hơn (set on less than)**

slt **rd,rs,rt** if (rs < rt) rd = 1 else rd = 0

sltu **rd,rs,rt** **unsigned <**

slti **rt,rs,im¹⁶** if (rs < im¹⁶) rt = 1 else rt = 0

sltiu **rt,rs,im¹⁶** **unsigned <**

- ❖ So sánh **có dấu / không dấu (Signed / Unsigned)**

Có thể sinh ra các kết quả **khác nhau**

Giả sử **\$s0 = 1** và **\$s1 = -1 = 0xffffffff**

slt **\$t0,\$s0,\$s1** kết quả **\$t0 = 0**

sltu **\$t0,\$s0,\$s1** kết quả **\$t0 = 1**

Các lệnh rẽ nhánh khác

- ❖ Phần cứng MIPS KHÔNG cung cấp các lệnh rẽ nhánh ...

blt, bltu branch if less than (signed/unsigned)

ble, bleu branch if less or equal (signed/unsigned)


bgt, bgtu branch if greater than (signed/unsigned)

bge, bgeu branch if greater or equal (signed/unsigned)

Có thể thực hiện bằng **2 lệnh**

- ❖ Thực hiện?


- ❖ Lời giải:



```
blt $s0,$s1,label  
slt $at,$s0,$s1  
bne $at,$zero,label
```

- ❖ Thực hiện?

- ❖ Lời giải:



```
ble $s2,$s3,label  
slt $at,$s3,$s2  
beq $at,$zero,label
```

Các lệnh giả Pseudo-Instructions

- ❖ Cung cấp bởi assembler và được tự động chuyển đổi sang lệnh có thật
 - ✧ Mục đích để hỗ trợ lập trình hợp ngữ được dễ dàng

Pseudo-Instructions	Các lệnh Thật tương ứng
<code>move \$s1, \$s2</code>	<code>addu \$s1, \$s2, \$zero</code>
<code>not \$s1, \$s2</code>	<code>nor \$s1, \$s2, \$s2</code>
<code>li \$s1, 0xabcd</code>	<code>ori \$s1, \$zero, 0xabcd</code>
<code>li \$s1, 0xabcd1234</code>	<code>lui \$s1, 0xabcd</code> <code>ori \$s1, \$s1, 0x1234</code>
<code>sgt \$s1, \$s2, \$s3</code>	<code>slt \$s1, \$s3, \$s2</code>
<code>blt \$s1, \$s2, label</code>	<code>slt \$at, \$s1, \$s2</code> <code>bne \$at, \$zero, label</code>

- ❖ Assembler dùng thanh ghi `$at` = \$1 trong các chuyển đổi
 - ✧ `$at` được gọi là thanh ghi **assembler temporary**

Các lệnh Jump, Branch và SLT

Instruction		Meaning	Format			
j	label	jump to label	$op^6 = 2$	imm^{26}		
beq	rs, rt, label	branch if (rs == rt)	$op^6 = 4$	rs^5	rt^5	imm^{16}
bne	rs, rt, label	branch if (rs != rt)	$op^6 = 5$	rs^5	rt^5	imm^{16}
blez	rs, label	branch if (rs ≤ 0)	$op^6 = 6$	rs^5	0	imm^{16}
bgtz	rs, label	branch if (rs > 0)	$op^6 = 7$	rs^5	0	imm^{16}
bltz	rs, label	branch if (rs < 0)	$op^6 = 1$	rs^5	0	imm^{16}
bgez	rs, label	branch if (rs ≥ 0)	$op^6 = 1$	rs^5	1	imm^{16}

Instruction		Meaning	Format					
slt	rd, rs, rt	$rd = (rs < rt ? 1 : 0)$	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
sltu	rd, rs, rt	$rd = (rs < rt ? 1 : 0)$	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2b
slti	rt, rs, imm^{16}	$rt = (rs < imm ? 1 : 0)$	0xa	rs^5	rt^5	imm^{16}		
sltiu	rt, rs, imm^{16}	$rt = (rs < imm ? 1 : 0)$	0xb	rs^5	rt^5	imm^{16}		

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý, dịch có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Chuyển phát biểu IF

❖ Xét phát biểu IF sau:

```
if (a == b) c = d + e;
```

```
else c = d - e;
```

Giả sử a, b, c, d, e là các thanh ghi $\$s0, \dots, \$s4$ tương ứng

❖ Chuyển phát biểu IF trên sang hợp ngữ MIPS?

```
        bne    $s0, $s1, else
        addu   $s2, $s3, $s4
        j      exit
else:    subu   $s2, $s3, $s4
exit:    . . .
```

Biểu thức điều kiện kết hợp AND

- ❖ Ngôn ngữ lập trình sử dụng **short-circuit evaluation**
- ❖ Nếu biểu thức đầu **false**, biểu thức thứ 2 được **bỏ qua**

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

One Possible Implementation ...

```
    bgtz    $s1, L1      # first expression
    j       next        # skip if false
L1:  bltz    $s2, L2      # second expression
    j       next        # skip if false
L2:  addiu   $s3,$s3,1    # both are true
next:
```


Cách hiện thực biểu thức AND tốt hơn

```
if (($s1 > 0) && ($s2 < 0)) {$s3++;}
```

Hiện thực dưới đây dùng ít lệnh hơn

Đảo ngược phép toán quan hệ (> thành <=, < thành >=)

Nhảy thẳng đến biểu thức thứ 2 nếu biểu thức 1 **false**

Số lượng lệnh giảm từ 5 xuống 3

```
# Better Implementation ...  
    blez    $s1, next    # skip if false  
    bgez    $s2, next    # skip if false  
    addiu   $s3,$s3,1    # both are true  
next:
```

Biểu thức điều kiện kết hợp OR

- ❖ Short-circuit evaluation cho phép OR
- ❖ Nếu biểu thức 1 **true**, biểu thức kế tiếp được **bỏ qua**

```
if (($s1 > $s2) || ($s2 > $s3)) {$s4 = 1;}
```

- ❖ Lời giải:

```
    bgt $s1, $s2, L1      # yes, execute if part
    ble $s2, $s3, next    # no: skip if part
L1:  li  $s4, 1           # set $s4 to 1
next:
```

- ❖ **bgt**, **ble**, và **li** là những lệnh giả **pseudo-instructions**

✧ Được assembler chuyển tự động sang các lệnh thật (Slide 44)

Bài tập. . .

- ❖ Chuyển phát biểu IF sang hợp ngữ MIPS
- ❖ \$s1 và \$s2 là giá trị **không dấu (unsigned)**

```
if( $s1 <= $s2 ) {  
    $s3 = $s4  
}
```

```
bgtu $s1, $s2, next  
move $s3, $s4  
next:
```

- ❖ \$s3, \$s4, và \$s5 là giá trị **có dấu (signed)**

```
if ( ($s3 <= $s4) &&  
    ($s4 > $s5) ) {  
    $s3 = $s4 + $s5  
}
```

```
bgt $s3, $s4, next  
ble $s4, $s5, next  
addu $s3, $s4, $s5  
next:
```

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý, dịch có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Lệnh Load và Store

❖ Lệnh dùng để di chuyển dữ liệu qua lại giữa **bộ nhớ** \Leftrightarrow **thanh ghi**

❖ Chương trình có biến kiểu mảng, đối tượng

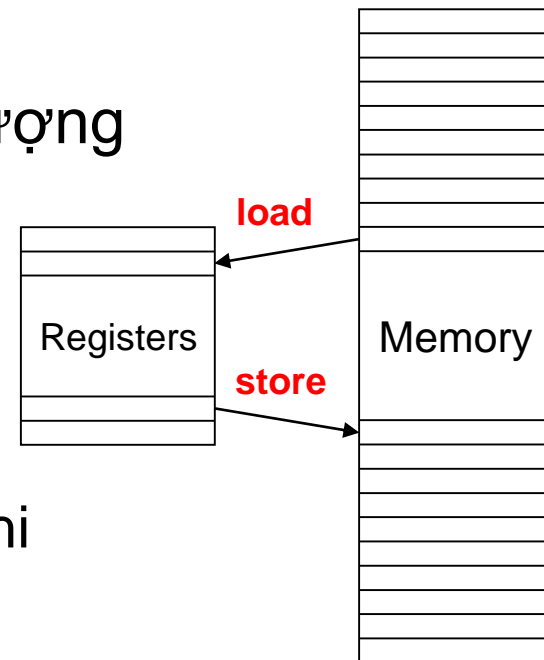
❖ Các biến này được lưu vào bộ nhớ

❖ Lệnh **Đọc (Load)** :

✧ Chuyển dữ liệu từ bộ nhớ đến thanh ghi

❖ Lệnh **Ghi (Store)** :

✧ Chuyển dữ liệu từ thanh ghi xuống bộ nhớ



❖ **Địa chỉ ô nhớ** được chỉ ra bởi lệnh load và store

Load và Store dữ liệu Word (32-bit)

- ❖ Lệnh Load Word (Word = 4 bytes in MIPS)

`lw Rt, imm16(Rs) # Rt ← MEMORY[Rs+imm16]`

- ❖ Lệnh Store Word

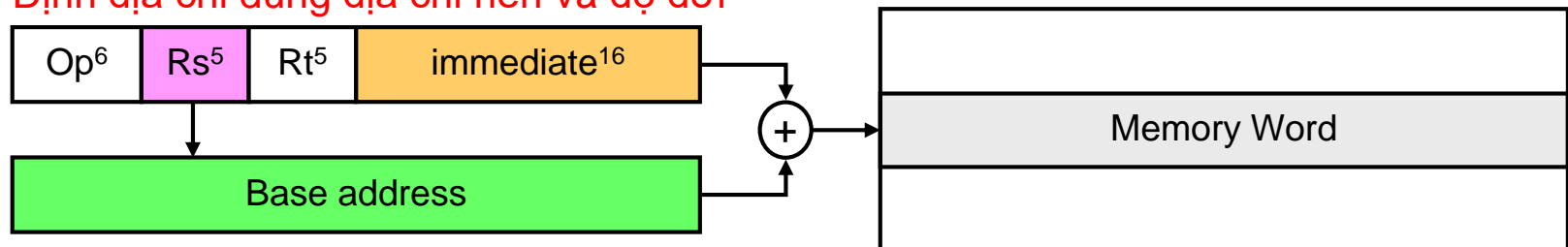
`sw Rt, imm16(Rs) # Rt → MEMORY[Rs+imm16]`

- ❖ Các xác định địa chỉ ô nhớ dùng **địa chỉ nền và độ dời**:

✧ Memory Address = Rs (**base**) + Immediate¹⁶ (**offset**)

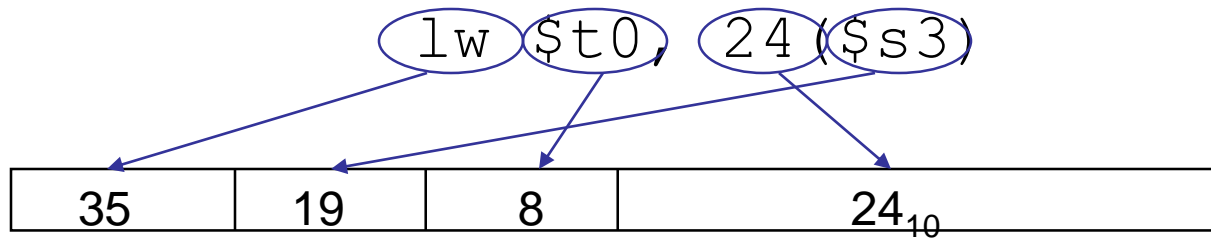
✧ Độ dời immediate¹⁶ được **mở rộng dấu** thành số 32 bit

Định địa chỉ dùng địa chỉ nền và độ dời



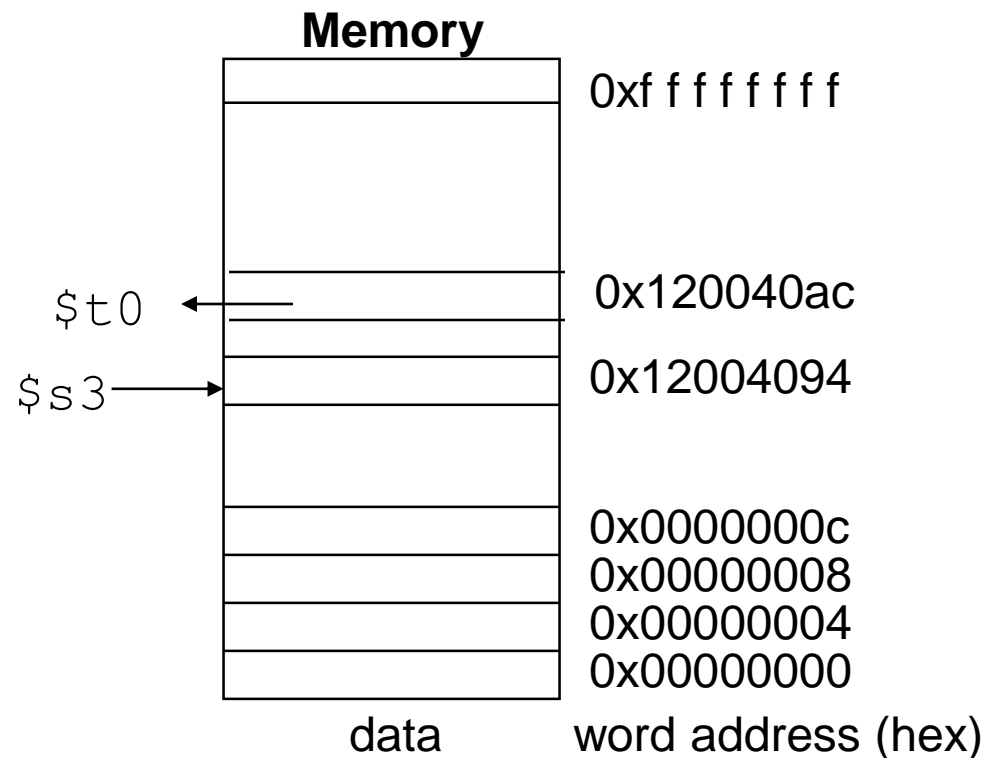
Ví dụ lệnh Load

❖ Lệnh Load/Store có định dạng I-Type:



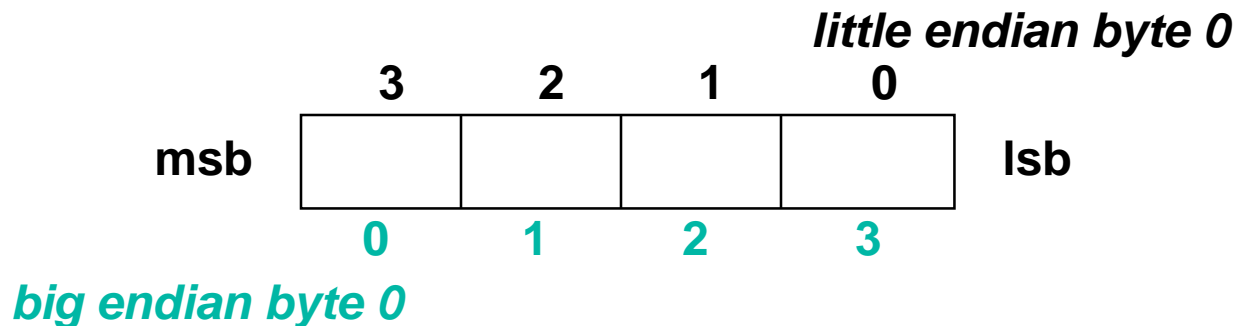
$$24_{10} + \$s3 =$$

$$\begin{array}{r}
 \dots 0001\ 1000 \\
 + \dots 1001\ 0100 \\
 \hline
 \dots 1010\ 1100 = \\
 \quad 0x120040ac
 \end{array}$$



Địa chỉ theo Byte

- ❖ Dữ liệu 8-bit bytes vẫn hữu dụng, hầu hết các kiến trúc hỗ trợ định địa chỉ bộ nhớ theo **bytes**
 - ✧ **Alignment restriction** – địa chỉ của một ô nhớ **word** phải là số chẵn cho kích thước một word (4 byte đối với MIPS-32)
- ❖ **Big Endian:** leftmost byte is word address
IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- ❖ **Little Endian:** rightmost byte is word address
Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Ví dụ chi tiết dữ liệu lệnh Load

Example

$$\$12 = \text{Memory}[0 + [\$3]]$$

$$\text{Address} = 0 + 0x10010000$$

lw \$12, 0(\$3)

word = 32 bits = 4 B

Data Memory

Register File

\$3	0x10010000
	...
\$12	33 22 11 00

0x10010000	0x00
0x10010001	0x11
0x10010002	0x22
0x10010003	0x33

} 4 B

Ví dụ chi tiết dữ liệu lệnh Store

Example

Memory $[0 + r[\$3]] = \12

Address = $0 + 0x10010000$

sw \$12, 0(\$3)

✓ word = 32 bits = 4B

Data Memory

Register File

\$3	0x10010000
	...
\$12	0xAABBCCDD

0x10010000
0x10010001
0x10010002
0x10010003

DD
CC
BB
AA

Ví dụ sử dụng Load & Store

❖ Chuyển $A[1] = A[2] + 5$ (A là mảng kiểu word)

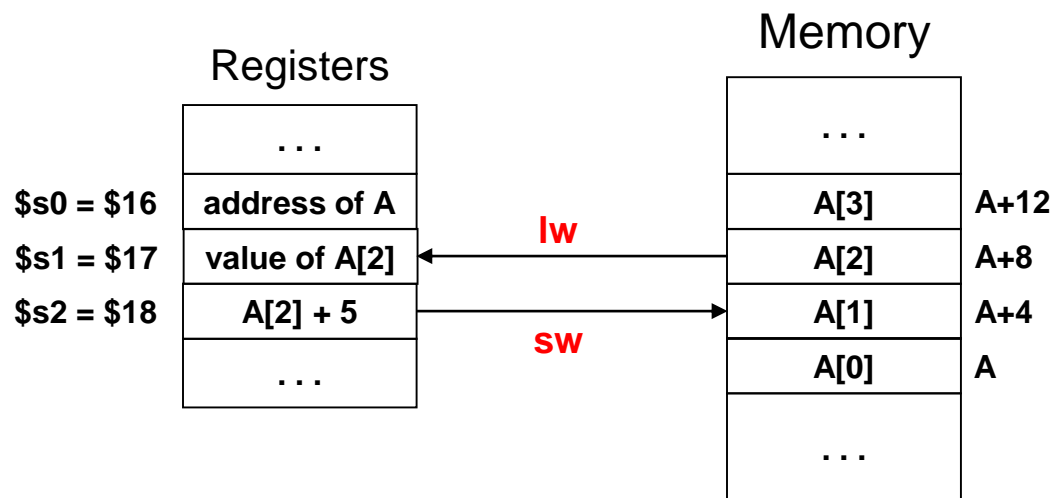
✧ Giả sử địa chỉ mảng A được lưu trong \$s0

lw **\$s1, 8(\$s0)** **# \$s1 = A[2]**

addiu **\$s2, \$s1, 5** **# \$s2 = A[2] + 5**

sw **\$s2, 4(\$s0)** **# A[1] = \$s2**

❖ Địa chỉ của $a[2]$ và $a[1]$ được nhân 4. Tại sao?



Load/Store Byte và Halfword

❖ MIPS hỗ trợ kiểu dữ liệu:

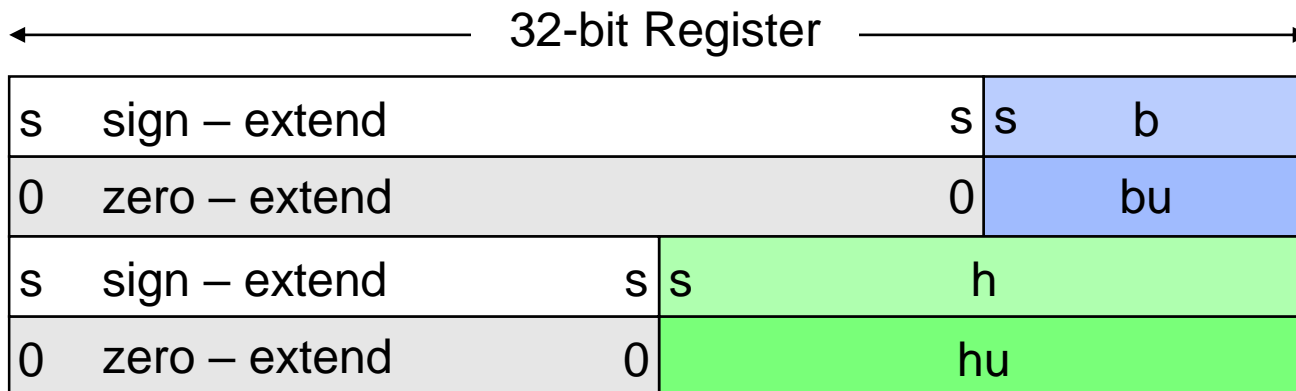
✧ Byte = 8 bits, Halfword = 16 bits, Word = 32 bits

❖ Lệnh Load & store cho bytes và halfwords

✧ lb = load byte, lbu = load byte unsigned, sb = store byte

✧ lh = load half, lhu = load half unsigned, sh = store halfword

❖ Load **mở rộng** giá trị ô nhớ thành số 32-bit trong thanh ghi



Các lệnh Load & Store

Instruction		Meaning	I-Type Format			
lb	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x20	rs ⁵	rt ⁵	imm ¹⁶
lh	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x21	rs ⁵	rt ⁵	imm ¹⁶
lw	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x23	rs ⁵	rt ⁵	imm ¹⁶
lbu	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x24	rs ⁵	rt ⁵	imm ¹⁶
lhu	rt, imm ¹⁶ (rs)	rt = MEM[rs+imm ¹⁶]	0x25	rs ⁵	rt ⁵	imm ¹⁶
sb	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x28	rs ⁵	rt ⁵	imm ¹⁶
sh	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x29	rs ⁵	rt ⁵	imm ¹⁶
sw	rt, imm ¹⁶ (rs)	MEM[rs+imm ¹⁶] = rt	0x2b	rs ⁵	rt ⁵	imm ¹⁶

❖ Định địa chỉ theo địa chỉ nền và độ dời

✧ Memory Address = Rs (**base**) + Immediate¹⁶ (**offset**)

Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý, dịch có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

Chuyển đổi khối lặp WHILE

❖ Xét phát biểu WHILE:

`i = 0; while (A[i] != k) i = i+1;`

A là mảng số nguyên 4 byte

Giả sử địa chỉ *A*, *i*, *k* tương ứng \$s0, \$s1, \$s2

Memory

...	
A[i]	A+4×i
...	
A[2]	A+8
A[1]	A+4
A[0]	A
...	

❖ Chuyển phát biểu WHILE?

```

        xor      $s1, $s1, $s1      # i = 0
        move     $t0, $s0           # $t0 = address A
loop:    lw      $t1, 0($t0)         # $t1 = A[i]
        beq     $t1, $s2, exit      # exit if (A[i]== k)
        addiu   $s1, $s1, 1         # i = i+1
        sll     $t0, $s1, 2         # $t0 = 4*i
        addu    $t0, $s0, $t0       # $t0 = address A[i]
        j       loop
exit:    . . .

```

Sử dụng con trỏ để duyệt Arrays

❖ Xét phát biểu WHILE:

```
i = 0; while (A[i] != k) i = i+1;
```

A là mảng số nguyên 4 byte

Giả sử địa chỉ A, i, k tương ứng \$s0, \$s1, \$s2

❖ Sử dụng **con trỏ (Pointer)** để duyệt mảng A

Pointer được tăng lên 4 (faster than indexing)

```
        move    $t0, $s0          # $t0 = $s0 = addr A
        j        cond             # test condition
loop:    addiu   $s1, $s1, 1        # i = i+1
        addiu   $t0, $t0, 4        # point to next
cond:    lw      $t1, 0($t0)        # $t1 = A[i]
        bne     $t1, $s2, loop     # loop if A[i] != k
```

❖ Chỉ còn 4 lệnh (thay vì 6) trong thân vòng lặp

Copying a String

Copy chuỗi nguồn sang chuỗi đích

Địa chỉ củ chuỗi nguồn là \$s0 và đích là \$s1

Chuỗi kết thúc bằng ký tự null (C strings)

```
i = 0;  
do {target[i]=source[i]; i++;} while (source[i]!=0);
```

```
        move    $t0, $s0           # $t0 = pointer to source  
        move    $t1, $s1           # $t1 = pointer to target  
L1:  lb      $t2, 0($t0)           # load byte into $t2  
        sb      $t2, 0($t1)         # store byte into target  
        addiu   $t0, $t0, 1         # increment source pointer  
        addiu   $t1, $t1, 1         # increment target pointer  
        bne     $t2, $zero, L1      # loop until NULL char
```

Tính tổng của một mảng nguyên

```
sum = 0;  
for (i=0; i<n; i++) sum = sum + A[i];
```

Giả sử \$s0 = địa chỉ của A, \$s1 = kích thước mảng = n

```
move    $t0, $s0           # $t0 = address A[i]  
xor      $t1, $t1, $t1      # $t1 = i = 0  
xor      $s2, $s2, $s2      # $s2 = sum = 0  
L1: lw   $t2, 0($t0)        # $t2 = A[i]  
addu     $s2, $s2, $t2      # sum = sum + A[i]  
addiu    $t0, $t0, 4        # point to next A[i]  
addiu    $t1, $t1, 1        # i++  
bne      $t1, $s1, L1       # loop if (i != n)
```

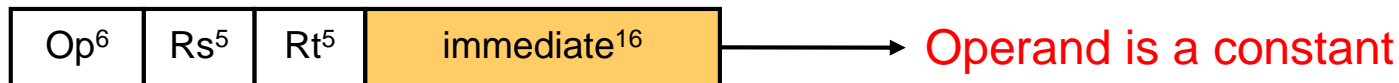
Tiếp theo ...

- ❖ Kiến trúc tập lệnh (Instruction Set Architecture)
- ❖ Sơ bộ kiến trúc bộ xử lý MIPS
- ❖ R-Type Các lệnh số học, luận lý, dịch
- ❖ I-Type Các lệnh số học, luận lý, dịch có hằng số
- ❖ Các lệnh nhảy và rẽ nhánh
- ❖ Chuyển phát biểu If và các biểu thức boolean
- ❖ Các lệnh truy xuất bộ nhớ Load & Store
- ❖ Chuyển đổi khối lập và duyệt mảng
- ❖ Các chế độ định địa chỉ

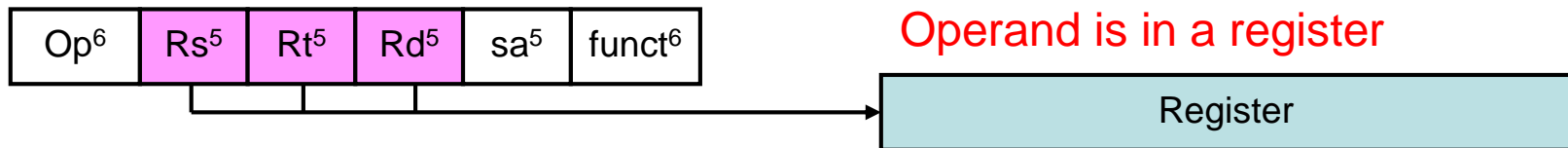
Các chế độ định địa chỉ

- ❖ Địa chỉ (vị trí) của toán hạng?
- ❖ Địa chỉ của ô nhớ được tính như thế nào?

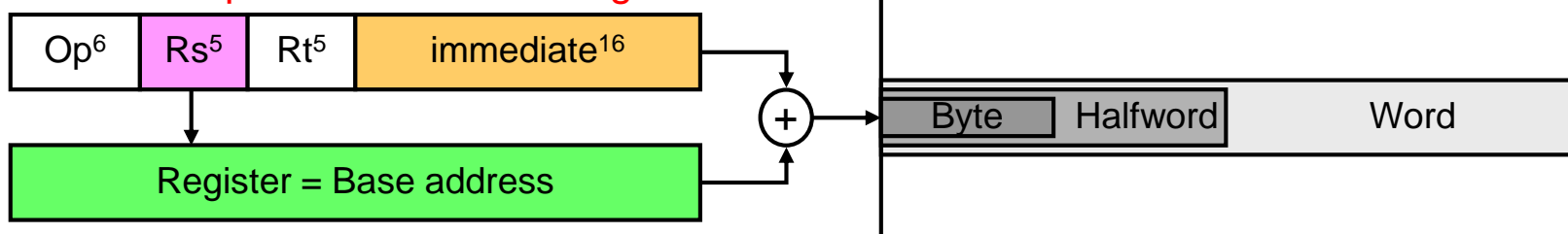
Immediate Addressing



Register Addressing

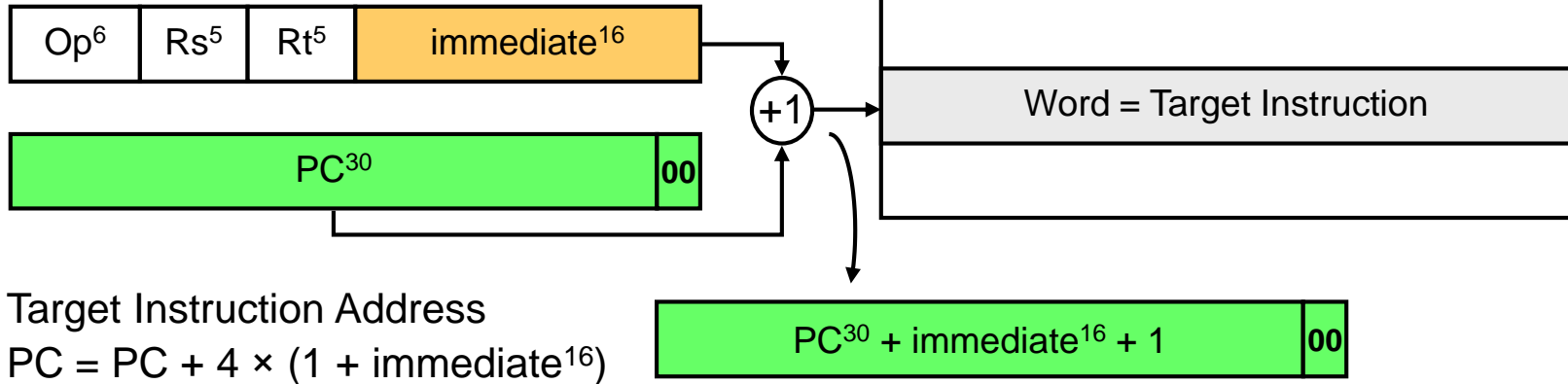


Base or Displacement Addressing

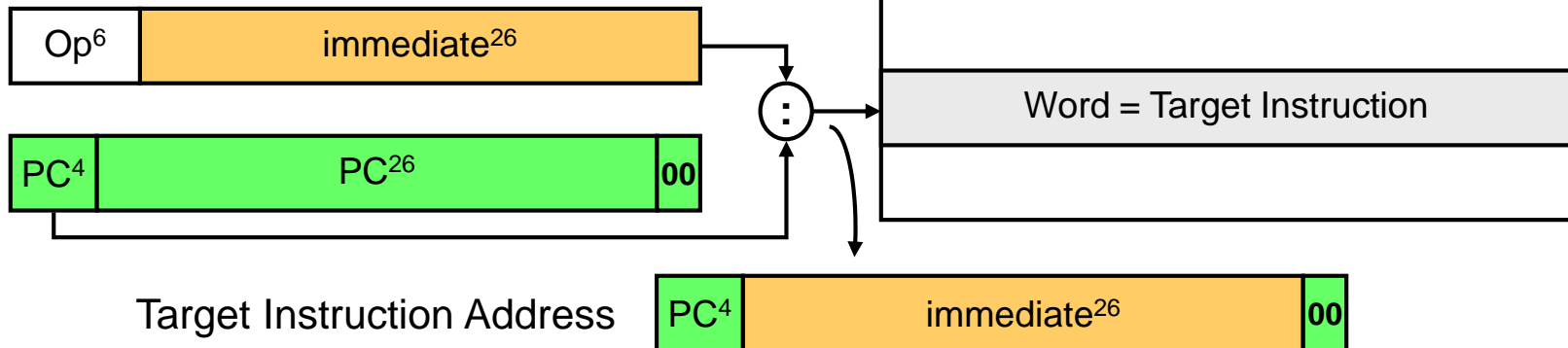


Chế độ định địa chỉ của lệnh nhảy, rẽ nhánh

PC-Relative Addressing



Pseudo-direct Addressing



Giới hạn của lệnh nhảy và rẽ nhánh

❖ Phạm vi địa chỉ của lệnh Jump = 2^{26} lệnh = 256 MB

✧ Nhảy đến vị trí không quá 2^{26} lệnh hoặc 256 MB

✧ 4 bit cao của PC không đổi

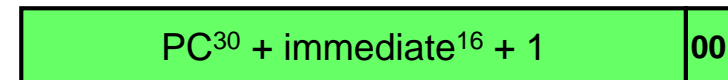
Địa chỉ đích của lệnh Jump



❖ Phạm vi địa chỉ của lệnh rẽ nhánh

✧ Lệnh rẽ nhánh có định dạng I-Type (16-bit hằng số)

✧ Định địa chỉ tương đối với PC:



▪ Địa chỉ đích = $PC + 4 \times (1 + \text{immediate}^{16})$

▪ Là số lệnh tính từ vị trí lệnh kế của lệnh rẽ nhánh

▪ **Hằng số dương** => **Forward**, **Hằng số âm** => **Backward**

▪ Khoảng cách xa nhất $\pm 2^{15}$ lệnh (thông thường lệnh rẽ nhánh có đích chỉ đích lân cận vị trí lệnh)

Summary of RISC Design

- ❖ All instructions are typically of one size
- ❖ Few instruction formats
- ❖ All operations on data are register to register
 - ✧ Operands are read from registers
 - ✧ Result is stored in a register
- ❖ General purpose integer and floating point registers
 - ✧ Typically, 32 integer and 32 floating-point registers
- ❖ Memory access only via **load** and **store** instructions
 - ✧ Load and store: bytes, half words, words, and double words
- ❖ Few simple addressing modes

Four Design Principles

1. Simplicity favors regularity
 - ✧ Fix the size of instructions (simplifies fetching & decoding)
 - ✧ Fix the number of operands per instruction
 - Three operands is the natural number for a typical instruction
2. Smaller is faster
 - ✧ Limit the number of registers for faster access (typically 32)
3. Make the common case fast
 - ✧ Include constants inside instructions (faster than loading them)
 - ✧ Design most instructions to be register-to-register
4. Good design demands good compromises
 - ✧ Fixed-size instructions compromise the size of constants