



Kiến trúc MIPS-32 bit

Môn học: Kiến trúc máy tính & Hợp ngữ

Giới thiệu

- Nhiệm vụ cơ bản nhất của CPU là phải thực hiện các lệnh được yêu cầu, gọi là **instruction**
- Các CPU sẽ sử dụng các tập lệnh (instruction set) khác nhau để có thể giao tiếp với nó

Kích thước lệnh

- Kích thước lệnh bị ảnh hưởng bởi:
 - Cấu trúc đường truyền bus
 - Kích thước và tổ chức bộ nhớ
 - Tốc độ CPU
- Giải pháp tối ưu lệnh:
 - Dùng lệnh có kích thước ngắn, mỗi lệnh chỉ nên được thực thi trong đúng 1 chu kỳ CPU
 - Dùng bộ nhớ cache

Bộ lệnh MIPS

- Chúng ta sẽ làm quen với tập lệnh cho kiến trúc MIPS (PlayStation 1, 2; PSP; Windows CE, Routers...)
- Được xây dựng theo kiến trúc (RISC) với **4 nguyên tắc:**
 - Càng đơn giản, càng ổn định
 - Càng nhỏ gọn, xử lý càng nhanh
 - Tăng tốc xử lý cho những trường hợp thường xuyên xảy ra
 - Thiết kế đòi hỏi sự thỏa hiệp tốt

Cấu trúc cơ bản của 1 chương trình hợp ngữ trên MIPS

.data # khai báo các data label (có thể hiểu là các biến)
sau chỉ thị này

label1: <kiểu lưu trữ> <giá trị khởi tạo>

label2: <kiểu lưu trữ> <giá trị khởi tạo>

...

.text # viết các lệnh sau chỉ thị này

.globl <các text label toàn cục, có thể truy xuất từ các file khác>

.globl main # Đây là text label toàn cục bắt buộc của program

...

main: # điểm text label bắt đầu của program

...

Hello.asm

```
.data          # data segment
str: .asciiz "Hello asm !"

.text          # text segment

.globl main

main:          # starting point of program
    addi $v0, $0, 4      # $v0 = 0 + 4 = 4 → print str syscall
    la $a0, str          # $a0 = address(str)
    syscall               # execute the system call
```

Bộ lệnh MIPS – Thanh ghi

- Là đơn vị lưu trữ data duy nhất trong CPU
- Trong kiến trúc MIPS:
 - Có tổng cộng 32 thanh ghi đánh số từ \$0 → \$31
 - Càng ít càng dễ quản lý, tính toán càng nhanh
 - Có thể truy xuất thanh ghi qua tên của nó (slide sau)
 - Mỗi thanh ghi có kích thước cố định 32 bit
 - Bị giới hạn bởi khả năng tính toán của chip xử lý
 - Kích thước toán hạng trong các câu lệnh MIPS bị giới hạn ở 32 bit, nhóm 32 bit gọi là từ (word)

Thanh ghi toán hạng

- Như chúng ta đã biết khi lập trình, biến (variable) là khái niệm rất quan trọng khi muốn biểu diễn các toán hạng để tính toán
- Trong kiến trúc MIPS không tồn tại khái niệm biến, thay vào đó là thanh ghi toán hạng

Thanh ghi toán hạng

- Ngôn ngữ cấp cao (C, Java...): toán hạng = biến (variable)
 - Các biến lưu trong bộ nhớ chính
- Ngôn ngữ cấp thấp (Hợp ngữ): toán hạng chứa trong các thanh ghi
 - Thanh ghi không có kiểu dữ liệu
 - Kiểu dữ liệu thanh ghi được quyết định bởi thao tác trên thanh ghi
- So sánh:
 - **Ưu:** Thanh ghi truy xuất nhanh hơn nhiều bộ nhớ chính
 - **Khuyết:** Không như bộ nhớ chính, thanh ghi là phần cứng có

Một số thanh ghi toán hạng quan tâm

- Save register:
 - MIPS lấy ra 8 thanh ghi (\$16 - \$23) dùng để thực hiện các phép tính số học, được đặt tên tương ứng là **\$s0 - \$s7**
 - Tương ứng trong C, để chứa giá trị biến (variable)
- Temporary register:
 - MIPS lấy ra 8 thanh ghi (\$8 - \$15) dùng để chứa kết quả trung gian, được đặt tên tương ứng là **\$t0 - \$t7**
 - Tương ứng trong C, để chứa giá trị biến tạm (temporary variable)

Bảng danh sách thanh ghi MIPS

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2–3	values for results and expression evaluation
\$a0-\$a3	4–7	arguments
\$t0-\$t7	8–15	temporaries
\$s0-\$s7	16–23	saved
\$t8-\$t9	24–25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Thanh ghi 1 (\$at) để dành cho assembler. Thanh ghi 26 – 27 (\$k0 - \$k1) để dành cho OS

Bộ lệnh MIPS – 4 thao tác chính

- Phần 1: Phép toán số học (Arithmetic)
- Phần 2: Di chuyển dữ liệu (Data transfer)
- Phần 3: Thao tác luận lý (Logical)
- Phần 4: Rẽ nhánh (Un/Conditional branch)

Phần 1: Phép toán số học

- Cú pháp:

opt opr, opr1, opr2

- **opt** (operator): Tên thao tác (toán tử, tác tử)
- **opr** (operand): Thanh ghi (toán hạng, tác tố đích)
chứa kết quả
- **opr1** (operand 1): Thanh ghi (toán hạng nguồn
1)
- **opr2** (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Ví dụ

- Giả sử xét câu lệnh sau:

add a, b, c

– Chỉ thị cho CPU thực hiện phép cộng

$a \leftarrow b + c$

– a, b, c được gọi là thanh ghi toán hạng

– Phép toán trên chỉ có thể thực hiện với đúng

3 toán hạng (không nhiều cũng không ít hơn)

Cộng, trừ số nguyên

- Cộng (Add):
 - Cộng có dấu: **add \$s0, \$s1, \$s2**
 - Cộng không dấu: **addu \$s0, \$s1, \$s2** (u: unsigned)
 - Diễn giải: $\$s0 \leftarrow \$s1 + \$s2$
C/C++: (a = b + c)
- Trừ (Subtract):
 - Trừ có dấu: **sub \$s0, \$s1, \$s2**
 - Trừ không dấu: **subu \$s0, \$s1, \$s2** (u: unsigned)
 - Diễn giải: $\$s0 \leftarrow \$s1 - \$s2$
C/C++: (a = b - c)

Nhận xét

- Toán hạng trong các lệnh trên phải là thanh ghi
- Trong MIPS, lệnh thao tác với số nguyên có dấu được biểu diễn dưới dạng bù 2
- Làm sao biết 1 phép toán được biên dịch từ C (ví dụ $a = b + c$) là thao tác có dấu hay không dấu? → Dựa vào trình biên dịch
- Có thể dùng 1 toán hạng vừa là nguồn vừa là đích
add \$s0, \$s0, \$s1
- Cộng, trừ với hằng số? → \$s2 sẽ đóng vai trò là hằng số
 - Cộng: addi \$s0, \$s1, 3 (addi = add immediate)

Ví dụ 1

- Chuyển thành lệnh MIPS từ lệnh C:

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

- Chia nhỏ thành nhiều lệnh MIPS:

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

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

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

- Tại sao dùng nhiều lệnh hơn C?

→ Bị giới hạn bởi số lượng cổng mạch toán tử và thiết kế bên trong
cổng mạch

- Ký tự "#" dùng để chú thích trong hợp ngữ cho MIPS

Ví dụ 2

- Chuyển thành lệnh MIPS từ lệnh C:

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

- Chia nhỏ thành nhiều lệnh MIPS:

add \$t0, \$s1, \$s2 # temp1 = g + h

add \$t1, \$s3, \$s4 # temp2 = i + j

sub \$s0, \$t0, \$t1 # f = temp1 -

Lưu ý: Phép gán ?

- Kiến trúc MIPS không có cổng mạch dành riêng cho phép gán

→ Giải pháp: Dùng thanh ghi zero (\$0 hay \$zero) luôn mang giá trị 0

- Ví dụ:

add \$s0, \$s1, \$zero

Tương đương: $\$s0 = \$s1 + 0 = \$s1$ (gán)

- Lệnh “add \$zero, \$zero, \$s0” có hợp lệ ?

Phép nhân, chia số nguyên

- Thao tác nhân / chia của MIPS có kết quả chứa trong cặp 2 thanh ghi tên là \$hi và \$lo Bit 0-31 thuộc \$lo và 32-63 thuộc \$hi

$$\boxed{} \quad \times \quad \boxed{} \quad = \quad \boxed{} \quad | \quad \boxed{}$$

op1 op2 hi lo

$$\boxed{} \quad \div \quad \boxed{} \quad = \quad \begin{matrix} \text{remainder} & \text{quotient} \end{matrix}$$

op1 op2 hi lo

Phép nhân

- Cú pháp:
`mult $s0, $s1`
- Kết quả (64 bit) chứa trong 2 thanh ghi
 - `$lo (32 bit)` = `(($s_0 * s_1$) << 32) >> 32`
 - `$hi (32 bit)` = `($s_0 * s_1$) >> 32`
- Câu hỏi: Làm sao truy xuất giá trị 2 thanh ghi `$lo` và `$hi`? → Dùng 2 cặp lệnh `mflo` (move from lo), `mfhi` (move from hi) - `mtlo` (move to lo), `mthi` (move to high)

Phép chia

- Cú pháp:

div \$s0, \$s1

- Kết quả (64 bit) chứa trong 2 thanh ghi
 - \$lo (32 bit) = \$s0 / \$s1 (thương)
 - \$hi (32 bit) = \$s0 % \$s1 (số dư)

Thao tác số dấu chấm động

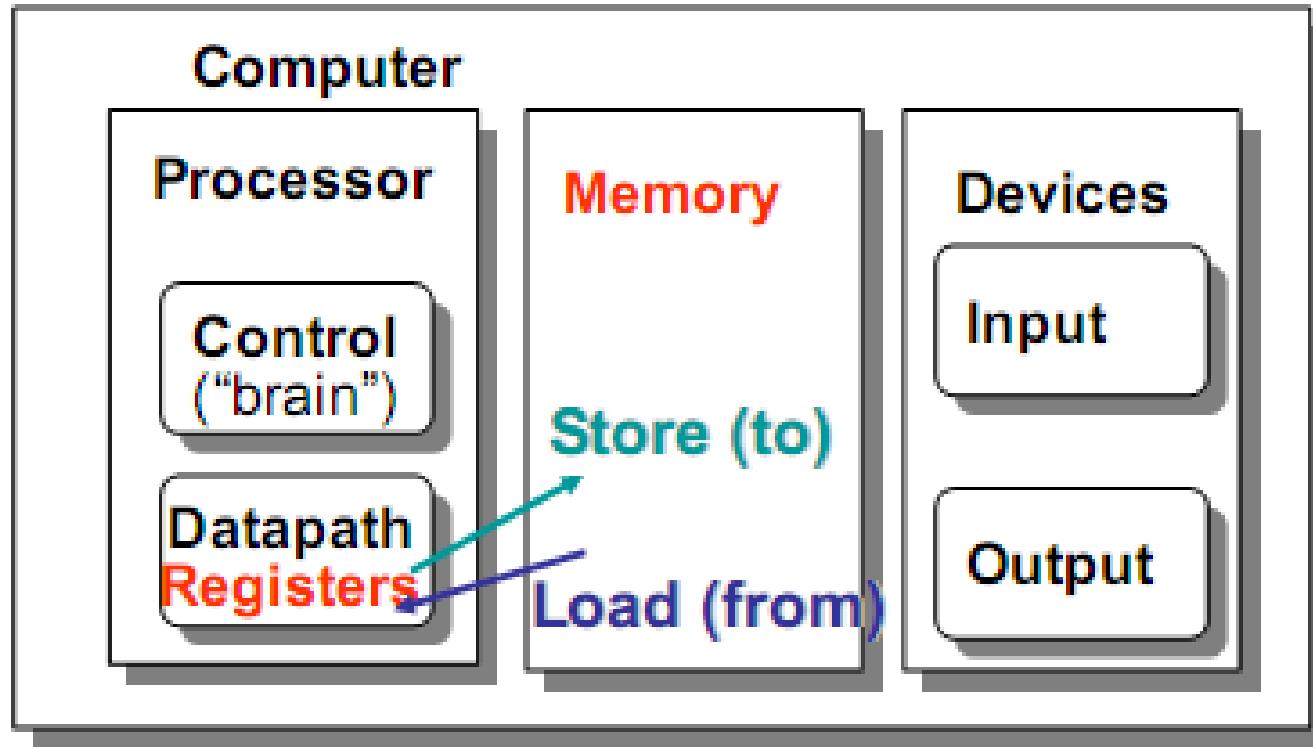
- MIPS sử dụng 32 thanh ghi dấu phẩy động để biểu diễn độ chính xác đơn của số thực. Các thanh ghi này có tên là : \$f0 – \$f31.
- Để biểu diễn độ chính xác kép (double precision) thì MIPS sử dụng sự ghép đôi của 2 thanh ghi có độ chính xác đơn.

Vấn đề tràn số

- Kết quả phép tính vượt qua miền giá trị cho phép → Tràn số xảy ra
- Một số ngôn ngữ có khả năng phát hiện tràn số (Ada), một số không (C)
- MIPS cung cấp 2 loại lệnh số học:
 - **add, addi, sub**: Phát hiện tràn số
 - **addu, addiu, subu**: Không phát hiện tràn số
- Trình biên dịch sẽ lựa chọn các lệnh số học tương ứng
 - Trình biên dịch C trên kiến trúc MIPS sử dụng addu,

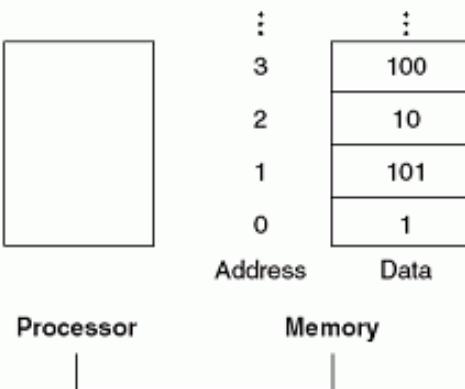
Phần 2: Di chuyển dữ liệu

- Một số nhận xét:
 - Ngoài các biến đơn, còn có các **biến phức tạp** thể hiện nhiều kiểu cấu trúc dữ liệu khác nhau, ví dụ như **array**
 - Các cấu trúc dữ liệu phức tạp có số phần tử dữ liệu nhiều hơn số thanh ghi của CPU → làm sao lưu ??
→ **Lưu phần nhiều data trong RAM, chỉ load 1 ít vào thanh ghi của CPU khi cần xử lý**



Bộ nhớ chính

- Có thể được xem như là array 1 chiều rất lớn, mỗi phần tử là 1 ô nhớ có kích thước bằng nhau
- Các ô nhớ được đánh số thứ tự từ 0 trở đi
→ Gọi là **địa chỉ (address) ô nhớ**
- Để truy xuất dữ liệu trong ô nhớ cần phải cung cấp địa chỉ ô nhớ đó



Cấu trúc lệnh

- Cú pháp:

opt opr, opr1 (opr2)

- opt (operator): Tên thao tác (Load / Save)
- opr (operand): Thanh ghi lưu từ nhớ (word)
- opr1 (operand 1): Hằng số nguyên
- opr2 (operand 2): Thanh ghi chứa địa chỉ vùng nhớ cơ sở (địa chỉ nền)

Hai thao tác chính

- **lw**: Nạp 1 từ dữ liệu, từ bộ nhớ, vào 1 thanh ghi trên CPU (Load Word - lw)



lw \$t0, 12 (\$s0)

Nạp từ nhớ có địa chỉ ($\$s0 + 12$) chứa vào thanh ghi
\$t0

- **sw**: Lưu 1 từ dữ liệu  từ thanh ghi trên CPU, ra bộ nhớ (Store Word – sw)

sw \$t0, 12 (\$s0)

Lưu ý 1

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

Lưu ý 2

- Một thanh ghi có lưu bất kỳ giá trị 32 bit nào, có thể là số nguyên (có dấu / không dấu), có thể là địa chỉ của 1 vùng nhớ trên RAM
- Ví dụ:
 - add \$t2, \$t1, \$t0 → \$t0, \$t1 lưu giá trị
 - lw \$t2, 4(\$t0) → \$t0 lưu địa chỉ (C: con trỏ)

Lưu ý 3

- Số biến cần dùng của chương trình nếu nhiều hơn số thanh ghi của CPU?
- Giải pháp:
 - Thanh ghi chỉ chứa các biến đang xử lý hiện hành và các biến thường sử dụng
 - Kỹ thuật **spilling register**

Ví dụ 1

- Giả sử **A** là 1 array gồm 100 từ với **địa chỉ bắt đầu** (**địa chỉ nền – base address**) chứa trong thanh ghi **\$s3**. Giá trị các biến **g**, **h** lần lượt chứa trong các thanh ghi **\$s1** và **\$s2**
- Hãy chuyển thành mã hợp ngữ MIPS:

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

- Trả lời:

Iw \$t0, 32(\$s3) # Chứa **A[8]** vào **\$t0**

add \$s1, \$s2, \$t0

Ví dụ 2

- Hãy chuyển thành mã hợp ngữ MIPS:

$$A[12] = h - A[8]$$

- Trả lời:

lw \$t0, 32(\$s3) # Chứa A[8] vào

\$t0

sub \$t0, \$s2, \$t0

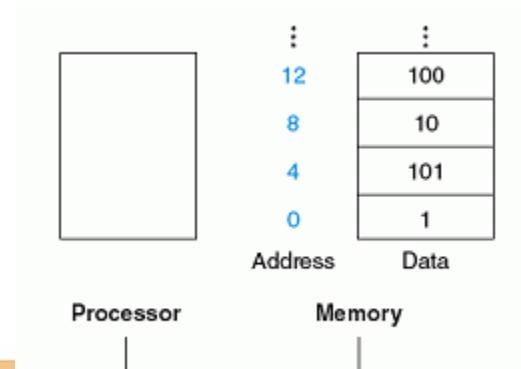
sw \$t0, 48(\$s3) # Kết quả vào

Nguyên tắc lưu trữ dữ liệu trong bộ nhớ

- MIPS thao tác và lưu trữ dữ liệu trong bộ nhớ theo 2 nguyên tắc:
 - Alignment Restriction
 - Big Endian

Alignment Restriction

- MIPS lưu dữ liệu trong bộ nhớ theo nguyên tắc **Alignment Restriction**
 - Các đối tượng lưu trong bộ nhớ (từ nhớ) phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng
 - Mà mỗi từ nhớ có kích thước là 32 bit = 4 byte = kích thước lưu trữ của 1 thanh ghi trong CPU
- Như vậy, từ nhớ phải bắt đầu tại địa chỉ là bội số của 4



Big Endian

- MIPS lưu trữ thứ tự các byte trong 1 word trong bộ nhớ theo nguyên tắc **Big Endian** (Kiến trúc x86 sử dụng Little Endian)
- Ví dụ: Lưu trữ giá trị 4 byte: **12345678h** trong bộ nhớ

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

Lưu ý

- Để truy xuất vào 1 từ nhớ sau 1 từ nhớ thì cần **tăng 1 lượng 4 byte** chứ không phải 1 byte
- Do đó luôn nhớ rằng các lệnh **lw** và **sw** thì **độ dời (offset) phải là bội số của 4**
- Tuy nhiên bộ nhớ các máy tính cá nhân ngày nay lại được đánh địa chỉ theo từng byte (8

Mở rộng: Load, Save 1 byte

- Ngoài việc hỗ trợ load, save 1 từ (lw, sw), MIPS còn hỗ trợ load, save từng byte (ASCII)
 - Load byte: lb
 - Save byte: sb
 - Cú pháp lệnh tương tự lw, sw
- Ví dụ:

lb \$s0, 3 (\$s1)

Lệnh này nạp giá trị byte nhớ có địa chỉ ($\$s1 + 3$) vào byte thấp của thanh ghi \$s0

Nguyên tắc

- Giả sử nạp 1 byte có giá trị **x₇zzz z₈zzz** vào thanh ghi trên CPU (x: bit dấu của byte đó)
- Giá trị thanh ghi trên CPU (32 bit) sau khi nạp có dạng:

xxxx xxxx xxxx xxxx xxxx xxxx x₇zzz z₈zzz

→ Tất cả các bit từ phải sang sẽ có giá trị = bit dấu của giá trị 1 byte vừa nạp (sign-extended)

→ Nếu muốn các bit còn lại từ phải sang có giá trị không theo bit dấu (=0) thì dùng lệnh:

Mở rộng: Load, Save 2 byte (1/2 Word)

- MIPS còn hỗ trợ load, save 1/2 word (2 byte) (Unicode)
 - Load half: lh (nạp 2 byte nhớ vào 2 byte thấp của thanh ghi \$s0)
 - Store half: sh
 - Cú pháp lệnh tương tự lw, sw
- Ví dụ:

lh \$s0, 3 (\$s1)

Lệnh này nạp giá trị 2 byte nhớ có địa chỉ ($\$s1 + 3$) vào 2 byte thấp của thanh ghi \$s0

Phần 3: Thao tác luận lý

- Chúng ta đã xem xét các thao tác số học (+, -, *, /)
 - Dữ liệu trên thanh ghi như **1 giá trị đơn** (số nguyên có dấu / không dấu)
- Cần thao tác trên từng bit của dữ liệu → **Thao tác luận lý**
 - Các thao tác luận lý xem dữ liệu trong thanh ghi là dãy 32 bit riêng lẻ thay vì 1 giá trị đơn
- Có 2 loại thao tác luận lý:
 - **Phép toán luận lý**
 - **Phép dịch luận lý**

Phép toán luận lý

- Cú pháp:

opt opr, opr1, opr2

- opt (operator): Tên thao tác
- opr (operand): Thanh ghi (toán hạng đích)
chứa kết quả
- opr1 (operand 1): Thanh ghi (toán hạng nguồn
1)
- opr2 (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Phép toán luận lý

- MIPS hỗ trợ 2 nhóm lệnh cho các phép toán luận lý trên bit:
 - **and, or, nor:** Toán hạng nguồn thứ 2 (opr2) phải là thanh ghi
 - **andi, ori:** Toán hạng nguồn thứ 2 (opr2) là hằng số
- **Lưu ý:** MIPS không hỗ trợ lệnh cho các phép luận lý NOT, XOR, NAND...
- **Lý do:** Vì với 3 phép toán luận lý and, or, nor ta có thể tạo ra tất cả các phép luận lý khác → Tiết kiệm

Phép dịch luận lý

- Cú pháp:

opt opr, opr1, opr2

- opt (operator): Tên thao tác
- opr (operand): Thanh ghi (toán hạng đích)
chứa kết quả
- opr1 (operand 1): Thanh ghi (toán hạng nguồn
1)
- opr2 (operand 2): Hằng số < 32 (Số bit dịch)

Phép dịch luận lý

- MIPS hỗ trợ 2 nhóm lệnh cho các phép dịch luận lý trên bit:
 - **Dịch luận lý**
 - Dịch trái (**sll** – shift left logical): Thêm vào các bit 0 bên phải
 - Dịch phải (**srl** – shift right logical): Thêm vào các bit 0 bên trái
 - **Dịch số học**
 - Không có dịch trái số học
 - Dịch phải (**sra** – shift right arithmetic): Thêm các bit = giá trị

Ví dụ

- **sll \$s1, \$s2, 2** # dịch trái luận lý \$s2 2 bit

$\$s2 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 0101 = 85$

$\$s1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0101\ 0100 = 340 \quad (85 * 2^2)$

- **srl \$s1, \$s2, 2** # dịch phải luận lý \$s2 2 bit

$\$s2 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0101\ 0101 = 85$

$\$s1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0101 = 21 \quad (85 / 2^2)$

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

$\$s2 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000 = -16$

$\$s1 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100 = -4 \quad (-16 / 2^2)$

Phần 4: Rẽ nhánh

- Tương tự lệnh if trong C: Có 2 loại

- if (condition) clause

- if (condition)

- clause1

- else

- clause2

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

- if (condition) goto L1 // if → Làm clause1

- clause2 // else → Làm clause2

- goto L2 // Làm tiếp các lệnh khác

- L1: clause1

Rẽ nhánh trong MIPS

- Rẽ nhánh có điều kiện
 - **beq opr1, opr2, label**
 - beq: Branch if (register are) equal
 - if (`opr1 == opr2`) goto `label`
 - **bne opr1, opr2, label**
 - bne: Branch if (register are) not equal
 - if (`opr1 != opr2`) goto `label`
- Rẽ nhánh không điều kiện
 - **j label**
 - Jump to label
 - Tương ứng trong C: goto label
 - Có thể viết lại thành: **beq \$0, \$0, label**

Ví dụ

- Biên dịch câu lệnh sau trong C thành lệnh hợp ngữ MIPS:

if (i == j) f = g + h;

else f = g - h;

- Ánh xạ biến f, g, h, i, j tương ứng vào các thanh ghi: \$s0, \$s1, \$s2, \$s3, \$s4
- Lệnh hợp ngữ MIPS:

beq \$s3, \$s4, TrueCase # branch (i == j)

sub \$s0, \$s1, \$s2 # f = g - h (false)

j Fin # goto "Fin" label

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

Xử lý vòng lặp

- Xét mảng int A[]. Giả sử ta có vòng lặp trong C:

do {

g = g + A[i];

i = i + j;

} while (i != h);

- Ta có thể viết lại:

Loop: **g = g + A[i];**

i = i + j;

if (i != h) goto Loop;

→ Sử dụng lệnh rẽ có điều kiện để biểu diễn vòng lặp!

Xử lý vòng lặp

- Ánh xạ biến vào các thanh ghi như sau:

g h i j base address of A

\$s1 \$s2 \$s3 \$s4 \$s5

- Trong ví dụ trên có thể viết lại thành lệnh MIPS như sau:

Loop: sll \$t1, \$s3, 2 # \$t1 = i * 2²

 add \$t1, \$t1, \$s5 # \$t1 = addr A[i]

 lw \$t1, 0 (\$t1) # \$t1 = A[i]

 add \$s1, \$s1, \$t1 # g = g + A[i]

 add \$s3, \$s3, \$s4 # i = i + j

 bne \$s3, \$s2, Loop # if (i != j) goto Label

Xử lý vòng lặp

- Tương tự cho các vòng lặp phổ biến khác trong C:
 - while
 - for
 - do...while
- Nguyên tắc chung:
 - Viết lại vòng lặp dưới dạng goto

So sánh không bằng ?

- **beq** và **bne** được sử dụng để **so sánh bằng** (**==** và **!=** trong C)

→ Muốn so sánh lớn hơn hay nhỏ hơn?

- MIPS hỗ trợ lệnh **so sánh không bằng**:

- **slt** opr1, opr2, opr3

- **slt**: Set on Less Than

- if (opr2 < opr3)

- opr1 = 1;

- else

- opr1 = 0;

So sánh không bằng

- Trong C, câu lệnh sau:

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

- Được chuyển thành lệnh MIPS như sau:

slt \$t0, \$s0, \$s1 # if (g < h) then \$t0 = 1

bne \$t0, \$0, Less # if (\$t0 != 0) goto Less
if (g < h) goto Less

- Nhận xét:** Thanh ghi \$0 luôn chứa giá trị 0, nên lệnh **bne** và **bep** thường dùng để so sánh sau lệnh

Các lệnh so sánh khác?

- Các phép so sánh còn lại như $>$, \geq , \leq thì sao?
- MIPS **không trực tiếp hỗ trợ** cho các phép so sánh trên, tuy nhiên dựa vào các lệnh **slt, bne, beq** ta hoàn toàn có thể biểu diễn chúng!

a: \$s0, b: \$s1

- **a < b**

```
slt $t0, $s0, $s1      # if (a < b) then $t0 = 1
bne $t0, $0, Label      # if (a < b) then goto Label
<do something>          # else then do something
```
- **a > b**

```
slt $t0, $s1, $s0      # if (b < a) then $t0 = 1
bne $t0, $0, Label      # if (b < a) then goto Label
<do something>          # else then do something
```
- **a ≥ b**

```
slt $t0, $s0, $s1      # if (a < b) then $t0 = 1
beq $t0, $0, Label      # if (a ≥ b) then goto Label
<do something>          # else then do something
```
- **a ≤ b**

```
slt $t0, $s1, $s0      # if (b < a) then $t0 = 1
beq $t0, $0, Label      # if (b ≥ a) then goto Label
<do something>          # else then do something
```

Nhận xét

- So sánh == → Dùng lệnh **beq**
- So sánh != → Dùng lệnh **bne**
- So sánh < và > → Dùng cặp lệnh (**slt** → **bne**)
- So sánh ≤ và ≥ → Dùng cặp lệnh (**slt** → **beq**)

So sánh với hàng số

- So sánh bằng: beq / bne
 - So sánh không bằng: MIPS hỗ trợ sẵn
- lệnh slti
- slti opr, opr1, const
 - Thường dùng cho switch...case, vòng lặp for

Ví dụ: switch...case trong C

- **switch (k) {**
case 0: f = i + j; break;
case 1: f = g + h; break;
case 2: f = g - h; break;

}

- Ta có thể viết lại thành các lệnh if lồng nhau:

```
if (k == 0)    f = i + j;  
else if (k == 1)      f = g + h;  
else if (k == 2) f = g - h;
```

- Ánh xạ giá trị biến vào các thanh ghi:

f	g	h	i	j	k
\$s0	\$s1	\$s2	\$s3	\$s4	\$s5

Ví dụ: switch...case trong C

- Chuyển thành lệnh hợp ngữ MIPS:

	bne	\$s5, \$0, L1	# if (k != 0) then goto L1
	add	\$s0, \$s3, \$s4	# else (k == 0) then f = i + j
	j	Exit	# end of case → Exit (break)
L1:	addi	\$t0, \$s5, -1	# \$t0 = k - 1
	bne	\$t0, \$0, L2	# if (k != 1) then goto L2
	add	\$s0, \$s1, \$s2	# else (k == 1) then f = g+ h
	j	Exit	# end of case → Exit (break)
L2:	addi	\$t0, \$s5, -2	# \$t0 = k - 2
	bne	\$t0, \$0, Exit	# if (k != 2) then goto Exit
	sub	\$s0, \$s1, \$s2	# else (k == 2) then f = g - h
Exit:		

Trình con (Thủ tục)

- Hàm (function) trong C → (Biên dịch) → Trình con (Thủ tục) trong hợp ngữ
- Giả sử trong C, ta viết như sau:

void main()

{

int a, b;

...

• Hàm được chuyển thành lệnh hợp ngữ như thế nào ?

• Dữ liệu được lưu trữ ra sao ?

sum(a, b);

...

}

int sum(int x, int y)

{

return (x + y);

}

c ... sum (a, b); ... /* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

 return x + y;

}

Địa chỉ Lệnh

M
I
P
S

1000 add \$a0, \$s0, \$zero # x = a

1004 add \$a1, \$s1, \$zero # y = b

1008 addi \$ra, \$zero, 1016 # lưu địa chỉ lát sau quay về vào \$ra = 1016

1012 j sum # nhảy đến nhãn sum

1016 [Làm tiếp thao tác khác...]

....

2000 sum: add \$v0, \$a0, \$a1 # thực hiện thủ tục "sum"

2024 jr \$ra # nhảy tới địa chỉ trong \$ra

Thanh ghi lưu trữ dữ liệu trong thủ tục

- MIPS hỗ trợ 1 số thanh ghi để lưu trữ dữ liệu cho thủ tục:
 - Đôi số input (argument input): \$a0 \$a1 \$a2 \$a3
 - Kết quả trả về (return ...): \$v0 \$v1
 - Biến cục bộ trong thủ tục: \$s0 \$s1 ... \$s7
 - Địa chỉ quay về (return address): \$ra
- Nếu có nhu cầu lưu nhiều dữ liệu (đôi số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên?
 - Bao nhiêu thanh ghi là đủ ?
 - Sử dụng ngăn xếp (stack)

C

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

[Làm tiếp thao tác khác...]

}

```
int sum (int x, int y) {
```

```
    return x + y;
```

}

NHẬN XÉT 1

M

M	Địa chỉ	Lệnh	
I	1000	add \$a0, \$s0, \$zero	# x = a
P	1004	add \$a1, \$s1, \$zero	• Tại sao không dùng lệnh j cho đơn giản, mà lại dùng jr ?
S	1008	addi \$ra, \$zero, 1016	→ Thủ tục “sum” có thể được gọi ở nhiều chỗ khác nhau, do vậy vị trí quay về mỗi lần gọi sẽ khác nhau → Lệnh mới: jr
	1012	j sum	
	1016	[Làm tiếp thao tác khác...]	
	...		
	2000	sum: add \$v0, \$a0, \$a1	# thực hiện thủ tục “sum”
	2024	jr \$ra	# nhảy tới địa chỉ trong \$ra

C ... sum (a, b); ... /* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

 return x + y;

}

NHẬN XÉT 2

M	Địa chỉ	Lệnh	
I	1000	add	\$a0, \$s0, \$zero
P	1004	add	\$a1, \$s1, \$zero
S	1008	addi	\$ra, \$zero, 1016
	1012	j	sum
	1016	[Làm tiếp thao tác khác...]	
	...		
	2000	sum:	add \$v0, \$a0, \$
	2024	jr	\$ra

- Thay vì dùng 2 lệnh để lưu địa chỉ quay về vào thanh ghi \$ra và nhảy đến thủ tục “sum”:

1008 addi \$ra, \$zero, 1016 # \$ra = 1016

1012 j sum # goto sum

→ MIPS hỗ trợ lệnh mới: **jal** (jump and link) để thực hiện 2 công việc trên:

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

→ Tại sao không cần xác định tường minh địa chỉ quay về trong \$ra ?

Các lệnh nhảy mới

- jr (jump register)
 - Cú pháp: **jr register**
 - Diễn giải: Nhảy đến địa chỉ nằm trong thanh ghi register thay vì nhảy đến 1 nhãn như lệnh j (jump)
- jal (jump and link)
 - Cú pháp: **jal label**
 - Diễn giải: Thực hiện 2 bước:
 - **Bước 1 (link):** Lưu địa chỉ của lệnh kế tiếp vào thanh ghi \$ra (Tại sao không phải là địa chỉ của lệnh hiện tại?)
 - **Bước 2 (jump):** Nhảy đến nhãn label
- Hai lệnh này được sử dụng hiệu quả trong thủ tục
 - **jal:** tự động lưu địa chỉ quay về chương trình chính vào thanh ghi \$ra và nhảy đến thủ tục con
 - **jr \$ra:** Quay lại thân chương trình chính bằng cách nhảy đến địa chỉ đã được lưu trước đó trong \$ra

Bài tập

- Chuyển đoạn chương trình sau thành mã hợp ngữ MIPS:

```
void main()
{
    int i, j, k, m;
    ...
    i = mult (j, k); ...
    m = mult (i, i); ...
}
```

```
int mult (int mcand, int mlier)
{
    int product = 0;
    while (mlier > 0)
    {
        product = product + mcand;
        mlier = mlier - 1;
    }
    return product;
}
```

Thủ tục lồng nhau

- Vấn đề đặt ra khi chuyển thành mã hợp ngữ của đoạn lệnh sau:

```
int sumSquare (int x, int y)
{
    return mult (x, x) + y;
}
```
- Thủ tục `sumSquare` sẽ gọi thủ tục `mult` trong thân hàm của nó
- Vấn đề:
 - Địa chỉ quay về của thủ tục `sumSquare` lưu trong thanh ghi \$ra sẽ bị đè bởi địa chỉ quay về của thủ tục `mult` khi thủ tục này được gọi!
 - Như vậy cần phải lưu lại (`backup`) trong bộ nhớ chính địa chỉ quay về của thủ tục `sumSquare` (trong thanh ghi \$ra) **trước khi gọi thủ tục mult**
→ Sử dụng ngăn xếp (Stack)

Ngăn xếp (Stack)

- Là ngăn xếp gồm nhiều ô nhớ kết hợp (vùng nhớ) nằm trong bộ nhớ chính
- Cấu trúc dữ liệu lý tưởng để chứa tạm các giá trị trong thanh ghi
 - Thường chứa **địa chỉ trả về**, các **biến cục bộ** của trình con, nhất là các biến có cấu trúc (array, list...) không chứa vừa trong các thanh ghi trong CPU
- Được định vị và quản lý bởi **stack pointer**
- Có 2 tác vụ hoạt động cơ bản:
 - **push**: Đưa dữ liệu từ thanh ghi vào stack
 - **pop**: Lấy dữ liệu từ stack chép vào thanh ghi
- Trong MIPS dành sẵn 1 thanh ghi **\$sp** để lưu trữ stack pointer
- Để sử dụng Stack, cần khai báo kích vùng Stack bằng cách tăng (push) giá trị con trả ngăn xếp stack pointer (lưu trữ trong thanh ghi **\$sp**)
 - Lưu ý: Stack pointer tăng theo chiều **giảm địa chỉ**

C
int sumSquare (int x, int y) { return mult (x, x) + y; }
/* x: \$a0, y: \$a1 */

sumSquare:

M	init	addi \$sp, \$sp, -8	# khai báo kích thước stack cần dùng = 8 byte
I	push	sw \$ra, 4 (\$sp)	# cất địa chỉ quay về của thủ tục sumSquare đưa vào stack
P	push	sw \$a1, 0 (\$sp)	# cất giá trị y vào stack
S		add \$a1, \$a0, \$zero	# gán tham số thứ 2 là x (ban đầu là y) để phục vụ cho thủ tục mult sắp gọi
		jal mult	# nhảy đến thủ tục mult
		lw \$a1, 0 (\$sp)	# sau khi thực thi xong thủ tục mult , khôi phục lại tham số thứ 2 = y
	pop		# dựa trên giá trị đã lưu trước đó trong stack
		add \$v0, \$v0, \$a1	# mult() + y
		lw \$ra, 4 (\$sp)	# khôi phục địa chỉ quay về của thủ tục sumSquare từ stack, đưa lại vào \$ra
	pop	addi \$sp, \$sp, 8	# khôi phục 8 byte giá trị \$sp ban đầu đã "mượn", kết thúc stack
		jr \$ra	# nhảy đến đoạn lệnh ngay sau khi gọi thủ tục sumSquare trong chương trình chính, để thao tác tiếp các lệnh khác.
	free		
	mult:	...	# lệnh xử lý cho thủ tục mult
		jr \$ra	# nhảy lại đoạn lệnh ngay sau khi gọi thủ tục mult trong thủ tục sumSquare

Tổng quát: Thao tác với stack

- Khởi tạo stack (init)
- Lưu trữ tạm các dữ liệu cần thiết vào stack (push)
- Gán các đối số (nếu có)
- Gọi lệnh jal để nhảy đến các thủ tục con
- Khôi phục các dữ liệu đã lưu tạm từ stack (pop)

Cụ thể hóa

- **Đầu thủ tục:**

Procedure_Label:

```
addi $sp, $sp, -framesize      # khởi tạo stack, dịch chuyển stack pointer $sp lùi  
sw $ra, framesize - 4 ($sp)    # cất $ra (kích thước 4 byte) vào stack (push)
```

Lưu tạm các thanh ghi khác (nếu cần)

- **Thân thủ tục:**

```
jal other_procedure           # Gọi các thủ tục khác (nếu cần)
```

- **Cuối thủ tục:**

```
lw $ra, frame_size - 4 ($sp)  # khôi phục $ra từ stack (pop)
```

```
lw ...                         # khôi phục các thanh ghi khác (nếu cần)
```

```
addi $sp, $sp, framesize       # khôi phục $sp, giải phóng stack
```

```
jr $ra                          # nhảy đến lệnh tiếp theo “Procedure Label”
```

```
# trong chương trình chính
```

Một số nguyên tắc khi thực thi thủ tục

- Nhảy đến thủ tục bằng lệnh **jal** và quay về nơi trước đó đã gọi nó bằng lệnh **jr \$ra**
- 4 thanh ghi chứa đối số của thủ tục: **\$a0, \$a1, \$a2, \$a3**
- Kết quả trả về của thủ tục chứa trong thanh ghi **\$v0** (và **\$v1** nếu cần)
- Phải tuân theo **nguyên tắc sử dụng các thanh ghi** (register conventions)

Nguyên tắc sử dụng thanh ghi

- **\$0:** (Không thay đổi) Luôn bằng 0
- **\$s0 - \$s7:** (Khôi phục lại nếu thay đổi) Rất quan trọng, nếu thủ tục được gọi (callee) thay đổi các thanh ghi này thì nó phải khôi phục lại giá trị các thanh ghi này trước khi kết thúc
- **\$sp:** (Khôi phục lại nếu thay đổi) Thanh ghi con trả stack phải có giá trị không đổi trước và sau khi gọi lệnh “jal”, nếu không thủ tục gọi (caller) sẽ không quay về được.

Nguyên tắc sử dụng thanh ghi

- \$ra: (Có thể thay đổi) Khi gọi lệnh “jal” sẽ làm thay đổi giá trị thanh ghi này. Thủ tục gọi (caller) lưu lại (backup) giá trị của thanh ghi \$ra vào stack nếu cần
- \$v0 - \$v1: (Có thể thay đổi) Chứa kết quả trả về của thủ tục
- \$a0 - \$a1: (Có thể thay đổi) Chứa đối số của thủ tục
- \$t0 - \$t9: (Có thể thay đổi) Đây là các thanh ghi

Tóm tắt

- Nếu thủ tục R gọi thủ tục E:
 - R phải lưu vào stack các thanh ghi tạm có thể bị sử dụng trong E trước khi gọi lệnh **jal E** (goto E)
 - E phải lưu lại giá trị các thanh ghi lưu trữ (**\$s0 - \$s7**) nếu nó muốn sử dụng các thanh ghi này → trước khi kết thúc E sẽ khôi phục lại giá trị của chúng
 - **Nhớ:** Thủ tục gọi **R (caller)** và Thủ tục được gọi **E (callee)** chỉ cần lưu các thanh ghi tạm / thanh ghi lưu trữ mà nó muốn dùng, không phải tất cả các thanh ghi!

Bảng tóm tắt

Name	Register number	Usage	Preserved on call?
\$zero	0	the constant value 0	n.a.
\$v0-\$v1	2-3	values for results and expression evaluation	no
\$a0-\$a3	4-7	arguments	no
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

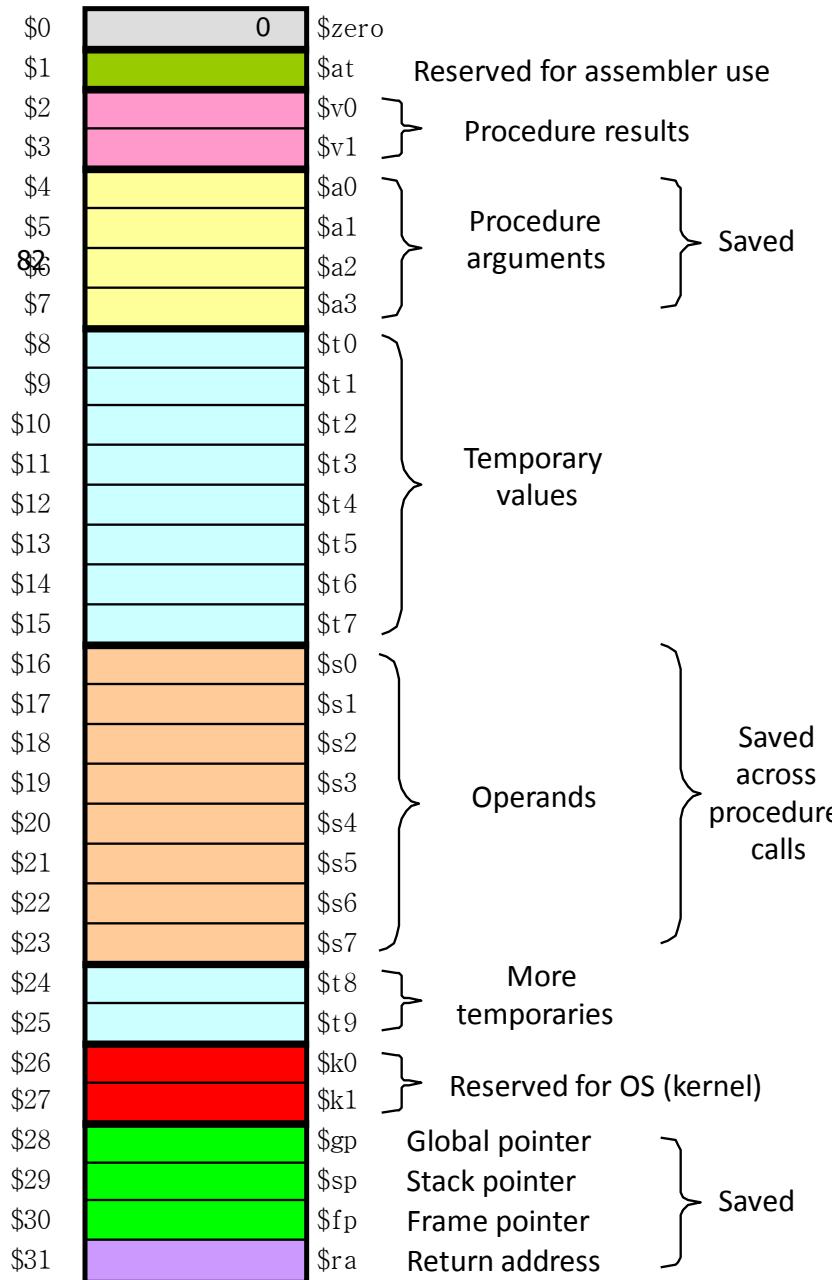
System call

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = integer	
read_character	12		char (in \$v0)

Hello.asm

```
.data          # data segment
str: .asciiz "Hello asm !"
.text          # text segment
.globl main
main:          # starting point of program
    addi $v0, $0, 4      # $v0 = 0 + 4 = 4 → print str syscall
    la $a0, str          # $a0 = address(str)
    syscall              # execute the system call
```

PHỤ LỤC



Phụ lục 1: 40 lệnh cơ bản

MTPS

Instruction	Usage
Load upper immediate	lui rt, imm
Add	add rd, rs, rt
Subtract	sub rd, rs, rt
Set less than	slt rd, rs, rt
Add immediate	addi rt, rs, imm
Set less than immediate	slti rd, rs, imm
AND	and rd, rs, rt
OR	or rd, rs, rt
XOR	xor rd, rs, rt
NOR	nor rd, rs, rt
AND immediate	andi rt, rs, imm
OR immediate	ori rt, rs, imm
XOR immediate	xori rt, rs, imm
Load word	lw rt, imm(rs)
Store word	sw rt, imm(rs)
Jump	j L
Jump register	jr rs
Branch less than 0	bltz rs, L
Branch equal	beq rs, rt, L
Branch not equal	bne rs, rt, L

Instruction	Usage
Move from Hi	mfhi rd
Move from Lo	mflo rd
Add unsigned	addu rd, rs, rt
Subtract unsigned	subu rd, rs, rt
Multiply	mult rs, rt
Multiply unsigned	multu rs, rt
Divide	div rs, rt
Divide unsigned	divu rs, rt
Add immediate unsigned	addiu rs, rt, imm
Shift left logical	sll rd, rt, sh
Shift right logical	srl rd, rt, sh
Shift right arithmetic	sra rd, rt, sh
Shift left logical variable	sllv rd, rt, rs
Shift right logical variable	srlv rd, rt, rs
Shift right arith variable	sraw rd, rt, rs
Load byte	lb rt, imm(rs)
Load byte unsigned	lbu rt, imm(rs)
Store byte	sb rt, imm(rs)
Jump and link	jal L
System call	syscall

Phụ lục 2: Pseudo Instructions

- “Lệnh giả”: Mặc định không được hỗ trợ bởi MIPS
 - Là những lệnh cần phải biên dịch thành rất nhiều câu lệnh thật trước khi được thực hiện bởi phần cứng
- Lệnh giả = Thủ tục
- Dùng để hỗ trợ lập trình viên thao tác nhanh chóng với những thao tác phức tạp gồm nhiều bước

Ví dụ: Tính $\$s1 = |\$s0|$

- Để tính được trị tuyệt đối của $\$s0 \rightarrow \$s1$, ta có lệnh giả là: **abs \$s1, \$s0**
- Thực sự MIPS không có lệnh này, khi chạy sẽ biên dịch lệnh này thành các lệnh thật sau:

Trị tuyệt đối của X là $-X$ nếu $X < 0$, là X nếu $X \geq 0$

abs:

```
sub    $s1, $zero, $s0  
slt    $t0, $s0, $zero  
bne    $t0, $zero, done  
add    $s1, $s0, $zero
```

done:

```
jr    $ra
```

Một số lệnh giả phổ biến của MIPS

Name	instruction syntax	meaning
Move	move rd, rs	rd = rs
Load Address	la rd, rs	rd = address (rs)
Load Immediate	li rd, imm	rd = 32 bit Immediate value
Branch greater than	bgt rs, rt, Label	if($R[rs] > R[rt]$) PC=Label
Branch less than	blt rs, rt, Label	if($R[rs] < R[rt]$) PC=Label
Branch greater than or equal	bge rs, rt, Label	if($R[rs] \geq R[rt]$) PC=Label
branch less than or equal	ble rs, rt, Label	if($R[rs] \leq R[rt]$) PC=Label
branch greater than unsigned	bgtu rs, rt, Label	if($R[rs] \leq R[rt]$) PC=Label
branch greater than zero	bgtz rs, Label	if($R[rs] \geq 0$) PC=Label

Phụ lục 3: Biểu diễn lệnh trong ngôn ngữ máy

- Chúng ta đã học 1 số nhóm lệnh hợp ngữ thao tác trên CPU tuy nhiên...
- CPU có hiểu các lệnh hợp ngữ đã học này không?
→ Tất nhiên là không vì nó chỉ hiểu được ngôn ngữ máy gồm toàn bit 0 và 1
- Dãy bit đó gọi là **lệnh máy** (machine language instruction)
- Mỗi lệnh máy có kích thước **32 bit**, được chia thành các **nhóm bit**, **gọi là trường** (field), mỗi nhóm có 1 vai trò trong lệnh máy
- Lệnh máy có 1 cấu trúc xác định gọi là **cấu trúc lệnh** (Instruction Format)

MIPS Instruction Format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address				Jump instruction format	

- Có 3 format lệnh trong MIPS:
 - **R-format:** Dùng trong các lệnh tính toán số học (add, sub, and, or, nor, sll, srl, sra...)
 - **I-format:** Dùng trong các lệnh thao tác với hằng số, chuyển dữ liệu với bộ nhớ, rẽ nhánh
 - **J-format:** Dùng trong các lệnh nhảy (jump – C: goto)

R-format

6 bits	5	5	5	5	6
opcode	rs	rt	rd	shmat	funct

- **opcode** (operation code): mã thao tác, cho biết lệnh làm gì
- **funct** (function code): kết hợp với opcode để xác định lệnh làm gì (trường hợp các lệnh có cùng mã thao tác với opcode)
- **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- **rt** (target register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 2
- **rd** (destination register): thanh ghi đích, thường chứa kết quả lệnh
- **shamt**: chứa số bit cần dịch trong các lệnh dịch, nếu không phải lệnh dịch thì trường này có giá trị 0

Nhận xét

- Các trường lưu địa chỉ thanh ghi **rs, rt, rd** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để biểu diễn 32 thanh ghi của MIPS
- Trường lưu số bit cần dịch **shamt** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để dịch hết 32 bit lưu trữ của 1 thanh ghi

Ví dụ R-format (1)

- Biểu diễn machine code của lệnh: **add \$t0, \$t1, \$t2**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	9	10	8	0	32
000000	01001	01010	01000	00000	100000

- $\text{opcode} = 0$ } Xác định thao tác cộng
- $\text{funct} = 32$ } (tất cả các lệnh theo cấu trúc R-format đều có $\text{opcode} = 0$)
- $\text{rs} = 9$ (toán hạng nguồn thứ 1 là $\$t1 \sim \9)
- $\text{rt} = 10$ (toán hạng nguồn thứ 2 là $\$t2 \sim \10)
- $\text{rd} = 8$ (toán hạng đích là $\$t0 \sim \8)
- $\text{shmat} = 0$ (không phải lệnh đích)

Ví dụ R-format (2)

- Biểu diễn machine code của lệnh: **sll \$t2, \$s0, 4**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

- $\text{opcode} = 0$] Xác định thao tác dịch trái luận lý
- $\text{funct} = 0$] (tất cả các lệnh theo cấu trúc R-format đều có $\text{opcode} = 0$)
- $\text{rs} = 0$ (không dùng trong phép dịch)
- $\text{rt} = 16$ (toán hạng nguồn là $\$s0 \sim \16)
- $\text{rd} = 10$ (toán hạng đích là $\$t2 \sim \10)
- $\text{shmat} = 4$ (số bit dịch = 4)

Vấn đề của R-format

- Làm sao giải quyết trường hợp nếu câu lệnh đòi hỏi trường dành cho toán hạng phải lớn hơn 5 bit?
- Ví dụ:
 - Lệnh **addi** cộng giá trị thanh ghi với 1 hằng số, nếu giới hạn trường hằng số ở 5 bit
→ hằng số không thể lớn hơn $2^5 = 32$
→ Giới hạn khả năng tính toán số học!
 - Lệnh **lw, sw** cần biểu diễn 2 thanh ghi và 1 hằng số offset, nếu giới hạn ở 5 bit
→ Giới hạn khả năng truy xuất dữ liệu trong bộ nhớ
 - Lệnh **beq, bne** cần biểu diễn 2 thanh ghi và 1 hằng số chứa địa chỉ (nhãn) cần nhảy, nếu giới hạn ở 5 bit
→ Giới hạn lưu trữ chương trình trong bộ nhớ
- Giải pháp: Dùng **I-format** cho các lệnh thao tác hằng số, truy xuất dữ liệu bộ nhớ và rẽ nhánh

I-format

6 bits	5	5	16
opcode	rs	rt	immediate

- **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format, chỉ khác không cần thêm trường funct)
 - Đây cũng là lý do tại sao R-format có 2 trường 6 bit để xác định lệnh làm gì thay vì 1 trường 12 bit → Để nhất quán với các cấu trúc lệnh khác (I-format) trong khi kích thước mỗi trường vẫn hợp lý
- **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- **rt** (target register): thanh ghi đích, thường chứa kết quả lệnh
- **immediate**: 16 bit, có thể biểu diễn số nguyên từ -2^{15} đến $(2^{15} - 1)$
 - I-format đã có thể lưu hằng số 16 bit (thay vì 5 bit như R-format)

Ví dụ I-format

- Biểu diễn machine code của lệnh: **addi \$s0, \$s1, 10**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	immediate
8	17	16	10
001000	10001	10000	0000 0000 0000 0000 1010

- opcode = 8: Xác định thao tác cộng hằng số
- rs = 17 (toán hạng nguồn thứ 1 là \$s1 ~ \$17)
- rt = 16 (toán hạng đích là \$s0 ~ \$16)
- immediate = 10 (toán hạng nguồn thứ 2 = hằng số = 10)

Vấn đề I-format

- Trường hằng số (immediate) có kích thước 16 bit
- Nếu muốn thao tác với các hằng số 32 bit?
 - Tăng kích thước trường immediate thành 32 bit?
 - Tăng kích thước các lệnh thao tác với hằng số có cấu trúc I-format

Vấn đề I-format (tt)

- Giải pháp: MIPS cung cấp lệnh mới “lui”
 - lui register, immediate
 - Load Upper Immediate
 - Đưa hằng số 16 bit vào 2 byte cao của 1 thanh ghi
 - Giá trị 2 byte thấp của thanh ghi đó gán = 0
 - Lệnh này có cấu trúc I-format

Ví dụ

- Muốn cộng giá trị 32 bit **0xABABCD** vào thanh ghi **\$t0** ?
 - Không thể dùng:

addi \$t0, \$t0, 0xABABCD

- Giải pháp dùng lệnh **lui**:

lui \$at, 0xABAB

ori \$at, \$at, 0xCD

add \$t0, \$0, \$at

Vấn đề rẽ nhánh có điều kiện trong I-format

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

6 bits	5	5	16
opcode	rs	rt	immediate

- opcode:** xác định lệnh beq hay bne
- rs, rt:** chứa các giá trị của thanh ghi cần so sánh
- immediate** chứa địa chỉ (nhãn) cần nhảy tới?
 - immediate chỉ có 16 bit → chỉ có thể nhảy tới địa chỉ từ 0 – 2^{16} (65535) ?
- Chương trình bị giới hạn không gian rất nhiều
- Câu trả lời:** immediate KHÔNG phải chứa địa chỉ cần nhảy tới

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

- Trong MIPS, thanh ghi **PC** (Program Counter) sẽ chứa địa chỉ của lệnh **đang được thực hiện**
- **immediate**: số có dấu, chứa khoảng cách so với địa chỉ lệnh đang thực hiện nằm trong thanh ghi PC
 - immediate + PC → địa chỉ cần nhảy tới
- Cách xác định địa chỉ này gọi là **PC-Relative Addressing** (định vị theo thanh ghi PC)
 - Xem slide “Addressing Mode” (phần sau) để biết thêm về các Addressing mode trong MIPS

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

- Mỗi lệnh trong MIPS có kích thước 32 bit (1 word – 1 từ nhớ)
- MIPS truy xuất bộ nhớ theo nguyên tắc Alignment Restriction

→Đơn vị của immediate, khoảng cách so với PC, là từ nhớ (word = 4 byte) chứ không phải là byte

→Các lệnh rẽ nhánh có thể nhảy tới các địa chỉ có khoảng cách $\pm 2^{15}$ word tính từ địa chỉ lưu trong PC ($\pm 2^{17}$ byte)

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

- Cách tính địa chỉ rẽ nhánh:
 - Nếu không rẽ nhánh:
$$PC = PC + 4 = \text{địa chỉ lệnh kế tiếp trong bộ nhớ}$$
 - Nếu thực hiện rẽ nhánh:
$$PC = (PC + 4) + (\text{immediate} * 4)$$
- Vì sao cộng immediate với $(PC + 4)$ thay vì PC ? \rightarrow Khi rẽ nhánh bị delayed 1 lệnh kế với lệnh rẽ nhánh
- Nhận xét: immediate cho biết số lệnh cần nhảy qua để đến được nhãn

Ví dụ I-format

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

End: ...

opcode	rs	rt	immediate
4	9	0	3
000100	01001	00000	0000 0000 0000 0000 0011

- rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)
- rt = 0 (toán hạng nguồn thứ 2 là \$0 ~ \$0)
- immediate = 3 (nhảy qua 3 lệnh kể từ lệnh rẽ nhánh có điều kiện)

Vấn đề I-format

- Mỗi lệnh trong MIPS có kích thước 32 bit
- Mong muốn: Có thể nhảy đến bất kỳ lệnh nào (MIPS hỗ trợ các hàm nhảy không điều kiện như **j**)
 - Nhảy trong khoảng 2^{32} (4 GB) bộ nhớ
 - I-format bị hạn chế giới hạn vùng nhảy
 - Dùng **J-format**
- Tuy nhiên, dù format nào cũng phải **cần tối thiểu 6 bit cho opcode** để nhất quán lệnh với các format khác
 - **J-format** chỉ có thể dùng $32 - 6 = 26$ bit để biểu diễn khoảng cách nhảy

J-format

6 bits	26
opcode	target address

- ❑ **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format và I-format)
 - ❑ Để nhất quán với các cấu trúc lệnh khác (R-format và I-format)
- ❑ **target address**: Lưu địa chỉ đích của lệnh nhảy
 - ❑ Tương tự lệnh rẽ nhánh, địa chỉ đích của lệnh nhảy tính theo đơn vị word

Nhận xét

- Trong J-format, các lệnh nhảy có thể nhảy tới các lệnh có địa chỉ trong khoảng 2^{26}
- Muốn nhảy tới các lệnh có địa chỉ lớn hơn từ 2^{27} đến 2^{32} ?
 - MIPS hỗ trợ lệnh **jr** (đọc trong phần thủ tục)
 - Tuy nhiên nhu cầu này không cần thiết lắm vì chương trình thường không quá lớn như vậy

Bảng tóm tắt Format

	op	rs	rt	rd	sh	fn				
R	31 6 bits	25 5 bits	20 5 bits	15 5 bits	10 5 bits	5 6 bits	0			
	Opcode	Source register 1	Source register 2	Destination register	Shift amount	Opcode extension				
I	31 6 bits	25 5 bits	20 5 bits	15	operand / offset					
	Opcode	Source or base	Destination or data		16 bits					
J	31 6 bits	25	jump target address							
	Opcode		26 bits							
			Memory word address (byte address divided by 4)							

Phụ lục 4: Addressing mode

- Là phương thức định vị trí (địa chỉ hóa) các toán hạng trong kiến trúc MIPS
- Có 5 phương pháp chính:
 - **Immediate addressing** (Vd: addi \$t0, \$t0, 5)
Toán hạng = hằng số 16 bit trong câu lệnh
 - **Register addressing** (Vd: add \$t0, \$t0, \$t1)
Toán hạng = nội dung thanh ghi
 - **Base addressing** (Vd: lw \$t1, 8(\$t0))
Toán hạng = nội dung ô nhớ (địa chỉ ô nhớ = nội dung thanh ghi + hằng số 16 bit trong câu lệnh)
 - **PC-relative addressing** (Vd: beq \$t0, \$t1, Label)
Toán hạng = địa chỉ đích lệnh nhảy = nội dung thanh ghi PC + hằng số 16 bit trong câu lệnh
 - **Pseudodirect addressing** (Vd: j 2500)
Toán hạng = địa chỉ đích lệnh nhảy = các bit cao thanh ghi PC + hằng số 26 bit trong

Addressing mode

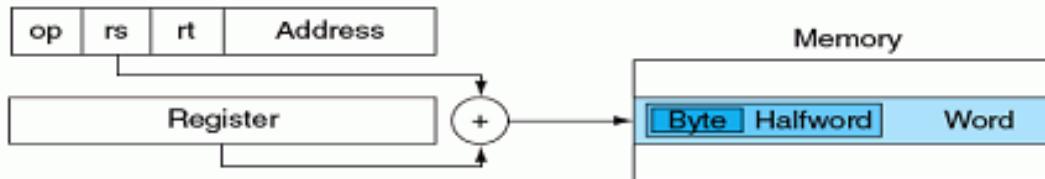
1. Immediate addressing



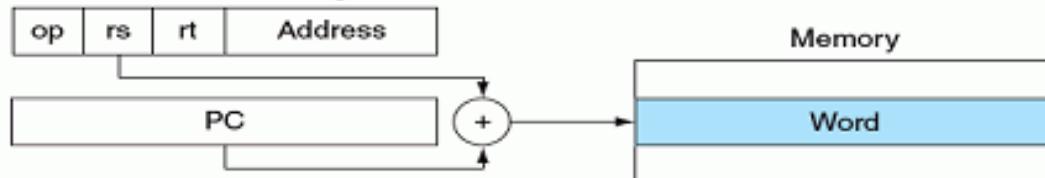
2. Register addressing



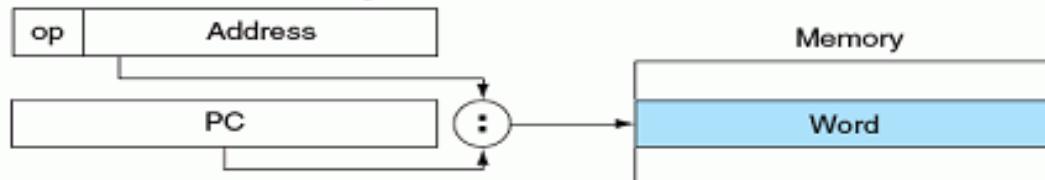
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Homework

- Sách Pettersson & Hennessy: Đọc hết chương 2
- Tài liệu tham khảo: Đọc “08_HP_AppA.pdf”