



TRƯỜNG ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

KIẾN TRÚC MÁY TÍNH

Computer Architecture

Course ID: IT3030

Nguyễn Kim Khánh

Nội dung học phần

Chương 1. Giới thiệu chung

Chương 2. Hệ thống máy tính

Chương 3. Số học và logic máy tính

Chương 4. Kiến trúc tập lệnh

Chương 5. Bộ xử lý

Chương 6. Bộ nhớ máy tính

Chương 7. Hệ thống vào-ra

Chương 8. Các kiến trúc song song

Chương 4

KIẾN TRÚC TẬP LỆNH

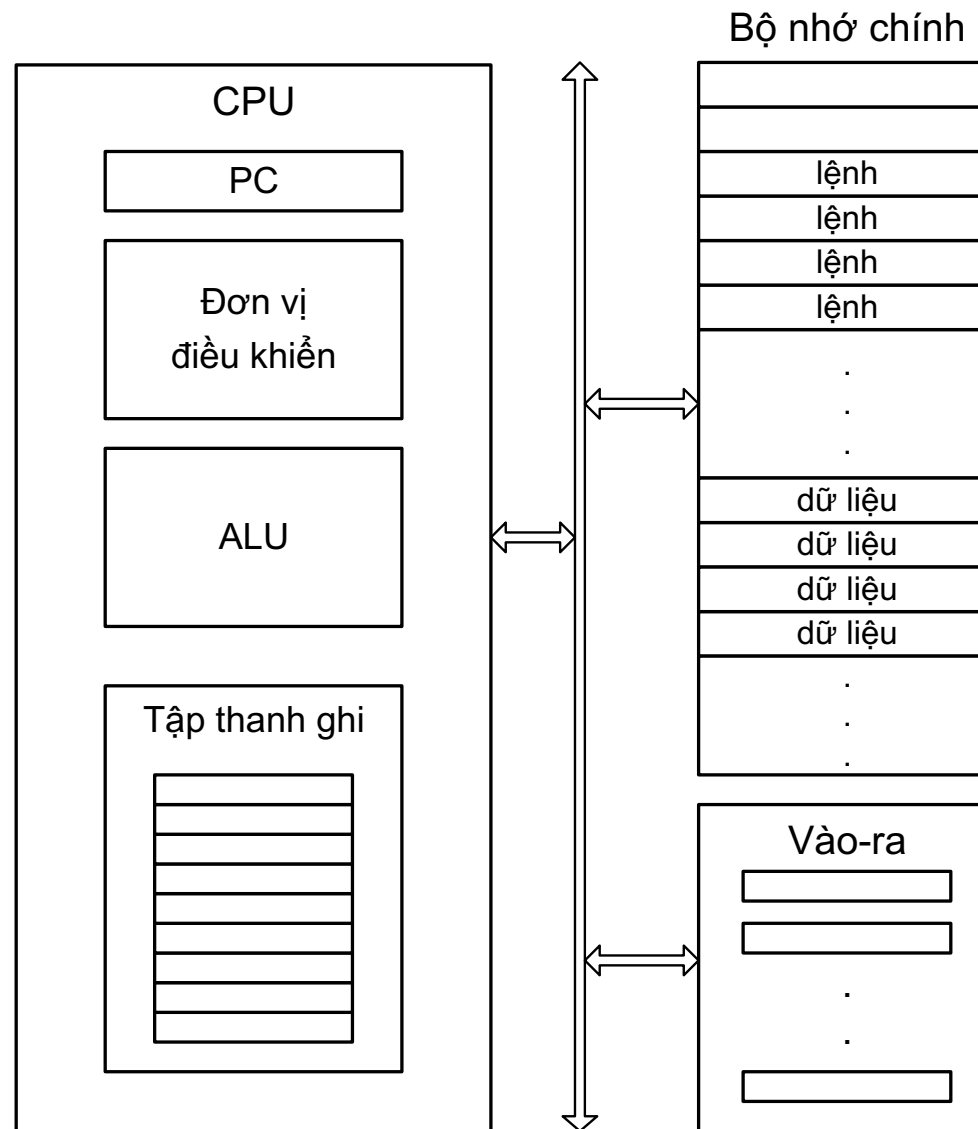
Nội dung của chương 4

- 4.1. Giới thiệu chung về kiến trúc tập lệnh
- 4.2. Lệnh hợp ngữ và toán hạng
- 4.3. Mã máy
- 4.4. Các lệnh logic
- 4.5. Tạo các cấu trúc điều khiển
- 4.6. Chương trình con
- 4.7. Các lệnh xử lý ký tự
- 4.8. Các lệnh nhân chia số nguyên
- 4.9. Xử lý số dấu phẩy động
- 4.10. Mảng và con trỏ
- 4.11. Các phương pháp định địa chỉ
- 4.12. Dịch và chạy chương trình hợp ngữ

4.1. Giới thiệu chung về kiến trúc tập lệnh

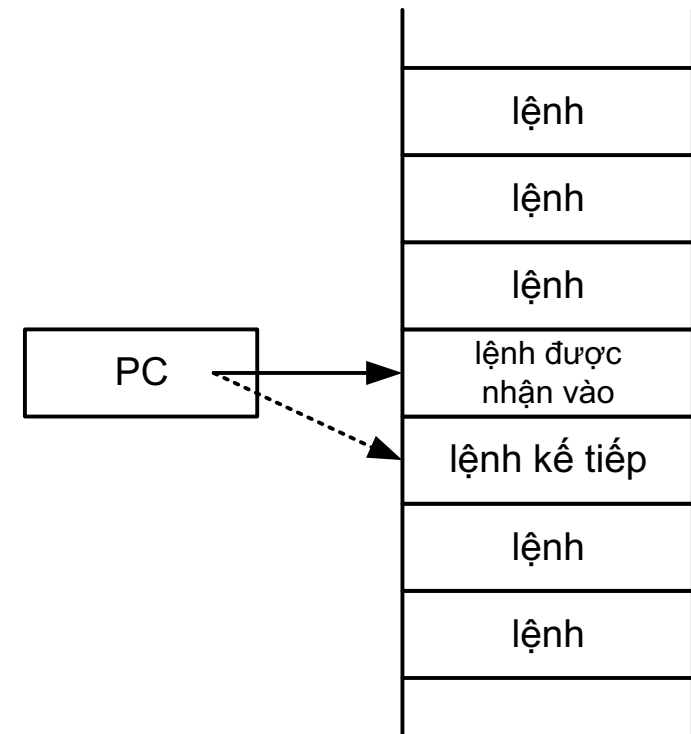
- **Kiến trúc tập lệnh** (Instruction Set Architecture): cách nhìn máy tính bởi người lập trình
- **Vi kiến trúc** (Microarchitecture): cách thực hiện kiến trúc tập lệnh bằng phần cứng
- **Ngôn ngữ trong máy tính:**
 - **Hợp ngữ (assembly language):**
 - dạng lệnh có thể đọc được bởi con người
 - biểu diễn dạng text
 - **Ngôn ngữ máy (machine language):**
 - còn gọi là mã máy (machine code)
 - dạng lệnh có thể đọc được bởi máy tính
 - biểu diễn bằng các bit 0 và 1

Mô hình lập trình của máy tính



CPU nhận lệnh từ bộ nhớ

- Bộ đếm chương trình PC (Program Counter) là thanh ghi của CPU giữ địa chỉ của lệnh cần nhận vào để thực hiện
- CPU phát địa chỉ từ PC đến bộ nhớ, lệnh được nhận vào
- Sau khi lệnh được nhận vào, nội dung PC tự động tăng để trở sang lệnh kế tiếp
- PC tăng bao nhiêu?
 - Tùy thuộc vào độ dài của lệnh vừa được nhận
 - MIPS: lệnh có độ dài 32-bit, PC tăng



Giải mã và thực hiện lệnh

- Bộ xử lý giải mã lệnh đã được nhận và phát các tín hiệu điều khiển thực hiện thao tác mà lệnh yêu cầu
- Các kiểu thao tác chính của lệnh:
 - Trao đổi dữ liệu giữa CPU với bộ nhớ chính hoặc với cổng vào-ra
 - Thực hiện các phép toán số học hoặc phép toán logic với các dữ liệu (được thực hiện bởi ALU)
 - Chuyển điều khiển trong chương trình (rẽ nhánh, nhảy)

CPU đọc/ghi dữ liệu bộ nhớ

- Với các lệnh trao đổi dữ liệu với bộ nhớ, CPU cần biết và phát ra địa chỉ của ngăn nhớ cần đọc/ghi
- Địa chỉ đó có thể là:
 - Hằng số địa chỉ được cho trực tiếp trong lệnh
 - Giá trị địa chỉ nằm trong thanh ghi con trỏ
 - Địa chỉ = Địa chỉ cơ sở + giá trị dịch chuyển

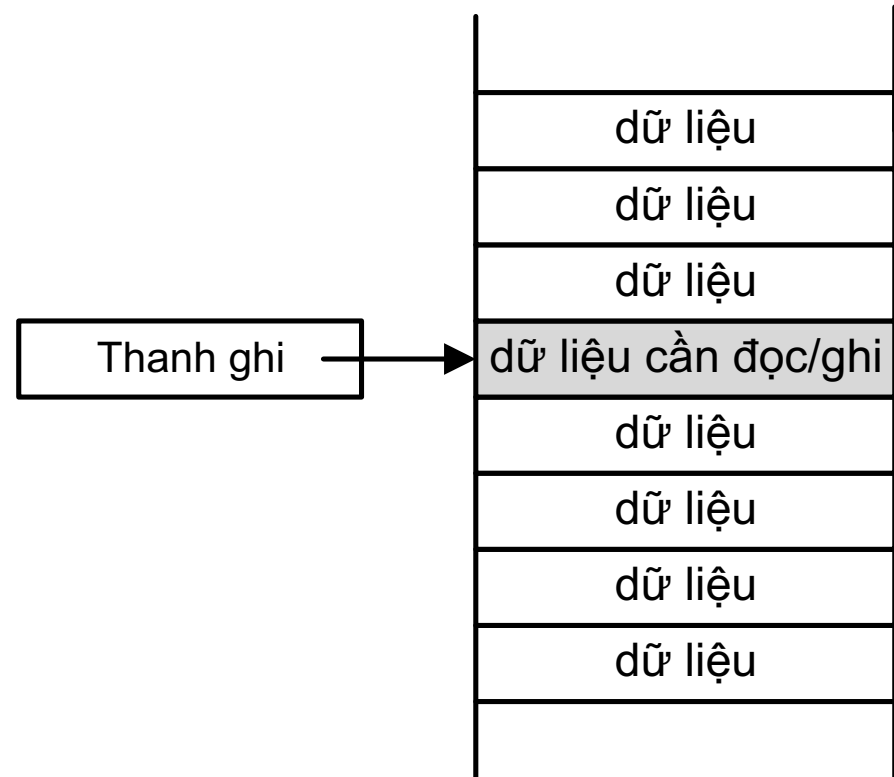
Hàng số địa chỉ

- Trong lệnh cho hàng số địa chỉ cụ thể
- CPU phát giá trị địa chỉ này đến bộ nhớ để tìm ra ngăn nhớ dữ liệu cần đọc/ghi



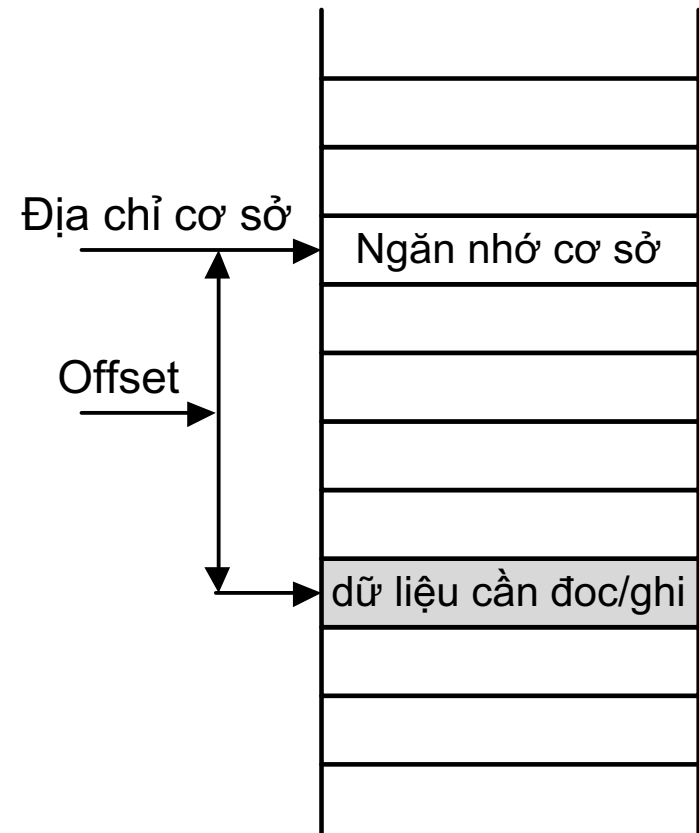
Sử dụng thanh ghi con trỏ

- Trong lệnh cho biết tên thanh ghi con trỏ
- Thanh ghi con trỏ chứa giá trị địa chỉ
- CPU phát địa chỉ này ra để tìm ra ngăn nhớ dữ liệu cần đọc/ghi



Sử dụng địa chỉ cơ sở và dịch chuyển

- Địa chỉ cơ sở (base address): địa chỉ của ngăn nhớ cơ sở
- Giá trị dịch chuyển địa chỉ (offset): giá số địa chỉ giữa ngăn nhớ cần đọc/ghi so với ngăn nhớ cơ sở
- Địa chỉ của ngăn nhớ cần đọc/ghi = (địa chỉ cơ sở) + (offset)
- Có thể sử dụng các thanh ghi để quản lý các tham số này
- Trường hợp riêng:
 - Địa chỉ cơ sở = 0
 - Offset = 0

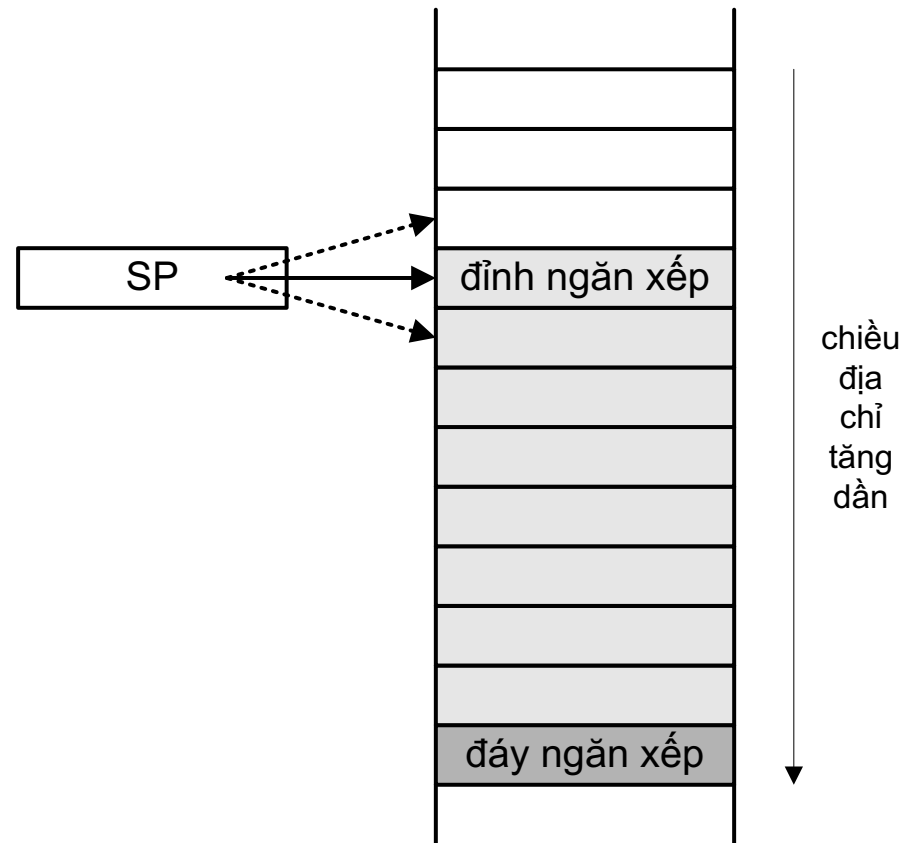


Ngăn xếp (Stack)

- Ngăn xếp là vùng nhớ dữ liệu có cấu trúc LIFO (Last In - First Out vào sau - ra trước)
- Ngăn xếp thường dùng để phục vụ cho chương trình con
- Đáy ngăn xếp là một ngăn nhớ xác định
- Đỉnh ngăn xếp là thông tin nằm ở vị trí trên cùng trong ngăn xếp
- Đỉnh ngăn xếp có thể bị thay đổi

Con trỏ ngăn xếp SP (Stack Pointer)

- SP là thanh ghi chứa địa chỉ của ngăn nhớ đỉnh ngăn xếp
- Khi cất thêm một thông tin vào ngăn xếp:
 - Giảm nội dung của SP
 - Thông tin được cất vào ngăn nhớ được trỏ bởi SP
- Khi lấy một thông tin ra khỏi ngăn xếp:
 - Thông tin được đọc từ ngăn nhớ được trỏ bởi SP
 - Tăng nội dung của SP
- Khi ngăn xếp rỗng, SP trỏ vào đáy



Thứ tự lưu trữ các byte trong bộ nhớ chính

- Bộ nhớ chính được đánh địa chỉ cho từng byte
- Hai cách lưu trữ thông tin nhiều byte:
 - **Đầu nhỏ** (Little-endian): Byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ lớn.
 - **Đầu to** (Big-endian): Byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ lớn.
- Các sản phẩm thực tế:
 - Intel x86: little-endian
 - Motorola 680x0, SunSPARC: big-endian
 - MIPS, IA-64: bi-endian (cả hai kiểu)

Ví dụ lưu trữ dữ liệu 32-bit

Số
nhị phân

0001	1010	0010	1011	0011	1100	0100	1101
------	------	------	------	------	------	------	------

Số Hexa

1A

2B

3C

4D

4D	4000
3C	4001
2B	4002
1A	4003

little-endian

1A	4000
2B	4001
3C	4002
4D	4003

big-endian

Tập lệnh

- Mỗi bộ xử lý có một tập lệnh xác định
- Tập lệnh thường có hàng chục đến hàng trăm lệnh
- Mỗi lệnh máy (mã máy) là một chuỗi các bit (0,1) mà bộ xử lý hiểu được để thực hiện một thao tác xác định.
- Các lệnh được mô tả bằng các ký hiệu gợi nhớ dạng text, đó chính là các lệnh của hợp ngữ (assembly language)

Dạng lệnh hợp ngữ

- Mã C:

$$a = b + c;$$

- Ví dụ lệnh hợp ngữ:

`add a, b, c # a = b + c`

trong đó:

- `add`: ký hiệu gợi nhớ chỉ ra thao tác (phép toán) cần thực hiện.
 - Chú ý: mỗi lệnh chỉ thực hiện một thao tác
- `b, c`: các toán hạng nguồn cho thao tác
- `a`: toán hạng đích (nơi ghi kết quả)
- phần sau dấu `#` là lời giải thích (chỉ có tác dụng đến hết dòng)

Các thành phần của lệnh máy

Mã thao tác	Địa chỉ toán hạng
-------------	-------------------

- Mã thao tác (operation code hay opcode): mã hóa cho thao tác mà bộ xử lý phải thực hiện
 - Các thao tác chuyển dữ liệu
 - Các phép toán số học
 - Các phép toán logic
 - Các thao tác chuyển điều khiển (rẽ nhánh, nhảy)
- Địa chỉ toán hạng: chỉ ra nơi chứa các toán hạng mà thao tác sẽ tác động
 - Toán hạng có thể là:
 - Hằng số nằm ngay trong lệnh
 - Nội dung của thanh ghi
 - Nội dung của ngăn nhớ (hoặc cổng vào-ra)

Số lượng địa chỉ toán hạng trong lệnh

- Ba địa chỉ toán hạng:
 - `add r1, r2, r3` $\# r1 = r2 + r3$
 - Sử dụng phổ biến trên các kiến trúc hiện nay
- Hai địa chỉ toán hạng:
 - `add r1, r2` $\# r1 = r1 + r2$
 - Sử dụng trên Intel x86, Motorola 680x0
- Một địa chỉ toán hạng:
 - `add r1` $\# Acc = Acc + r1$
 - Được sử dụng trên kiến trúc thể hệ trước
- 0 địa chỉ toán hạng:
 - Các toán hạng đều được ngầm định ở ngăn xếp
 - Không thông dụng

Các kiến trúc tập lệnh CISC và RISC

- CISC: Complex Instruction Set Computer
 - Máy tính với tập lệnh phức tạp
 - Các bộ xử lý: Intel x86, Motorola 680x0
- RISC: Reduced Instruction Set Computer
 - Máy tính với tập lệnh thu gọn
 - SunSPARC, Power PC, MIPS, ARM ...
 - RISC đối nghịch với CISC
 - Kiến trúc tập lệnh tiên tiến

Các đặc trưng của kiến trúc RISC

- Số lượng lệnh ít
- Hầu hết các lệnh truy nhập toán hạng ở các thanh ghi
- Truy nhập bộ nhớ bằng các lệnh LOAD/STORE (nạp/lưu)
- Thời gian thực hiện các lệnh là như nhau
- Các lệnh có độ dài cố định (thường là 32 bit)
- Số lượng dạng lệnh ít
- Có ít phương pháp định địa chỉ toán hạng
- Có nhiều thanh ghi
- Hỗ trợ các thao tác của ngôn ngữ bậc cao

Kiến trúc tập lệnh MIPS

- MIPS viết tắt cho:
Microprocessor without Interlocked Pipeline Stages
- Được phát triển bởi John Hennessy và các đồng nghiệp ở đại học Stanford (1984)
- Được thương mại hóa bởi MIPS Technologies
- Năm 2013 công ty này được bán cho Imagination Technologies (imaginationtech.com)
- Là kiến trúc RISC điển hình, dễ học
- Được sử dụng trong nhiều sản phẩm thực tế
- Các phần tiếp theo trong chương này sẽ nghiên cứu kiến trúc tập lệnh MIPS 32-bit
 - Tài liệu: MIPS Reference Data Sheet và Chapter 2 – COD

4.2. Lệnh hợp ngữ và các toán hạng

- Thực hiện phép cộng: 3 toán hạng
 - Là phép toán phổ biến nhất
 - Hai toán hạng nguồn và một toán hạng đích

`add a, b, c # a = b + c`

- Hầu hết các lệnh số học/logic có dạng trên
- Các lệnh số học sử dụng toán hạng thanh ghi hoặc hằng số

Tập thanh ghi của MIPS

- MIPS có tập 32 thanh ghi 32-bit
 - Được sử dụng thường xuyên
 - Được đánh số từ 0 đến 31 (mã hóa bằng 5-bit)
- Chương trình hợp dịch Assembler đặt tên:
 - Bắt đầu bằng dấu \$
 - \$t0, \$t1, ..., \$t9 chứa các giá trị tạm thời
 - \$s0, \$s1, ..., \$s7 cất các biến
- Qui ước gọi dữ liệu trong MIPS:
 - Dữ liệu 32-bit được gọi là “word”
 - Dữ liệu 16-bit được gọi là “halfword”

Tập thanh ghi của MIPS

Tên thanh ghi	Số hiệu thanh ghi	Công dụng
\$zero	0	the constant value 0, chứa hằng số = 0
\$at	1	assembler temporary, giá trị tạm thời cho hợp ngữ
\$v0-\$v1	2-3	procedure return values, các giá trị trả về của thủ tục
\$a0-\$a3	4-7	procedure arguments, các tham số vào của thủ tục
\$t0-\$t7	8-15	temporaries, chứa các giá trị tạm thời
\$s0-\$s7	16-23	saved variables, lưu các biến
\$t8-\$t9	24-25	more temporarie, chứa các giá trị tạm thời
\$k0-\$k1	26-27	OS temporaries, các giá trị tạm thời của OS
\$gp	28	global pointer, con trỏ toàn cục
\$sp	29	stack pointer, con trỏ ngăn xếp
\$fp	30	frame pointer, con trỏ khung
\$ra	31	procedure return address, địa chỉ trở về của thủ tục

Toán hạng thanh ghi

- Lệnh add, lệnh sub (subtract) chỉ thao tác với toán hạng thanh ghi

- `add rd, rs, rt` # $(rd) = (rs) + (rt)$

- `sub rd, rs, rt` # $(rd) = (rs) - (rt)$

- Ví dụ mã C:

`f = (g + h) - (i + j);`

- giả thiết: f, g, h, i, j nằm ở \$s0, \$s1, \$s2, \$s3, \$s4

- Được dịch thành mã hợp ngữ MIPS:

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

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

`sub $s0, $t0, $t1` # $f = (g+h) - (i+j)$

Toán hạng ở bộ nhớ

- Muốn thực hiện phép toán số học với toán hạng ở bộ nhớ, cần phải:
 - Nạp (load) giá trị từ bộ nhớ vào thanh ghi
 - Thực hiện phép toán trên thanh ghi
 - Lưu (store) kết quả từ thanh ghi ra bộ nhớ
- Bộ nhớ được đánh địa chỉ theo byte
 - MIPS sử dụng 32-bit để đánh địa chỉ cho các byte nhớ và các cổng vào-ra
 - Không gian địa chỉ: **0x00000000 – 0xFFFFFFFF**
 - Mỗi word có độ dài 32-bit chiếm 4-byte trong bộ nhớ, địa chỉ của các word là bội của 4 (địa chỉ của byte đầu tiên)
- MIPS cho phép lưu trữ trong bộ nhớ theo kiểu đầu to (big-endian) hoặc kiểu đầu nhỏ (little-endian)

Địa chỉ byte nhớ và word nhớ

Dữ liệu hoặc lệnh	Địa chỉ byte (theo Hexa)
byte (8-bit)	0x0000 0000
byte	0x0000 0001
byte	0x0000 0002
byte	0x0000 0003
byte	0x0000 0004
byte	0x0000 0005
byte	0x0000 0006
byte	0x0000 0007
.	
.	
.	
byte	0xFFFF FFFB
byte	0xFFFF FFFC
byte	0xFFFF FFFD
byte	0xFFFF FFFE
byte	0xFFFF FFFF

2^{32} bytes

Dữ liệu hoặc lệnh	Địa chỉ word (theo Hexa)
word (32-bit)	0x0000 0000
word	0x0000 0004
word	0x0000 0008
word	0x0000 000C
word	0x0000 0010
word	0x0000 0014
word	0x0000 0018
.	
.	
.	
word	0xFFFF FFF4
word	0xFFFF FFF8
word	0xFFFF FFFC

2^{30} words



Lệnh load và lệnh store

- Để đọc word dữ liệu 32-bit từ bộ nhớ đưa vào thanh ghi, sử dụng lệnh **load word**

`lw rt, imm(rs) # (rt) = mem[(rs)+imm]`

- rs: thanh ghi chứa địa chỉ cơ sở (base address)

- imm (immediate): hằng số (offset)

→ địa chỉ của word dữ liệu cần đọc = địa chỉ cơ sở + hằng số

- rt: thanh ghi đích, chứa word dữ liệu được đọc vào

- Để ghi word dữ liệu 32-bit từ thanh ghi đưa ra bộ nhớ, sử dụng lệnh **store word**

`sw rt, imm(rs) # mem[(rs)+imm] = (rt)`

- rt: thanh ghi nguồn, chứa word dữ liệu cần ghi ra bộ nhớ

- rs: thanh ghi chứa địa chỉ cơ sở (base address)

- imm: hằng số (offset)

→ địa chỉ nơi ghi word dữ liệu = địa chỉ cơ sở + hằng số

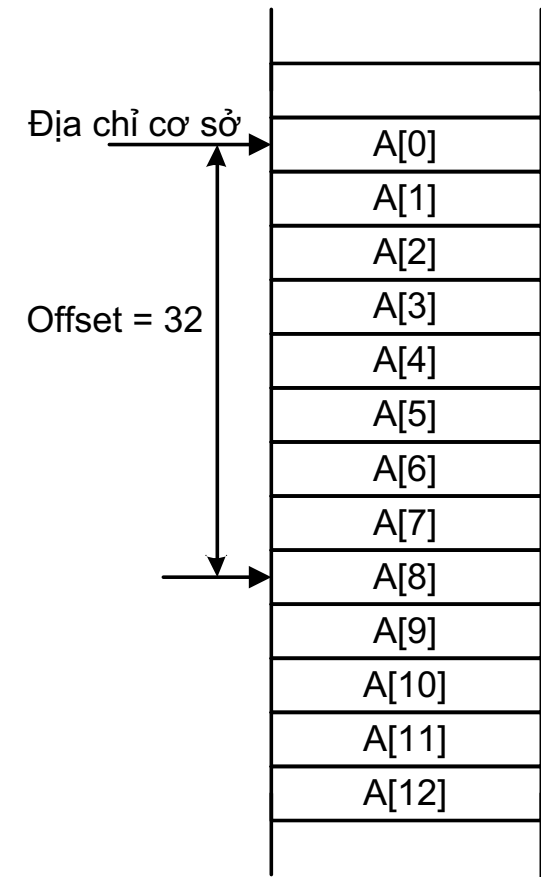
Ví dụ toán hạng bộ nhớ

- Mã C:

// A là mảng các phần tử 32-bit

$g = h + A[8];$

- Cho g ở \$s1, h ở \$s2
- \$s3 chứa địa chỉ cơ sở của mảng A



Ví dụ toán hạng bộ nhớ

■ Mã C:

// A là mảng các phần tử 32-bit

`g = h + A[8];`

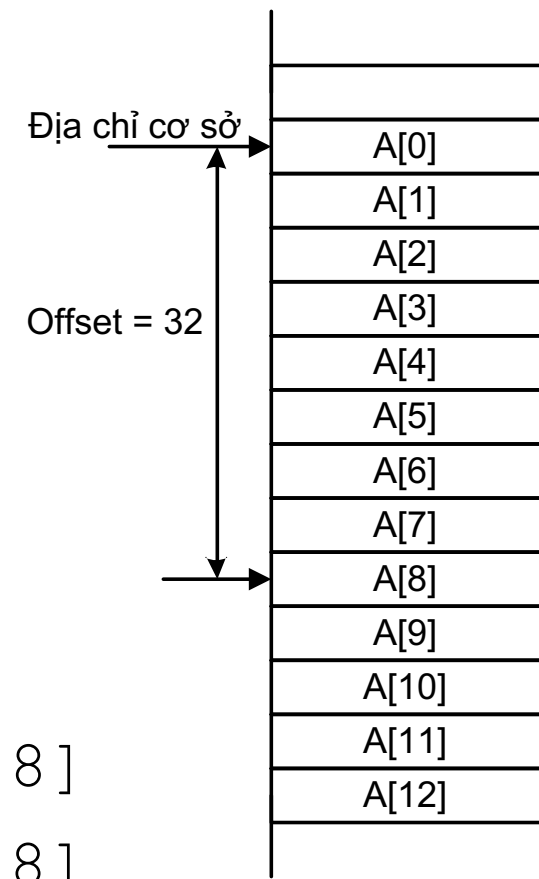
- Cho g ở \$s1, h ở \$s2
- \$s3 chứa địa chỉ cơ sở của mảng A

■ Mã hợp ngữ MIPS:

Chỉ số 8, do đó offset = 32

`lw $t0, 32($s3) # $t0 = A[8]`

`add $s1, $s2, $t0 # g = h + A[8]`



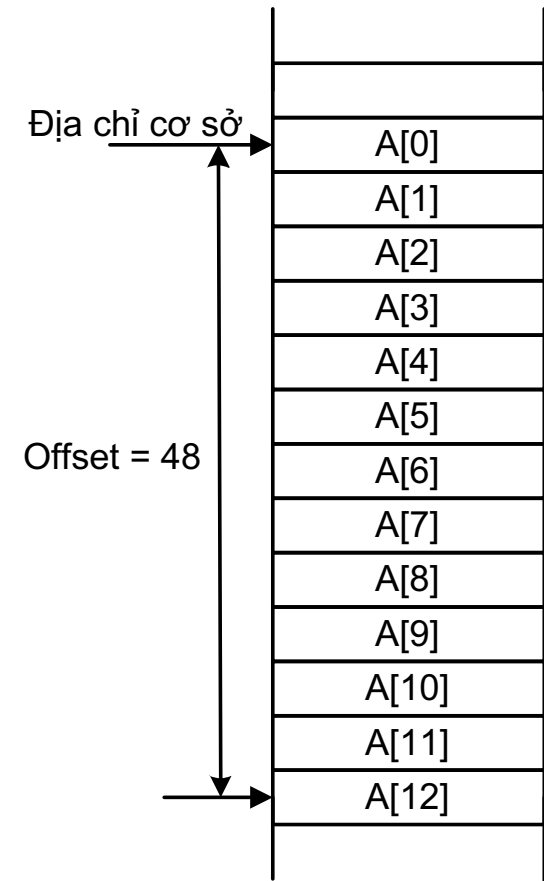
(Chú ý: offset phải là hằng số, có thể dương hoặc âm)

Ví dụ toán hạng bộ nhớ (tiếp)

- Mã C:

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

- h ở $\$s2$
- $\$s3$ chứa địa chỉ cơ sở của mảng A



Ví dụ toán hạng bộ nhớ (tiếp)

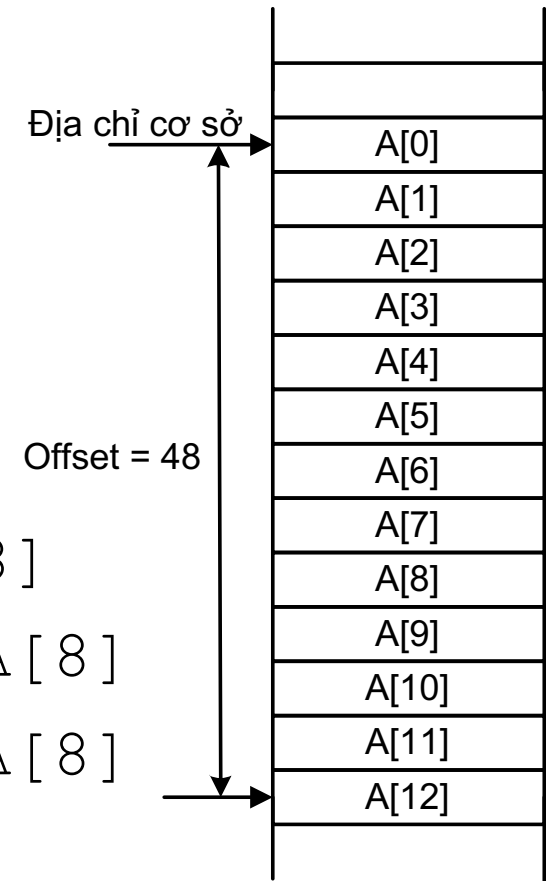
■ Mã C:

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

- h ở \$s2
- \$s3 chứa địa chỉ cơ sở của mảng A

■ Mã hợp ngữ MIPS:

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



Thanh ghi với Bộ nhớ

- Truy nhập thanh ghi nhanh hơn bộ nhớ
- Thao tác dữ liệu trên bộ nhớ yêu cầu nạp (load) và lưu (store)
 - Cần thực hiện nhiều lệnh hơn
- Chương trình dịch sử dụng các thanh ghi cho các biến nhiều nhất có thể
 - Chỉ sử dụng bộ nhớ cho các biến ít được sử dụng
 - Cần tối ưu hóa sử dụng thanh ghi

Toán hạng tức thì (immediate)

- Dữ liệu hằng số được xác định ngay trong lệnh

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

- Không có lệnh trừ (subi) với giá trị hằng số

- Sử dụng hằng số âm trong lệnh `addi` để thực hiện phép trừ

`addi $s2, $s1, -1` # $\$s2 = \$s1 - 1$

Xử lý với số nguyên

- Số nguyên có dấu (biểu diễn bằng bù hai):
 - Với n bit, dải biểu diễn: $[-2^{n-1}, +(2^{n-1}-1)]$
 - Overflow: tràn với số nguyên có dấu
 - Các lệnh **add**, **sub**, **addi**: nếu có overflow xảy ra thì báo lỗi tràn và dừng lại
- Số nguyên không dấu:
 - Với n bit, dải biểu diễn: $[0, 2^n - 1]$
 - Các lệnh **addu**, **subu**, **addiu**: bỏ qua overflow
- Qui ước biểu diễn hằng số nguyên trong hợp ngữ MIPS:
 - số thập phân: 12; 3456; -18
 - số Hexa (bắt đầu bằng **0x**): 0x12 ; 0x3456; 0x1AB6

Hằng số Zero

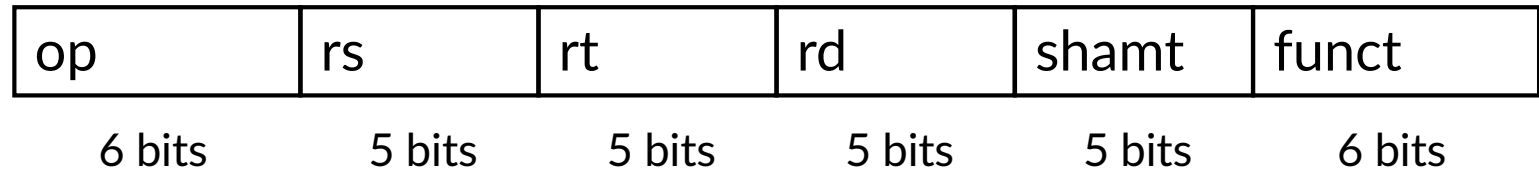
- Thanh ghi 0 của MIPS (\$zero hay \$0) luôn chứa hằng số 0
 - Không thể thay đổi giá trị
- Hữu ích cho một số thao tác thông dụng
 - Chẳng hạn, chuyển dữ liệu giữa các thanh ghi
`add $t2, $s1, $zero # $t2 = $s1`

4.3. Mã máy (Machine code)

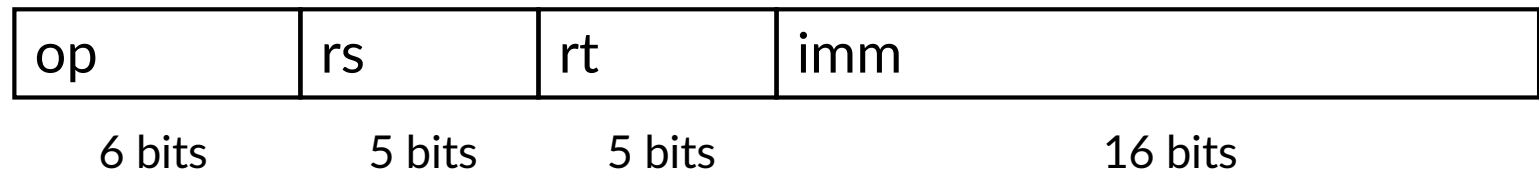
- Các lệnh được mã hóa dưới dạng nhị phân được gọi là mã máy
- Các lệnh của MIPS:
 - Được mã hóa bằng các từ lệnh 32-bit
 - Mỗi lệnh chiếm 4-byte trong bộ nhớ, do vậy địa chỉ của lệnh trong bộ nhớ là bội của 4
 - Có ít dạng lệnh
- Số hiệu thanh ghi được mã hóa bằng 5-bit
 - \$t0 – \$t7 có số hiệu từ 8 – 15
 - \$t8 – \$t9 có số hiệu từ 24 – 25
 - \$s0 – \$s7 có số hiệu từ 16 – 23

Các kiểu lệnh máy của MIPS

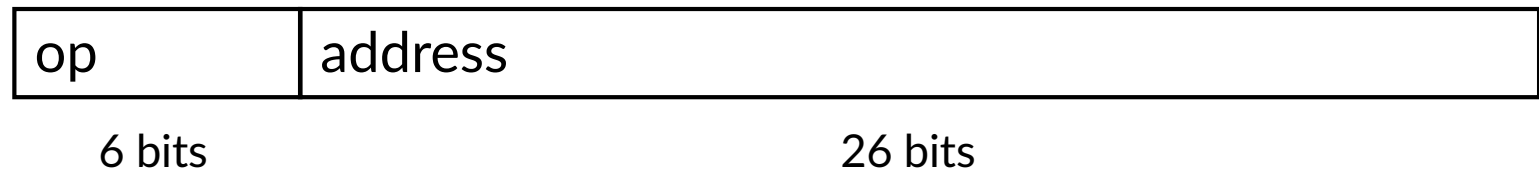
Lệnh kiểu R



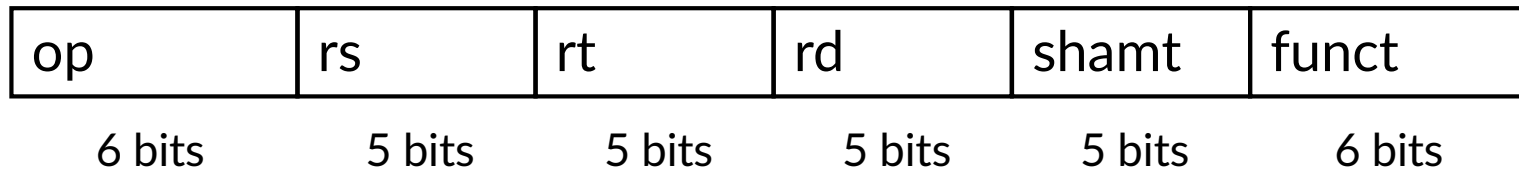
Lệnh kiểu I



Lệnh kiểu J



Lệnh kiểu R (Registers)



- Các trường của lệnh

- op (operation code - opcode): mã thao tác
 - với các lệnh kiểu R, op = 000000
- rs: số hiệu thanh ghi nguồn thứ nhất
- rt: số hiệu thanh ghi nguồn thứ hai
- rd: số hiệu thanh ghi đích
- shamt (shift amount): số bit được dịch, chỉ dùng cho lệnh dịch bit, với các lệnh khác shamt = 00000
- funct (function code): mã hàm → mã hóa cho thao tác cụ thể

Ví dụ mã máy của lệnh add, sub

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

0	\$s1	\$s2	\$t0	0	add
---	------	------	------	---	-----

0	17	18	8	0	32 (0x20)
---	----	----	---	---	-----------

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

(0x02324020)

sub \$s0, \$t3, \$t5

0	\$t3	\$t5	\$s0	0	sub
---	------	------	------	---	-----

0	11	13	16	0	34 (0x22)
---	----	----	----	---	-----------

000000	01011	01101	10000	00000	100010
--------	-------	-------	-------	-------	--------

(0x016D8022)

Lệnh kiểu I (Immediate)



- Dùng cho các lệnh số học/logic với toán hạng tức thì và các lệnh **load/store** (nạp/lưu)
 - rs: số hiệu thanh ghi nguồn (addi) hoặc thanh ghi cơ sở (lw, sw)
 - rt: số hiệu thanh ghi đích (addi, lw) hoặc thanh ghi nguồn (sw)
 - imm (immediate): hằng số nguyên 16-bit

`addi rt, rs, imm` # $(rt) = (rs) + \text{SignExtImm}$

`lw rt, imm(rs)` # $(rt) = \text{mem}[(rs) + \text{SignExtImm}]$

`sw rt, imm(rs)` # $\text{mem}[(rs) + \text{SignExtImm}] = (rt)$

(SignExtImm: hằng số imm 16-bit được mở rộng theo kiểu số có dấu thành 32-bit)

Mở rộng bit cho hằng số theo số có dấu

- Với các lệnh addi, lw, sw cần cộng nội dung thanh ghi với hằng số:
 - Thanh ghi có độ dài 32-bit
 - Hằng số imm 16-bit, cần mở rộng thành 32-bit theo kiểu số có dấu (Sign-extended)
- Ví dụ mở rộng số 16-bit thành 32-bit theo kiểu số có dấu:

+5 =	0000 0000 0000 0101	16-bit
+5 =	0000 0000 0000 0000 0000 0000 0000 0101	32-bit
-12 =	1111 1111 1111 0100	16-bit
-12 =	1111 1111 1111 1111 1111 1111 1111 0100	32-bit

Ví dụ mã máy của lệnh addi

op	rs	rt	imm
----	----	----	-----

6 bits

5 bits

5 bits

16 bits

addi \$s0, \$s1, 5

8	\$s1	\$s0	5
---	------	------	---

8	17	16	5
---	----	----	---

001000	10001	10000	0000 0000 0000 0101
--------	-------	-------	---------------------

(0x22300005)

addi \$t1, \$s2, -12

8	\$s2	\$t1	-12
---	------	------	-----

8	18	9	-12
---	----	---	-----

001000	10010	01001	1111 1111 1111 0100
--------	-------	-------	---------------------

(0x2249FFF4)



Ví dụ mã máy của lệnh load và lệnh store

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

lw \$t0, 32(\$s3)

35	\$s3	\$t0	32
----	------	------	----

35	19	8	32
----	----	---	----

100011	10011	01000	0000 0000 0010 0000
--------	-------	-------	---------------------

(0x8E680020)

sw \$s1, 4(\$t1)

43	\$t1	\$s1	4
----	------	------	---

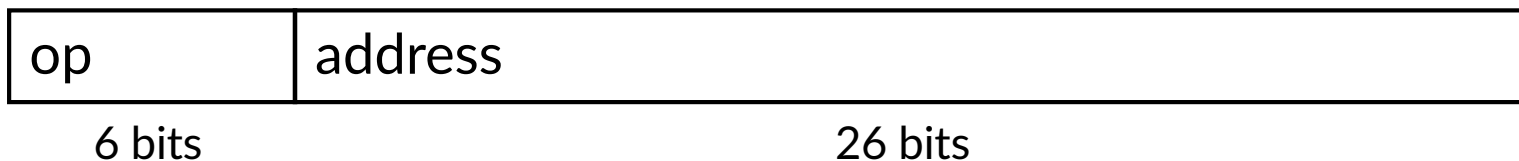
43	9	17	4
----	---	----	---

101011	01001	10001	0000 0000 0000 0100
--------	-------	-------	---------------------

(0xAD310004)

Lệnh kiểu J (Jump)

- Toán hạng 26-bit địa chỉ
- Được sử dụng cho các lệnh nhảy
 - `j (jump)` → `op = 000010`
 - `jal (jump and link)` → `op = 000011`



4.4. Các lệnh logic

- Các lệnh logic để thao tác trên các bit của dữ liệu

Phép toán logic	Toán tử trong C	Lệnh của MIPS
Shift left	<<	sll
Shift right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR		or, ori
Bitwise XOR	^	xor, xori
Bitwise NOT	~	nor

Ví dụ lệnh logic kiểu R

Nội dung các thanh ghi nguồn

\$s1	0100	0110	1010	0001	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

\$s2	1111	1111	1111	1111	0000	0000	0000	0000
------	------	------	------	------	------	------	------	------

Mã hợp ngữ

Kết quả thanh ghi đích

and \$s3, \$s1, \$s2

\$s3								
------	--	--	--	--	--	--	--	--

or \$s4, \$s1, \$s2

\$s4								
------	--	--	--	--	--	--	--	--

xor \$s5, \$s1, \$s2

\$s5								
------	--	--	--	--	--	--	--	--

nor \$s6, \$s1, \$s2

\$s6								
------	--	--	--	--	--	--	--	--

Ví dụ lệnh logic kiểu R

Nội dung các thanh ghi nguồn

\$s1	0100	0110	1010	0001	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

\$s2	1111	1111	1111	1111	0000	0000	0000	0000
------	------	------	------	------	------	------	------	------

Mã hợp ngữ

Kết quả thanh ghi đích

and \$s3, \$s1, \$s2

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
------	------	------	------	------	------	------	------	------

or \$s4, \$s1, \$s2

\$s4	1111	1111	1111	1111	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

xor \$s5, \$s1, \$s2

\$s5	1011	1001	0101	1110	1100	0000	1011	0111
------	------	------	------	------	------	------	------	------

nor \$s6, \$s1, \$s2

\$s6	0000	0000	0000	0000	0011	1111	0100	1000
------	------	------	------	------	------	------	------	------

Ví dụ lệnh logic kiểu I

Giá trị các toán hạng nguồn

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

← Zero-extended →

Mã hợp ngữ

Kết quả thanh ghi đích

andi \$s2,\$s1,0xFA34

\$s2

--	--	--	--	--	--	--	--

ori \$s3,\$s1,0xFA34

\$s3

--	--	--	--	--	--	--	--

xori \$s4,\$s1,0xFA34

\$s4

--	--	--	--	--	--	--	--

Chú ý: Với các lệnh logic kiểu I, hằng số imm 16-bit được mở rộng thành 32-bit theo số không dấu (zero-extended)

Ví dụ lệnh logic kiểu I

Giá trị các toán hạng nguồn

\$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100
<div>← Zero-extended →</div>								

Mã hợp ngữ

Kết quả thanh ghi đích

andi \$s2,\$s1,0xFA34

\$s2	0000	0000	0000	0000	0000	0000	0011	0100
------	------	------	------	------	------	------	------	------

ori \$s3,\$s1,0xFA34

\$s3	0000	0000	0000	0000	1111	1010	1111	1111
------	------	------	------	------	------	------	------	------

xori \$s4,\$s1,0xFA34

\$s4	0000	0000	0000	0000	1111	1010	1100	1011
------	------	------	------	------	------	------	------	------

Ý nghĩa của các phép toán logic

- Phép AND dùng để giữ nguyên một số bit trong word, xóa các bit còn lại về 0
- Phép OR dùng để giữ nguyên một số bit trong word, thiết lập các bit còn lại lên 1
- Phép XOR dùng để giữ nguyên một số bit trong word, đảo giá trị các bit còn lại
- Phép NOT dùng để đảo các bit trong word
 - Đổi 0 thành 1, và đổi 1 thành 0
 - MIPS không có lệnh NOT, nhưng có lệnh NOR với 3 toán hạng
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

```
nor    $t0, $t1, $zero    # $t0 = not($t1)
```

Lệnh logic dịch bit

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *shamt*: chỉ ra dịch bao nhiêu vị trí (shift amount)
- *rs*: không sử dụng, thiết lập = 00000
- Thanh ghi đích *rd* nhận giá trị thanh ghi nguồn *rt* đã được dịch trái hoặc dịch phải, *rt* không thay đổi nội dung
- **s11** - shift left logical (dịch trái logic)
 - Dịch trái các bit và điền các bit 0 vào bên phải
 - Dịch trái i bits là nhân với 2^i (nếu kết quả trong phạm vi biểu diễn 32-bit)
- **sr1** - shift right logical (dịch phải logic)
 - Dịch phải các bit và điền các bit 0 vào bên trái
 - Dịch phải i bits là chia cho 2^i (chỉ với số nguyên không dấu)

Ví dụ lệnh dịch trái sll

Lệnh hợp ngữ:

sll \$t2, \$s0, 4 # \$t2 = \$s0 << 4

Mã máy:

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

(0x00105100)

Ví dụ kết quả thực hiện lệnh:

\$s0

0000	0000	0000	0000	0000	0000	0000	1101
------	------	------	------	------	------	------	------

 = 13

\$t2

0000	0000	0000	0000	0000	0000	1101	0000
------	------	------	------	------	------	------	------

 = 208
(13x16)

Chú ý: Nội dung thanh ghi \$s0 không bị thay đổi

Ví dụ lệnh dịch phải srl

Lệnh hợp ngữ:

srl \$s2, \$s1, 2 # \$s2 = \$s1 >> 2

Mã máy:

op	rs	rt	rd	shamt	funct
0	0	17	18	2	2
000000	00000	10001	10010	00010	000010

(0x00119082)

Ví dụ kết quả thực hiện lệnh:

\$s1

0000	0000	0000	0000	0000	0000	0101	0110
------	------	------	------	------	------	------	------

 = 86

\$s2

0000	0000	0000	0000	0000	0000	0001	0101
------	------	------	------	------	------	------	------

 = 21
[86/4]

Nạp hằng số vào thanh ghi

- Trường hợp hằng số 16-bit → sử dụng lệnh **addi**:
 - Ví dụ: nạp hằng số 0x4F3C vào thanh ghi \$s0:
addi \$s0, \$0, 0x4F3C #\$s0 = 0x4F3C
- Trong trường hợp hằng số 32-bit → sử dụng lệnh **lui** và lệnh **ori**:

lui rt, constant_hi16bit

- Copy 16 bit cao của hằng số 32-bit vào 16 bit trái của rt
- Xóa 16 bits bên phải của rt về 0

ori rt,rt,constant_low16bit

- Đưa 16 bit thấp của hằng số 32-bit vào thanh ghi rt

Lệnh lui (load upper immediate)

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

lui \$s0, 0x21A0

15	0	\$s0	0x21A0
15	0	16	0x21A0

Lệnh mã máy

001111	00000	10000	0010 0001 1010 0000
--------	-------	-------	---------------------

(0x3C1021A0)

Nội dung \$s0 sau khi lệnh được thực hiện:

\$s0	0010	0001	1010	0000	0000	0000	0000
------	------	------	------	------	------	------	------

Ví dụ khởi tạo thanh ghi 32-bit

- Nạp vào thanh ghi \$s0 giá trị 32-bit sau:

0010 0001 1010 0000 0100 0000 0011 1011 = 0x21A0 403B

```
lui $s0, 0x21A0      # nạp 0x21A0 vào nửa cao
                     # của thanh ghi $s0
ori $s0, $s0, 0x403B  # nạp 0x403B vào nửa thấp
                     # của thanh ghi $s0
```

Nội dung \$s0 sau khi thực hiện lệnh **lui**

\$s0	0010	0001	1010	0000	0000	0000	0000
	0000	0000	0000	0000	0100	0000	0011 1011

or

Nội dung \$s0 sau khi thực hiện lệnh **ori**

\$s0	0010	0001	1010	0000	0100	0000	0011 1011
------	------	------	------	------	------	------	-----------

4.5. Tạo các cấu trúc điều khiển

- Các cấu trúc rẽ nhánh

- `if`
- `if/else`
- `switch/case`

- Các cấu trúc lặp

- `while`
- `do while`
- `for`

Các lệnh rẽ nhánh và lệnh nhảy

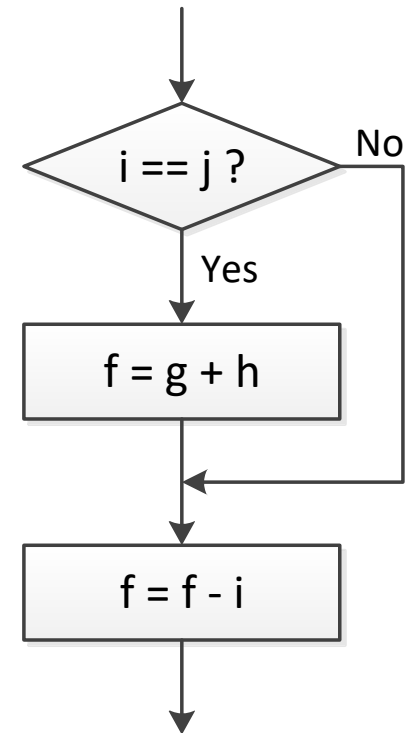
- Các lệnh rẽ nhánh: beq, bne
 - Rẽ nhánh đến lệnh được đánh nhãn nếu điều kiện là đúng, ngược lại, thực hiện tuần tự
 - `beq rs, rt, L1`
 - branch on equal
 - nếu $(rs == rt)$ rẽ nhánh đến lệnh ở nhãn L1
 - `bne rs, rt, L1`
 - branch on not equal
 - nếu $(rs != rt)$ rẽ nhánh đến lệnh ở nhãn L1
- Lệnh nhảy j
 - `j L1`
 - nhảy (jump) không điều kiện đến lệnh ở nhãn L1

Dịch câu lệnh if

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4



Dịch câu lệnh if

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4

- Mã hợp ngữ MIPS:

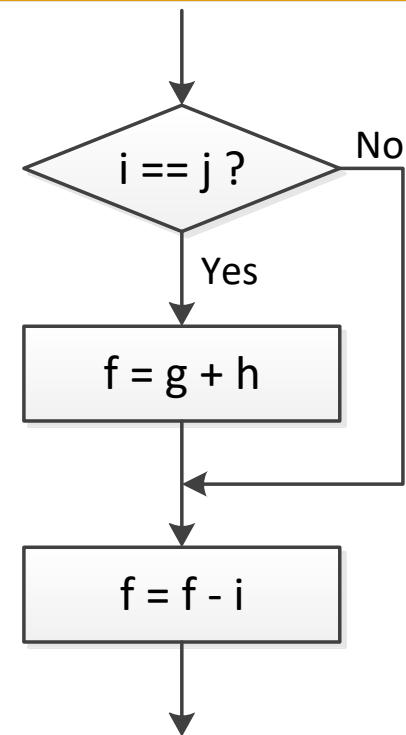
```
# $s0 = f, $s1 = g, $s2 = h
```

```
# $s3 = i, $s4 = j
```

```
bne $s3, $s4, L1    # Nếu i≠j
```

```
add $s0, $s1, $s2    # thì f=g+h
```

```
L1: sub $s0, $s0, $s3 # f=f-i
```



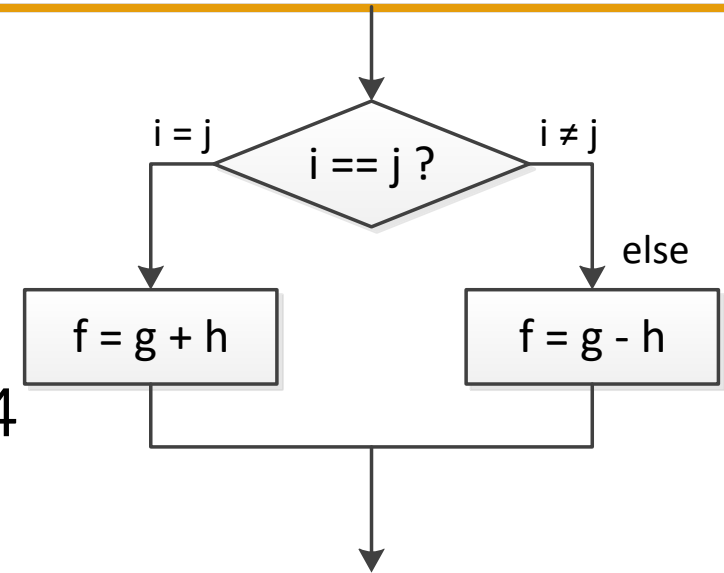
Điều kiện hợp ngữ ngược với điều kiện của ngôn ngữ bậc cao

Dịch câu lệnh if/else

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4

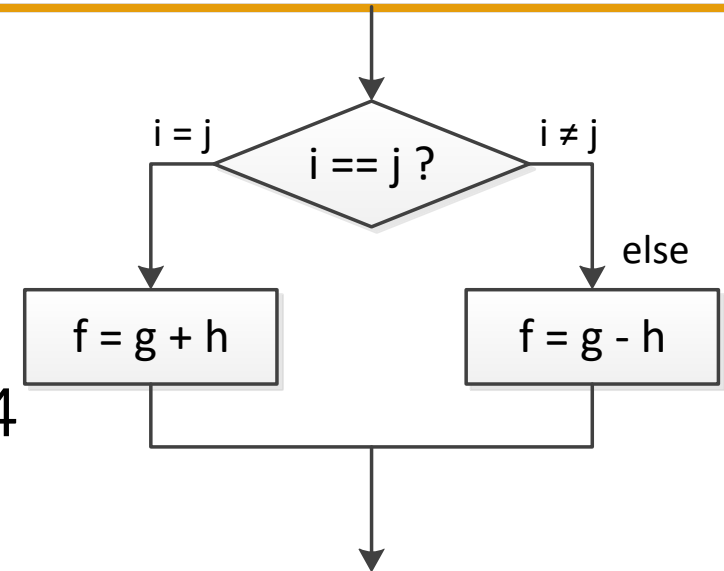


Dịch câu lệnh if/else

- Mã C:

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

- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4



- Mã hợp ngữ MIPS:

```
        bne $s3,$s4,Else      # Nếu i=j  
        add $s0,$s1,$s2      # thì f=g+h  
        j    Exit            # thoát  
Else:    sub $s0,$s1,$s2      # nếu i<>j thì f=g-h  
Exit:    ...
```

Dịch câu lệnh switch/case

Mã C:

```
switch (amount) {  
    case 20:    fee = 2; break;  
    case 50:    fee = 3; break;  
    case 100:   fee = 5; break;  
    default:    fee = 0;  
}
```

// tương đương với sử dụng các câu lệnh if/else

```
if (amount == 20) fee = 2;  
else if (amount == 50) fee = 3;  
else if (amount == 100) fee = 5;  
else fee = 0;
```

Dịch câu lệnh switch/case

Mã hợp ngữ MIPS

\$s0 = amount, \$s1 = fee

case20:

```
addi    $t0, $0, 20      # $t0 = 20
bne     $s0, $t0, case50  # amount == 20? if not, skip to case50
addi    $s1, $0, 2       # if so, fee = 2
j      done              # and break out of case
```

case50:

```
addi    $t0, $0, 50      # $t0 = 50
bne     $s0, $t0, case100 # amount == 50? if not, skip to case100
addi    $s1, $0, 3       # if so, fee = 3
j      done              # and break out of case
```

case100:

```
addi    $t0, $0, 100     # $t0 = 100
bne     $s0, $t0, default # amount == 100? if not, skip to default
addi    $s1, $0, 5       # if so, fee = 5
j      done              # and break out of case
```

default:

```
add     $s1, $0, $0      # fee = 0
```

done:

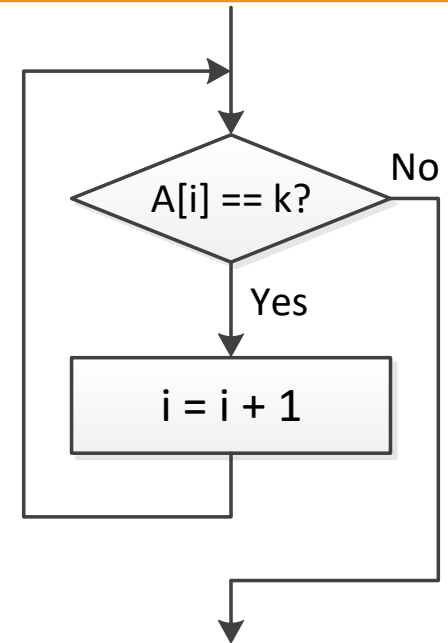


Dịch câu lệnh vòng lặp while

■ Mã C:

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

- i ở \$s3, k ở \$s5
- địa chỉ cơ sở của mảng A ở \$s6

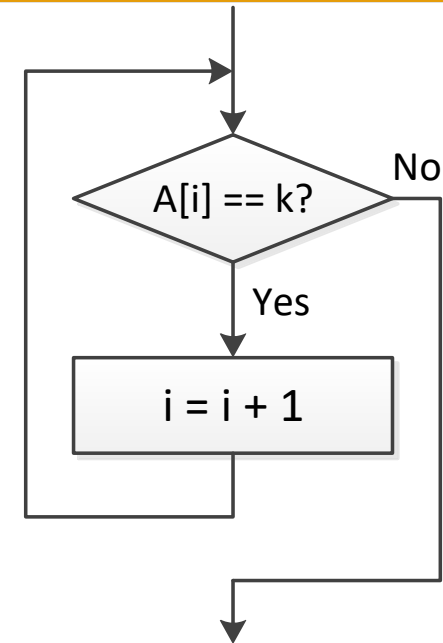


Dịch câu lệnh vòng lặp while

■ Mã C:

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

- i ở \$s3, k ở \$s5
- địa chỉ cơ sở của mảng A ở \$s6



■ Mã hợp ngữ MIPS:

```
Loop: sll    $t1, $s3, 2      # $t1 = 4*i
        add   $t1, $t1, $s6   # $t1 = địa chỉ A[i]
        lw    $t0, 0($t1)     # $t0 = A[i]
        bne   $t0, $s5, Exit  # nếu A[i] <> k thì Exit
        addi  $s3, $s3, 1     # nếu A[i] = k thì i = i + 1
        j     Loop           # quay lại Loop

Exit: ...
```

Dịch câu lệnh vòng lặp for

- Mã C:

```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i++) {
    sum = sum + i;
}
```

Dịch câu lệnh vòng lặp for

- Mã C:

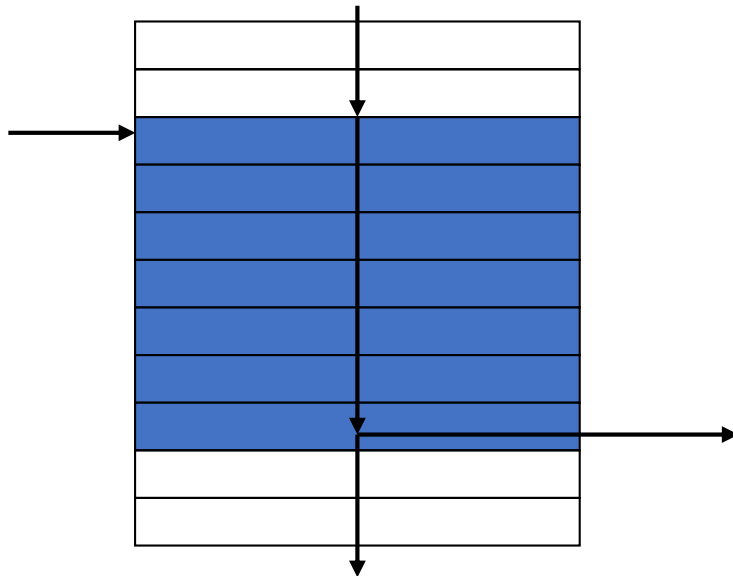
```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i++) {
    sum = sum + i;
}
```

- Mã hợp ngữ MIPS:

```
# $s0 = i, $s1 = sum
    addi    $s1, $0, 0           # sum = 0
    add     $s0, $0, $0          # i = 0
    addi    $t0, $0, 10          # $t0 = 10
for: beq     $s0, $t0, done       # Nếu i=10, thoát
    add     $s1, $s1, $s0        # Nếu i<10 thì sum = sum+i
    addi    $s0, $s0, 1          # tăng i thêm 1
    j       for                 # quay lại for
done: ...
```

Khối lệnh cơ sở (basic block)

- Khối lệnh cơ sở là dãy các lệnh với
 - Không có lệnh rẽ nhánh nhúng trong đó (ngoại trừ ở cuối)
 - Không có đích rẽ nhánh tới (ngoại trừ ở vị trí đầu tiên)



- Chương trình dịch xác định khối cơ sở để tối ưu hóa
- Các bộ xử lý tiên tiến có thể tăng tốc độ thực hiện khối cơ sở

Thêm các lệnh thao tác điều kiện

- Lệnh `slt` (set on less than)

`slt rd, rs, rt`

- Nếu $(rs < rt)$ thì $rd = 1$; ngược lại $rd = 0$;

- Lệnh `slti`

`slti rt, rs, constant`

- Nếu $(rs < \text{constant})$ thì $rt = 1$; ngược lại $rt = 0$;

- Sử dụng kết hợp với các lệnh `beq`, `bne`

`slt $t0, $s1, $s2 # nếu ($\$s1 < \$s2$)`

`bne $t0, $zero, L1 # rẽ nhánh đến L1`

`...`

`L1:`

So sánh số có dấu và không dấu

- So sánh số có dấu: `slt`, `slti`
- So sánh số không dấu: `sltu`, `sltiu`
- Ví dụ
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed`
 - $-1 < +1 \rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned`
 - $4,294,967,295 > 1 \rightarrow \$t0 = 0$

Ví dụ sử dụng lệnh `slt`

- Mã C

```
int sum = 0;
```

```
int i;
```

```
for (i=1; i < 101; i = i*2) {  
    sum = sum + i;  
}
```

Ví dụ sử dụng lệnh slt

■ Mã hợp ngữ MIPS

```
# $s0 = i, $s1 = sum
```

```
    addi $s1, $0, 0           # sum = 0
```

```
    addi $s0, $0, 1           # i = 1
```

```
    addi $t0, $0, 101         # t0 = 101
```

```
loop: slt  $t1, $s0, $t0       # Nếu i >= 101
```

```
    beq  $t1, $0, done         # thì thoát
```

```
    add  $s1, $s1, $s0         # nếu i < 101 thì sum = sum + i
```

```
    sll  $s0, $s0, 1           # i = 2 * i
```

```
    j    loop                 # lặp lại
```

```
done:
```

4.6. Chương trình con - thủ tục

- Các bước yêu cầu:
 1. Đặt các tham số vào các thanh ghi
 2. Chuyển điều khiển đến thủ tục
 3. Thực hiện các thao tác của thủ tục
 4. Đặt kết quả vào thanh ghi cho chương trình đã gọi thủ tục
 5. Trở về vị trí đã gọi

Sử dụng các thanh ghi

- \$a0 – \$a3: các tham số vào (các thanh ghi 4 – 7)
- \$v0, \$v1: các kết quả ra (các thanh ghi 2 và 3)
- \$t0 – \$t9: các giá trị tạm thời
 - Có thể được ghi lại bởi thủ tục được gọi
- \$s0 – \$s7: cất giữ các biến
 - Cần phải cất/khôi phục bởi thủ tục được gọi
- \$gp: global pointer - con trỏ toàn cục cho dữ liệu tĩnh (thanh ghi 28)
- \$sp: stack pointer - con trỏ ngăn xếp (thanh ghi 29)
- \$fp: frame pointer - con trỏ khung (thanh ghi 30)
- \$ra: return address - địa chỉ trở về (thanh ghi 31)

Các lệnh liên quan với thủ tục

- Gọi thủ tục: jump and link

`jal ProcedureAddress`

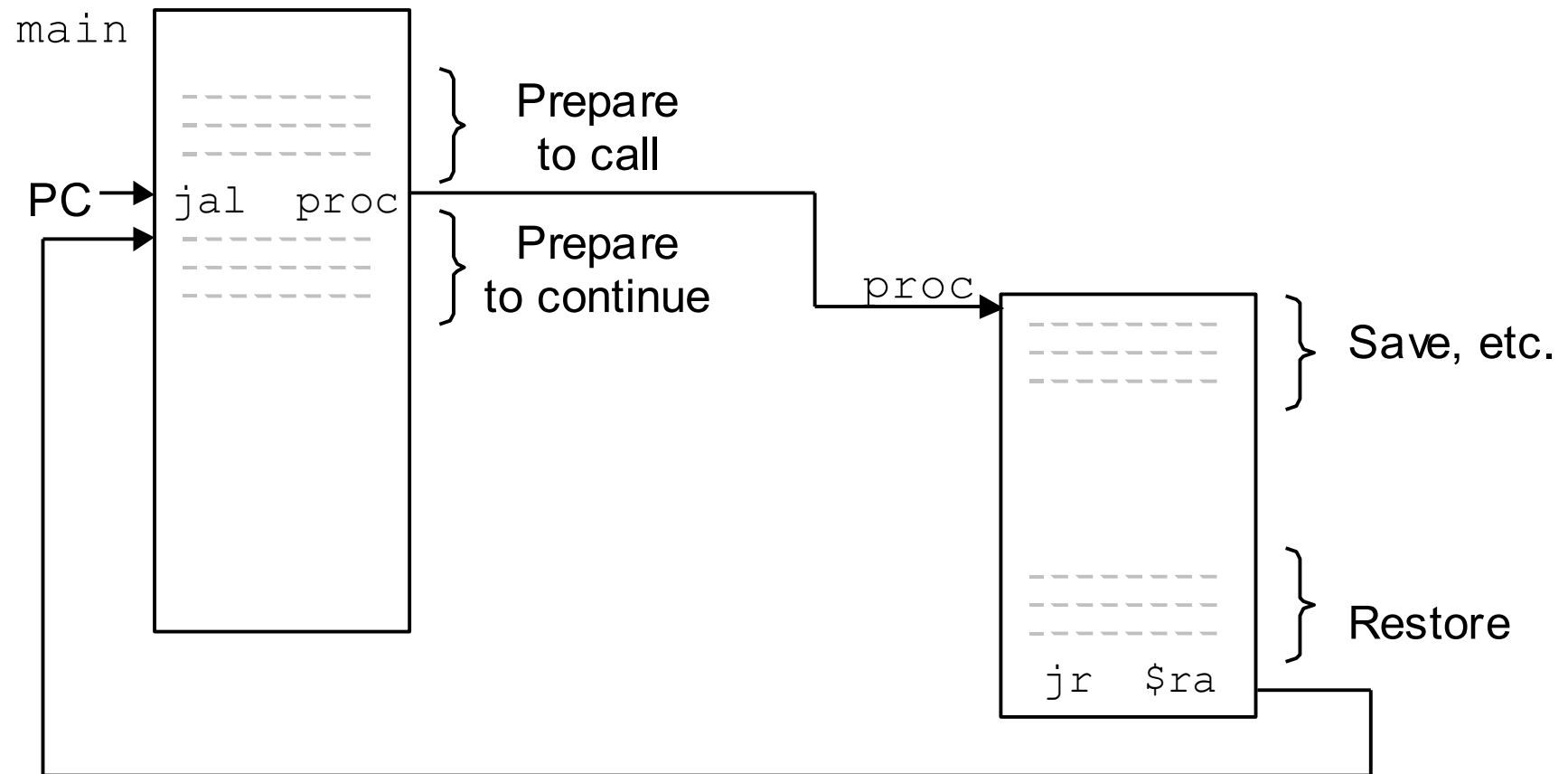
- Địa chỉ của lệnh kế tiếp (địa chỉ trở về) được cất ở thanh ghi \$ra
- Nhảy đến địa chỉ của thủ tục: nạp vào PC địa chỉ của lệnh đầu tiên của Thủ tục được gọi

- Trở về từ thủ tục: jump register

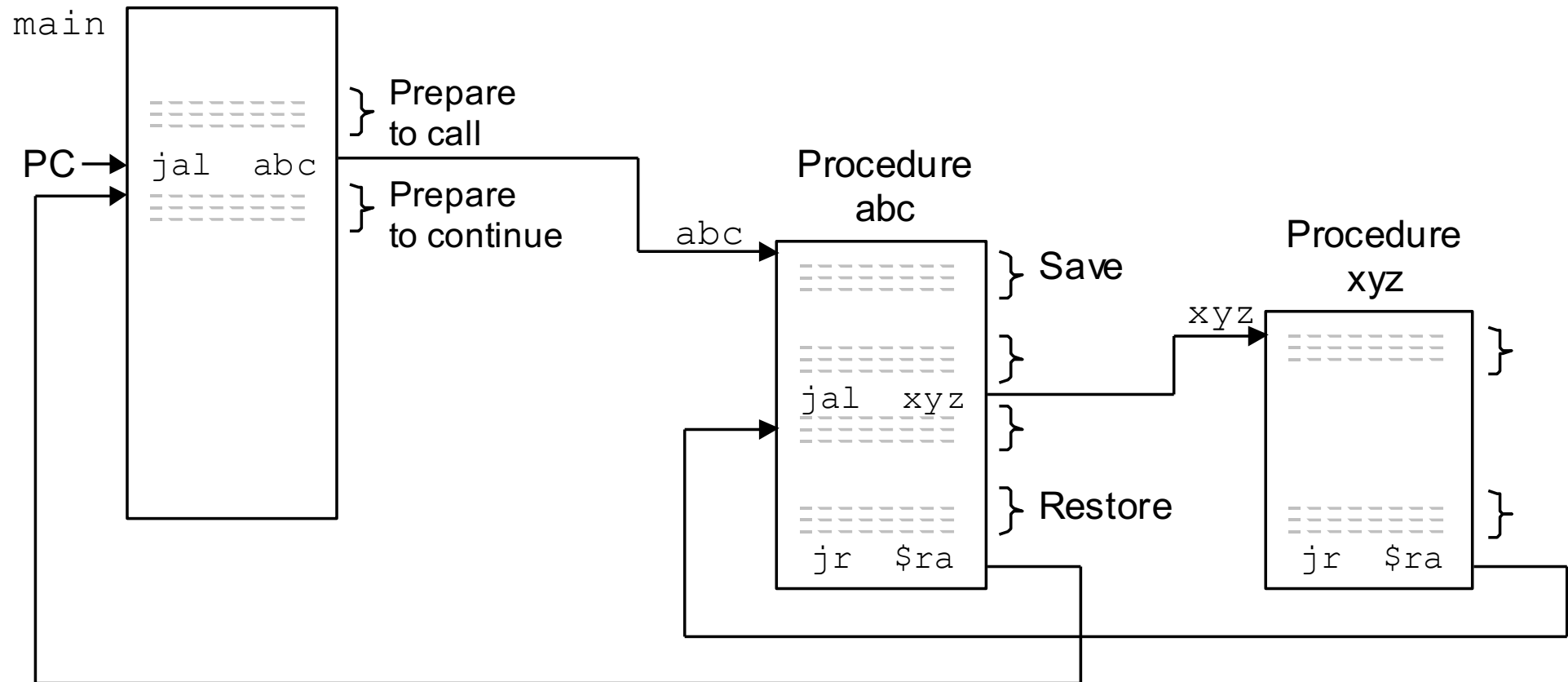
`jr $ra`

- Copy nội dung thanh ghi \$ra (đang chứa địa chỉ trở về) trả lại cho bộ đếm chương trình PC

Minh họa gọi Thủ tục



Gọi thủ tục lồng nhau



Ví dụ Thủ tục lá

- Thủ tục lá là thủ tục không có lời gọi thủ tục khác
- Mã C:

```
int leaf_example (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Các tham số g, h, i, j ở \$a0, \$a1, \$a2, \$a3
- f ở \$s0 (do đó, cần cất \$s0 ra ngăn xếp)
- \$t0 và \$t1 được thủ tục dùng để chứa các giá trị tạm thời, cũng cần cất trước khi sử dụng
- Kết quả ở \$v0

Mã hợp ngữ MIPS

leaf_example:		
addi	\$sp, \$sp, -12	# tạo 3 vị trí ở stack
sw	\$t1, 8(\$sp)	# cất nội dung \$t1
sw	\$t0, 4(\$sp)	# cất nội dung \$t0
sw	\$s0, 0(\$sp)	# cất nội dung \$s0
add	\$t0, \$a0, \$a1	# \$t0 = g+h
add	\$t1, \$a2, \$a3	# \$t1 = i+j
sub	\$s0, \$t0, \$t1	# \$s0 = (g+h)-(i+j)
add	\$v0, \$s0, \$zero	# trả kết quả sang \$v0
lw	\$s0, 0(\$sp)	# khôi phục \$s0
lw	\$t0, 4(\$sp)	# khôi phục \$t0
lw	\$t1, 8(\$sp)	# khôi phục \$t1
addi	\$sp, \$sp, 12	# xóa 3 mục ở stack
jr	\$ra	# trở về nơi đã gọi

Ví dụ Thủ tục đệ quy

- Là thủ tục có gọi thủ tục khác
- Mã C:

```
int fact (int n)
{
    if (n < 1) return (1);
    else return n * fact(n - 1);
}
```

- Tham số n ở \$a0
- Kết quả ở \$v0

Mã hợp ngữ MIPS

fact:		
addi	\$sp, \$sp, -8	# dành stack cho 2 mục
sw	\$ra, 4(\$sp)	# cất địa chỉ trở về
sw	\$a0, 0(\$sp)	# cất tham số n
slti	\$t0, \$a0, 1	# kiểm tra $n < 1$
beq	\$t0, \$zero, L1	
addi	\$v0, \$zero, 1	# nếu đúng, kết quả là 1
addi	\$sp, \$sp, 8	# xóa 2 mục khỏi stack
jr	\$ra	# và trở về
L1:	addi \$a0, \$a0, -1	# nếu không, giảm n
	jal fact	# gọi đệ qui
lw	\$a0, 0(\$sp)	# khôi phục n ban đầu
lw	\$ra, 4(\$sp)	# và địa chỉ trở về
addi	\$sp, \$sp, 8	# xóa 2 mục khỏi stack
mul	\$v0, \$a0, \$v0	# nhân để nhận kết quả
jr	\$ra	# và trở về

4.7. Dữ liệu ký tự

- Các tập ký tự được mã hóa theo byte
 - ASCII: 128 ký tự
 - 95 ký tự hiển thị , 33 mã điều khiển
 - Latin-1: 256 ký tự
 - ASCII và các ký tự mở rộng
- Unicode: Tập ký tự 32-bit
 - Được sử dụng trong Java, C++, ...
 - Hầu hết các ký tự của các ngôn ngữ trên thế giới và các ký hiệu

Các thao tác với byte/halfword

- Có thể sử dụng các phép toán logic
- MIPS có các lệnh để Nạp/Lưu byte/halfword:

`lb rt,imm(rs)` và `lh rt,imm(rs)`

- Nạp 1 byte hoặc 2 byte (halfword) từ bộ nhớ vào bên phải thanh ghi đích `rt`
- Phần còn lại của thanh ghi `rt` được mở rộng theo số có dấu thành 32 bits (Sign-extended)

`lbu rt,imm(rs)` và `lhu rt,imm(rs)`

- Phần còn lại của thanh ghi `rt` được mở rộng theo số không dấu thành 32 bits (Zero-extended)

`sb rt,imm(rs)` và `sh rt,imm(rs)`

- Chỉ lưu byte/halfword bên phải thanh ghi `rt` ra bộ nhớ

Ví dụ copy String

- Mã C:

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i]) != '\0')
    i += 1;
}
```

- Các địa chỉ của x, y ở \$a0, \$a1
- i ở \$s0

Ví dụ Copy String

■ Mã hợp ngữ MIPS

strcpy:

```
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw    $s0, 0($sp)     # save $s0
    add   $s0, $zero, $zero # i = 0
L1:  add   $t1, $s0, $a1    # addr of y[i] in $t1
     lbu   $t2, 0($t1)     # $t2 = y[i]
     add   $t3, $s0, $a0    # addr of x[i] in $t3
     sb    $t2, 0($t3)     # x[i] = y[i]
     beq   $t2, $zero, L2   # exit loop if y[i] == 0
     addi  $s0, $s0, 1      # i = i + 1
     j     L1              # next iteration of loop
L2:  lw    $s0, 0($sp)     # restore saved $s0
     addi  $sp, $sp, 4      # pop 1 item from stack
     jr    $ra             # and return
```

4.8. Các lệnh nhân và chia số nguyên

- MIPS có hai thanh ghi 32-bit: HI (high) và LO (low)
- Các lệnh liên quan:
 - `mult rs, rt` # nhân số nguyên có dấu
 - `multu rs, rt` # nhân số nguyên không dấu
 - Tích 64-bit nằm trong cặp thanh ghi HI/LO
 - `div rs, rt` # chia số nguyên có dấu
 - `divu rs, rt` # chia số nguyên không dấu
 - HI: chứa phần dư, LO: chứa thương
 - `mfhi rd` # Move from Hi to rd
 - `mflo rd` # Move from LO to rd

4.9. Các lệnh với số dấu phẩy động (FP)

- Các thanh ghi số dấu phẩy động
 - 32 thanh ghi 32-bit (single-precision): \$f0, \$f1, ... \$f31
 - Cặp đôi để chứa dữ liệu dạng 64-bit (double-precision): \$f0/\$f1, \$f2/\$f3, ...
- Các lệnh số dấu phẩy động chỉ thực hiện trên các thanh ghi số dấu phẩy động
- Lệnh load và store với FP
 - `lwc1, ldc1, swc1, sdc1`
 - Ví dụ: `ldc1 $f8, 32($s2)`

Các lệnh với số dấu phẩy động

- Các lệnh số học với số FP 32-bit (single-precision)
 - `add.s, sub.s, mul.s, div.s`
 - VD: `add.s $f0, $f1, $f6`
- Các lệnh số học với số FP 64-bit (double-precision)
 - `add.d, sub.d, mul.d, div.d`
 - VD: `mul.d $f4, $f4, $f6`
- Các lệnh so sánh
 - `c.xx.s, c.xx.d` (trong đó xx là eq, lt, le, ...)
 - Thiết lập hoặc xóa các bit mã điều kiện
 - VD: `c.lt.s $f3, $f4`
- Các lệnh rẽ nhánh dựa trên mã điều kiện
 - `bc1t, bc1f`
 - VD: `bc1t TargetLabel`

4.10 Lập trình với mảng dữ liệu

- Truy cập số lượng lớn các dữ liệu cùng loại
- Chỉ số (Index): truy cập từng phần tử của mảng
- Kích cỡ (Size): số phần tử của mảng

Ví dụ về mảng

- Mảng 5-phần tử, mỗi phần tử có độ dài 32-bit, chiếm 4 byte trong bộ nhớ
- Địa chỉ cơ sở = 0x12348000 (địa chỉ của phần tử đầu tiên của mảng array[0])
- Bước đầu tiên để truy cập mảng: nạp địa chỉ cơ sở vào thanh ghi

0x12348000	array[0]
0x12348004	array[1]
0x12348008	array[2]
0x1234800C	array[3]
0x12348010	array[4]

Ví dụ truy cập các phần tử

■ Mã C

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

■ Mã hợp ngữ MIPS

nạp địa chỉ cơ sở của mảng vào \$s0

```
lui    $s0, 0x1234          # 0x1234 vào nửa cao của $s0  
ori    $s0, $s0, 0x8000     # 0x8000 vào nửa thấp của $s0
```

```
lw     $t1, 0($s0)          # $t1 = array[0]  
sll    $t1, $t1, 1          # $t1 = $t1 * 2  
sw     $t1, 0($s0)          # array[0] = $t1
```

```
lw     $t1, 4($s0)          # $t1 = array[1]  
sll    $t1, $t1, 1          # $t1 = $t1 * 2  
sw     $t1, 4($s0)          # array[1] = $t1
```

Ví dụ vòng lặp truy cập mảng dữ liệu

- Mã C

```
int array[1000];  
int i;
```

```
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;
```

// giả sử địa chỉ cơ sở của mảng = 0x23b8f000

- Mã hợp ngữ MIPS

```
# $s0 = array base address (0x23b8f000), $s1 = i
```


Ví dụ vòng lặp truy cập mảng dữ liệu (tiếp)

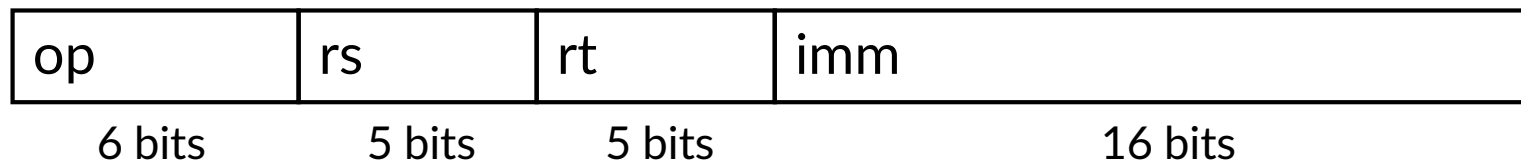
Mã hợp ngữ MIPS

```
# $s0 = array base address (0x23b8f000), $s1 = i
# khởi tạo các thanh ghi
    lui    $s0, 0x23b8                # $s0 = 0x23b80000
    ori    $s0, $s0, 0xf000          # $s0 = 0x23b8f000
    addi   $s1, $0, 0                 # i = 0
    addi   $t2, $0, 1000              # $t2 = 1000
# vòng lặp
loop: slt   $t0, $s1, $t2             # i < 1000?
    beq    $t0, $0, done              # if not then done
    sll    $t0, $s1, 2                 # $t0 = i*4
    add    $t0, $t0, $s0               # address of array[i]
    lw     $t1, 0($t0)                 # $t1 = array[i]
    sll    $t1, $t1, 3                 # $t1 = array[i]*8
    sw     $t1, 0($t0)                 # array[i] = array[i]*8
    addi   $s1, $s1, 1                 # i = i + 1
    j      loop                       # repeat
done:
```



4.11. Các phương pháp định địa chỉ của MIPS

- Các lệnh **Branch** chỉ ra:
 - Mã thao tác, hai thanh ghi, hằng số
- Hầu hết các đích rẽ nhánh là rẽ nhánh gần
 - Rẽ xuôi hoặc rẽ ngược



- Định địa chỉ tương đối với PC
 - PC-relative addressing
 - **Địa chỉ đích = PC + hằng số imm × 4**
 - Chú ý: trước đó PC đã được tăng thêm 4 (trở tới lệnh kế tiếp)
 - Hằng số imm 16-bit có giá trị trong dải $[-2^{15}, +2^{15} - 1]$

Lệnh beq, bne

op	rs	rt	imm
----	----	----	-----

6 bits

5 bits

5 bits

16 bits

beq \$s0, \$s1, Exit

bne \$s0, \$s1, Exit

4 or 5	16	17	Exit
--------	----	----	------

khoảng cách tương đối tính theo word

Lệnh mã máy

beq

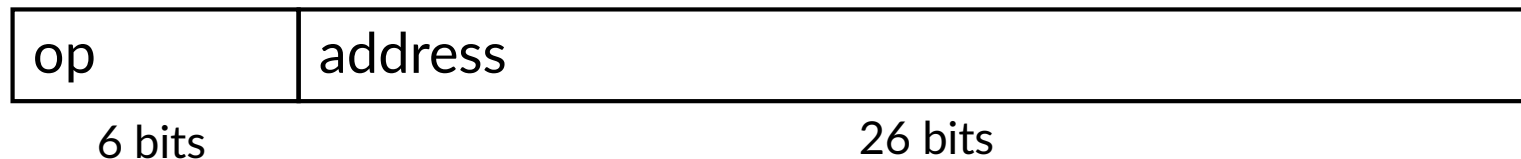
000100	10000	10001	0000 0000 0000 0110
--------	-------	-------	---------------------

bne

000101	10000	10001	0000 0000 0000 0110
--------	-------	-------	---------------------

Địa chỉ hóa cho lệnh Jump

- Đích của lệnh **Jump (j và jal)** có thể là bất kỳ chỗ nào trong chương trình
 - Cần mã hóa đầy đủ địa chỉ trong lệnh

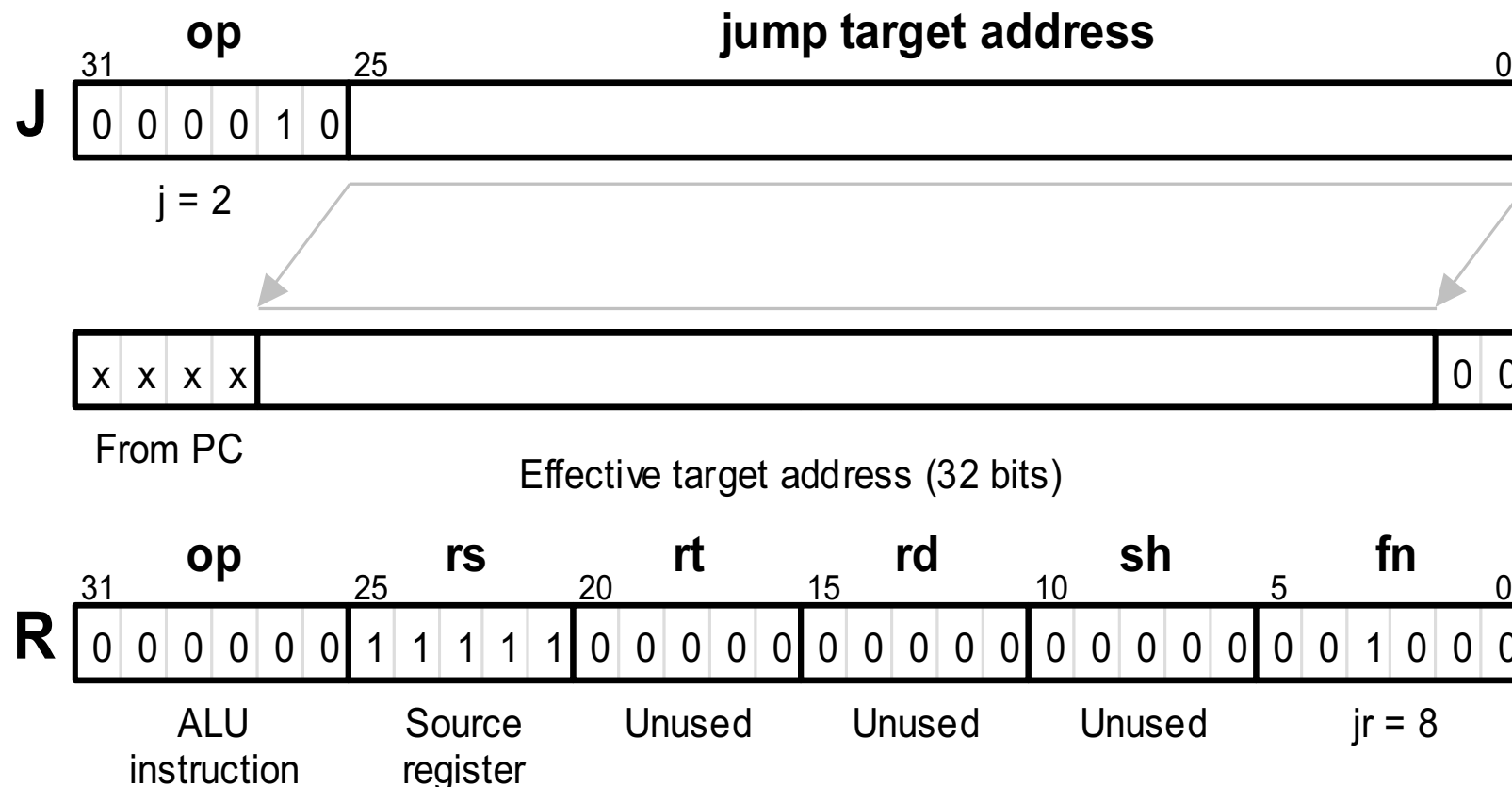


- Định địa chỉ nhảy (giả) trực tiếp (Pseudo)Direct jump addressing
 - Địa chỉ đích = $PC_{31...28} : (\text{address} \times 4)$

Ví dụ mã lệnh j và jr

j L1 # nhảy đến vị trí có nhãn L1

jr \$ra # nhảy đến vị trí có địa chỉ ở \$ra;
\$ra chứa địa chỉ trở về



Ví dụ mã hóa lệnh

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: ...
```

0x80000	0	0	19	9	2	0
0x80004	0	9	22	9	0	32
0x80008	35	9	8	0		
0x8000C	5	8	21	2		
0x80010	8	19	19	1		
0x80014	2	0x20000				
0x80018						

Rẽ nhánh xa

- Nếu đích rẽ nhánh là quá xa để mã hóa với offset 16-bit, assembler sẽ viết lại code
- Ví dụ

```
beq $s0, $s1, L1  
(lệnh kế tiếp)
```

...

L1:

sẽ được thay bằng đoạn lệnh sau:

```
bne $s0, $s1, L2  
j L1
```

L2: (lệnh kế tiếp)

...

L1:

Tóm tắt về các phương pháp định địa chỉ

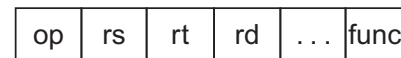
1. Định địa chỉ tức thì

1. Immediate addressing



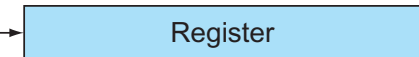
2. Định địa chỉ thanh ghi

2. Register addressing



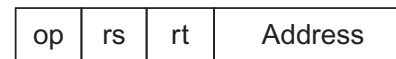
Registers

Register

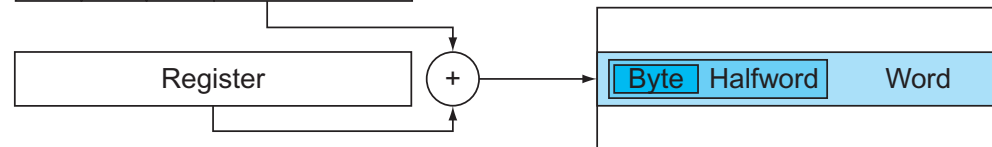


3. Định địa chỉ cơ sở

3. Base addressing

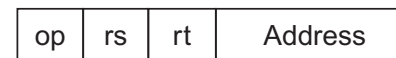


Memory

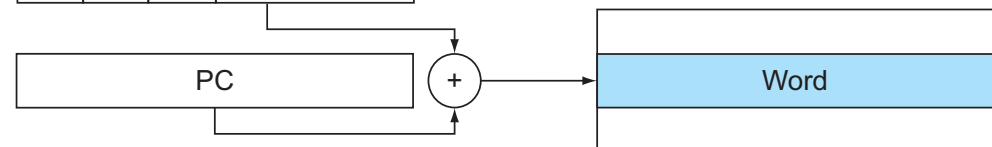


4. Định địa chỉ tương đối với PC

4. PC-relative addressing

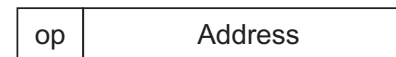


Memory

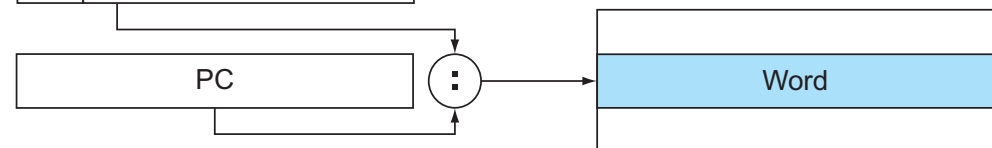


5. Định địa chỉ giả trực tiếp

5. Pseudodirect addressing



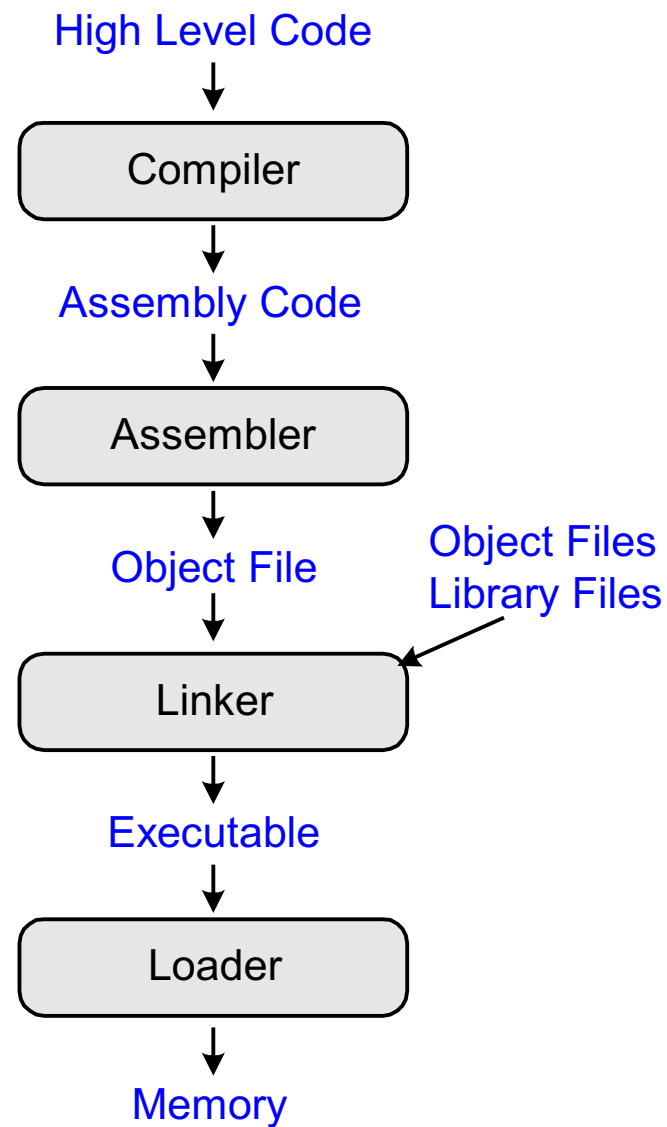
Memory



4.12. Dịch và chạy chương trình hợp ngữ

- Các phần mềm lập trình hợp ngữ MIPS:
 - MARS
 - MipsIt
 - QtSpim
- MIPS Reference Data

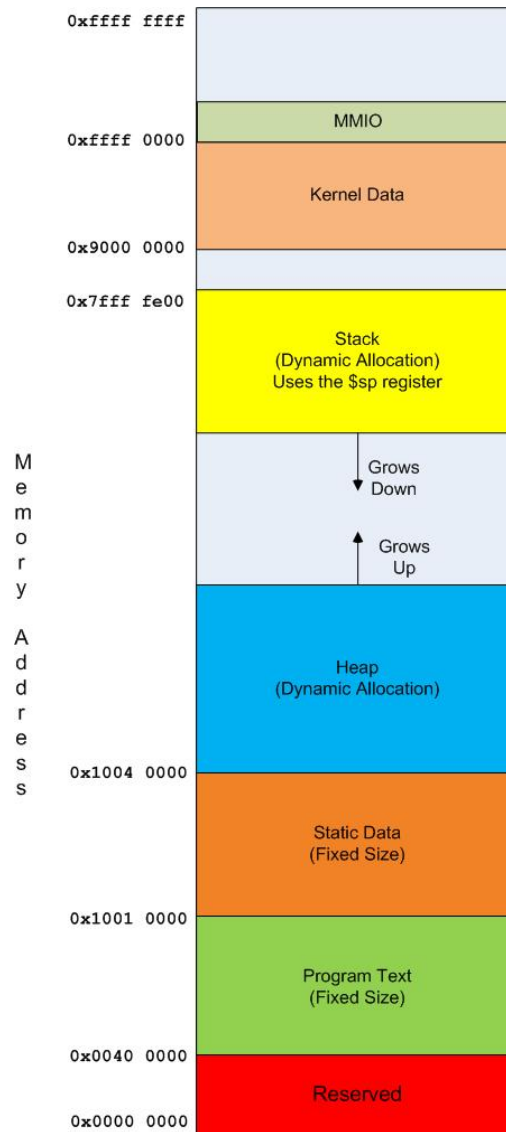
Dịch và chạy ứng dụng



Chương trình trong bộ nhớ

- Các lệnh (instructions)
- Dữ liệu
 - Toàn cục/tĩnh: được cấp phát trước khi chương trình bắt đầu thực hiện
 - Động: được cấp phát trong khi chương trình thực hiện
- Bộ nhớ:
 - 2^{32} bytes = 4 GiB
 - Địa chỉ từ 0x00000000 đến 0xFFFFFFFF

Bản đồ bộ nhớ của MIPS



Ví dụ: Mã C

```
int f, g, y; // global variables
```

```
int main(void)
```

```
{
```

```
    f = 2;
```

```
    g = 3;
```

```
    y = sum(f, g);
```

```
    return y;
```

```
}
```

```
int sum(int a, int b) {
```

```
    return (a + b);
```

```
}
```

Ví dụ chương trình hợp ngữ MIPS

```
.data
f: .word 0
g: .word 0
y: .word 0
.text
main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)    # store $ra
    addi $a0, $0, 2      # $a0 = 2
    sw   $a0, f          # f = 2
    addi $a1, $0, 3      # $a1 = 3
    sw   $a1, g          # g = 3
    jal  sum             # call sum
    sw   $v0, y          # y = sum()
    lw   $ra, 0($sp)    # restore $ra
    addi $sp, $sp, 4     # restore $sp
    li   $v0, 10
    syscall
sum:
    add  $v0, $a0, $a1   # $v0 = a + b
    jr   $ra             # return
```

Viết chương trình trên phần mềm MARS

The screenshot displays the MARS MIPS simulator interface. The main window shows assembly code for a program named `fullprogram.asm`. The code includes data declarations, a `main` function, and a `sum` function. The registers panel on the right shows the current state of the MIPS registers.

Assembly Code:

```
1 .data
2 f: .word 0
3 g: .word 0
4 y: .word 0
5 .text
6 main:
7     addi $sp, $sp, -4    # stack frame
8     sw   $ra, 0($sp)    # store $ra
9     addi $a0, $0, 2      # $a0 = 2
10    sw   $a0, f          # f = 2
11    addi $a1, $0, 3      # $a1 = 3
12    sw   $a1, g          # g = 3
13    jal  sum             # call sum
14    sw   $v0, y          # y = sum()
15    lw   $ra, 0($sp)    # restore $ra
16    addi $sp, $sp, 4     # restore $sp
17    li   $v0, 10        # syscall
18    syscall
19 sum:
20    add  $v0, $a0, $a1   # $v0 = a + b
```

Registers:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffcfc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Thực thi chương trình trên MARS

Mars

/Volumes/DATA/Kientrucmaytinh/MARS/fullprogram.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

Text Segment

Program Arguments:

Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x23bdfcfc	addi \$29,\$29,0xffff...	7: addi \$sp, \$sp, -4 # stack frame
<input type="checkbox"/>	0x00400004	0xafbf0000	sw \$31,0x00000000(\$29)	8: sw \$ra, 0(\$sp) # store \$ra
<input type="checkbox"/>	0x00400008	0x20040002	addi \$4,\$0,0x00000002	9: addi \$a0, \$0, 2 # \$a0 = 2
<input type="checkbox"/>	0x0040000c	0x3c011001	lui \$1,0x0001001	10: sw \$a0, f # f = 2
<input type="checkbox"/>	0x00400010	0xac240000	sw \$4,0x00000000(\$1)	
<input type="checkbox"/>	0x00400014	0x20050003	addi \$5,\$0,0x00000003	11: addi \$a1, \$0, 3 # \$a1 = 3
<input type="checkbox"/>	0x00400018	0x3c011001	lui \$1,0x0001001	12: sw \$a1, g # g = 3
<input type="checkbox"/>	0x0040001c	0xac250004	sw \$5,0x00000004(\$1)	
<input type="checkbox"/>	0x00400020	0xc10000f	jal 0x0040003c	13: jal sum # call sum
<input type="checkbox"/>	0x00400024	0x3c011001	lui \$1,0x0001001	14: sw \$v0, y # y = sum()
<input type="checkbox"/>	0x00400028	0xac220008	sw \$2,0x00000008(\$1)	
<input type="checkbox"/>	0x0040002c	0x8fbf0000	lw \$31,0x00000000(\$29)	15: lw \$ra, 0(\$sp) # restore \$ra

Labels

Label	Address
main	0x00400000
sum	0x0040003c
f	0x10010000
g	0x10010004
y	0x10010008

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffcfc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Mars Messages Run I/O

Clear

Vùng nhớ lệnh

Text Segment				
Program Arguments: <input type="text"/>				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x23bdfffc	addi \$29,\$29,0xffffffffc	7: addi \$sp, \$sp, -4 # stack frame
<input type="checkbox"/>	0x00400004	0xafbf0000	sw \$31,0x00000000(\$29)	8: sw \$ra, 0(\$sp) # store \$ra
<input type="checkbox"/>	0x00400008	0x20040002	addi \$4,\$0,0x00000002	9: addi \$a0, \$0, 2 # \$a0 = 2
<input type="checkbox"/>	0x0040000c	0x3c011001	lui \$1,0x00001001	10: sw \$a0, f # f = 2
<input type="checkbox"/>	0x00400010	0xac240000	sw \$4,0x00000000(\$1)	
<input type="checkbox"/>	0x00400014	0x20050003	addi \$5,\$0,0x00000003	11: addi \$a1, \$0, 3 # \$a1 = 3
<input type="checkbox"/>	0x00400018	0x3c011001	lui \$1,0x00001001	12: sw \$a1, g # g = 3
<input type="checkbox"/>	0x0040001c	0xac250004	sw \$5,0x00000004(\$1)	
<input type="checkbox"/>	0x00400020	0x0c10000f	jal 0x0040003c	13: jal sum # call sum
<input type="checkbox"/>	0x00400024	0x3c011001	lui \$1,0x00001001	14: sw \$v0, y # y = sum()
<input type="checkbox"/>	0x00400028	0xac220008	sw \$2,0x00000008(\$1)	
<input type="checkbox"/>	0x0040002c	0x8fbf0000	lw \$31,0x00000000(\$29)	15: lw \$ra, 0(\$sp) # restore \$ra
<input type="checkbox"/>	0x00400030	0x23bd0004	addi \$29,\$29,0x00000004	16: addi \$sp, \$sp, 4 # restore \$sp
<input type="checkbox"/>	0x00400034	0x2402000a	addiu \$2,\$0,0x0000000a	17: li \$v0, 10
<input type="checkbox"/>	0x00400038	0x0000000c	syscall	18: syscall
<input type="checkbox"/>	0x0040003c	0x00851020	add \$2,\$4,\$5	20: add \$v0, \$a0, \$a1 # \$v0 = a + b
<input type="checkbox"/>	0x00400040	0x03e00008	jr \$31	21: jr \$ra # return

Tập thanh ghi

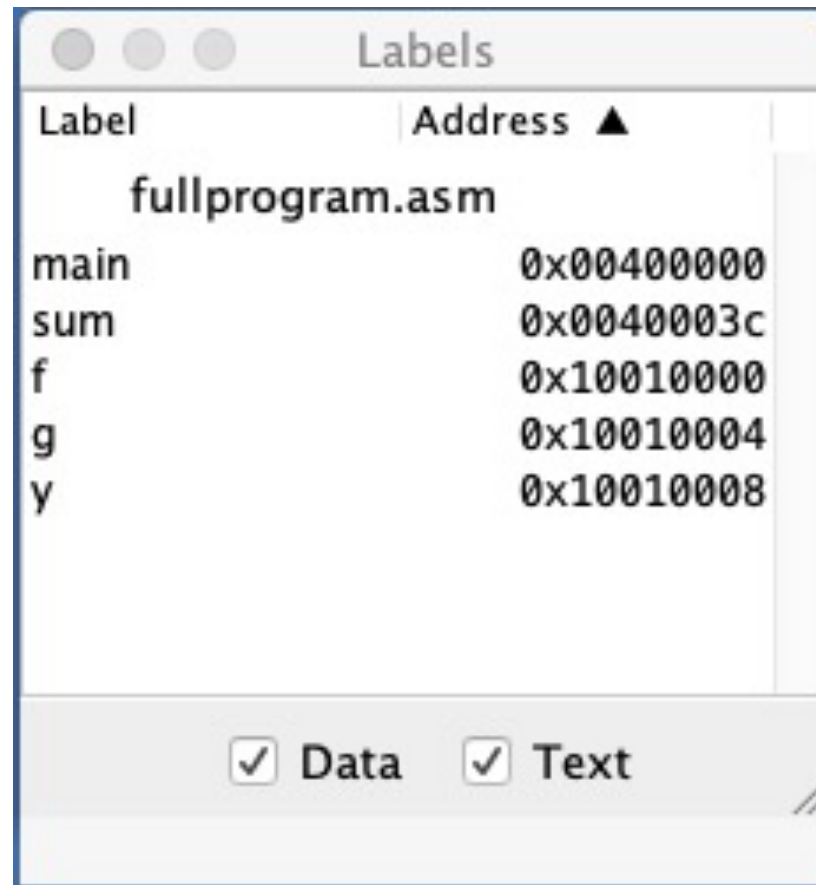
Registers			Coproc 1	Coproc 0
Name	Number	Value		
\$zero	0	0x00000000		
\$at	1	0x00000000		
\$v0	2	0x00000000		
\$v1	3	0x00000000		
\$a0	4	0x00000000		
\$a1	5	0x00000000		
\$a2	6	0x00000000		
\$a3	7	0x00000000		
\$t0	8	0x00000000		
\$t1	9	0x00000000		
\$t2	10	0x00000000		
\$t3	11	0x00000000		
\$t4	12	0x00000000		
\$t5	13	0x00000000		
\$t6	14	0x00000000		
\$t7	15	0x00000000		
\$s0	16	0x00000000		
\$s1	17	0x00000000		
\$s2	18	0x00000000		
\$s3	19	0x00000000		
\$s4	20	0x00000000		
\$s5	21	0x00000000		
\$s6	22	0x00000000		
\$s7	23	0x00000000		
\$t8	24	0x00000000		
\$t9	25	0x00000000		
\$k0	26	0x00000000		
\$k1	27	0x00000000		
\$gp	28	0x10008000		
\$sp	29	0x7ffefffc		
\$fp	30	0x00000000		
\$ra	31	0x00000000		
pc		0x00400000		
hi		0x00000000		
lo		0x00000000		

Vùng nhớ dữ liệu

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

← → 0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Nhãn và biến



The screenshot shows a window titled 'Labels' with a table of labels and their addresses. The table has two columns: 'Label' and 'Address'. The file name 'fullprogram.asm' is displayed above the table. The labels listed are 'main', 'sum', 'f', 'g', and 'y', with their corresponding addresses in hexadecimal. At the bottom of the window, there are two checked checkboxes labeled 'Data' and 'Text'.

Label	Address ▲
fullprogram.asm	
main	0x00400000
sum	0x0040003c
f	0x10010000
g	0x10010004
y	0x10010008

☒ Data ☒ Text

Ví dụ lệnh giả (Pseudoinstruction)

Pseudoinstruction	MIPS Instructions
<code>li \$s0, 0x1234AA77</code>	<code>lui \$at, 0x1234</code> <code>ori \$s0, \$at, 0xAA77</code>
<code>mul \$s0, \$s1, \$s2</code>	<code>mult \$s1, \$s2</code> <code>mflo \$s0</code>
<code>not \$t1, \$t2</code>	<code>nor \$t1, \$t2, \$0</code>
<code>move \$s1, \$s2</code>	<code>addu \$s1, \$0, \$s2</code>
<code>nop</code>	<code>sll \$0, \$0, 0</code>

Hết chương 4