



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
Hanoi University of Science and Technology

KIẾN TRÚC MÁY TÍNH

Computer Architecture

Nguyễn Kim Khánh

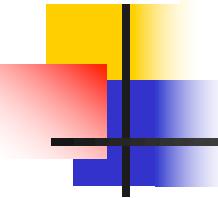
Bộ môn Kỹ thuật máy tính

Viện Công nghệ thông tin và Truyền thông

Department of Computer Engineering (DCE)

School of Information and Communication Technology (SoICT)

Version: IT3282 CA-HEDSPI2020



Contact Information

- Address: 502-B1, HUST Campus
- Mobile: 091-358-5533
- e-mail: khanhnk@soict.hust.edu.vn
khanh.nguyenkim@hust.edu.vn

Mục tiêu

- Hai học phần liên thông:
 - Kiến trúc máy tính (Computer Architecture)
 - Hệ thống máy tính (Computer Systems)
- Sinh viên được trang bị các kiến thức về kiến trúc tập lệnh và tổ chức của máy tính
- Sau khi học xong cả hai học phần, sinh viên có khả năng:
 - Tìm hiểu kiến trúc tập lệnh của các bộ xử lý cụ thể
 - Lập trình hợp ngữ
 - Đánh giá hiệu năng máy tính
 - Khai thác và quản trị hiệu quả các hệ thống máy tính
 - Phân tích và thiết kế hệ thống máy tính

Mục tiêu của từng học phần

- **Kiến trúc máy tính**

- Kiến trúc tập lệnh
- Chương trình nguồn được dịch ra thành mã máy như thế nào ?
- Phần cứng thực hiện chương trình mã máy như thế nào ?

- **Hệ thống máy tính**

- Đánh giá hiệu năng hệ thống máy tính
- Tổ chức các thành phần của hệ thống máy tính
- Các kiến trúc máy tính song song

Tài liệu học tập

- **Bài giảng Kiến trúc máy tính: CA-HEDSPI2020**
- **Sách giáo trình:**
 - [1] David A. Patterson, John L. Hennessy
Computer Organization and Design – 2012, Revised 4th edition
 - [2] William Stallings
Computer Organization and Architecture – 2013, 9th edition
 - [3] David Money Harris, Sarah L. Harris
Digital Design and Computer Architecture – 2013, 2nd edition
 - [4] Andrew S. Tanenbaum
Structured Computer Organization – 2013, 6th edition
- **Sách tham khảo:**
 - [2] William Stallings
Computer Organization and Architecture – 2013, 9th edition
 - [3] David Money Harris, Sarah L. Harris
Digital Design and Computer Architecture – 2013, 2nd edition
 - [4] Andrew S. Tanenbaum
Structured Computer Organization – 2013, 6th edition

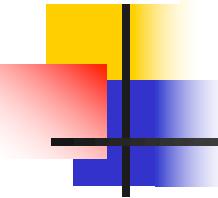
Nội dung học phần

Chương 1. Giới thiệu chung

Chương 2. Kiến trúc tập lệnh

Chương 3. Số học máy tính

Chương 4. Bộ xử lý

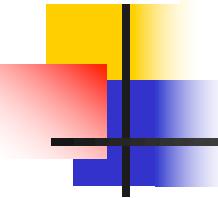


Kiến trúc máy tính

Chương 1

GIỚI THIỆU CHUNG

Nguyễn Kim Khánh
Trường Đại học Bách khoa Hà Nội



Nội dung

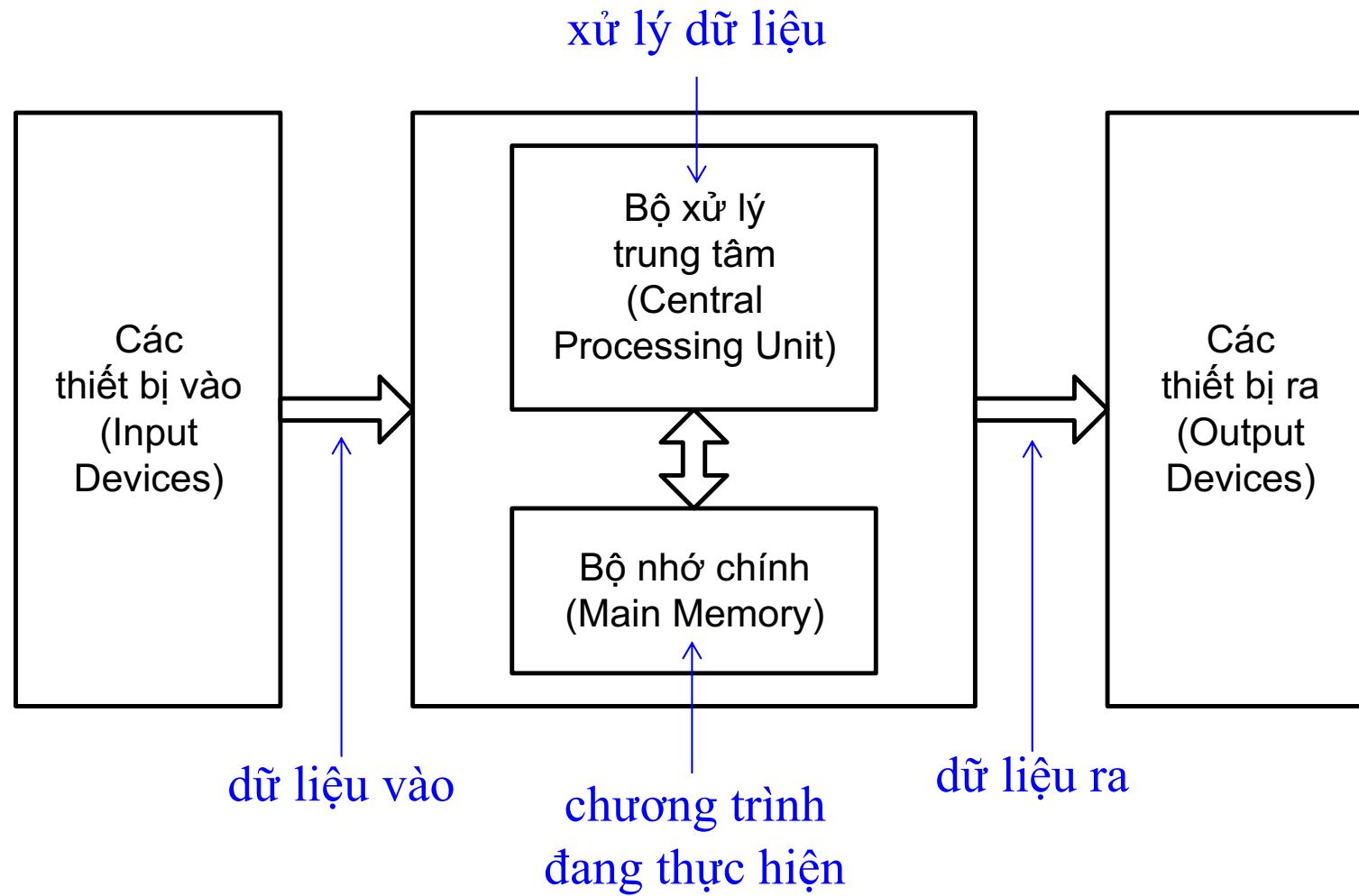
- 1.1. Máy tính và phân loại máy tính
- 1.2. Khái niệm kiến trúc máy tính
- 1.3. Sự tiến hóa của công nghệ máy tính

1.1. Máy tính và phân loại máy tính

- **Máy tính (Computer)** là thiết bị điện tử thực hiện các công việc sau:
 - Nhận dữ liệu vào,
 - Xử lý dữ liệu theo dãy các lệnh được nhớ sẵn bên trong,
 - Đưa dữ liệu (thông tin) ra.
- Dãy các lệnh nằm trong bộ nhớ để yêu cầu máy tính thực hiện công việc cụ thể gọi là **chương trình (program)**.

→ Máy tính hoạt động theo chương trình

Mô hình đơn giản của máy tính



Phân loại máy tính kỹ nguyên PC

■ Máy tính cá nhân (Personal Computers)

- Desktop computers, Laptop computers
- Máy tính đa dụng

■ Máy chủ (Servers) – máy phục vụ

- Dùng trong mạng để quản lý và cung cấp các dịch vụ
- Hiệu năng và độ tin cậy cao
- Hàng nghìn đến hàng triệu USD

■ Siêu máy tính (Supercomputers)

- Dùng cho tính toán cao cấp trong khoa học và kỹ thuật
- Hàng triệu đến hàng trăm triệu USD

■ Máy tính nhúng (Embedded Computers)

- Đặt ẩn trong thiết bị khác
- Được thiết kế chuyên dụng

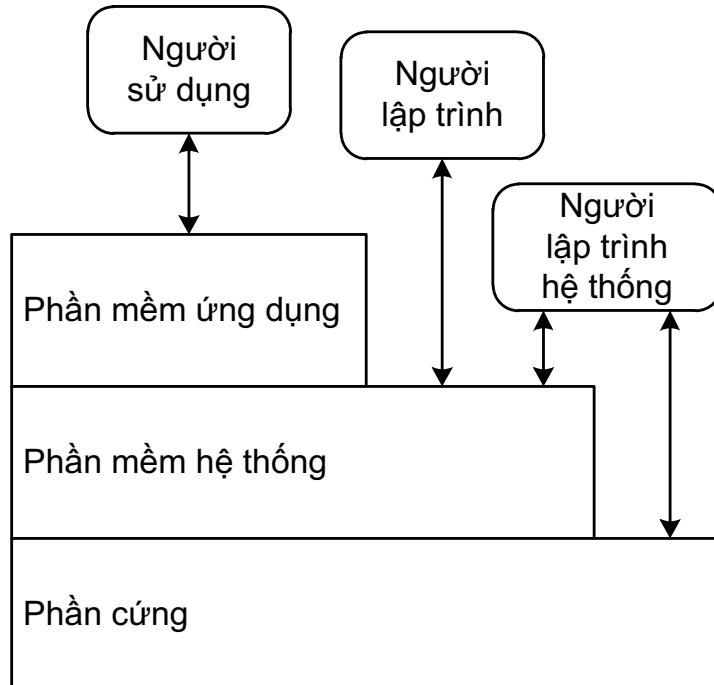
Phân loại máy tính kỷ nguyên sau PC

- Thiết bị di động cá nhân (PMD - Personal Mobile Devices)
 - Smartphones, Tablet
 - Kết nối Internet
- Điện toán đám mây (Cloud Computing)
 - Sử dụng máy tính qui mô lớn (Warehouse Scale Computers), gồm rất nhiều servers kết nối với nhau
 - Cho các công ty thuê một phần để cung cấp dịch vụ phần mềm
 - Software as a Service (SaaS): một phần của phần mềm chạy trên PMD, một phần chạy trên Cloud

1.2. Khái niệm kiến trúc máy tính

- **Kiến trúc máy tính bao gồm:**
 - **Kiến trúc tập lệnh** (Instruction Set Architecture): nghiên cứu máy tính theo cách nhìn của người lập trình
 - **Tổ chức máy tính** (Computer Organization) hay **Vi kiến trúc** (Microarchitecture): nghiên cứu thiết kế máy tính ở mức cao (thiết kế CPU, hệ thống nhớ, cấu trúc bus, ...)
 - **Phần cứng** (Hardware): nghiên cứu thiết kế logic chi tiết và công nghệ đóng gói của máy tính.
- Cùng một kiến trúc tập lệnh có thể có nhiều sản phẩm (tổ chức, phần cứng) khác nhau

Phân lớp máy tính



- **Phần mềm ứng dụng**
 - Được viết theo ngôn ngữ bậc cao
- **Phần mềm hệ thống**
 - Chương trình dịch (Compiler): dịch mã ngôn ngữ bậc cao thành ngôn ngữ máy
 - Hệ điều hành (Operating System)
 - Lập lịch cho các nhiệm vụ và chia sẻ tài nguyên
 - Quản lý bộ nhớ và lưu trữ
 - Điều khiển vào-ra
- **Phần cứng**
 - Bộ xử lý, bộ nhớ, mô-đun vào-ra

Các mức của mã chương trình

- Ngôn ngữ bậc cao
 - High-level language – HLL
 - Mức trừu tượng gần với vấn đề cần giải quyết
 - Hiệu quả và linh động
 - Hợp ngữ
 - Assembly language
 - Mô tả lệnh dưới dạng text
 - Ngôn ngữ máy
 - Machine language
 - Mô tả theo phần cứng
 - Các lệnh và dữ liệu được mã hóa theo nhị phân

High-level
language
program
(in C)

```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly language program (for MIPS)

```

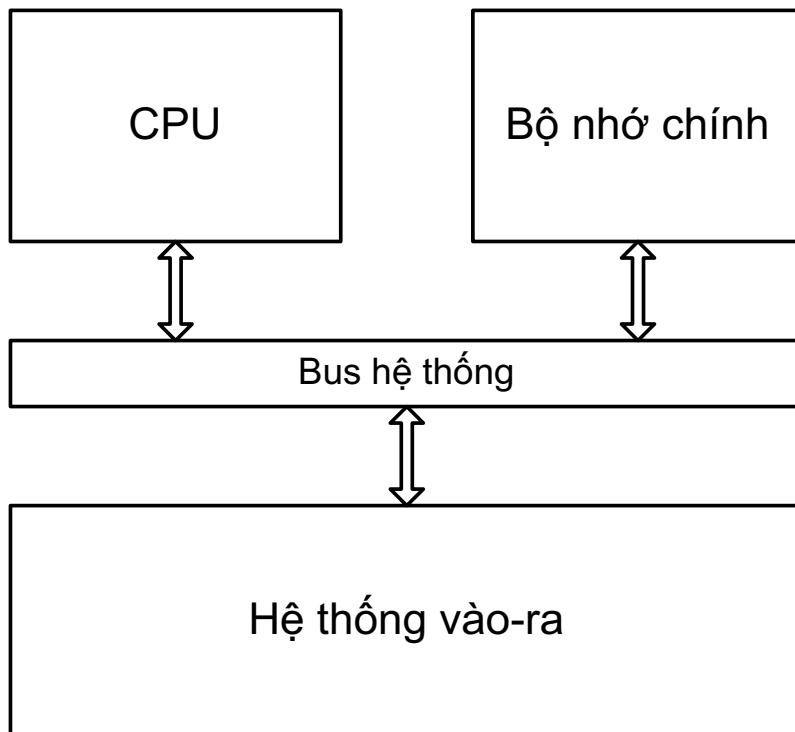
swap:
    multi $2, $5,4
    add   $2, $4,$2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
    jr

```

Binary machine
language
program
(for MIPS)

```
00000000101000100000000100011000  
00000000100000100001000000100001  
100011011110001000000000000000000  
100011100001001000000000000000100  
1010111000010010000000000000000000  
1010110111100010000000000000000100  
00000011111000000000000000000000100
```

Các thành phần cơ bản của máy tính



- Giống nhau với tất cả các loại máy tính
- **Bộ xử lý trung tâm (Central Processing Unit – CPU)**
 - Điều khiển hoạt động của máy tính và xử lý dữ liệu
- **Bộ nhớ chính (Main Memory)**
 - Chứa các chương trình đang thực hiện
- **Hệ thống vào-ra (Input/Output)**
 - Trao đổi thông tin giữa máy tính với bên ngoài
- **Bus hệ thống (System bus)**
 - Kết nối và vận chuyển thông tin

1.3. Sự tiến hóa của công nghệ máy tính

- Máy tính dùng đèn điện tử chân không (1950s)
 - Máy tính ENIAC: máy tính đầu tiên (1946)
 - Máy tính IAS: máy tính von Neumann (1952)
- Máy tính dùng transistors (1960s)
- Máy tính dùng vi mạch SSI, MSI và LSI (1970s)
 - SSI - Small Scale Integration
 - MSI - Medium Scale Integration
 - LSI - Large Scale Integration
- Máy tính dùng vi mạch VLSI (1980s)
 - VLSI - Very Large Scale Integration
- Máy tính dùng vi mạch ULSI (1990s-nay)
 - ULSI - Ultra Large Scale Integration

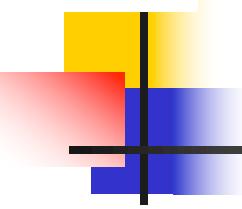
Một số loại vi mạch số điển hình

- **Bộ vi xử lý (Microprocessors)**
 - Một hoặc một vài CPU được chế tạo trên một chip
- **Vi mạch điều khiển tổng hợp (Chipset)**
 - Vi mạch thực hiện các chức năng nối ghép các thành phần của máy tính với nhau
- **Bộ nhớ bán dẫn (Semiconductor Memory)**
 - ROM, RAM, Flash memory
- **Hệ thống trên chip (SoC – System on Chip) hay Bộ vi điều khiển (Microcontrollers)**
 - Tích hợp các thành phần chính của máy tính trên một chip vi mạch
 - Được sử dụng chủ yếu trên smartphone, tablet và các máy tính nhúng

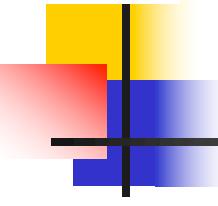
Sự phát triển của bộ vi xử lý

- 1971: bộ vi xử lý 4-bit Intel 4004
- 1972: các bộ xử lý 8-bit
- 1978: các bộ xử lý 16-bit
 - Máy tính cá nhân IBM-PC ra đời năm 1981
- 1985: các bộ xử lý 32-bit
- 2001: các bộ xử lý 64-bit
- 2006: các bộ xử lý đa lõi (multicores)
 - Nhiều CPU trên 1 chip





Hết chương 1



Kiến trúc máy tính

Chương 2

KIẾN TRÚC TẬP LỆNH

Nguyễn Kim Khánh
Trường Đại học Bách khoa Hà Nội

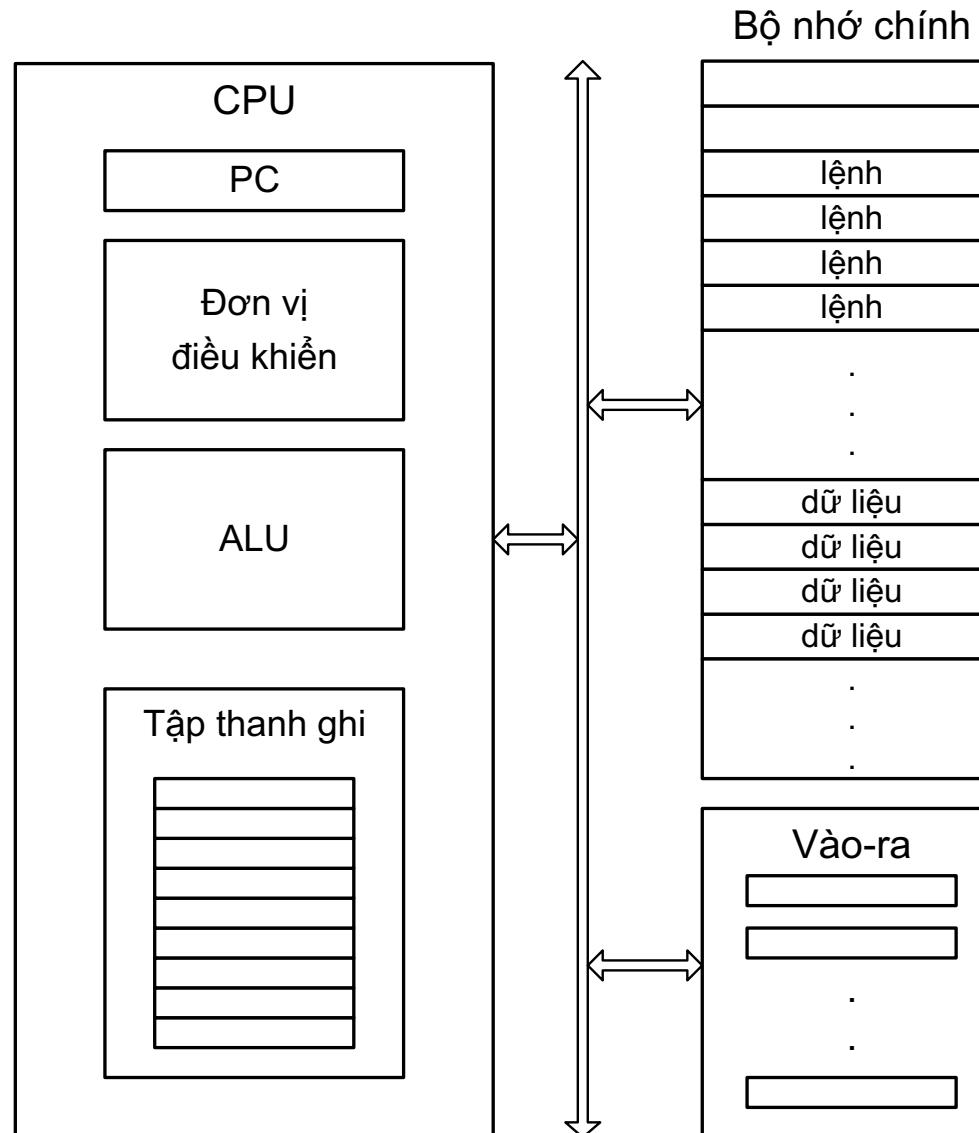
Nội dung

- 2.1. Giới thiệu chung về kiến trúc tập lệnh
- 2.2. Lệnh hợp ngữ và toán hạng
- 2.3. Mã máy
- 2.4. Cơ bản về lập trình hợp ngữ
- 2.5. Các phương pháp định địa chỉ
- 2.6. Dịch và chạy chương trình hợp ngữ

2.1. Giới thiệu chung về kiến trúc tập lệnh

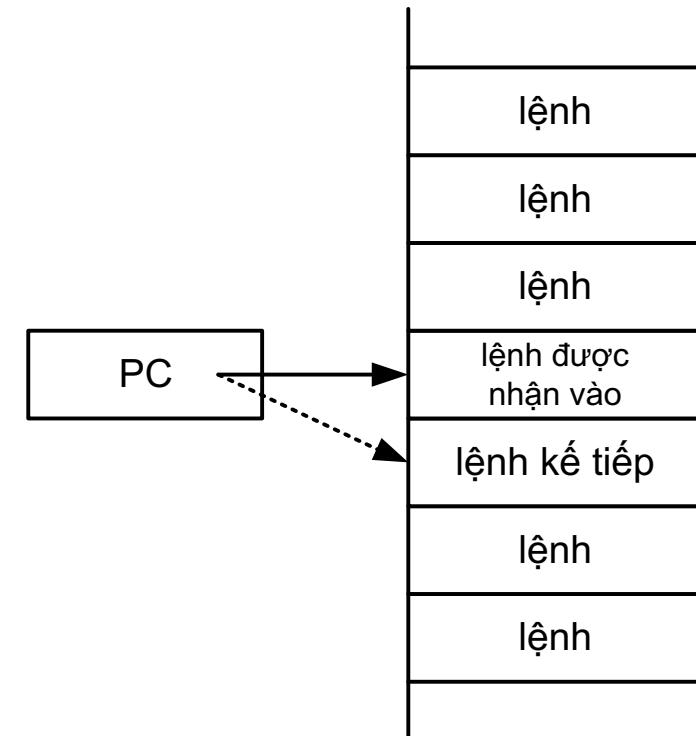
- **Kiến trúc tập lệnh** (Instruction Set Architecture): cách nhìn máy tính bởi người lập trình
- **Vi kiến trúc** (Microarchitecture): cách thực hiện kiến trúc tập lệnh bằng phần cứng
- Ngôn ngữ trong máy tính:
 - **Hợp ngữ** (assembly language):
 - dạng lệnh có thể đọc được bởi con người
 - biểu diễn 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 trỏ 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 4



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 và bộ nhớ chính hoặc 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

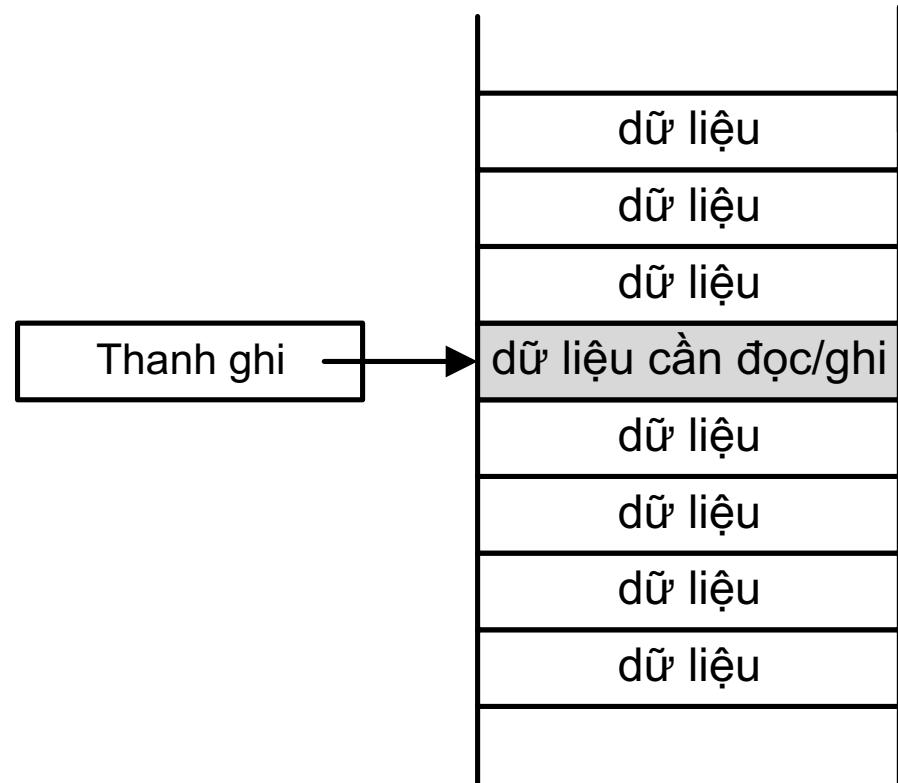
Hàng số địa chỉ

- 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



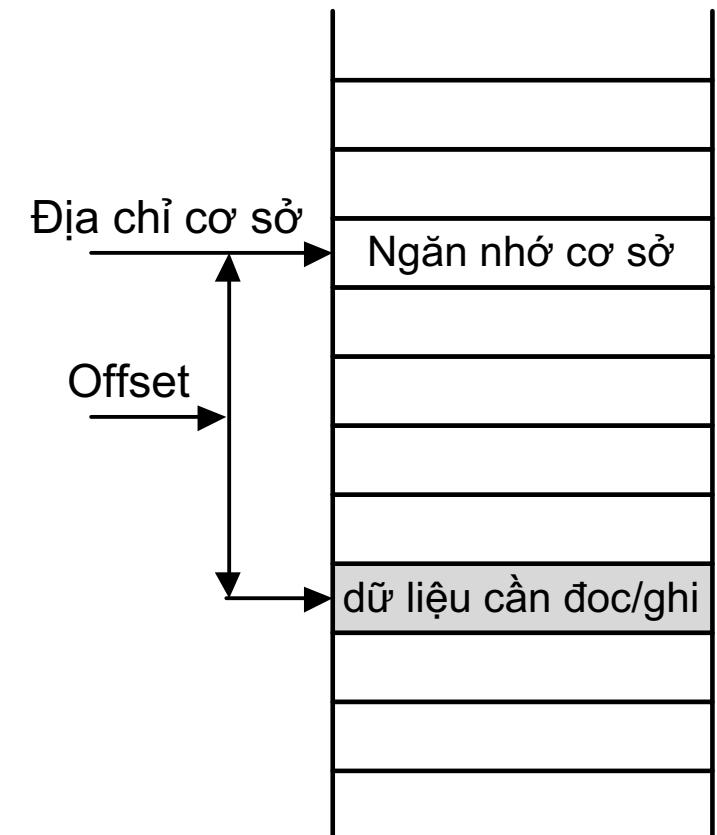
Sử dụng thanh ghi con trỏ

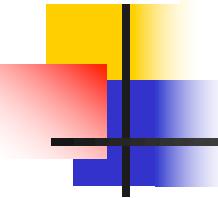
- Trong lệnh cho biết tên thanh ghi con trỏ
- Thanh ghi con trỏ chứa giá trị địa chỉ
- CPU phát địa chỉ này ra để tìm ra ngăn nhớ dữ liệu cần đọc/ghi



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):
gia số địa chỉ giữa ngăn nhớ cần
đọc/ghi so với ngăn nhớ cơ sở
- Địa chỉ của ngăn nhớ cần đọc/ghi
 $= (\text{địa chỉ cơ sở}) + (\text{offset})$
- Có thể sử dụng các thanh ghi để
quản lý các tham số này
- Trường hợp riêng:
 - Địa chỉ cơ sở = 0
 - Offset = 0



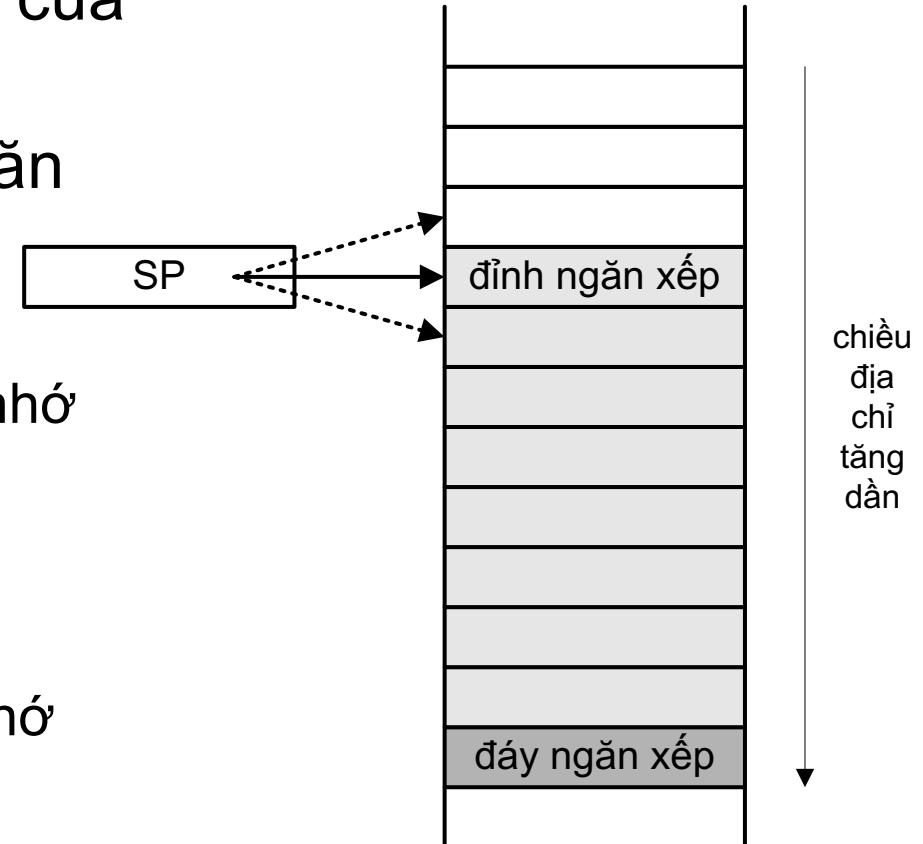


Ngăn xếp (Stack)

- 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
- Đây 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 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 trở vào đáy



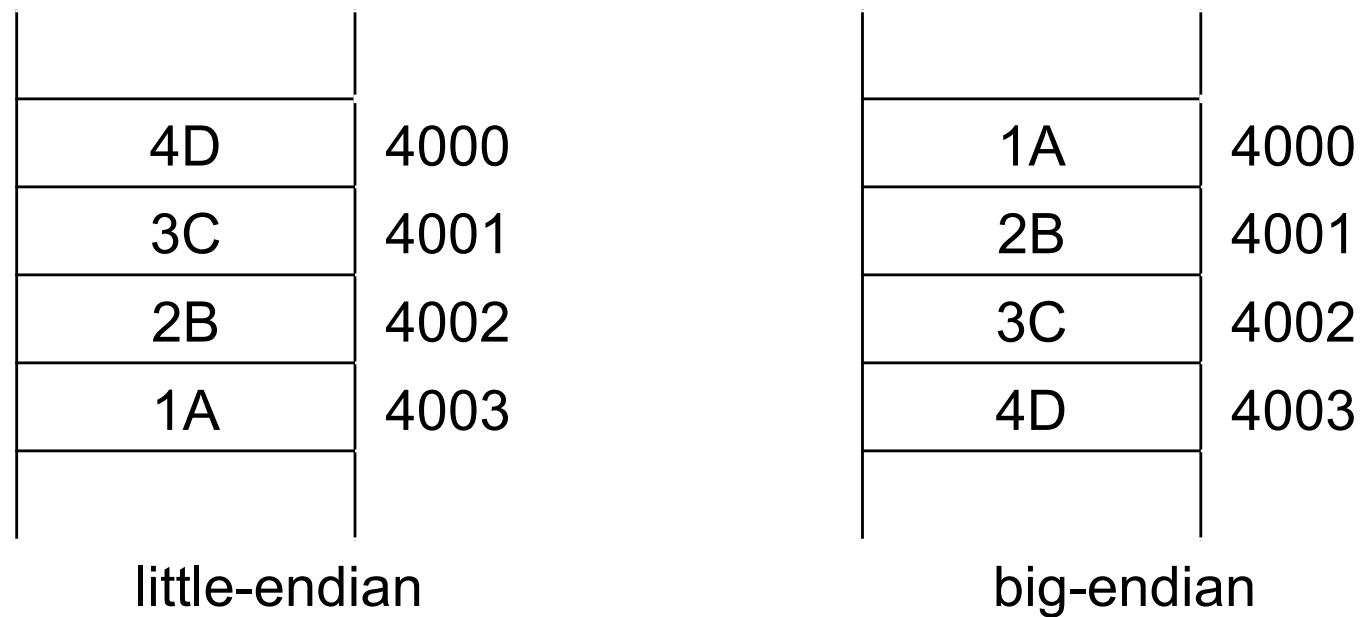
Thứ tự lưu trữ các byte trong bộ nhớ chính

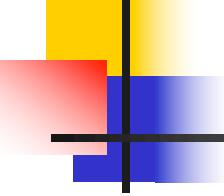
- Bộ nhớ chính được đánh địa chỉ cho từng byte
- Hai cách lưu trữ thông tin nhiều byte:
 - **Đầu nhỏ (Little-endian)**: Byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ lớn.
 - **Đầu to (Big-endian)**: Byte có ý nghĩa cao được lưu trữ 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
---------	----	----	----	----





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 gọi nhớ dạng text, đó chính là các lệnh của hợp ngữ (assembly language)

Dạng lệnh hợp ngũ

- Mã C:

$$a = b + c;$$

- Ví dụ lệnh hợp ngũ:

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 thế 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 **I**nterlocked **P**ipeline **S**tages

- 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 (imgtec.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*

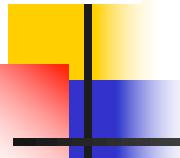
2.2. Lệnh hợp ngũ và các toán hạng

- 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 trả 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)	Dữ liệu hoặc lệnh	Địa chỉ word (theo Hexa)
byte (8-bit)	0x0000 0000	word (32-bit)	0x0000 0000
byte	0x0000 0001	word	0x0000 0004
byte	0x0000 0002	word	0x0000 0008
byte	0x0000 0003	word	0x0000 000C
byte	0x0000 0004	word	0x0000 0010
byte	0x0000 0005	word	0x0000 0014
byte	0x0000 0006	word	0x0000 0018
byte	0x0000 0007	.	.
.	.	.	.
.	.	.	.
byte	0xFFFF FFFF	word	0xFFFF FFF4
byte	0xFFFF FFFC	word	0xFFFF FFF8
byte	0xFFFF FFFD	word	0xFFFF FFFC
byte	0xFFFF FFFE		
byte	0xFFFF FFFF		

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

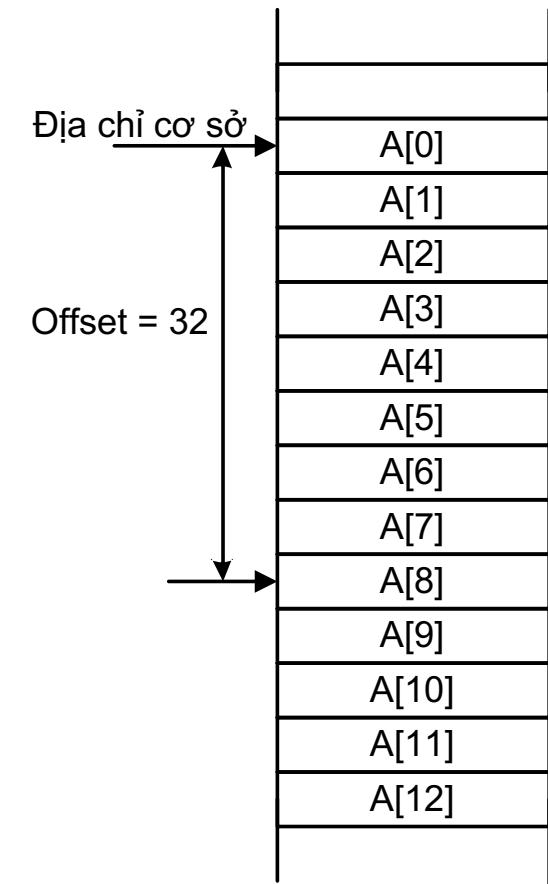
Ví dụ toán hạng bộ nhớ

- 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

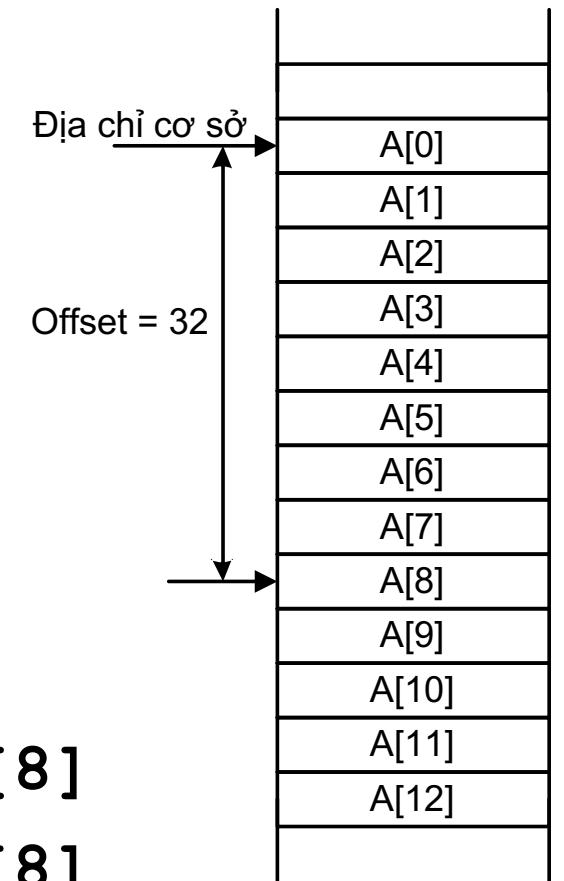
- Mã hợp ngữ MIPS:

Chỉ số 8, do đó offset = 32

```
lw    $t0, 32($s3)    # $t0 = A[8]
add $s1, $s2, $t0      # g = h+A[8]
```

offset

base register



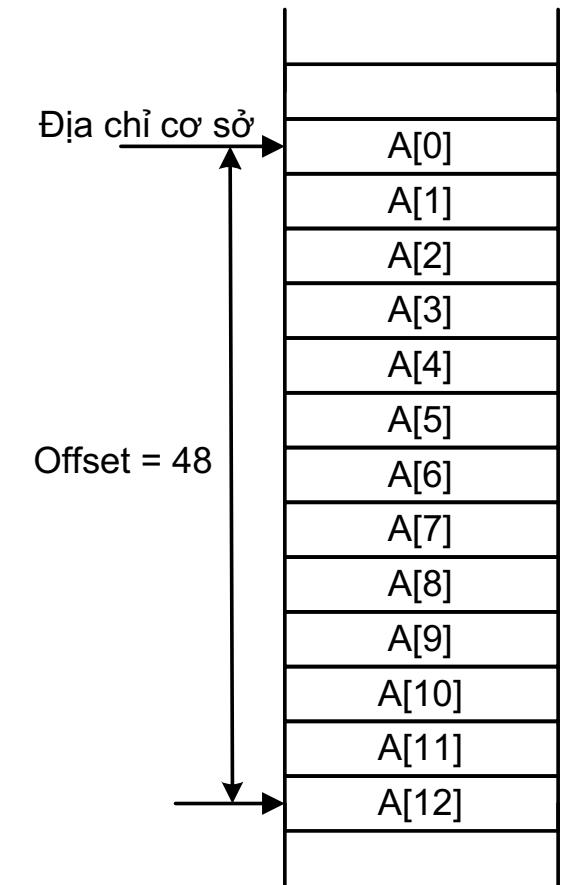
(Chú ý: offset phải là hằng số, có thể dương hoặc âm)

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];

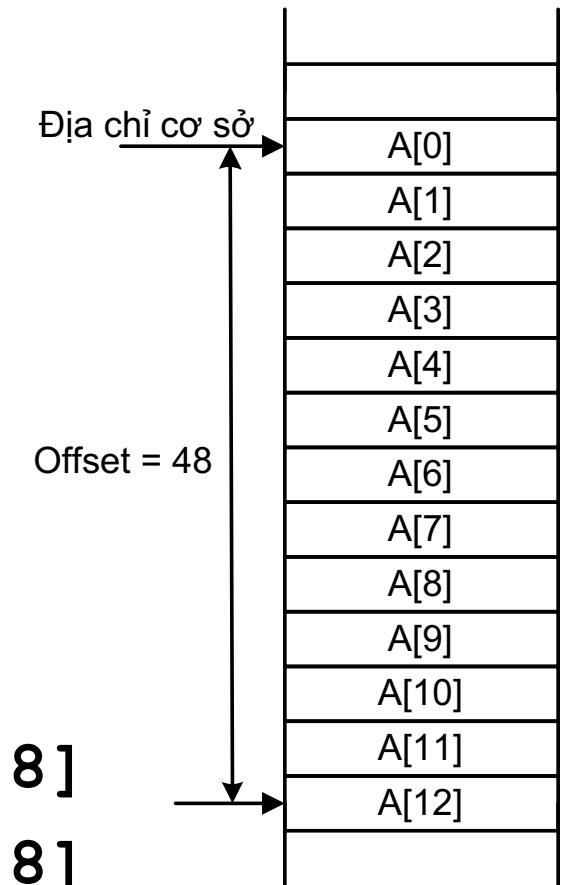
- h ở \$s2
- \$s3 chứa địa chỉ cơ sở của mảng A

- 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: $[-2^{n-1}, + (2^{n-1}-1)]$
 - Các lệnh **add**, **sub** dành cho số nguyên có dấu
- Số nguyên không dấu:
 - Với n bit, dải biểu diễn: $[0, 2^n - 1]$
 - Các lệnh **addu**, **subu** dành cho số nguyên không dấu
- Qui ước biểu diễn hàng 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

2.3. 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

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

Lệnh kiểu I

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

Lệnh kiểu J

op	address
6 bits	26 bits

Lệnh kiểu R (Registers)

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

■ 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

Ví dụ mã máy của lệnh add, sub



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

0	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

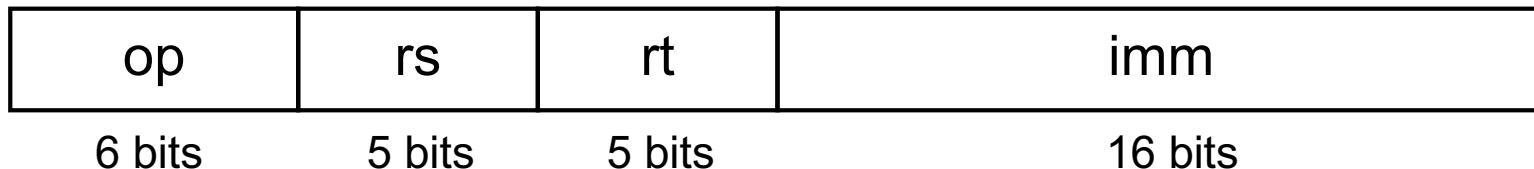
(0x02324020)

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

0	\$t3	\$t5	\$s0	0	sub
0	11	13	16	0	34
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



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



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

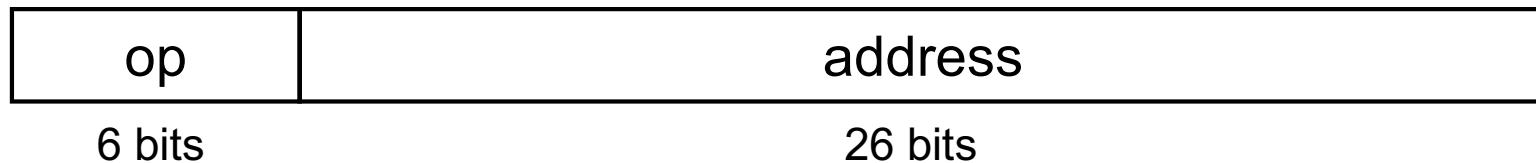
43	9	17	4
----	---	----	---

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



2.4. Cơ bản về lập trình hợp ngữ

1. Các lệnh logic
2. Nạp hàng số vào thanh ghi
3. Tạo các cấu trúc điều khiển
4. Lập trình mảng dữ liệu
5. Chương trình con
6. Dữ liệu ký tự
7. Lệnh nhân và lệnh chia
8. Các lệnh với số dấu phẩy động

1. 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

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

Kết quả thanh ghi đích

\$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

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

Kết quả thanh ghi đích

\$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

andi \$s2,\$s1,0xFA34 \$s2

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

Kết quả thanh ghi đích

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
← Zero-extended →								

Mã hợp ngữ

andi \$s2,\$s1,0xFA34

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

Kết quả thanh ghi đích

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
- **sll** - 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)
- **srl** - 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	1101	= 13
\$t2	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]

2. 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	0000	0100	0000	0011	1011

or

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

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

3. 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

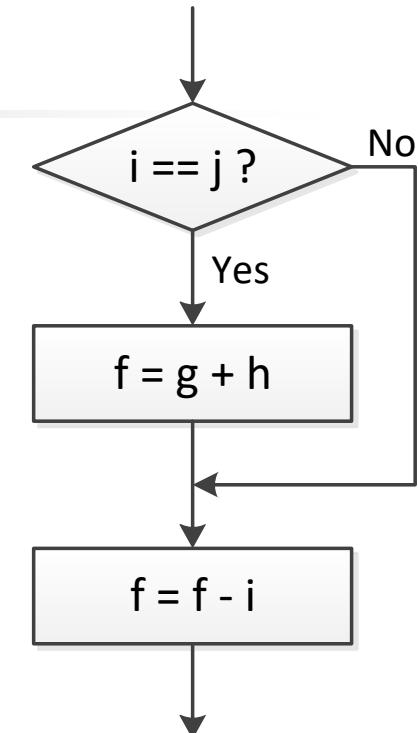
Dịch câu lệnh if

- 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;

```

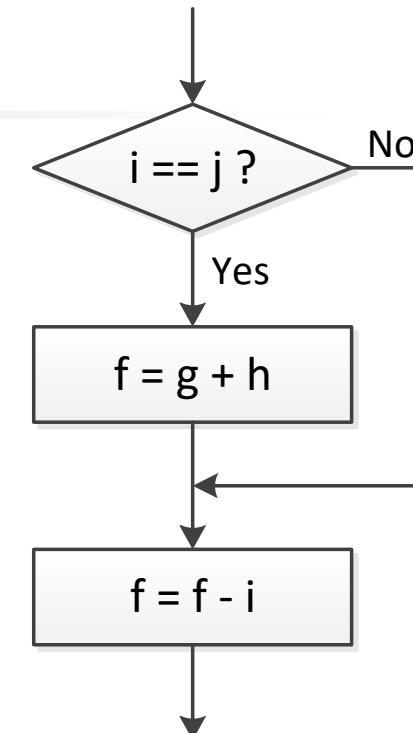
- f, g, h, i, j ở \$s0, \$s1, \$s2, \$s3, \$s4

- 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

```



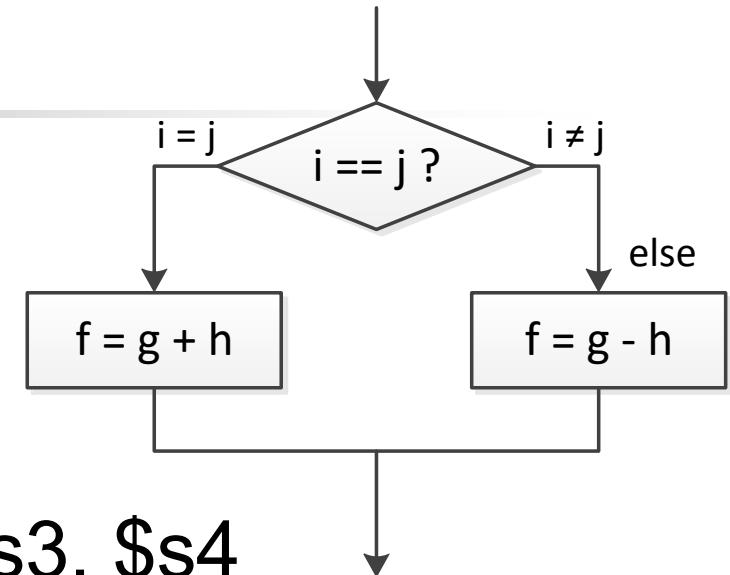
Điều kiện hợp ngữ ngược với điều kiện của ngôn ngữ bậc cao

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



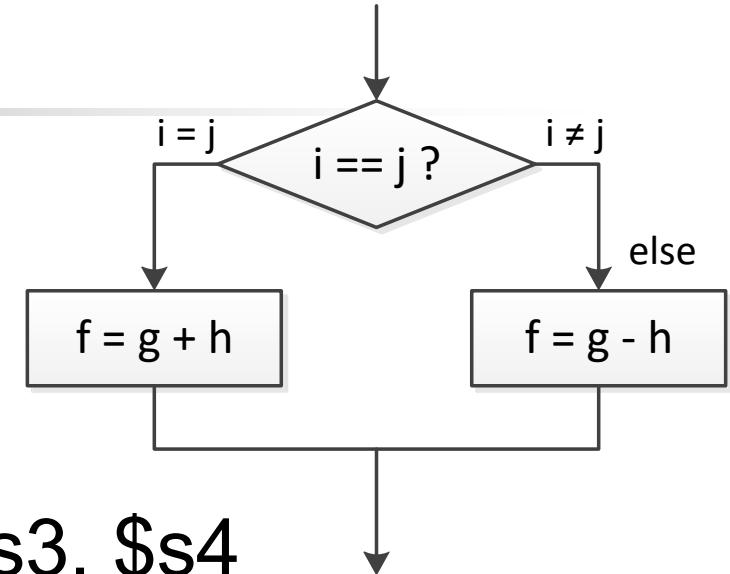
Dịch câu lệnh if/else

- 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

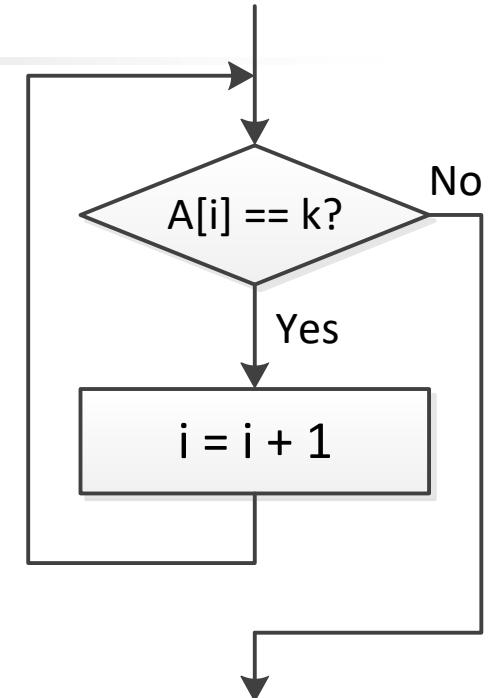
```
# $s0 = amount, $s1 = fee
case20:
    addi $t0, $0, 20          # $t0 = 20
    bne $s0, $t0, case50     # amount == 20? if not, skip to case50
    addi $s1, $0, 2            # if so, fee = 2
    j done                   # and break out of case
case50:
    addi $t0, $0, 50          # $t0 = 50
    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:

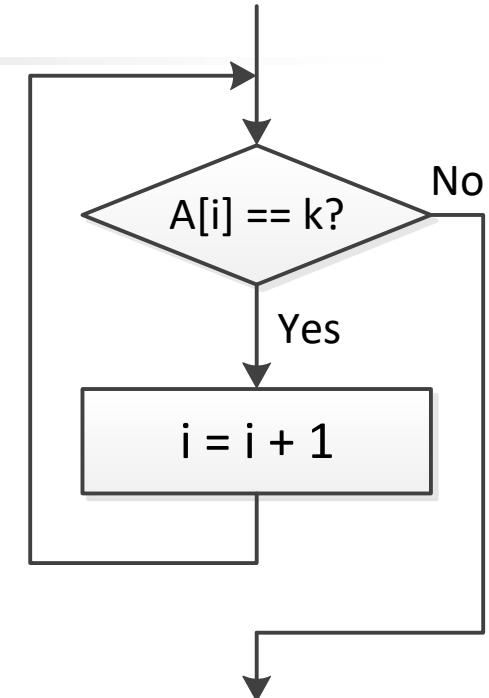
```
while (A[i] == k) i += 1;
```

- 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:

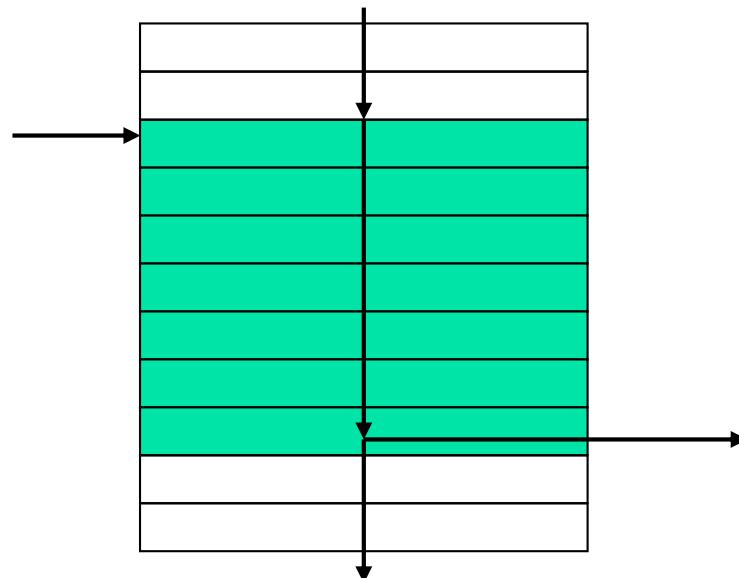
```
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i++) {
    sum = sum + i;
}
```

■ 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 dây 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 $i \geq 101$

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

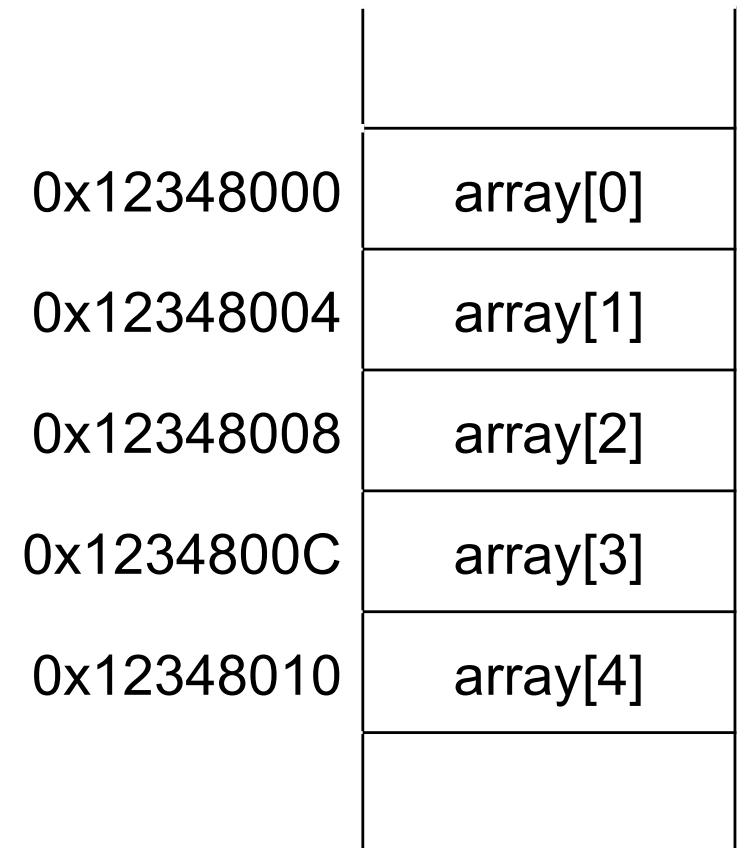
done:

4. 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



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:

```

5. 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 trỏ về (thanh ghi 31)

Các lệnh gọi thủ tục

- Gọi thủ tục: jump and link

jal ProcedureAddress

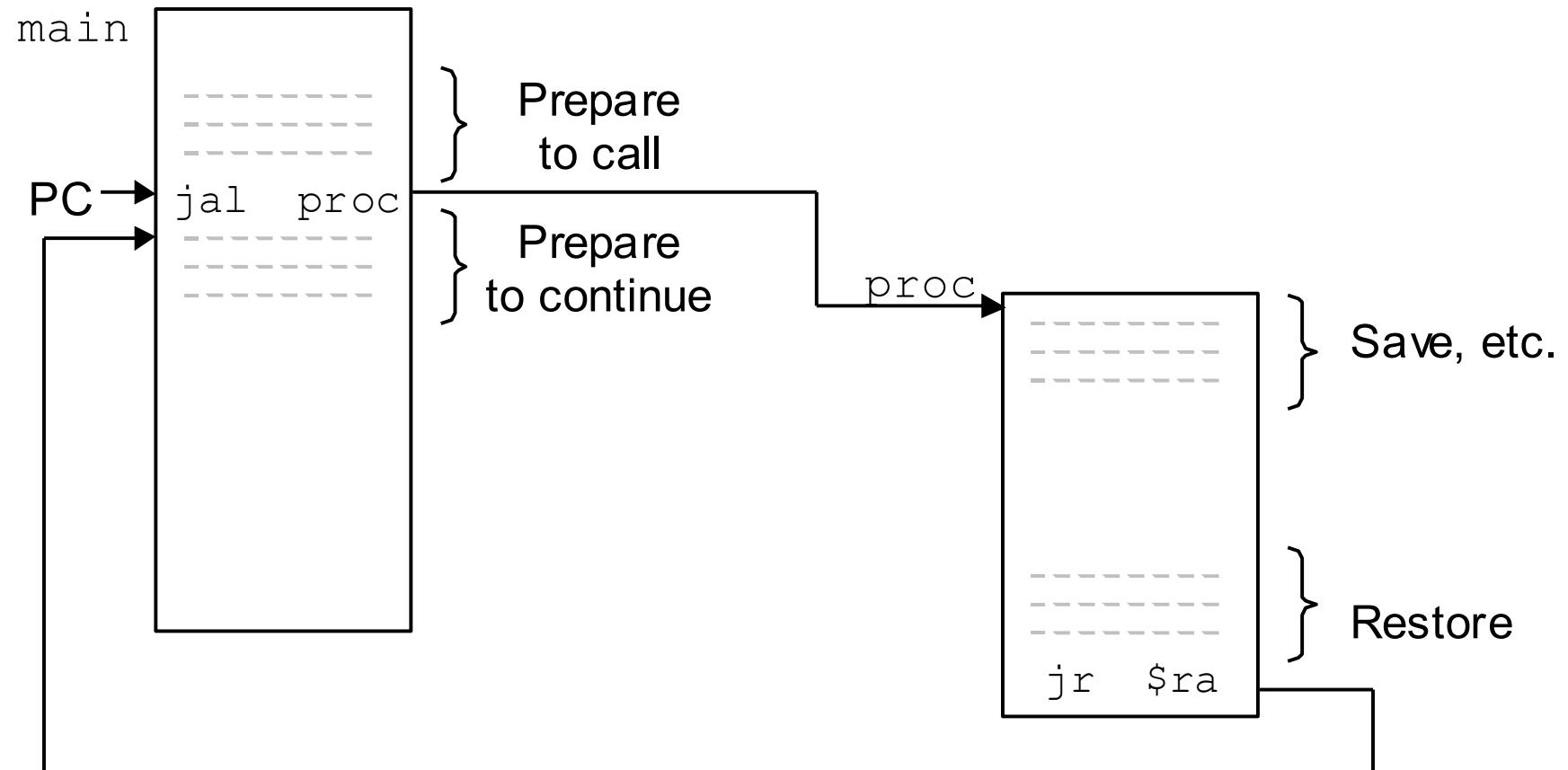
- Địa chỉ của lệnh kế tiếp (địa chỉ trả về) được cất ở thanh ghi \$ra
- Nhảy đến địa chỉ của thủ tục

- Trở về từ thủ tục: jump register

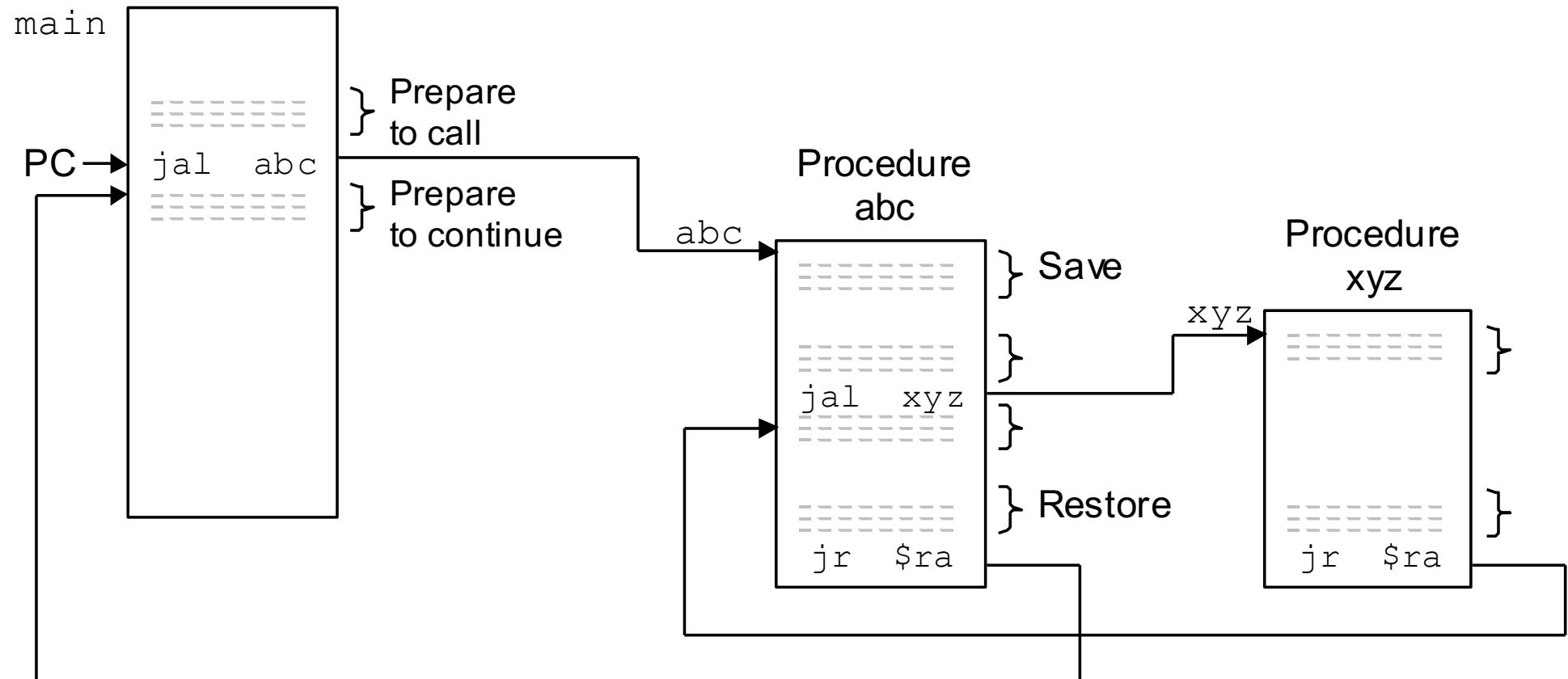
jr \$ra

- Copy nội dung thanh ghi \$ra (đang chứa địa chỉ trả 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)	# cắt nội dung \$t1
sw \$t0, 4(\$sp)	# cắt nội dung \$t0
sw \$s0, 0(\$sp)	# cắt 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

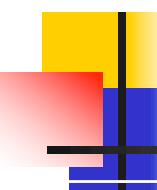
Ví dụ Thủ tục cành

- 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

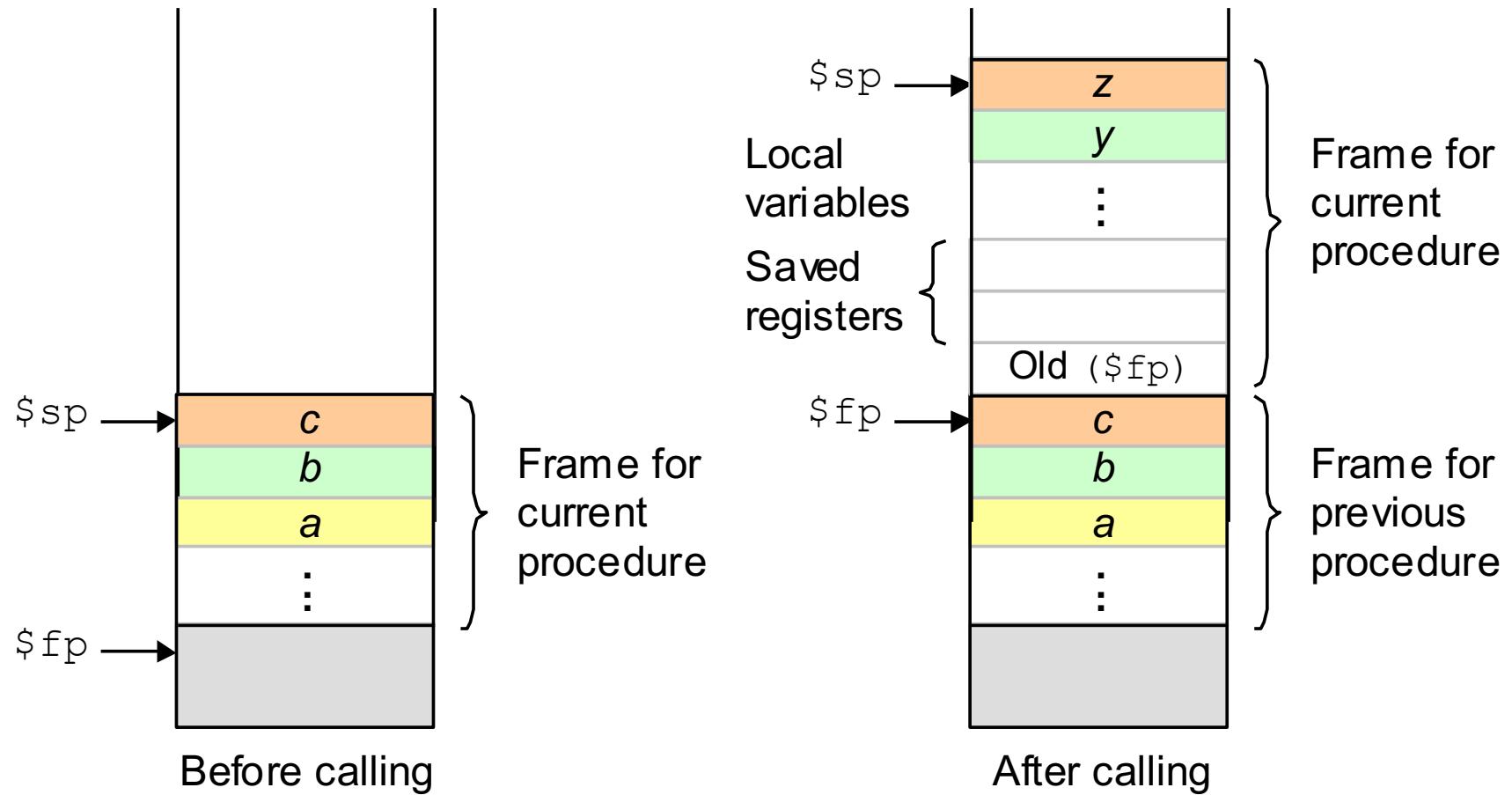


Mã hợp ngữ MIPS

fact:

addi \$sp, \$sp, -8	# dành stack cho 2 mục
sw \$ra, 4(\$sp)	# cất địa chỉ trả 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	# lấy 2 mục từ stack
jr \$ra	# và trả 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 trả về
addi \$sp, \$sp, 8	# lấy 2 mục từ stack
mul \$v0, \$a0, \$v0	# nhân để nhận kết quả
jr \$ra	# và trả về

Sử dụng Stack khi gọi thủ tục



6. Dữ liệu ký tự

- Các tập ký tự được mã hóa theo byte
 - ASCII: 128 ký tự
 - 95 ký thị 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
 - Nạp/Lưu byte/halfword trong MIPS
-
- **lb rt, offset(rs)** **lh rt, offset(rs)**
 - Mở rộng theo số có dấu thành 32 bits trong rt

 - **lbu rt, offset(rs)** **lhu rt, offset(rs)**
 - Mở rộng theo số không dấu thành 32 bits trong rt

 - **sb rt, offset(rs)** **sh rt, offset(rs)**
 - Chỉ lưu byte/halfword bên phải

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
```

7. 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`

8. 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**

2.5. 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 $\times 4$
 - Chú ý: trước đó PC đã được tăng lên
 - 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

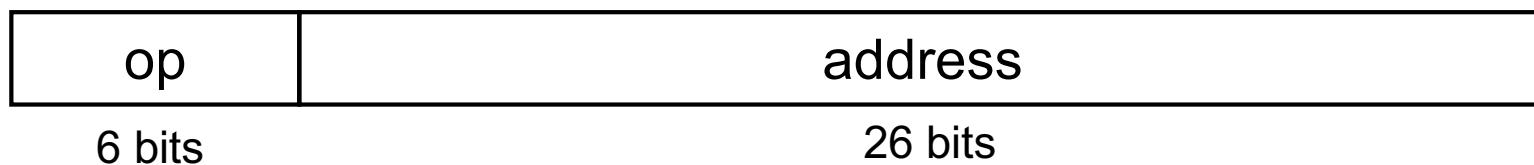
Lệnh mã máy

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

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

Địa chỉ hóa cho lệnh Jump

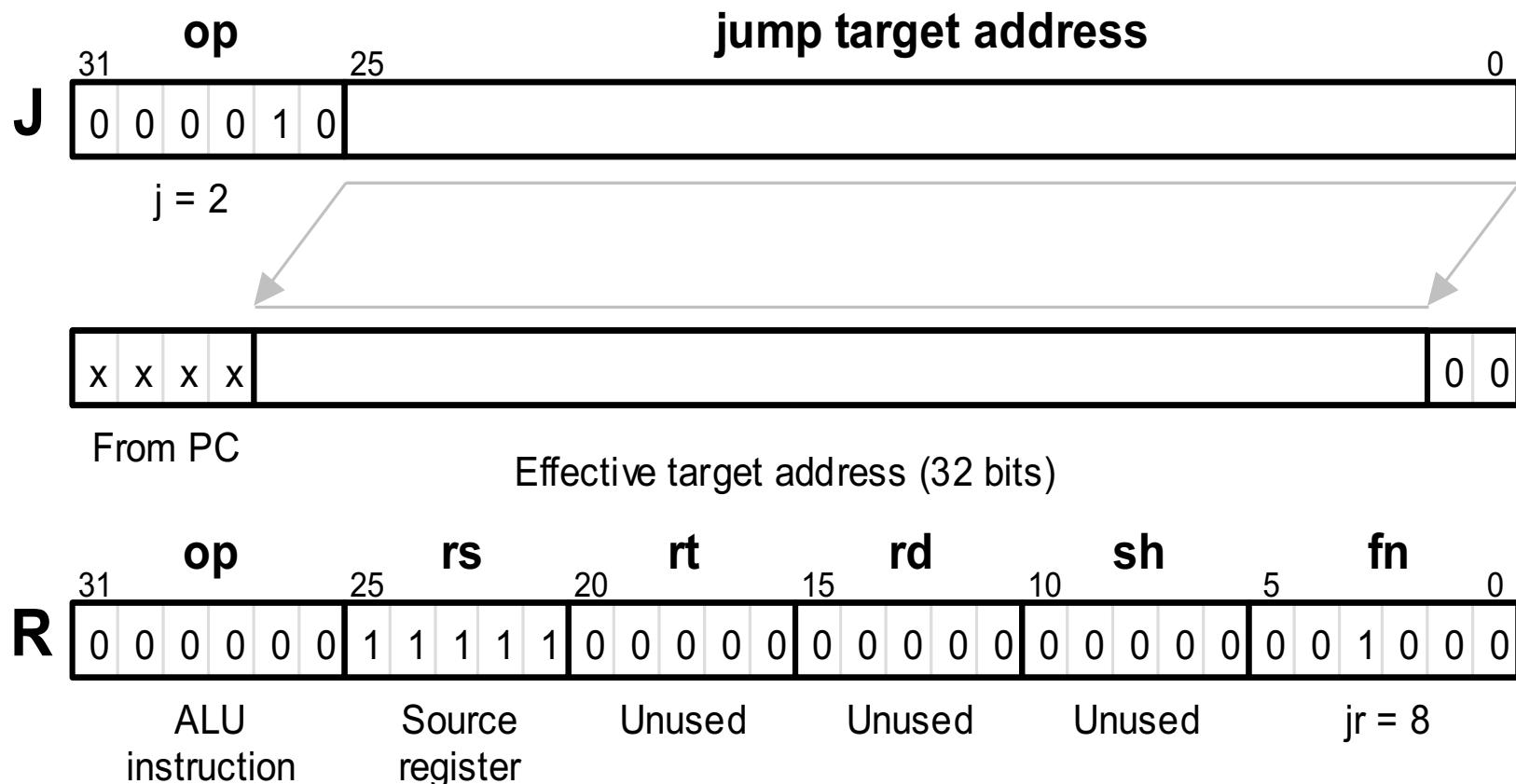
- Đích của lệnh Jump (**j** và **jal**) có thể là bất kỳ chỗ nào trong chương trình
 - Cần mã hóa đầy đủ địa chỉ trong lệnh



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

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 chưa địa chỉ trả về



Ví dụ mã hóa lệnh

Loop:	sll	\$t1, \$s3, 2	0x8000
	add	\$t1, \$t1, \$s6	0x8004
	lw	\$t0, 0(\$t1)	0x8008
	bne	\$t0, \$s5, Exit	0x800C
	addi	\$s3, \$s3, 1	0x8010
	j	Loop	0x8014
Exit:	...		0x8018

0	0	19	9	2	0
0	9	22	9	0	32
35	9	8		0	
5	8	21		2	
8	19	19		1	
2			0x2000		

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



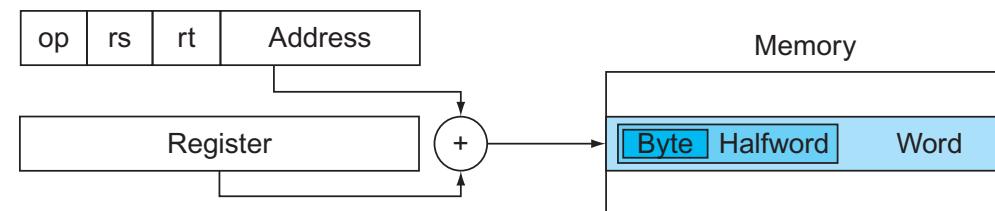
2. Định địa chỉ thanh ghi

2. Register addressing



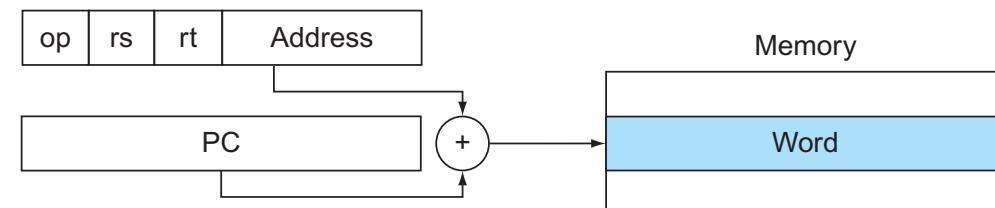
3. Định địa chỉ cơ sở

3. Base addressing



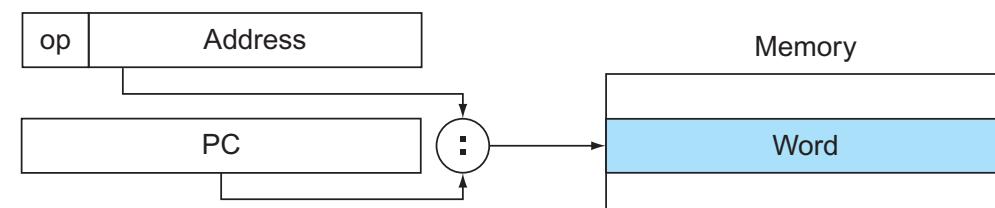
4. Định địa chỉ tương đối với PC

4. PC-relative addressing



5. Định địa chỉ giả trực tiếp

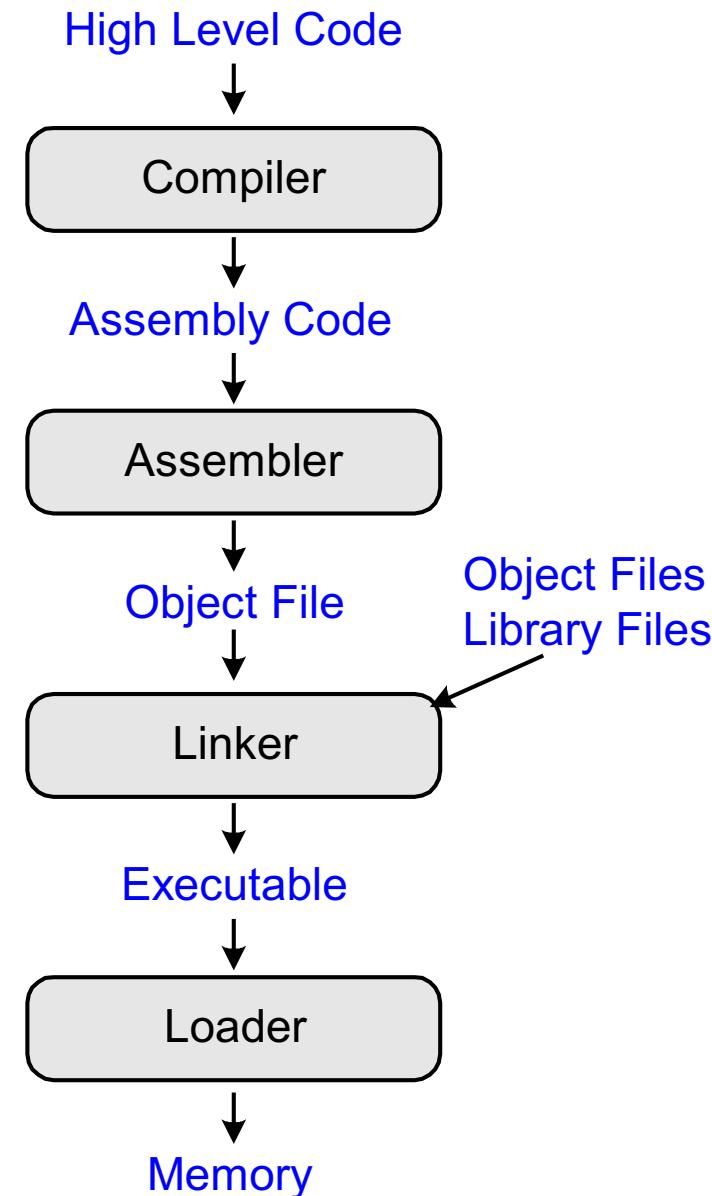
5. Pseudodirect addressing



2.6. Dịch và chạy chương trình hợp ngữ

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

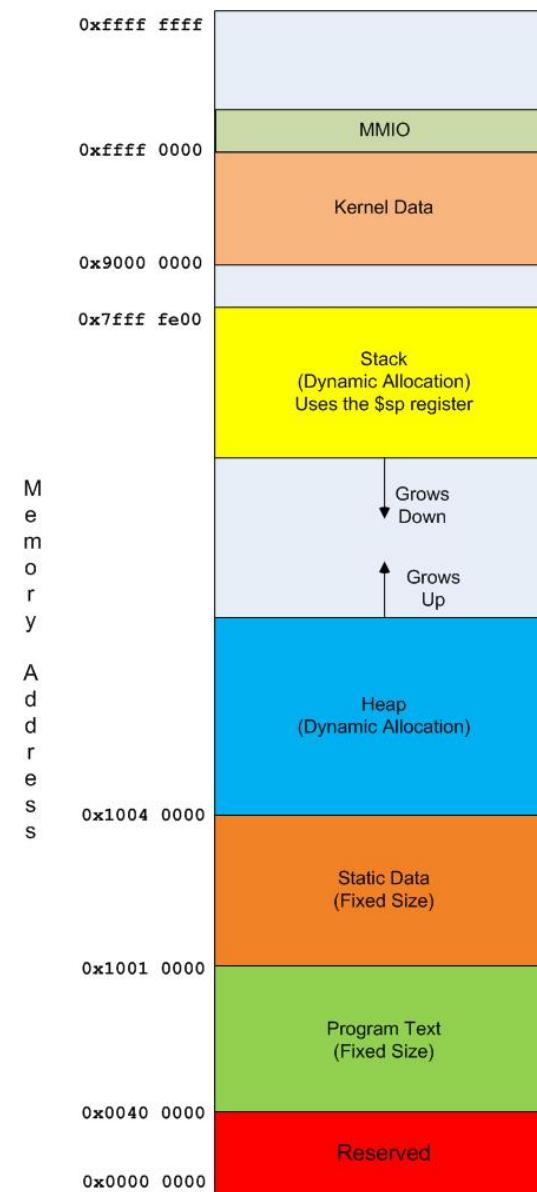
Dịch và chạy ứng dụng



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 shows the MARS 4.5 assembly debugger interface. The main window displays the assembly code for a program named 'fullprogram.asm'. The code initializes variables f, g, and y to 0, then performs a stack frame setup, calculates the sum of \$a0 and \$a1, and stores the result in \$v0. The right side of the interface features a register window showing the values of all general-purpose registers (\$zero to \$lo) and three coprocessor registers (Coproc 1 and Coproc 0). The bottom of the screen shows the Mac OS X dock with various application icons.

File Edit Run Settings Tools Help

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

Run speed at max (no interaction)

Edit Execute

fullprogram.asm

```
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
18 syscall
19 sum:
20 add $v0, $a0, $a1      # $v0 = a + b
```

Line: 1 Column: 1 Show Line Numbers

Mars Messages Run I/O

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	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Mars Messages Run I/O

Clear

46% Tue Feb 11 10:09 AM

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)

Text Segment

Program Arguments:

Bkpt	Address	Code	Basic	Source
	0x00400000	0x23bdffffc	addi \$29,\$29,0xffff...	7: addi \$sp, \$sp, -4 # stack frame
	0x00400004	0xaafb0000	sw \$31,0x0000000(\$29)	8: sw \$ra, 0(\$sp) # store \$ra
	0x00400008	0x20040002	addi \$4,\$0,0x00000002	9: addi \$a0, \$0, 2 # \$a0 = 2
	0x0040000c	0x3c011001	lui \$1,0x00001001	10: sw \$a0, f # f = 2
	0x00400010	0xac240000	sw \$4,0x0000000(\$1)	
	0x00400014	0x20050003	addi \$5,\$0,0x00000003	11: addi \$a1, \$0, 3 # \$a1 = 3
	0x00400018	0x3c011001	lui \$1,0x00001001	12: sw \$a1, g # g = 3
	0x0040001c	0xac250004	sw \$5,0x00000004(\$1)	
	0x00400020	0x0c10000f	jal 0x0040003c	13: jal sum # call sum
	0x00400024	0x3c011001	lui \$1,0x00001001	14: sw \$v0, y # y = sum()
	0x00400028	0xac220008	sw \$2,0x00000008(\$1)	
	0x0040002c	0x8fb00000	lw \$31,0x0000000(\$29)	15: lw \$ra, 0(\$sp) # restore \$ra

Labels

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

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

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	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Mars Messages Run I/O

Clear

Vùng nhớ lệnh

Text Segment

Program Arguments:

Bkpt	Address	Code	Basic	Source
	0x00400000	0x23bdffffc	addi \$29,\$29,0xffffffffc	7: addi \$sp, \$sp, -4 # stack frame
	0x00400004	0xafbf0000	sw \$31,0x00000000(\$29)	8: sw \$ra, 0(\$sp) # store \$ra
	0x00400008	0x20040002	addi \$4,\$0,0x00000002	9: addi \$a0, \$0, 2 # \$a0 = 2
	0x0040000c	0x3c011001	lui \$1,0x00001001	10: sw \$a0, f # f = 2
	0x00400010	0xac240000	sw \$4,0x00000000(\$1)	
	0x00400014	0x20050003	addi \$5,\$0,0x00000003	11: addi \$a1, \$0, 3 # \$a1 = 3
	0x00400018	0x3c011001	lui \$1,0x00001001	12: sw \$a1, g # g = 3
	0x0040001c	0xac250004	sw \$5,0x00000004(\$1)	
	0x00400020	0x0c10000f	jal 0x0040003c	13: jal sum # call sum
	0x00400024	0x3c011001	lui \$1,0x00001001	14: sw \$v0, y # y = sum()
	0x00400028	0xac220008	sw \$2,0x00000008(\$1)	
	0x0040002c	0x8fbf0000	lw \$31,0x00000000(\$29)	15: lw \$ra, 0(\$sp) # restore \$ra
	0x00400030	0x23bd0004	addi \$29,\$29,0x00000004	16: addi \$sp, \$sp, 4 # restore \$sp
	0x00400034	0x2402000a	addiu \$2,\$0,0x0000000a	17: li \$v0, 10
	0x00400038	0x0000000c	syscall	18: syscall
	0x0040003c	0x00851020	add \$2,\$4,\$5	20: add \$v0, \$a0, \$a1 # \$v0 = a + b
	0x00400040	0x03e00008	jr \$31	21: jr \$ra # return

Tập thanh ghi

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	0x7fffeffc
\$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	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

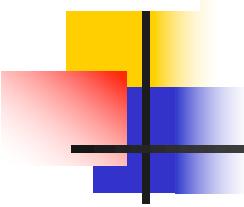
Nhãn và biến

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

Data Text

Ví dụ lệnh giả (Pseudoinstruction)

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



Hết chương 2

Chương 3 SỐ HỌC MÁY TÍNH

Nguyễn Kim Khánh
Trường Đại học Bách khoa Hà Nội

Nội dung

- 3.1. Biểu diễn số nguyên
- 3.2. Phép cộng và phép trừ số nguyên
- 3.3. Phép nhân và phép chia số nguyên
- 3.4. Số dấu phẩy động

3.1. Biểu diễn số nguyên

- Số nguyên không dấu (Unsigned Integer)
- Số nguyên có dấu (Signed Integer)

1. Biểu diễn số nguyên không dấu

- **Nguyên tắc tổng quát:** Dùng n bit biểu diễn số nguyên không dấu A:

$$a_{n-1}a_{n-2} \dots a_2a_1a_0$$

Giá trị của A được tính như sau:

$$A = \sum_{i=0}^{n-1} a_i 2^i$$

Dải biểu diễn của A: [0, $2^n - 1$]

Với $n = 8$ bit

Biểu diễn được các giá trị từ 0 đến 255 ($2^8 - 1$)

Chú ý:

$$\begin{array}{r}
 1111\ 1111 \\
 +\ 0000\ 0001 \\
 \hline
 1\ 0000\ 0000
 \end{array}$$

có *nhớ ra ngoài*
(*Carry out*)

$$255 + 1 = 0 ???$$

do vượt ra khỏi dải biểu diễn

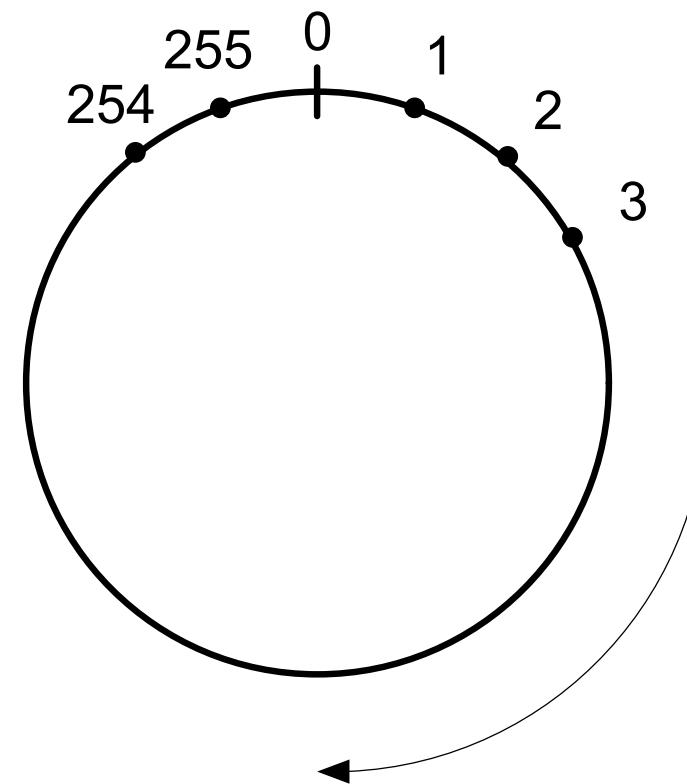
Biểu diễn nhị phân	Giá trị thập phân
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
0000 0100	4
...	
1111 1110	254
1111 1111	255

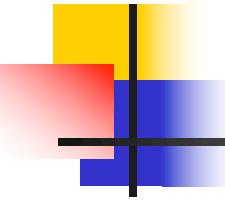
Trục số học với $n = 8$ bit

Trục số học:



Trục số học máy tính:





Với $n = 16$ bit, 32 bit, 64 bit

- $n= 16$ bit: dải biểu diễn từ 0 đến 65535 ($2^{16} - 1$)
- $n= 32$ bit: dải biểu diễn từ 0 đến $2^{32} - 1$
- $n= 64$ bit: dải biểu diễn từ 0 đến $2^{64} - 1$

2. Biểu diễn số nguyên có dấu

Số bù một và Số bù hai

- Cho một số nhị phân A được biểu diễn bằng n bit
 - Số bù một của A = đảo giá trị các bit của A
 - (Số bù hai của A) = (Số bù một của A) + 1

Biểu diễn số nguyên có dấu theo mã bù hai

Nguyên tắc tổng quát: Dùng n bit biểu diễn số nguyên có dấu A:

$$a_{n-1}a_{n-2}\dots a_2a_1a_0$$

- Với A là số dương: bit $a_{n-1} = 0$, các bit còn lại biểu diễn độ lớn như số không dấu
- Với A là số âm: được biểu diễn bởi số bù hai của số dương tương ứng, vì vậy bit $a_{n-1} = 1$

Xác định giá trị của số dương

- Dạng tổng quát của số dương:

$$0a_{n-2} \dots a_2a_1a_0$$

- Giá trị của số dương:

$$A = \sum_{i=0}^{n-2} a_i 2^i$$

- Dải biểu diễn cho số dương: $[0, (2^{n-1} - 1)]$

Xác định giá trị của số âm

- Dạng tổng quát của số âm:

$$1a_{n-2} \dots a_2a_1a_0$$

- Giá trị của số âm:

$$A = -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Dải biểu diễn cho số âm: [- 2^{n-1} , -1]

Công thức tổng quát cho số nguyên có dấu

- Dạng tổng quát của số nguyên có dấu A:

$$a_{n-1}a_{n-2} \dots a_2a_1a_0$$

- Giá trị của A được xác định như sau:

$$A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

- Dải biểu diễn: $[-(2^{n-1}), +(2^{n-1}-1)]$

Với $n = 8$ bit

- Biểu diễn được các giá trị từ -2^7 đến $+2^7 - 1$
 - 128 đến +127
 - Chỉ có một giá trị 0
 - Không biểu diễn cho giá trị +128

Chú ý:

$$+127 + 1 = -128$$

$$(-128) + (-1) = +127$$

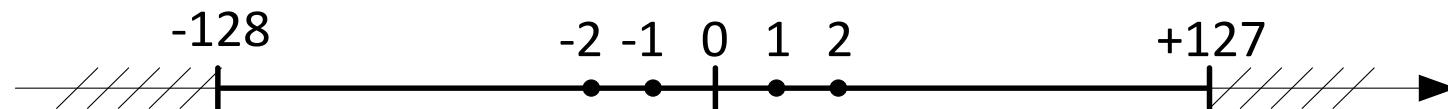
có tràn xảy ra (Overflow)

(do vượt ra khỏi dải biểu diễn)

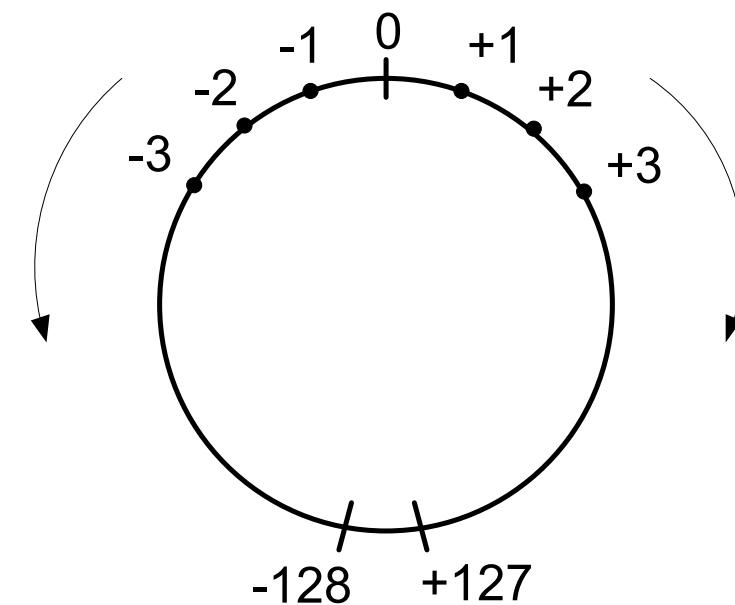
Giá trị thập phân	Biểu diễn bù hai
0	0000 0000
+1	0000 0001
+2	0000 0010
	...
+126	0111 1110
+127	0111 1111
-128	1000 0000
-127	1000 0001
	...
-2	1111 1110
-1	1111 1111

Trục số học số nguyên có dấu với $n = 8$ bit

- Trục số học:



- Trục số học máy tính:



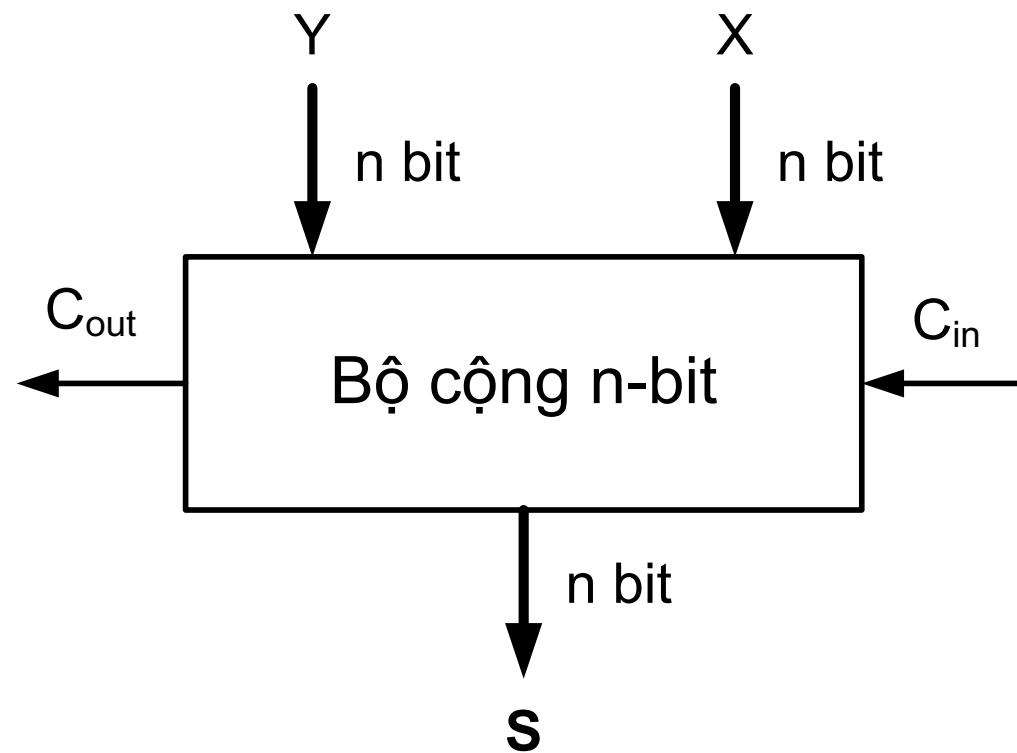
Với $n = 16$ bit, 32 bit, 64 bit

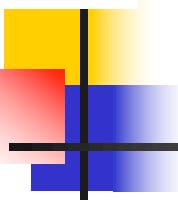
- Với $n = 16$ bit: biểu diễn từ -2^{15} đến $2^{15}-1$
- Với $n = 32$ bit: biểu diễn từ -2^{31} đến $2^{31}-1$
- Với $n = 64$ bit: biểu diễn từ -2^{63} đến $2^{63}-1$

3.2. Thực hiện phép cộng/trừ với số nguyên

1. Phép cộng số nguyên không dấu

Bộ cộng n-bit





Nguyên tắc cộng số nguyên không dấu

- Khi cộng hai số nguyên không dấu n-bit, kết quả nhận được là n-bit:
 - Nếu $C_{out} = 0 \rightarrow$ nhận được **kết quả đúng**
 - Nếu $C_{out} = 1 \rightarrow$ nhận được **kết quả sai**, do có *nhớ ra ngoài (Carry Out)*
- Hiện tượng *nhớ ra ngoài* xảy ra khi:
 $tổng > (2^n - 1)$

Ví dụ cộng số nguyên không dấu

- $$\begin{array}{r} 57 \\ + 34 \\ \hline 91 \end{array} = + \begin{array}{r} 0011\ 1001 \\ 0010\ 0010 \\ \hline 0101\ 1011 \end{array} = 64+16+8+2+1=91 \rightarrow \text{đúng}$$

- $$\begin{array}{r} 209 \\ + 73 \\ \hline 282 \end{array} = + \begin{array}{r} 1101\ 0001 \\ 0100\ 1001 \\ \hline 1\ 0001\ 1010 \end{array}$$

 kết quả = 0001 1010 = 16+8+2=26 → sai
 do có *nhớ ra ngoài* ($C_{out}=1$)

Để có kết quả đúng, ta thực hiện cộng theo 16-bit:

$$\begin{array}{r} 209 = 0000\ 0000\ 1101\ 0001 \\ + 73 = + 0000\ 0000\ 0100\ 1001 \\ \hline 0000\ 0001\ 0001\ 1010 \end{array} = 256+16+8+2 = 282$$

2. Phép đảo dấu

- Ta có:

$$\begin{array}{rcl} + 37 & = & 0010\ 0101 \\ \text{bù một} & = & 1101\ 1010 \\ & + & \hline & 1 & \\ \text{bù hai} & = & 1101\ 1011 & = -37 \end{array}$$

- Lấy bù hai của số âm:

$$\begin{array}{rcl} - 37 & = & 1101\ 1011 \\ \text{bù một} & = & 0010\ 0100 \\ & + & \hline & 1 & \\ \text{bù hai} & = & 0010\ 0101 & = +37 \end{array}$$

- Kết luận: *Phép đảo dấu số nguyên trong máy tính thực chất là lấy bù hai*

3. Cộng số nguyên có dấu

- Khi cộng hai số nguyên có dấu n-bit, kết quả nhận được là n-bit và *không cần quan tâm đến bit C_{out}*
 - Khi cộng hai số khác dấu thì **kết quả luôn luôn đúng**
 - Khi cộng hai số cùng dấu, nếu dấu kết quả cùng dấu với các số hạng thì **kết quả là đúng**
 - Khi cộng hai số cùng dấu, nếu kết quả có dấu ngược lại, khi đó có **tràn (Overflow)** xảy ra và **kết quả bị sai**
- Hiện tượng **tràn** xảy ra khi tổng nằm ngoài dải biểu diễn: [$-(2^{n-1})$, $+(2^{n-1}-1)$]

Ví dụ cộng số nguyên có dấu không tràn

- $(+ 70) = 0100\ 0110$
- $+ \underline{(+ 42)} = \underline{\textcolor{red}{0010}\ 1010}$
- $+ 112 = \textcolor{red}{0111}\ 0000 = +112$

- $(+ 97) = 0110\ 0001$
- $+ \underline{(- 52)} = \underline{\textcolor{blue}{1100}\ 1100} \quad (+52=0011\ 0100)$
- $+ 45 = \textcolor{teal}{1} \textcolor{red}{0010}\ 1101 = +45$

- $(- 90) = 1010\ 0110 \quad (+90=0101\ 1010)$
- $+ \underline{(+36)} = \underline{\textcolor{red}{0010}\ 0100}$
- $- 54 = \textcolor{blue}{1100}\ 1010 = - 54$

- $(- 74) = 1011\ 0110 \quad (+74=0100\ 1010)$
- $+ \underline{(- 30)} = \underline{\textcolor{blue}{1110}\ 0010} \quad (+30=0001\ 1110)$
- $-104 = \textcolor{teal}{1} \textcolor{blue}{1001}\ 1000 = -104$

Ví dụ cộng số nguyên có dấu bị tràn

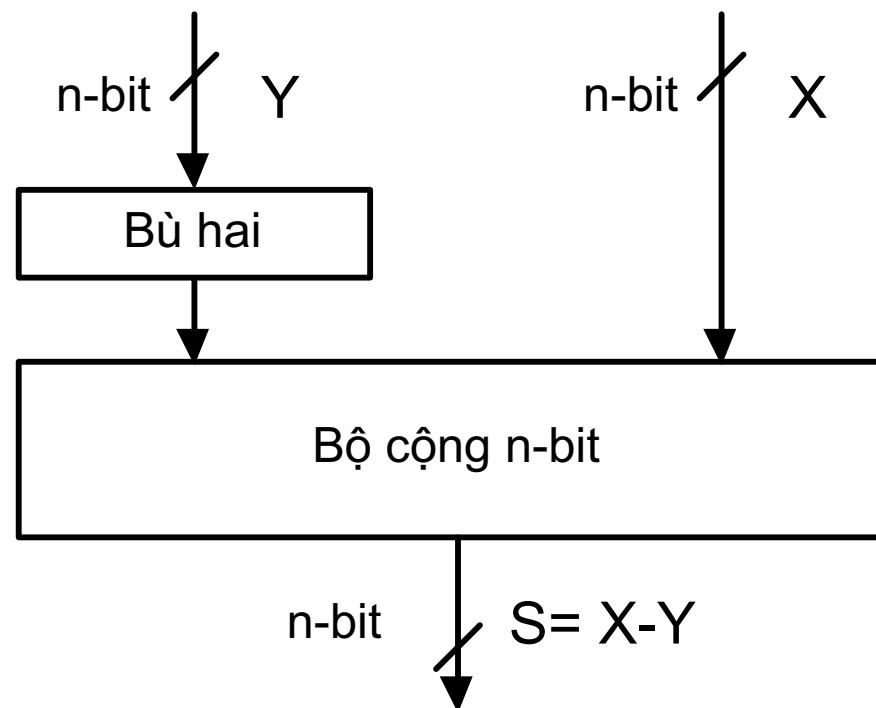
- $(+ 75) = 0100\ 1011$
- $+ (\underline{+ 82}) = \underline{0101\ 0010}$
- $+ 157 \quad \quad \quad 1001\ 1101$
- $= -128 + 16 + 8 + 4 + 1 = -99 \rightarrow \text{sai}$

- $(- 104) = 1001\ 1000 \quad (+104=0110\ 1000)$
- $+ (-\underline{43}) = \underline{1101\ 0101} \quad (+43=0010\ 1011)$
- $- 147 \quad \quad \quad 1\ 0110\ 1101$
- $= 64 + 32 + 8 + 4 + 1 = +109 \rightarrow \text{sai}$

- Cả hai ví dụ đều **tràn** vì tổng nằm ngoài dải biểu diễn $[-128, +127]$

4. Nguyên tắc thực hiện phép trừ

- Phép trừ hai số nguyên: $X-Y = X+(-Y)$
- Nguyên tắc: Lấy bù hai của Y để được $-Y$, rồi cộng với X



3.3. Phép nhân và phép chia số nguyên

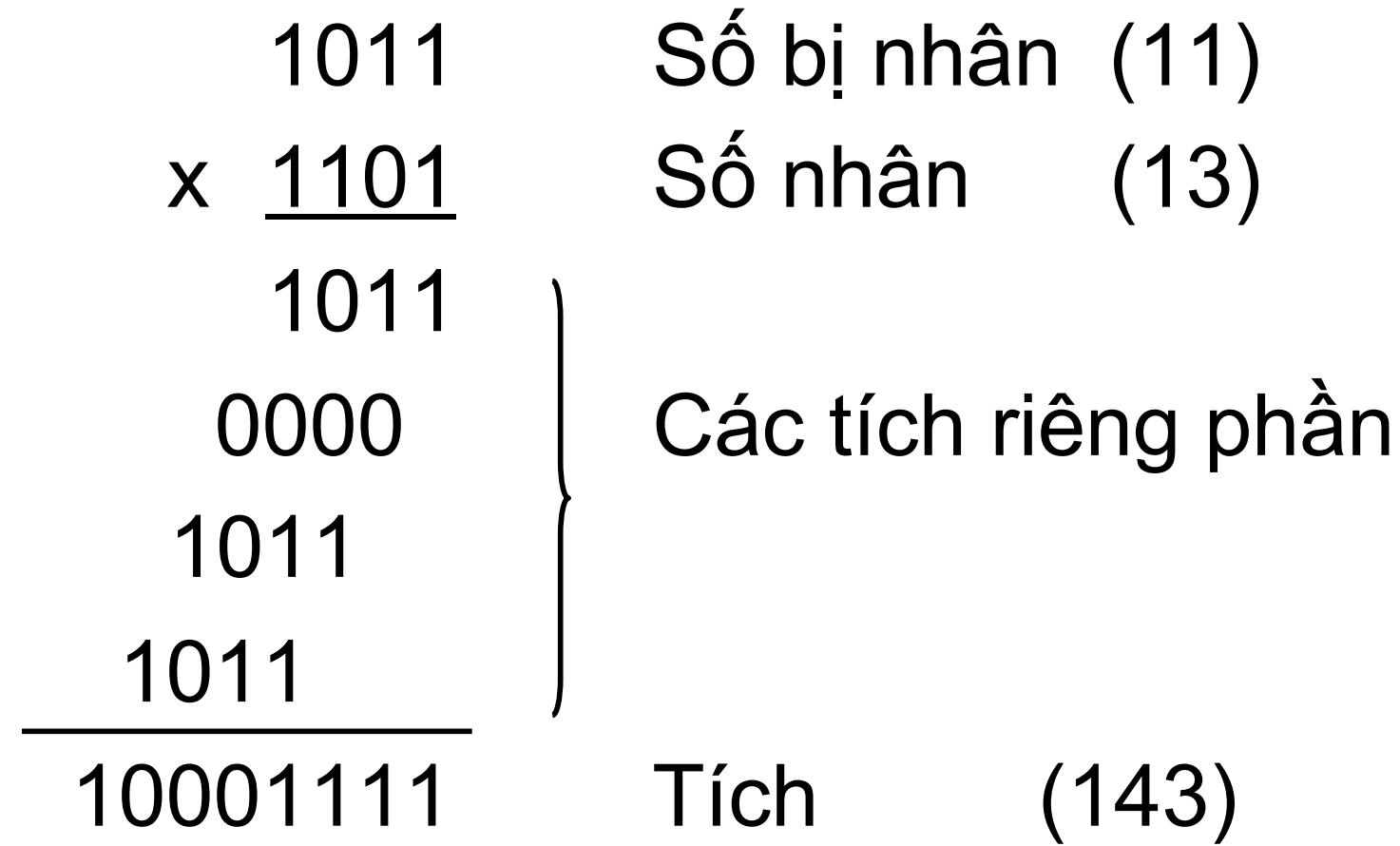
1. Nhân số nguyên không dấu

$$\begin{array}{r} 1011 \\ \times \underline{1101} \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

Số bị nhân (11) Số nhân (13)

Các tích riêng phần

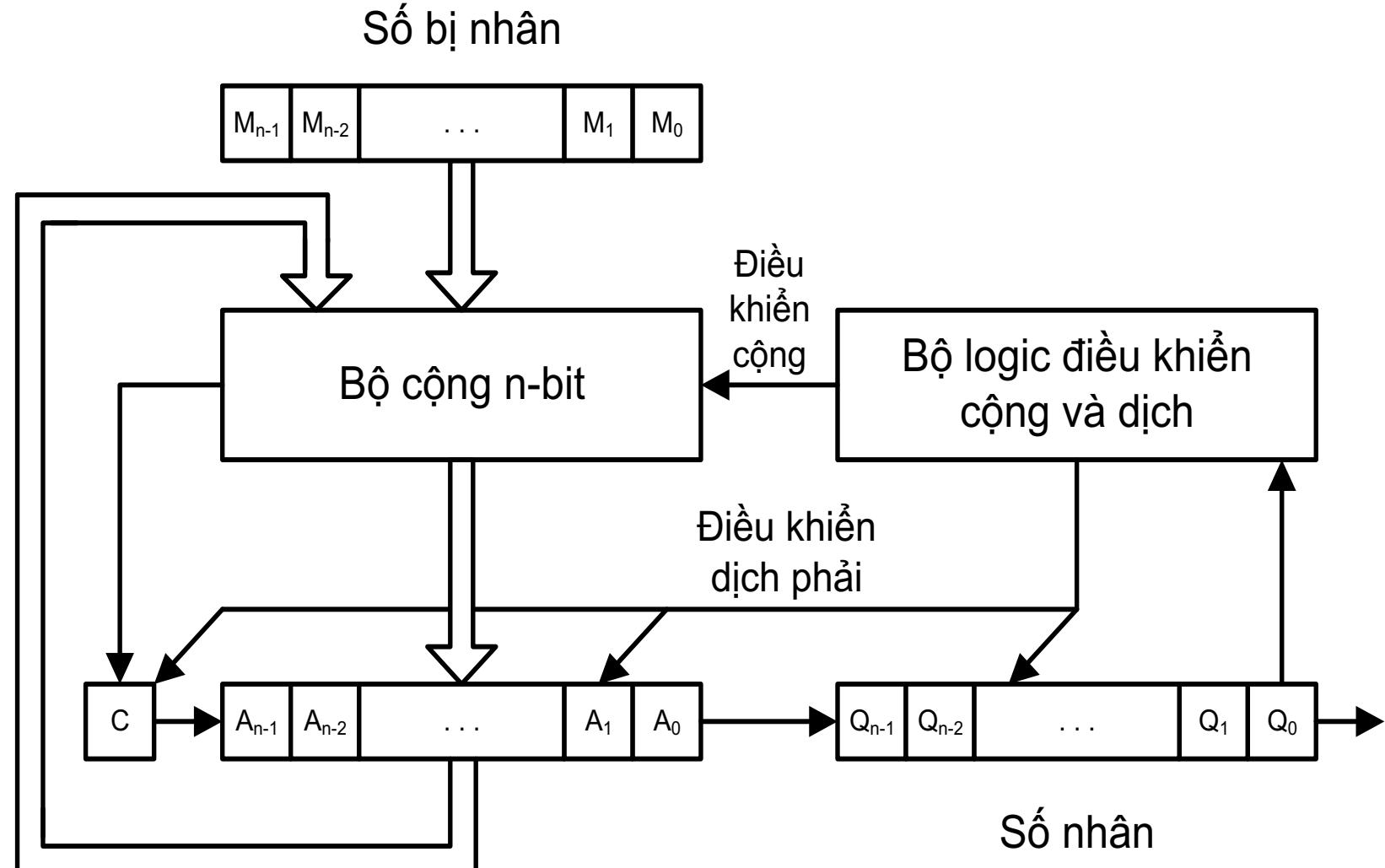
Tích (143)



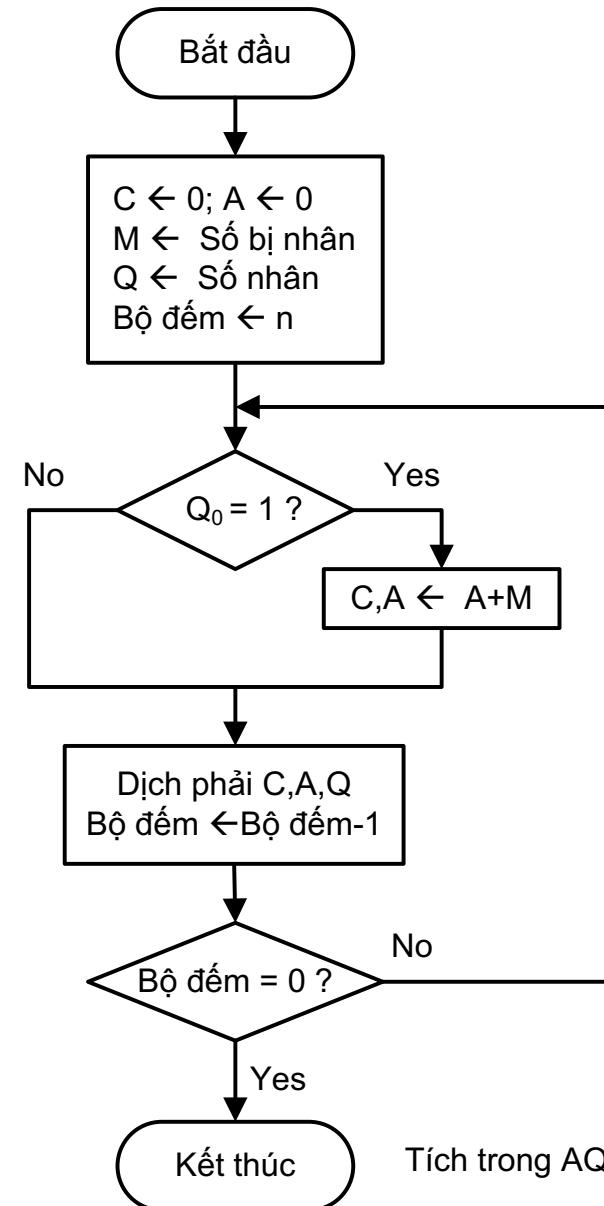
Nhân số nguyên không dấu (tiếp)

- Các **tích riêng phần** được xác định như sau:
 - Nếu bit của số nhân bằng 0 → tích riêng phần bằng 0
 - Nếu bit của số nhân bằng 1 → tích riêng phần bằng số bị nhân
 - Tích riêng phần tiếp theo được dịch trái một bit so với tích riêng phần trước đó
- **Tích bằng tổng các tích riêng phần**
- **Nhân hai số nguyên n-bit, tích có độ dài $2n$ bit (không bao giờ tràn)**

Bộ nhân số nguyên không dấu



Lưu đồ nhân số nguyên không dấu



Ví dụ nhân số nguyên không dấu

- Số bị nhân $M = 1011 \quad (11)$
- Số nhân $Q = 1101 \quad (13)$
- Tích $= 1000\ 1111 \quad (143)$

	C	A	Q	
■	0	0000	1101	Các giá trị khởi đầu
		+ 1011		
■	0	1011	1101	$A \leftarrow A + M$
■	0	0101	1110	Dịch phải
■	0	0010	1111	Dịch phải
		+ 1011		
■	0	1101	1111	$A \leftarrow A + M$
■	0	0110	1111	Dịch phải
		+ 1011		
■	1	0001	1111	$A \leftarrow A + M$
■	0	1000	1111	Dịch phải

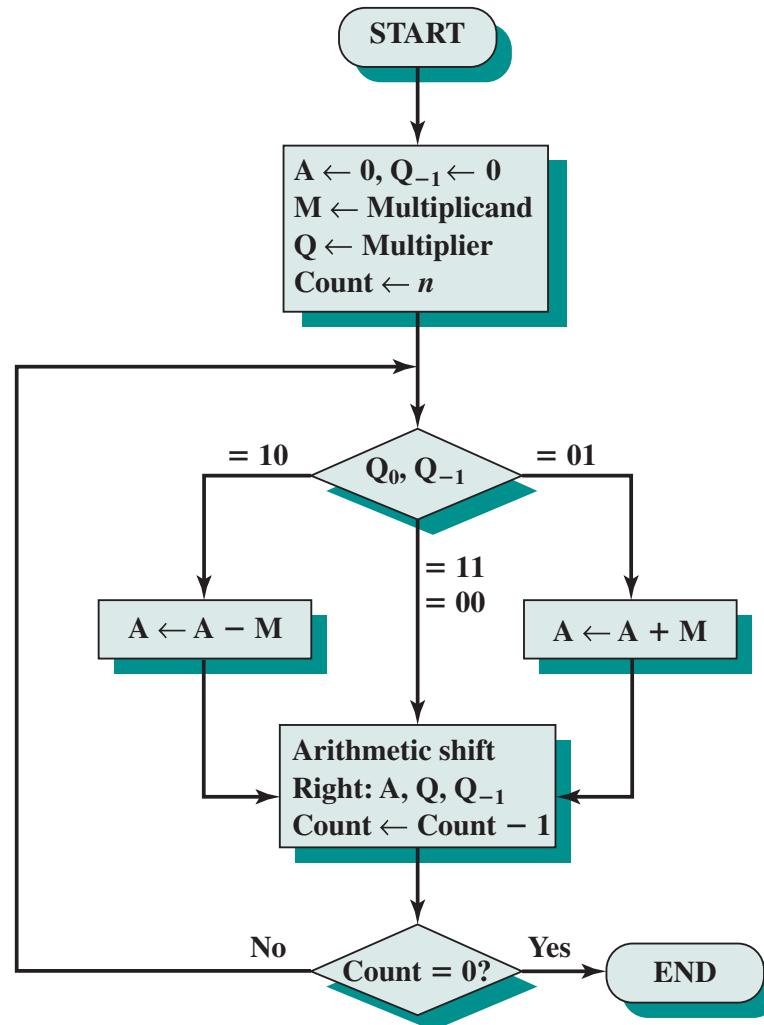
2. Nhân số nguyên có dấu

- Sử dụng thuật giải nhân không dấu
- Sử dụng thuật giải Booth

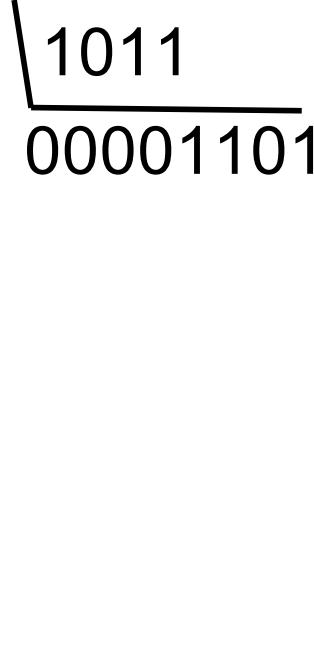
Sử dụng thuật giải nhân không dấu

- Bước 1. Chuyển đổi số bị nhân và số nhân thành số dương tương ứng
- Bước 2. Nhân hai số dương bằng thuật giải nhân số nguyên không dấu, được tích của hai số dương.
- Bước 3. Hiệu chỉnh dấu của tích:
 - Nếu hai thừa số ban đầu cùng dấu thì giữ nguyên kết quả ở bước 2
 - Nếu hai thừa số ban đầu là khác dấu thì đảo dấu kết quả của bước 2 (lấy bù hai)

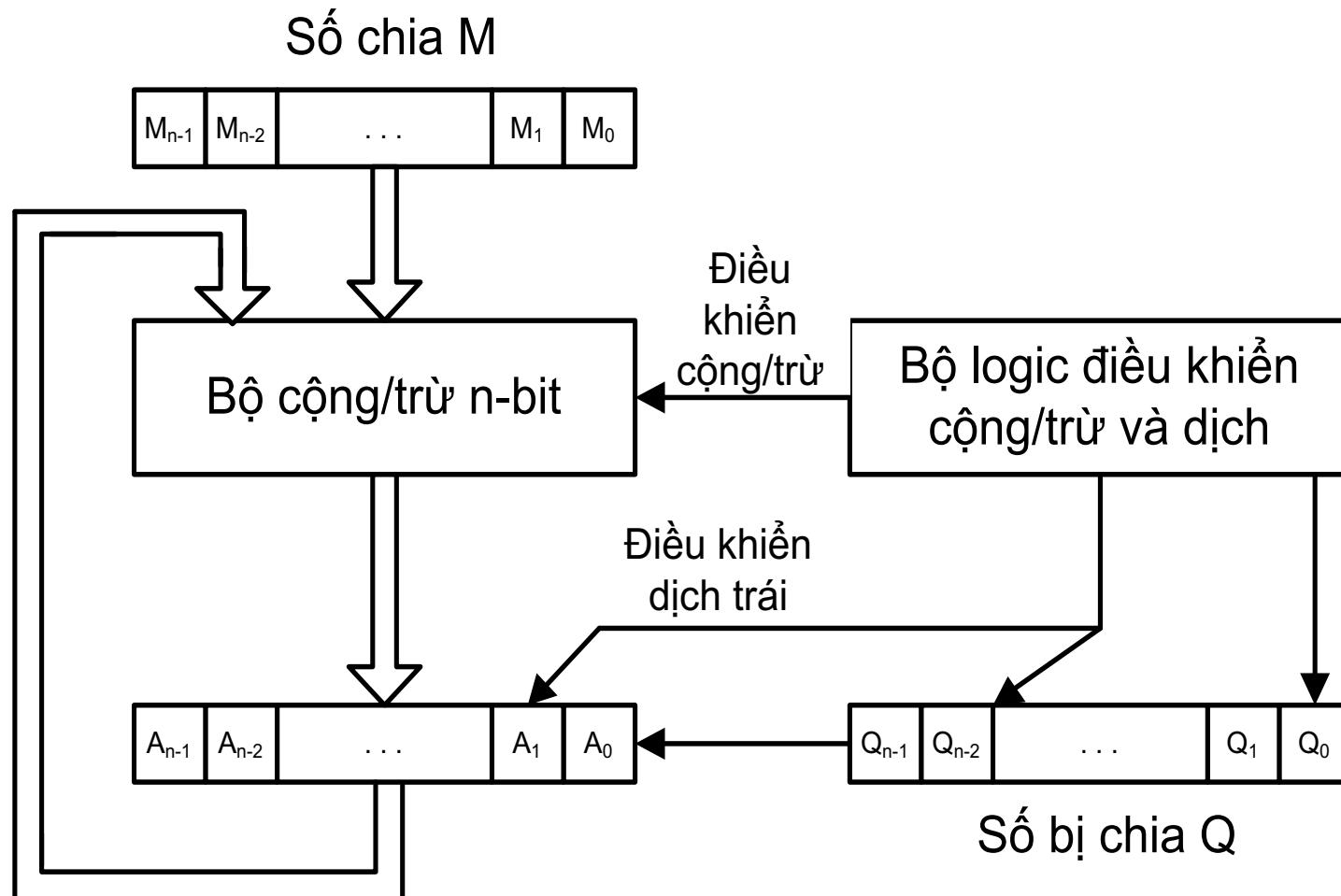
Thuật giải Booth (tham khảo sách COA)



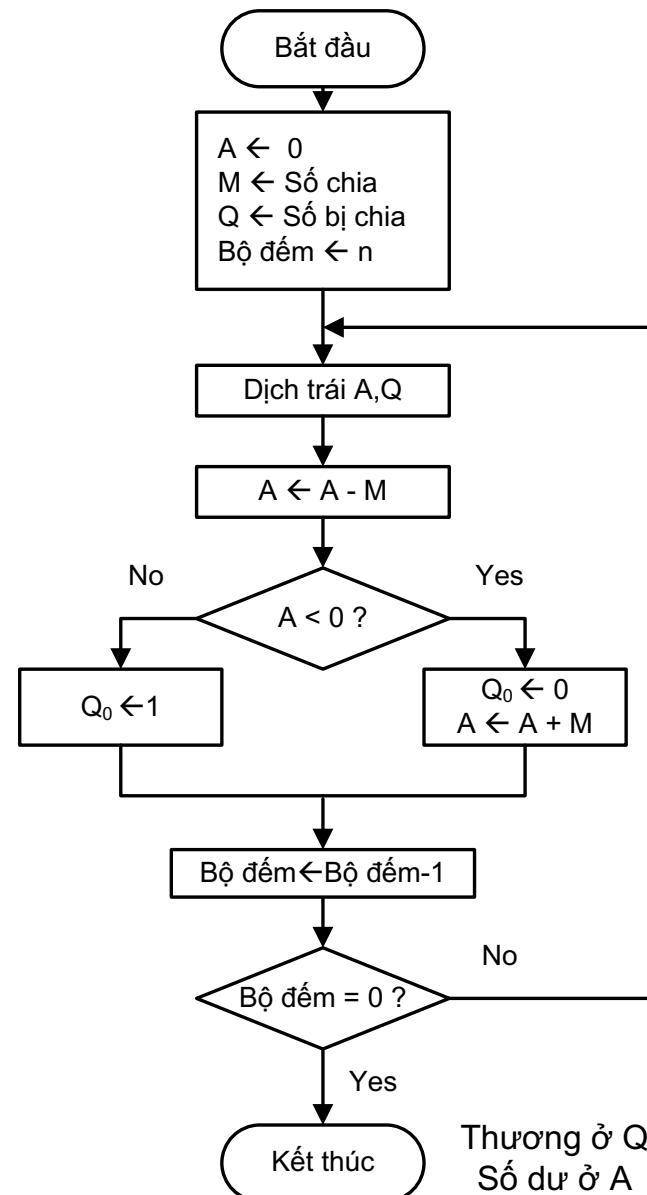
3. Chia số nguyên không dấu

Số bị chia	10010011		Số chia
	- <u>1011</u>	00001101	Thương
	001110		
	- <u>1011</u>		
	001111		
	- <u>1011</u>		
	100		Phần dư

Bộ chia số nguyên không dấu



Lưu đồ chia số nguyên không dấu



4. Chia số nguyên có dấu

- Bước 1. Chuyển đổi số bị chia và số chia về thành số dương tương ứng.
- Bước 2. Sử dụng thuật giải chia số nguyên không dấu để chia hai số dương, kết quả nhận được là thương Q và phần dư R đều là dương
- Bước 3. Hiệu chỉnh dấu của kết quả như sau:
(Lưu ý: phép đảo dấu thực chất là thực hiện phép lấy bù hai)

Số bị chia	Số chia	Thương	Số dư
dương	dương	giữ nguyên	giữ nguyên
dương	âm	đảo dấu	giữ nguyên
âm	dương	đảo dấu	đảo dấu
âm	âm	giữ nguyên	đảo dấu

3.4. Số dấu phẩy động

1. Nguyên tắc chung

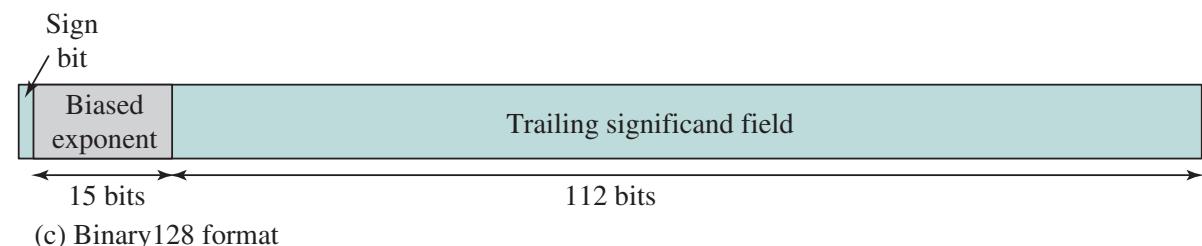
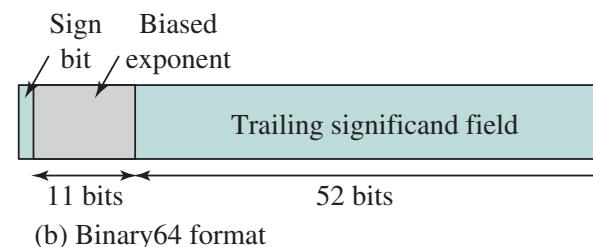
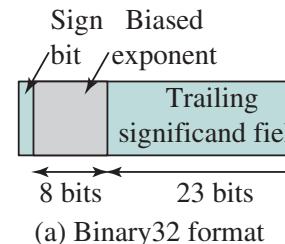
- Floating Point Number → biểu diễn cho số thực
- Tổng quát: một số thực X được biểu diễn theo kiểu số dấu phẩy động như sau:

$$X = \pm M * R^E$$

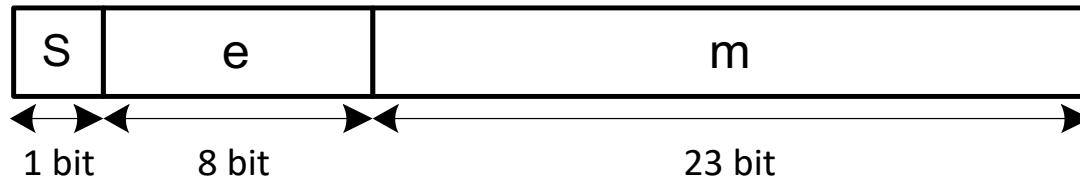
- M là phần định trị (Mantissa),
- R là cơ số (Radix),
- E là phần mũ (Exponent).

2. Chuẩn IEEE754-2008

- Cơ số $R = 2$
- Các dạng:
 - Dạng 32-bit
 - Dạng 64-bit
 - Dạng 128-bit



Dạng 32-bit



- S là bit dấu:
 - $S = 0 \rightarrow$ số dương
 - $S = 1 \rightarrow$ số âm
- e (8 bit) là giá trị dịch chuyển của phần mũ E:
 - $e = E + 127 \rightarrow$ phần mũ $E = e - 127$
- m (23 bit) là phần lẻ của phần định trị M:
 - $M = 1.m$
- Công thức xác định giá trị của số thực:

$$X = (-1)^S * 1.m * 2^{e-127}$$

Ví dụ 1

Xác định giá trị của các số thực được biểu diễn bằng 32-bit sau đây:

- **1100 0001 0101 0110 0000 0000 0000 0000**

- S = 1 → số âm

- e = **1000 0010**₍₂₎ = 130₍₁₀₎ → E = 130 - 127 = 3

Vậy

$$X = -1.10101100_{(2)} * 2^3 = -1101.011_{(2)} = -13.375_{(10)}$$

- **0011 1111 1000 0000 0000 0000 0000 0000** = ?

Ví dụ 2

Biểu diễn số thực $X = 83.75_{(10)}$ về dạng số dấu phẩy động IEEE754 32-bit

Giải:

- $X = 83.75_{(10)} = 1010011.11_{(2)} = 1.0100111 \times 2^6$
- Ta có:
 - $S = 0$ vì đây là số dương
 - $E = e - 127 = 6 \rightarrow e = 127 + 6 = 133_{(10)} = 1000\ 0101_{(2)}$
- Vậy:
$$X = 0100\ 0010\ 1010\ 0111\ 1000\ 0000\ 0000\ 0000$$

Các qui ước đặc biệt

- Các bit của e bằng 0, các bit của m bằng 0, thì $X = \pm 0$
 $x000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \rightarrow X = \pm 0$
- Các bit của e bằng 1, các bit của m bằng 0, thì $X = \pm \infty$
 $x111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000\ 0000 \rightarrow X = \pm \infty$
- Các bit của e bằng 1, còn m có ít nhất một bit bằng 1, thì nó không biểu diễn cho số nào cả (NaN - not a number)

Dải giá trị biểu diễn

- 2^{-127} đến 2^{+127}
- 10^{-38} đến 10^{+38}



Dạng 64-bit

- S là bit dấu
- e (11 bit) là giá trị dịch chuyển của phần mũ E:
 - $e = E + 1023 \rightarrow$ phần mũ E = $e - 1023$
- m (52 bit): phần lẻ của phần định trị M
- Giá trị số thực:
$$X = (-1)^S * 1.m * 2^{e-1023}$$
- Dải giá trị biểu diễn: 10^{-308} đến 10^{+308}

Dạng 128-bit

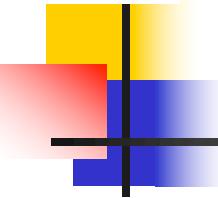
- S là bit dấu
- e (15 bit) là giá trị dịch chuyển của phần mũ E:
 - $e = E + 16383 \rightarrow \text{phần mũ } E = e - 16383$
- m (112 bit): phần lẻ của phần định trị M
- Giá trị số thực:
$$X = (-1)^S * 1.m * 2^{e-16383}$$
- Dải giá trị biểu diễn: 10^{-4932} đến 10^{+4932}

3. Thực hiện phép toán số dấu phẩy động

- $X_1 = M_1 * R^{E_1}$
- $X_2 = M_2 * R^{E_2}$
- Ta có
 - $X_1 * X_2 = (M_1 * M_2) * R^{E_1+E_2}$
 - $X_1 / X_2 = (M_1 / M_2) * R^{E_1-E_2}$
 - $X_1 \pm X_2 = (M_1 * R^{E_1-E_2} \pm M_2) * R^{E_2}$, với $E_2 \geq E_1$

Các khả năng tràn số

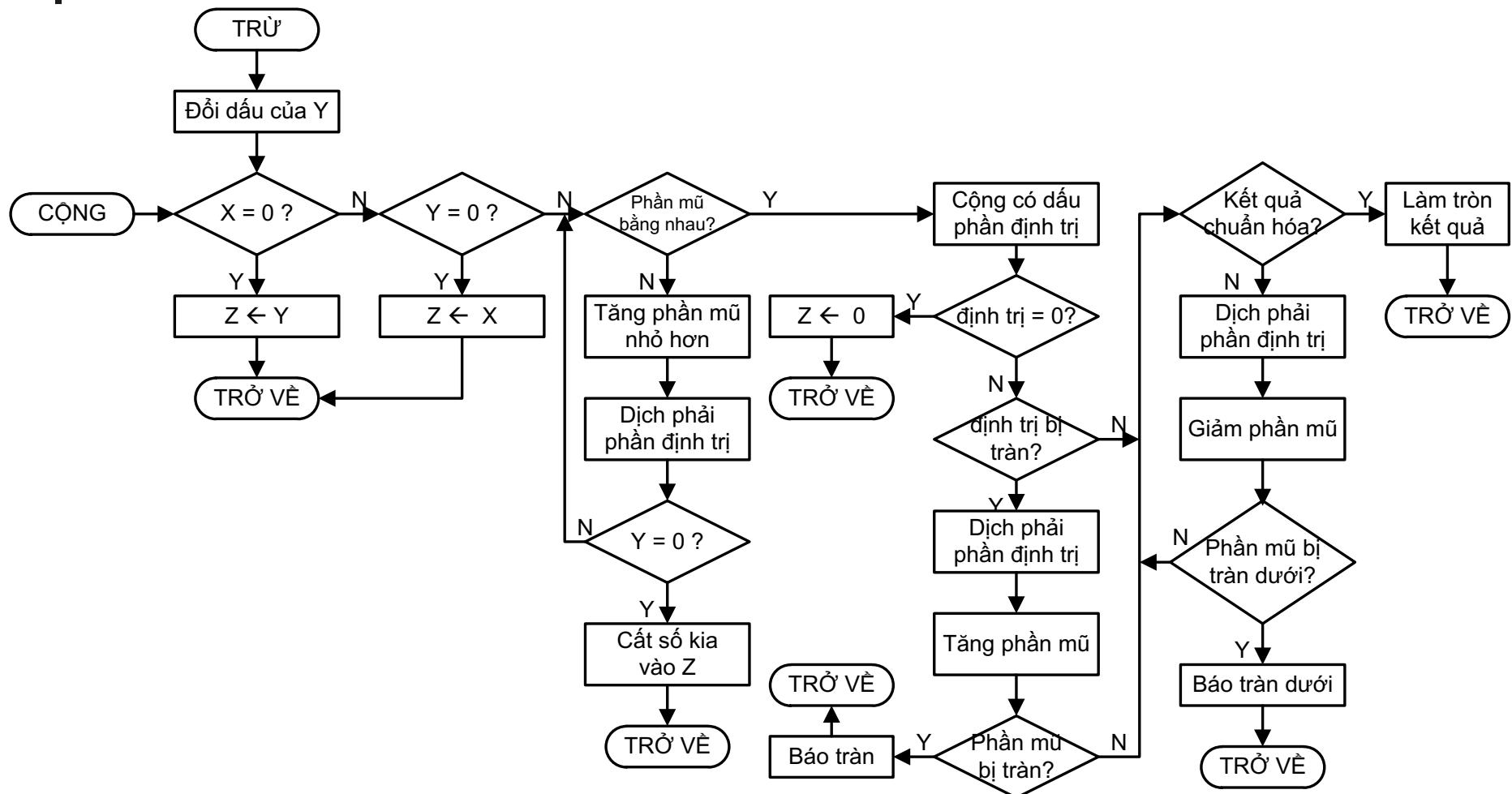
- Tràn trên số mũ (Exponent Overflow): mũ dương vượt ra khỏi giá trị cực đại của số mũ dương có thể ($\rightarrow \infty$)
- Tràn dưới số mũ (Exponent Underflow): mũ âm vượt ra khỏi giá trị cực đại của số mũ âm có thể ($\rightarrow 0$)
- Tràn trên phần định trị (Mantissa Overflow): cộng hai phần định trị có cùng dấu, kết quả bị nhớ ra ngoài bit cao nhất
- Tràn dưới phần định trị (Mantissa Underflow): Khi hiệu chỉnh phần định trị, các số bị mất ở bên phải phần định trị



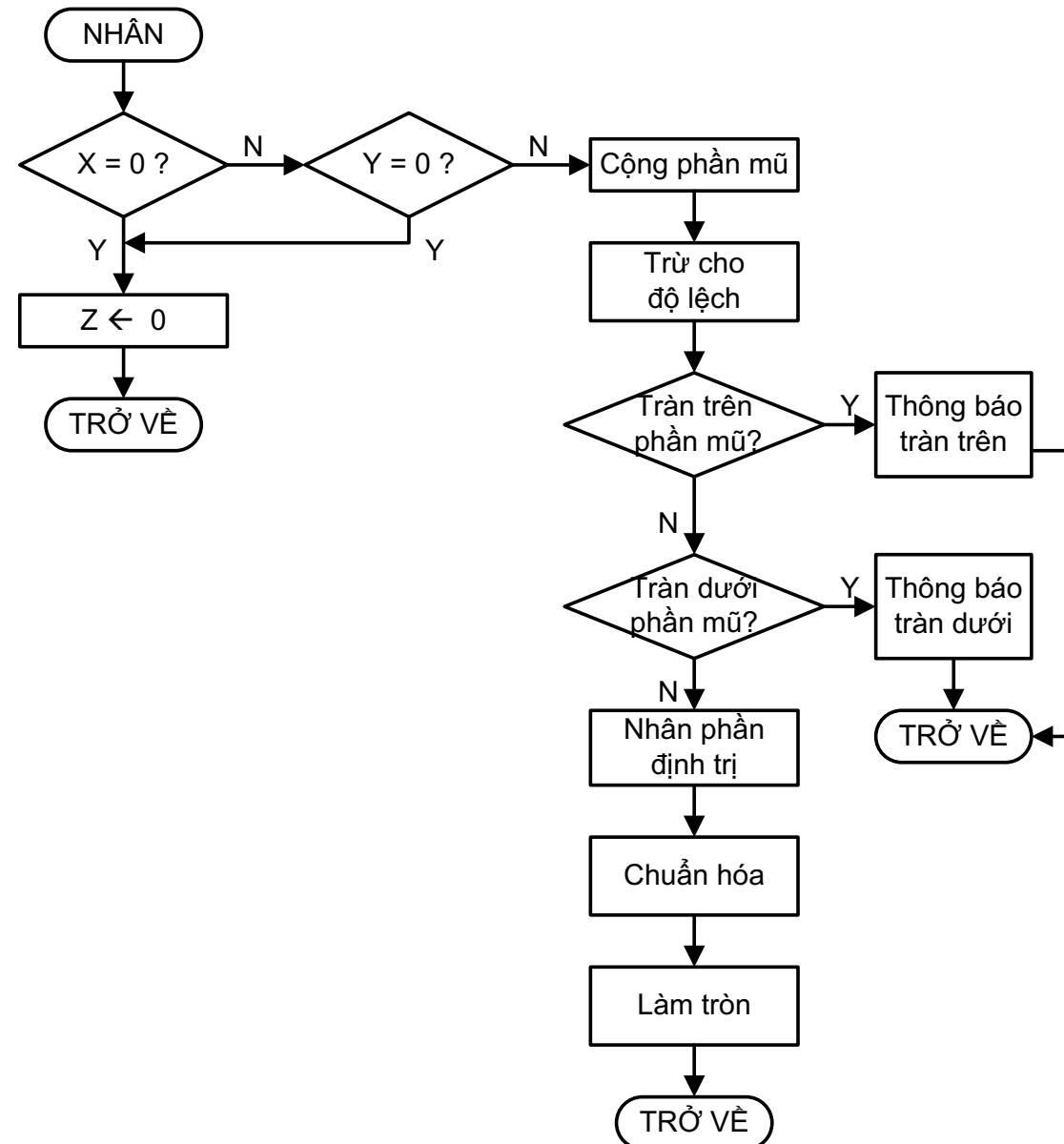
Phép cộng và phép trừ

- Kiểm tra các số hạng có bằng 0 hay không
- Hiệu chỉnh phần định trị
- Cộng hoặc trừ phần định trị
- Chuẩn hoá kết quả

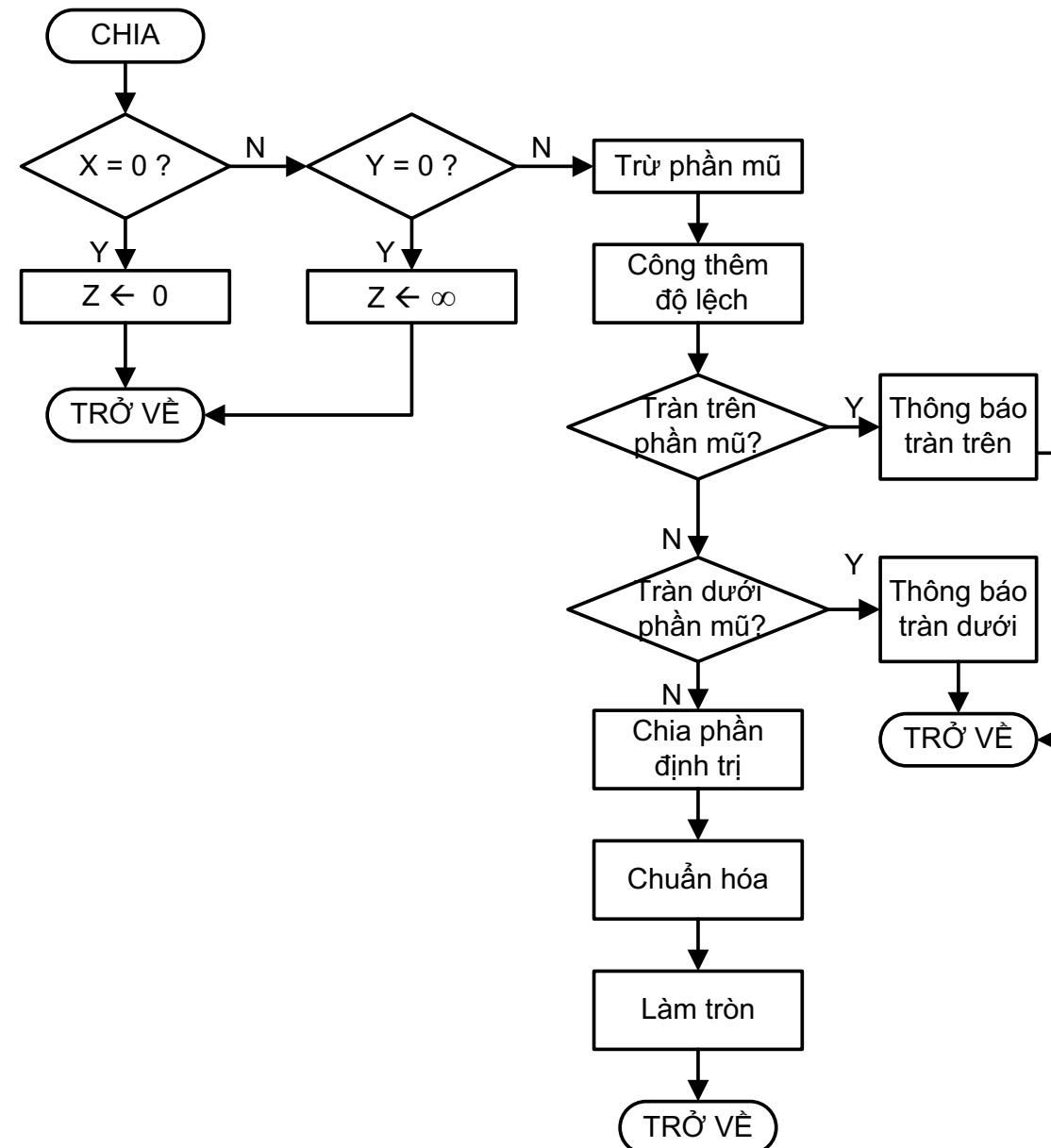
Thuật toán cộng/trừ số dấu phẩy động

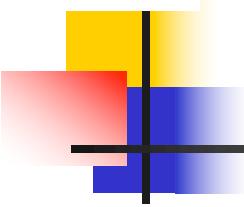


Thuật toán nhân số dấu phẩy động

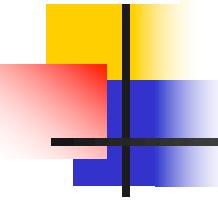


Thuật toán chia số dấu phẩy động





Hết chương 3



Kiến trúc máy tính

Chương 4 BỘ XỬ LÝ

Nguyễn Kim Khánh
Trường Đại học Bách khoa Hà Nội

Nội dung

- 4.1. Thực hiện bộ xử lý MIPS cơ bản
- 4.2. Thiết kế Datapath
- 4.3. Thiết kế Control Unit
- 4.4. Kỹ thuật đường ống lệnh

4.1. Thực hiện bộ xử lý MIPS cơ bản

- Xem xét hai cách thực hiện bộ xử lý theo kiến trúc MIPS:
 - Phiên bản đơn giản
 - Phiên bản được đường ống hóa (gần với thực tế)
- Chỉ thực hiện với một số lệnh cơ bản của MIPS, nhưng chỉ ra hầu hết các khía cạnh:
 - Các lệnh tham chiếu bộ nhớ: lw, sw
 - Các lệnh số học/logic: add, sub, and, or,slt
 - Các lệnh chuyển điều khiển: beq, j

Tổng quan quá trình thực hiện các lệnh

■ Hai bước đầu tiên với mỗi lệnh:

- Đưa địa chỉ từ bộ đếm chương trình **PC** đến bộ nhớ lệnh, tìm và nhận lệnh từ bộ nhớ này
- Sử dụng các số hiệu thanh ghi trong lệnh để chọn và đọc một hoặc hai thanh ghi:
 - Lệnh **Iw**: đọc 1 thanh ghi
 - Các lệnh khác (không kể lệnh jump): đọc 2 thanh ghi

Tổng quan quá trình thực hiện các lệnh (tiếp)

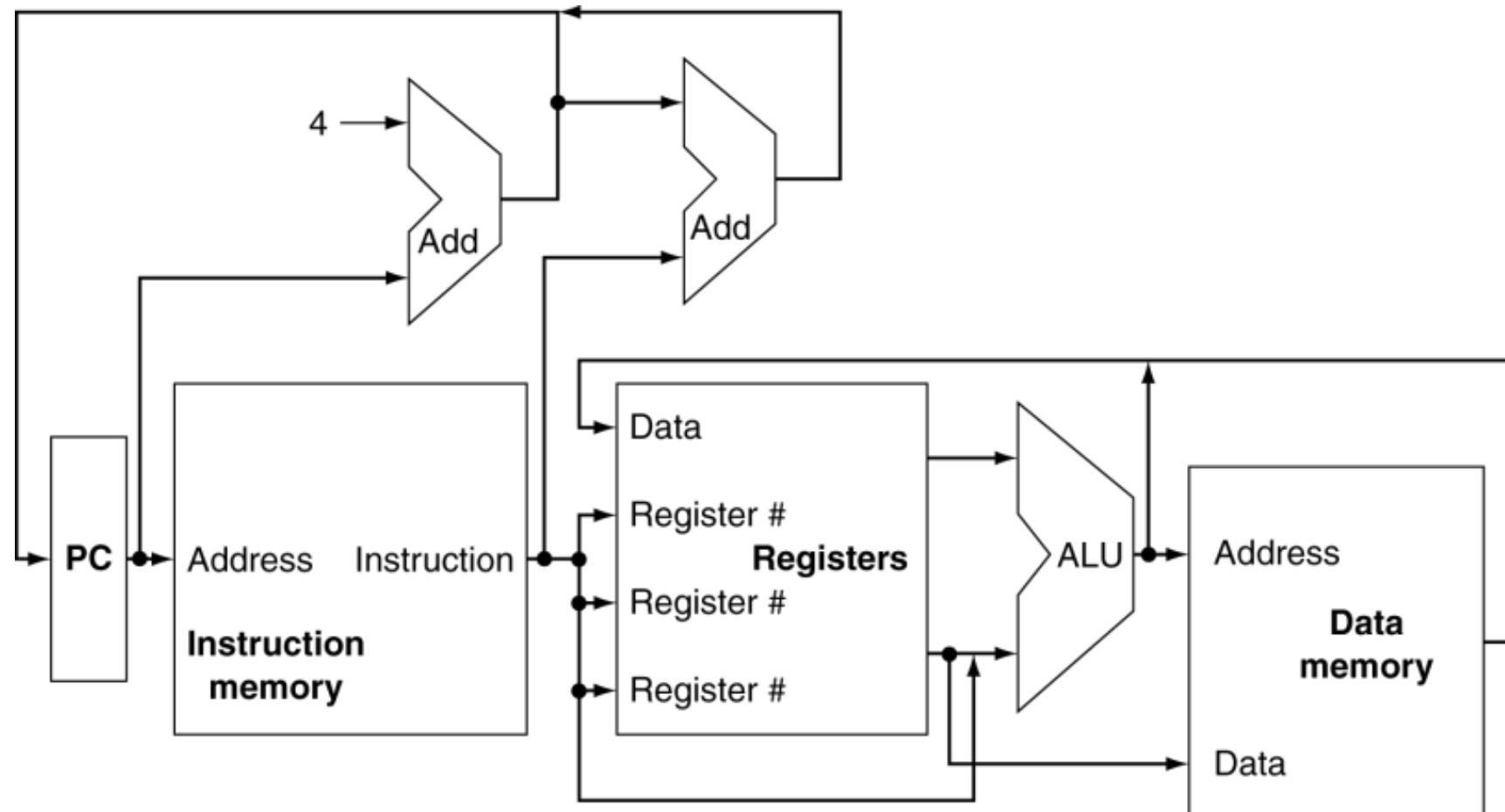
Các bước tiếp theo tùy thuộc vào loại lệnh:

- Sử dụng ALU hoặc bộ cộng Add để:
 - Tính kết quả phép toán với các lệnh số học/logic
 - So sánh các toán hạng với lệnh branch
 - Tính địa chỉ đích với các lệnh branch
 - Tính địa chỉ ngăn nhớ dữ liệu với lệnh load/store
- Truy cập bộ nhớ dữ liệu với lệnh load/store
 - Lệnh lw: đọc dữ liệu từ bộ nhớ
 - Lệnh sw: ghi dữ liệu ra bộ nhớ
- Ghi dữ liệu đến thanh ghi đích:
 - Các lệnh số học/logic: kết quả phép toán
 - Lệnh lw: dữ liệu được đọc từ bộ nhớ dữ liệu

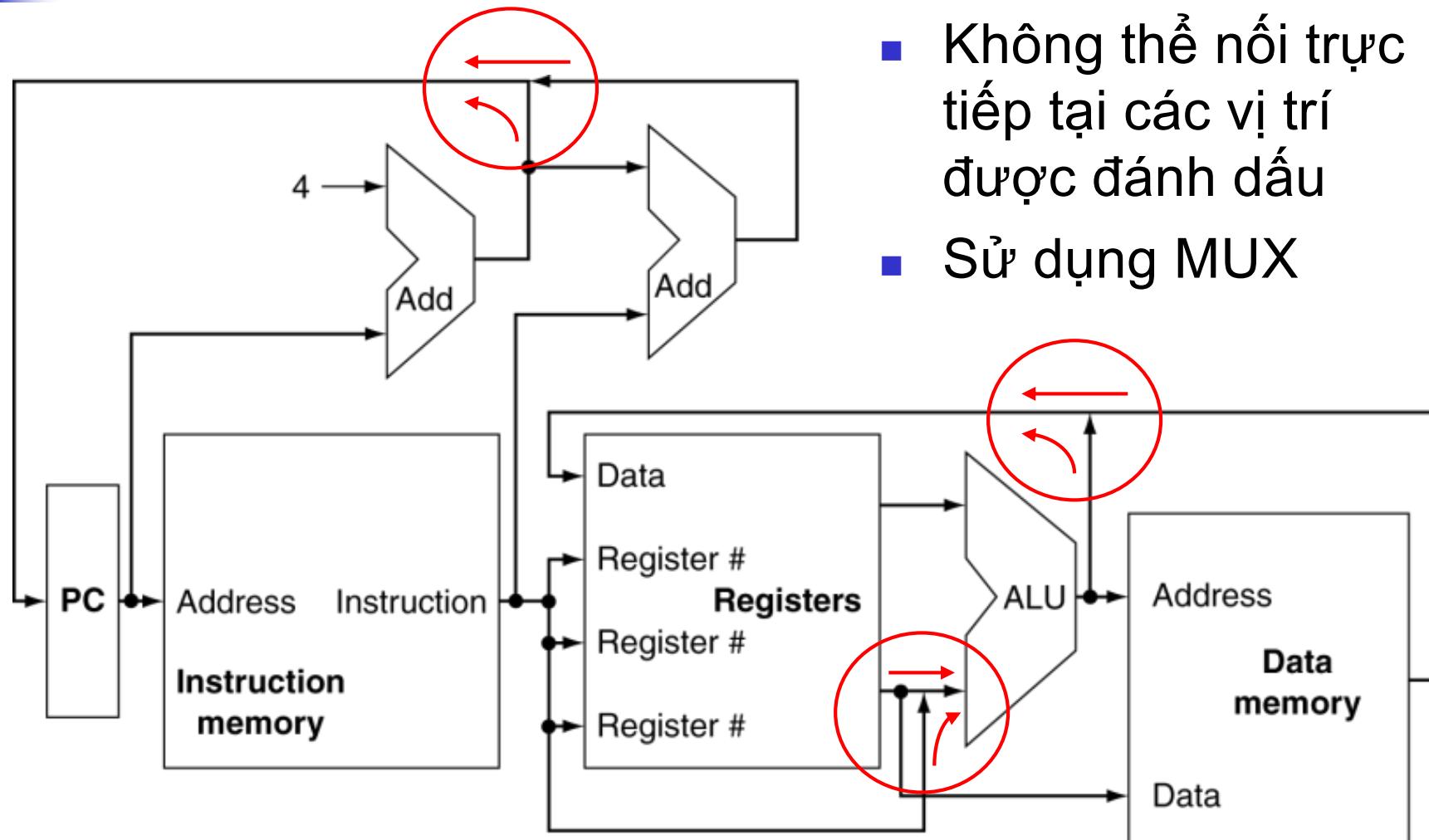
Tổng quan quá trình thực hiện các lệnh (tiếp)

- Thay đổi nội dung bộ đếm chương trình PC:
 - Với các lệnh rẽ nhánh (branch), tùy thuộc vào kết quả so sánh:
 - Điều kiện thỏa mãn: $PC \leftarrow \text{địa chỉ đích}$ (địa chỉ của lệnh cần rẽ tới)
 - Điều kiện không thỏa mãn: $PC \leftarrow PC + 4$ (địa chỉ của lệnh kế tiếp)
 - Với các lệnh còn lại (không kể các lệnh jump)
 - $PC \leftarrow PC + 4$ (địa chỉ của lệnh kế tiếp)

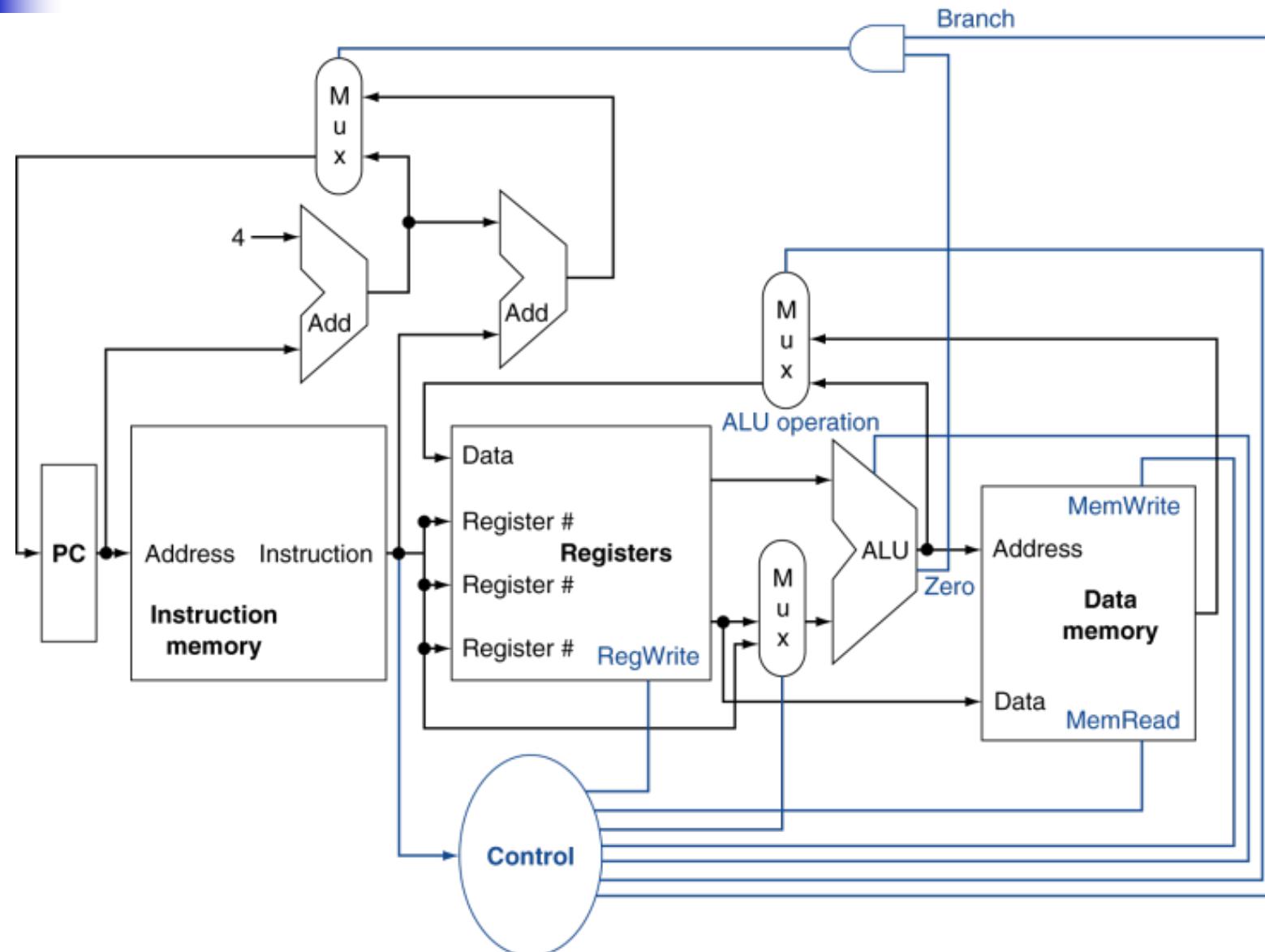
Sơ đồ khái quát của bộ xử lý MIPS



Sử dụng bộ chọn kênh (MUX)



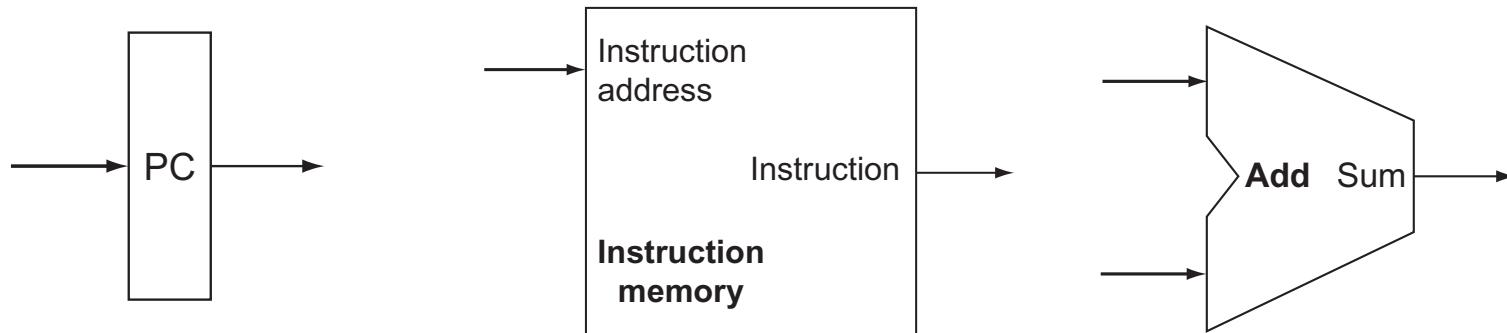
Bộ xử lý với các đường điều khiển chính



4.2. Thiết kế Datapath

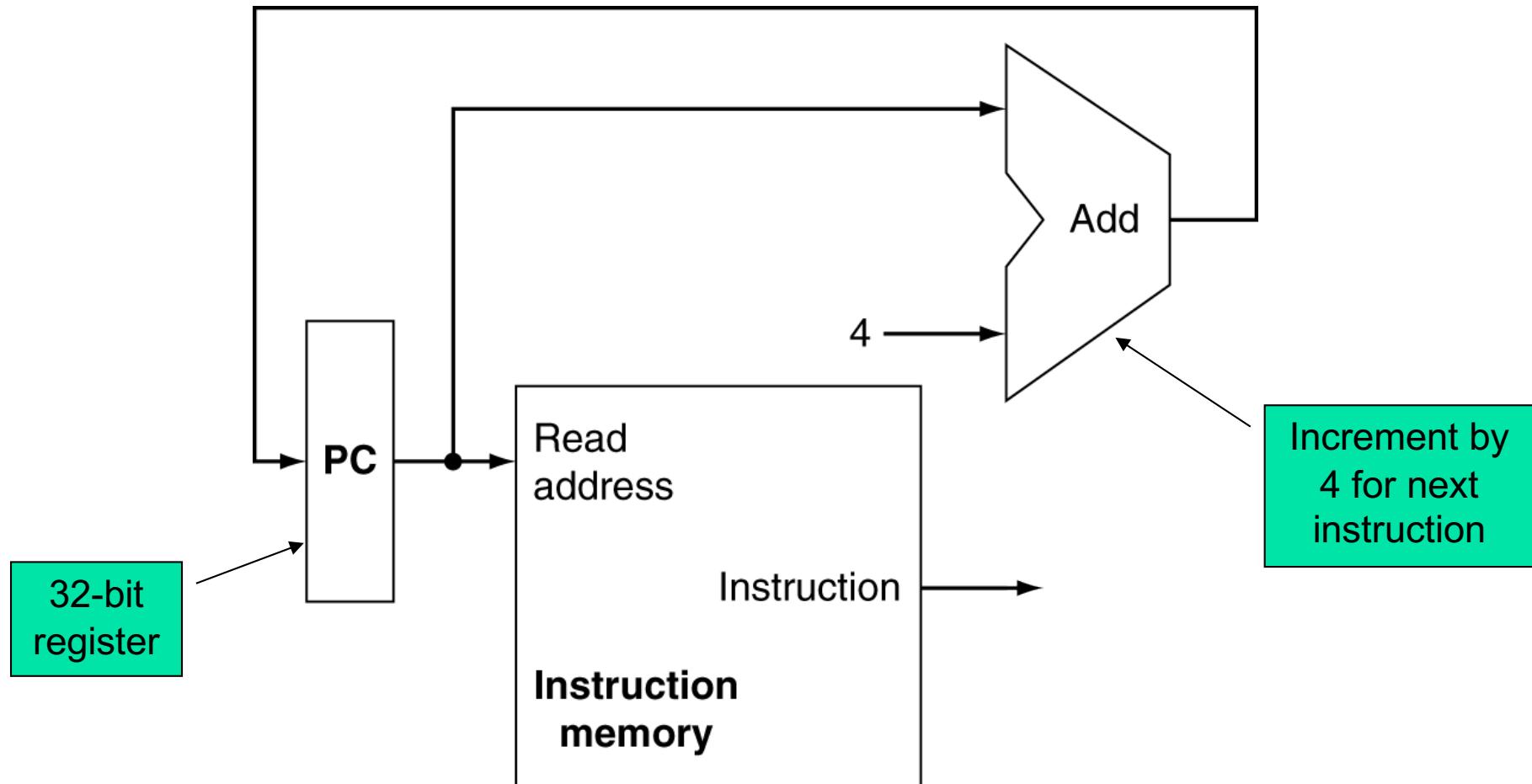
- Datapath: gồm các thành phần để xử lý dữ liệu và địa chỉ
 - Tập thanh ghi, ALUs, MUX's, bộ nhớ, ...
- Sẽ xây dựng tăng dần Datapath cho MIPS

Các thành phần để thực hiện nhận lệnh



- **Bộ đếm chương trình *PC*:**
 - Thanh ghi 32-bit chứa địa chỉ của lệnh hiện tại
 - Địa chỉ khởi động = 0xBFC0 0000
- **Bộ nhớ lệnh (*Instruction memory*):**
 - Chứa các lệnh của chương trình
 - Khi có địa chỉ lệnh từ PC đưa đến thì lệnh được đọc ra
- **Bộ cộng (*Add*):** được sử dụng tăng nội dung PC thêm 4 để trả tới lệnh kế tiếp

Thực hiện phần nhận lệnh

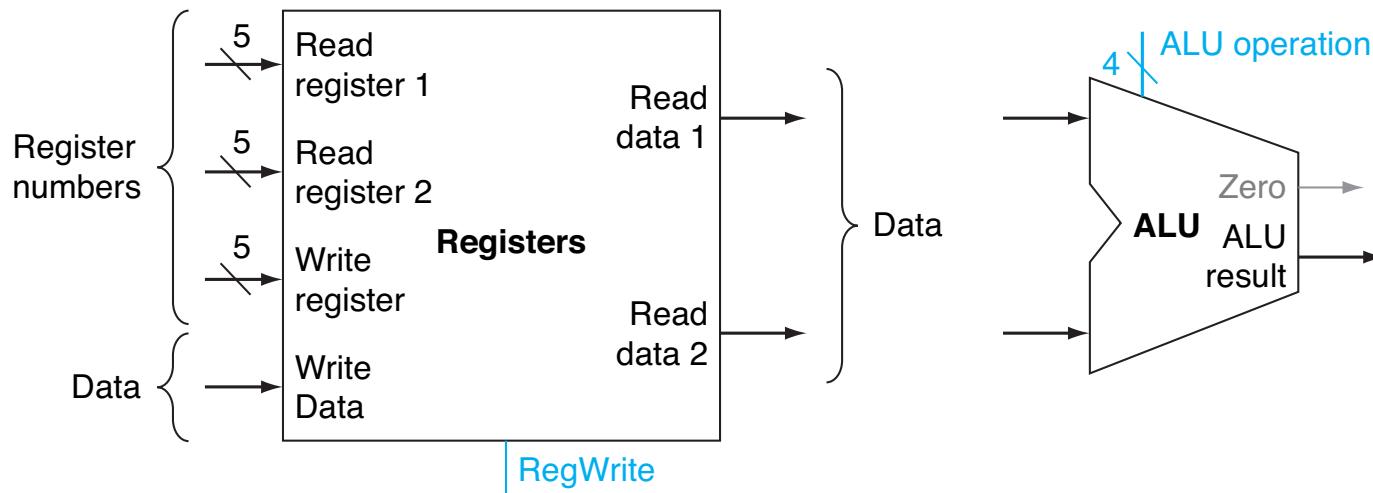


Thực hiện lệnh số học/logic kiểu R

op	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

- bits 31:26: mã thao tác (opcode)
 - 000000 với các lệnh kiểu R
- bits 25:21: số hiệu thanh ghi nguồn thứ nhất rs
- bits 20:16: số hiệu thanh ghi nguồn thứ hai rt
- bits 15:11: số hiệu thanh ghi đích rd
- bits 10:6: số bit được dịch với các lệnh dịch bit
 - 00000 với các lệnh khác
- bits 5:0: mã hàm để xác định phép toán (function code)

Các thành phần thực hiện lệnh kiểu R

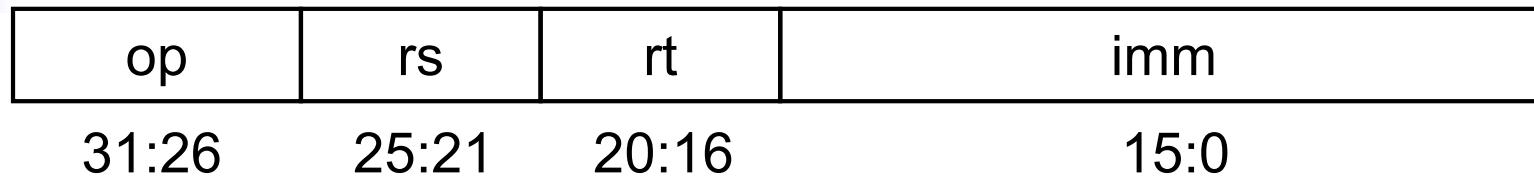


- **Tập thanh ghi (*Registers*):** có 32 thanh ghi 32-bit, mỗi thanh ghi được xác định bởi số hiệu 5-bit
 - *Read register 1, Read register 2:* các đầu vào để chọn các thanh ghi cần đọc
 - *Write register:* đầu vào để chọn thanh ghi cần ghi
 - *Read data 1, Read data 2:* hai đầu ra dữ liệu đọc từ thanh ghi (32-bit)
 - *Write Data:* đầu vào dữ liệu ghi vào thanh ghi (32-bit)
 - *RegWrite:* tín hiệu điều khiển ghi dữ liệu vào thanh ghi
- **ALU** để thực hiện các phép toán số học/logic

Mô tả thực hiện lệnh số học/logic kiểu R

- Hai số hiệu thanh ghi *rs* và *rt* đưa đến hai đầu vào *Read register 1*, *Read register 2* để chọn hai thanh ghi nguồn
- Số hiệu thanh ghi *rd* đưa đến đầu vào *Write register* để chọn thanh ghi đích
- Hai dữ liệu từ hai thanh ghi nguồn được đọc ra 2 đầu ra *Read data 1* và *Read data 2*, rồi đưa đến đầu vào của *ALU*
- *ALU operation* (4-bit): tín hiệu điều khiển chọn phép toán ở *ALU*
- *ALU* thực hiện phép toán tương ứng, kết quả phép toán ở đầu ra *ALU result* được đưa về đầu vào *Write Data* của tập thanh ghi để ghi vào thanh ghi đích
- *RegWrite*: tín hiệu điều khiển ghi dữ liệu vào thanh ghi đích

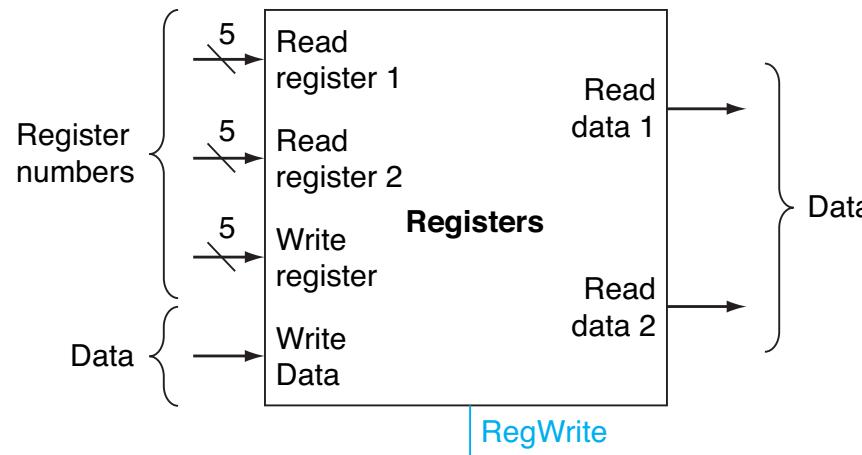
Thực hiện các lệnh lw/sw



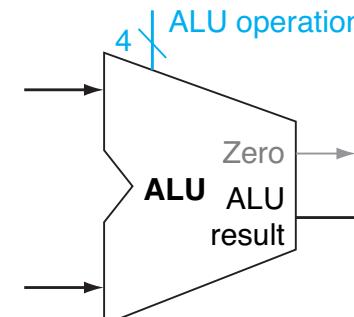
- bits 31:26 là mã thao tác
 - 100011 (35) với lệnh lw
 - 101011 (43) với lệnh sw
- bits 25:21: số hiệu thanh ghi cơ sở rs
- bits 20:16: số hiệu thanh ghi rt
 - thanh ghi đích với lệnh lw
 - thanh ghi nguồn với lệnh sw
- bits 15:0: hằng số imm có giá trị trong dải $[-2^{15}, +2^{15} - 1]$
- Địa chỉ từ nhớ dữ liệu = nội dung thanh ghi rs + imm
- **lw rt, imm(rs) # (rt) = mem[(rs)+SignExtImm]**
- **sw rt, imm(rs) #mem[(rs)+SignExtImm] = (rt)**

Các thành phần thực hiện các lệnh lw/sw

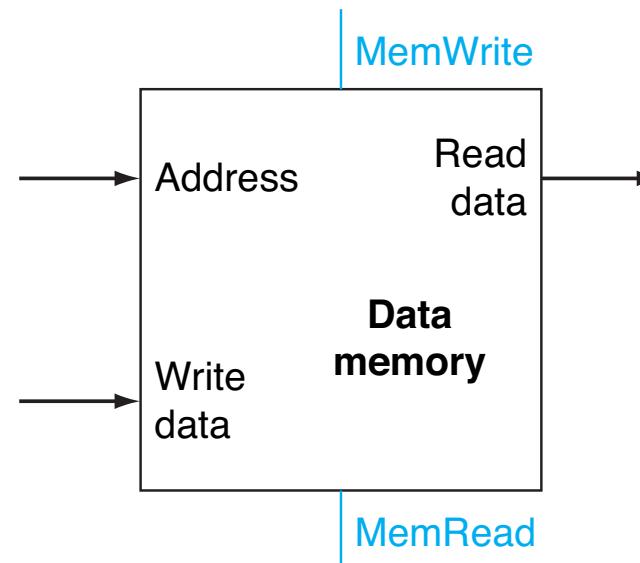
Tập thanh ghi



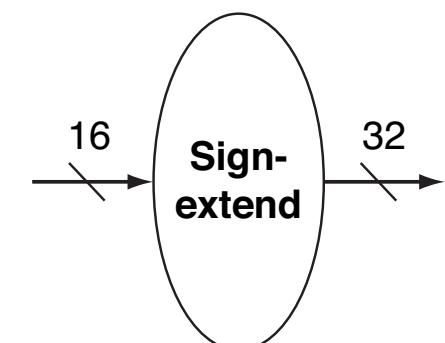
Bộ số học/logic



Bộ nhớ dữ liệu



Bộ mở rộng số có dấu
từ 16-bit → 32-bit



Thực hiện lệnh lw

- Số hiệu thanh ghi **rs** đưa đến đầu vào *Read register 1* để chọn thanh ghi cơ sở **rs**, nội dung **rs** được đưa ra đầu ra *Read Data 1*, rồi chuyển đến *ALU*
- Hàng số imm 16-bit đưa đến bộ *Sign-extend* để mở rộng thành 32-bit, rồi chuyển đến *ALU*
- ALU* cộng hai giá trị trên đưa ra *ALU result* chính là địa chỉ của dữ liệu cần đọc từ bộ nhớ dữ liệu; địa chỉ này được đưa đến đầu vào *Address* của bộ nhớ dữ liệu
- Số hiệu thanh ghi **rt** đưa đến đầu vào *Write Register* để chọn thanh ghi đích
- Dữ liệu 32-bit ở bộ nhớ dữ liệu, tại vị trí địa chỉ đã được tính, được đọc ra ở đầu ra *Read data* của bộ nhớ dữ liệu nhờ tín hiệu điều khiển *MemRead*, rồi đưa về đầu vào *Write Data* của tập thanh ghi để ghi vào thanh ghi đích **rt**

Thực hiện lệnh sw

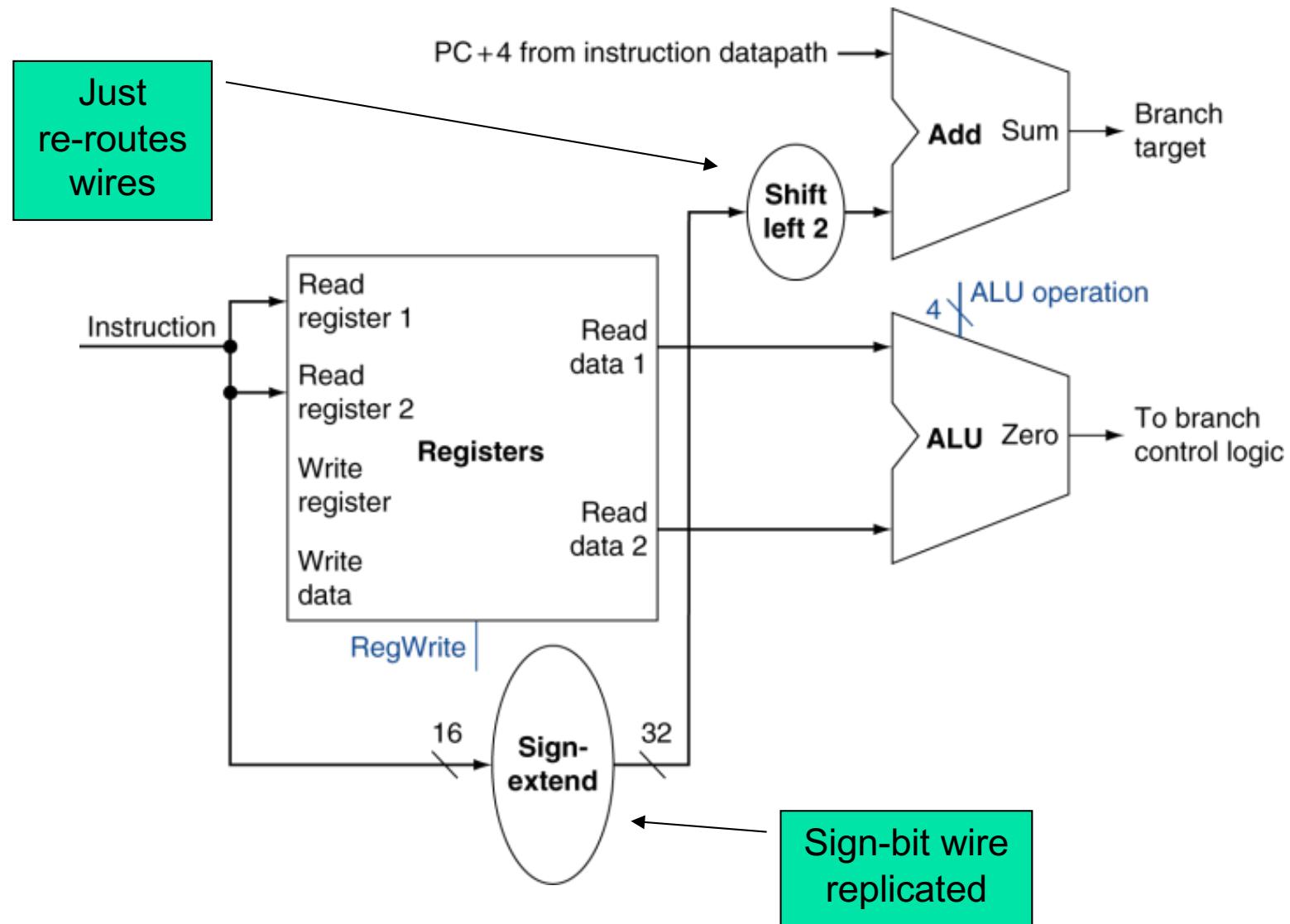
- Số hiệu thanh ghi **rs** đưa đến đầu vào *Read register 1* để chọn thanh ghi cơ sở **rs**, nội dung **rs** được đưa ra đầu ra *Read Data 1*, rồi chuyển đến *ALU*
- Hàng số imm 16-bit đưa đến bộ *Sign-extend* để mở rộng thành 32-bit, rồi chuyển đến *ALU*
- ALU* cộng hai giá trị trên đưa ra *ALU result* chính là địa chỉ của vị trí ở bộ nhớ dữ liệu; địa chỉ này được đưa đến đầu vào *Address* của bộ nhớ dữ liệu
- Số hiệu thanh ghi **rt** đưa đến đầu vào *Read register 2* để chọn thanh ghi dữ liệu nguồn **rt**, nội dung **rt** được đưa ra đầu ra *Read data 2*
- Dữ liệu 32-bit này sẽ được đưa đến đầu vào *Write data* của bộ nhớ dữ liệu và được ghi vào vị trí nhớ có địa chỉ đã được tính, nhờ tín hiệu điều khiển *MemWrite*

Thực hiện lệnh Branch (beq, bne)

4 or 5	rs	rt	imm
31:26	25:21	20:16	15:0

- bits 31:26 mã thao tác
 - 000100 (4) với lệnh beq
 - 000101 (5) với lệnh bne
- bits 25:21 số hiệu thanh ghi nguồn rs
- bits 20:16 số hiệu thanh ghi nguồn rt
- bits 15:0 hằng số imm có giá trị trong dải $[-2^{15}, +2^{15} - 1]$
- So sánh nội dung hai thanh ghi rs và rt:
 - Nếu điều kiện đúng: rẽ nhánh đến nhãn đích
 - $PC \leftarrow (PC + 4) + \text{hằng số} \times 4$
 - Nếu điều kiện sai: chuyển sang thực hiện lệnh kế tiếp
 - $PC \leftarrow PC + 4$

Các thành phần thực hiện lệnh Branch



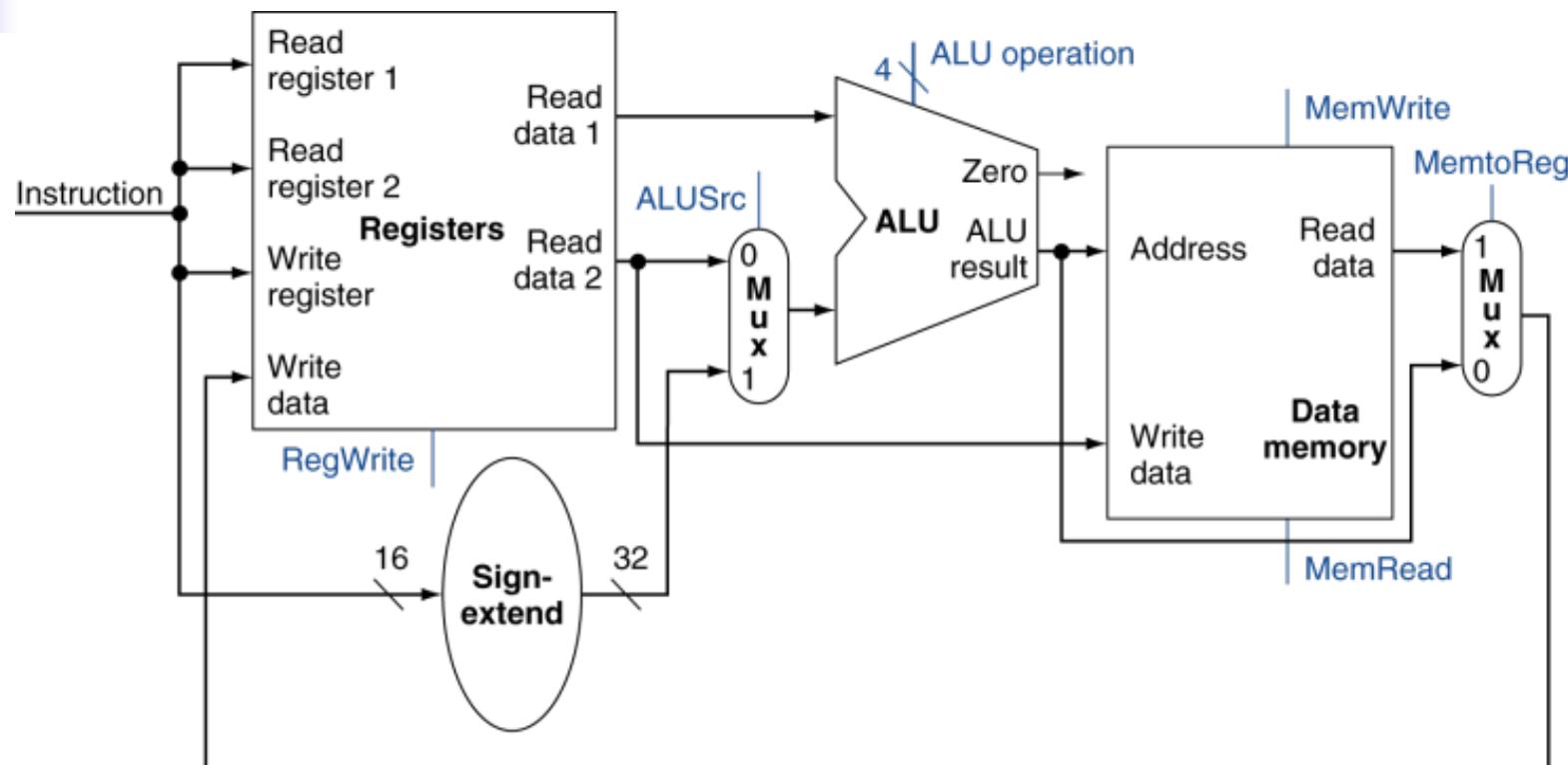
Thực hiện lệnh Branch (beq, bne)

- Nhờ các số hiệu thanh ghi **rs** và **rt**, hai toán hạng nguồn được đọc ra đưa đến **ALU**
- **ALU** trừ hai toán hạng và thiết lập giá trị ở đầu ra “Zero”
 - Hiệu = 0 → đầu ra Zero = 1
 - Hiệu \neq 0 → đầu ra Zero = 0
 - Đầu ra Zero này được đưa đến mạch logic điều khiển rẽ nhánh
- Bộ cộng **Add** tính địa chỉ đích rẽ nhánh
 - Hằng số imm 16-bit được mở rộng theo kiểu có dấu thành 32-bit, rồi dịch trái 2 bit
 - Cộng với PC (PC đã được tăng 4)
→ Địa chỉ đích = $(PC+4) + (\text{hằng số đã mở rộng 32-bit, dịch trái 2 bit})$
- Điều kiện đúng: $PC \leftarrow \text{địa chỉ đích rẽ nhánh}$ (rẽ nhánh xảy ra)
- Điều kiện sai: $PC \leftarrow PC+4$ (chuyển sang lệnh kế tiếp)

Hợp các thành phần cho các lệnh

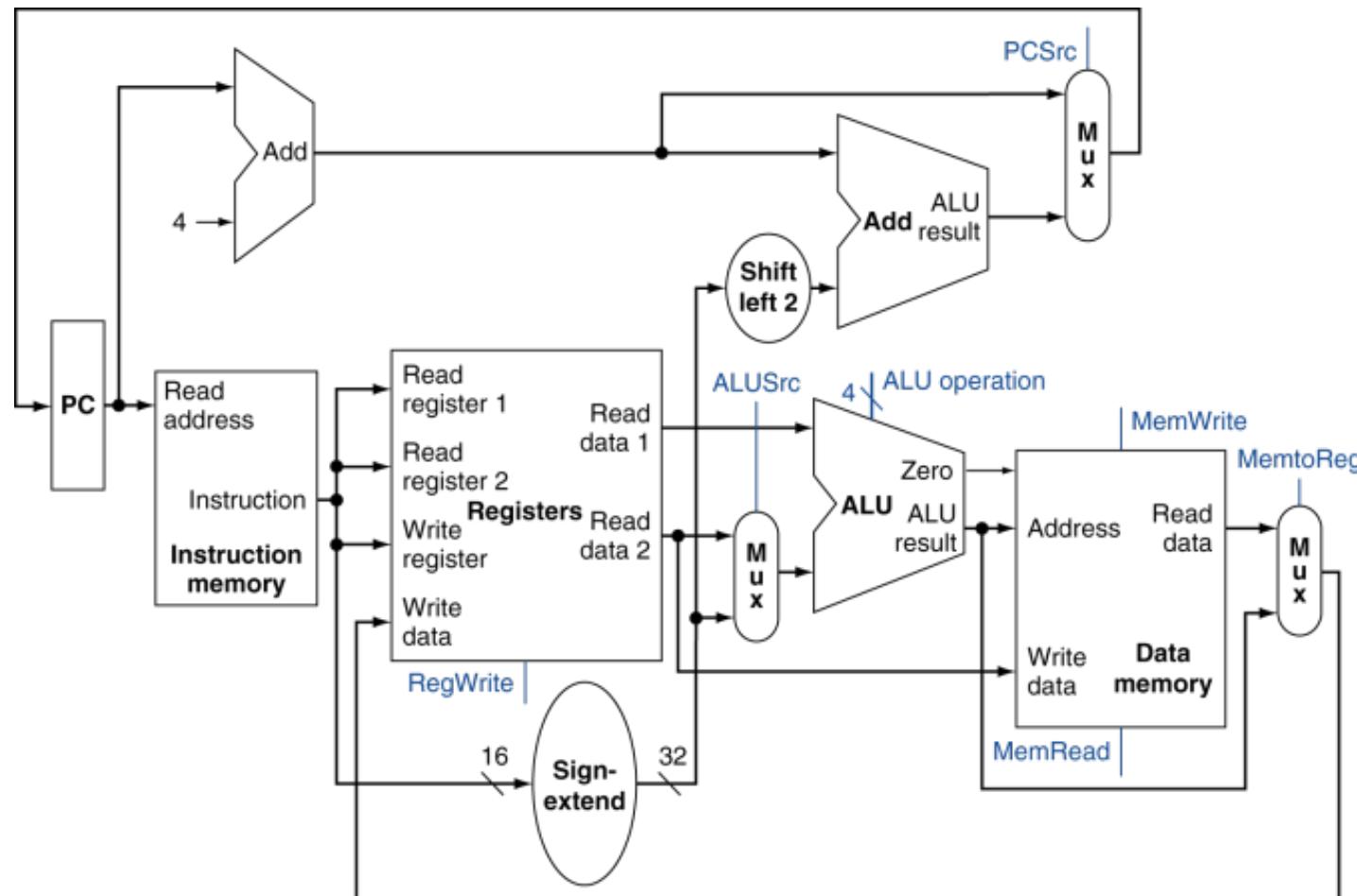
- Datapath cho các lệnh thực hiện trong 1 chu kỳ
 - Mỗi phần tử của datapath chỉ có thể làm một chức năng trong mỗi chu kỳ
 - Do đó, cần tách rời bộ nhớ lệnh và bộ nhớ dữ liệu
- Sử dụng các bộ chọn kênh để chọn dữ liệu nguồn cho các lệnh khác nhau

Datapath cho các lệnh R-Type/Load/Store



- **ALUSrc**: tín hiệu điều khiển chọn toán hạng đưa đến ALU:
 - Lệnh kiểu R: toán hạng từ thanh ghi nguồn thứ hai
 - Lệnh lw/sw: Hằng số imm 16-bit được mở rộng thành 32-bit (tính địa chỉ)
- **MemtoReg**: tín hiệu điều khiển chọn dữ liệu đưa về thanh ghi đích:
 - Lệnh kiểu R: lấy kết quả từ ALU result
 - Lệnh lw: dữ liệu đọc (Read data) từ bộ nhớ dữ liệu

Datapath đơn giản cho các lệnh R/lw/sw/branch



- PCSrc: tín hiệu điều khiển chọn giá trị cập nhật PC
 - Không rẽ nhánh: $PC \leftarrow PC + 4$
 - Rẽ nhánh: $PC \leftarrow (PC + 4) + (\text{hằng số imm đã mở rộng thành 32-bit} \ll 2)$

4.3. Thiết kế Control Unit

- Đơn vị điều khiển có hai phần:
 - Bộ điều khiển ALU
 - Bộ điều khiển chính

Thiết kế bộ điều khiển ALU

■ ALU được sử dụng để:

- Load/Store: F = add (xác định địa chỉ bộ nhớ dữ liệu)
- Branch: F = subtract (so sánh)
- Các lệnh số học/logic : F phụ thuộc vào funct code

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Tín hiệu điều khiển ALU

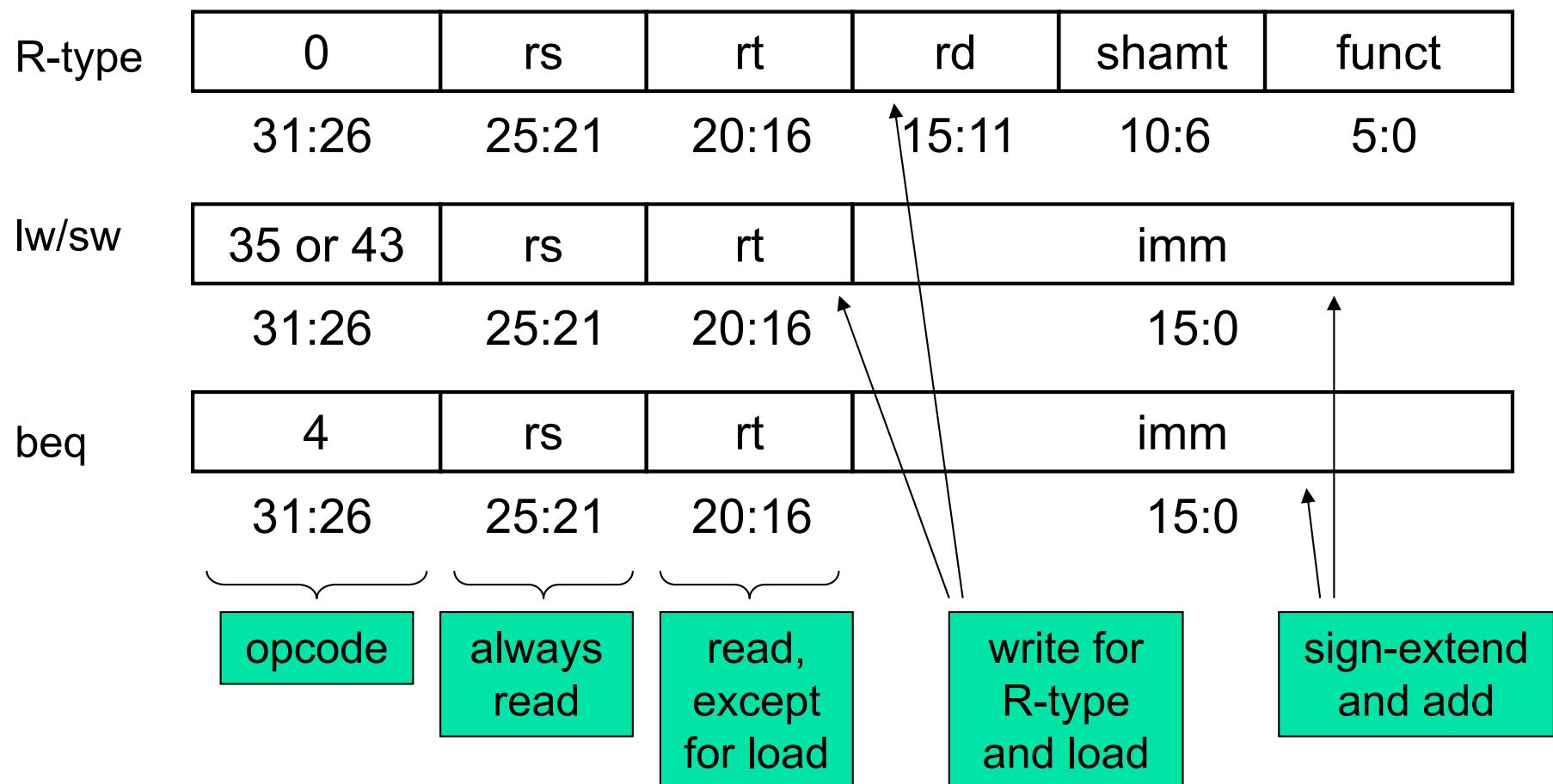
- Bộ điều khiển ALU sử dụng mạch logic tổ hợp:

- Đầu vào: 2-bit ALUOp được tạo ra từ opcode của lệnh và 6-bit của function code
- Đầu ra: các tín hiệu điều khiển ALU (ALU control) gồm 4 bit

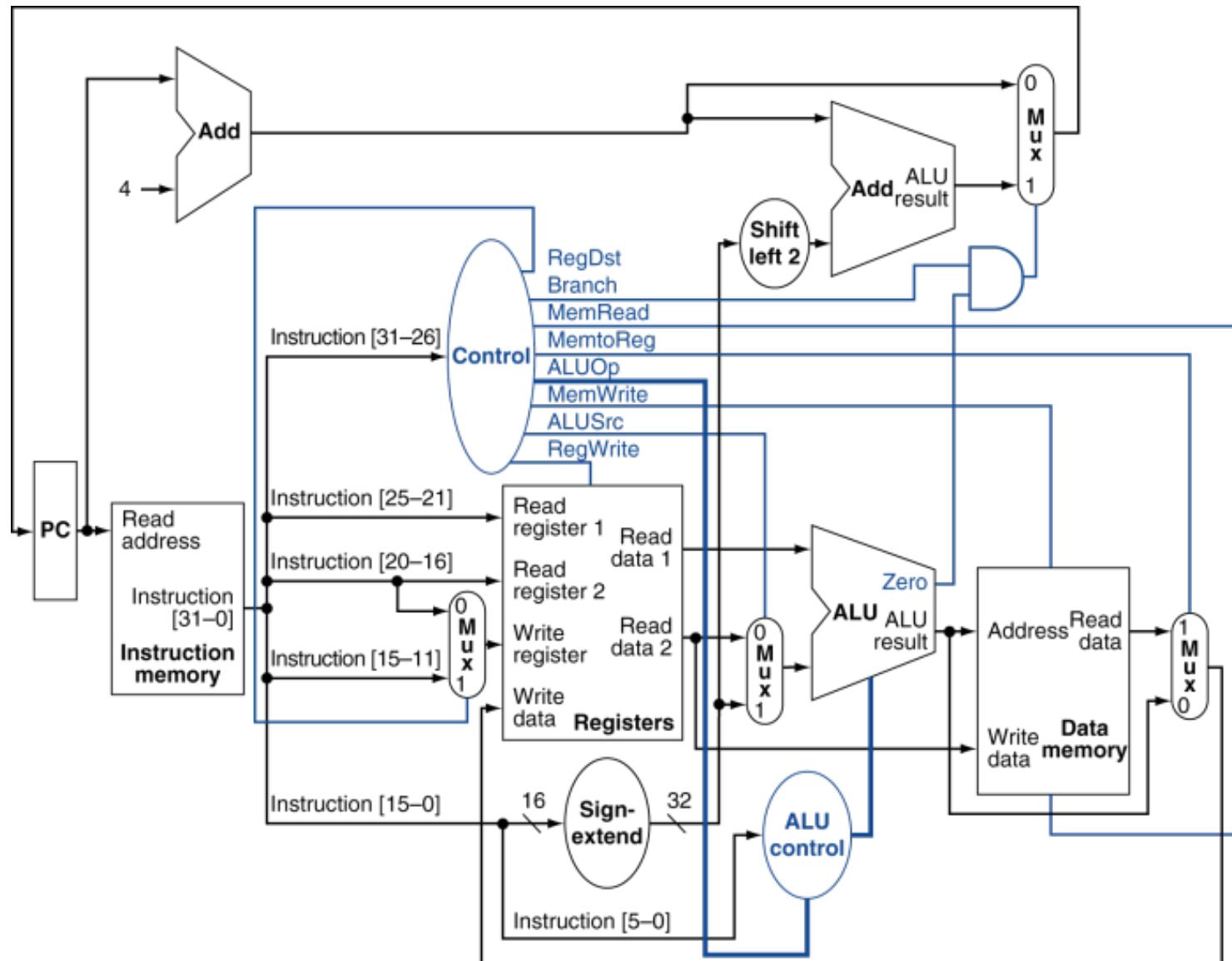
Opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

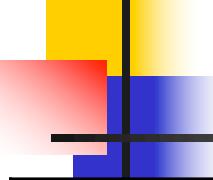
Thiết kế bộ điều khiển chính

- Các tín hiệu điều khiển được tạo ra từ lệnh



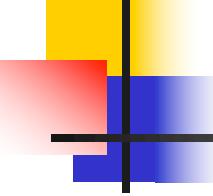
Datapath và Control Unit





Các tín hiệu điều khiển

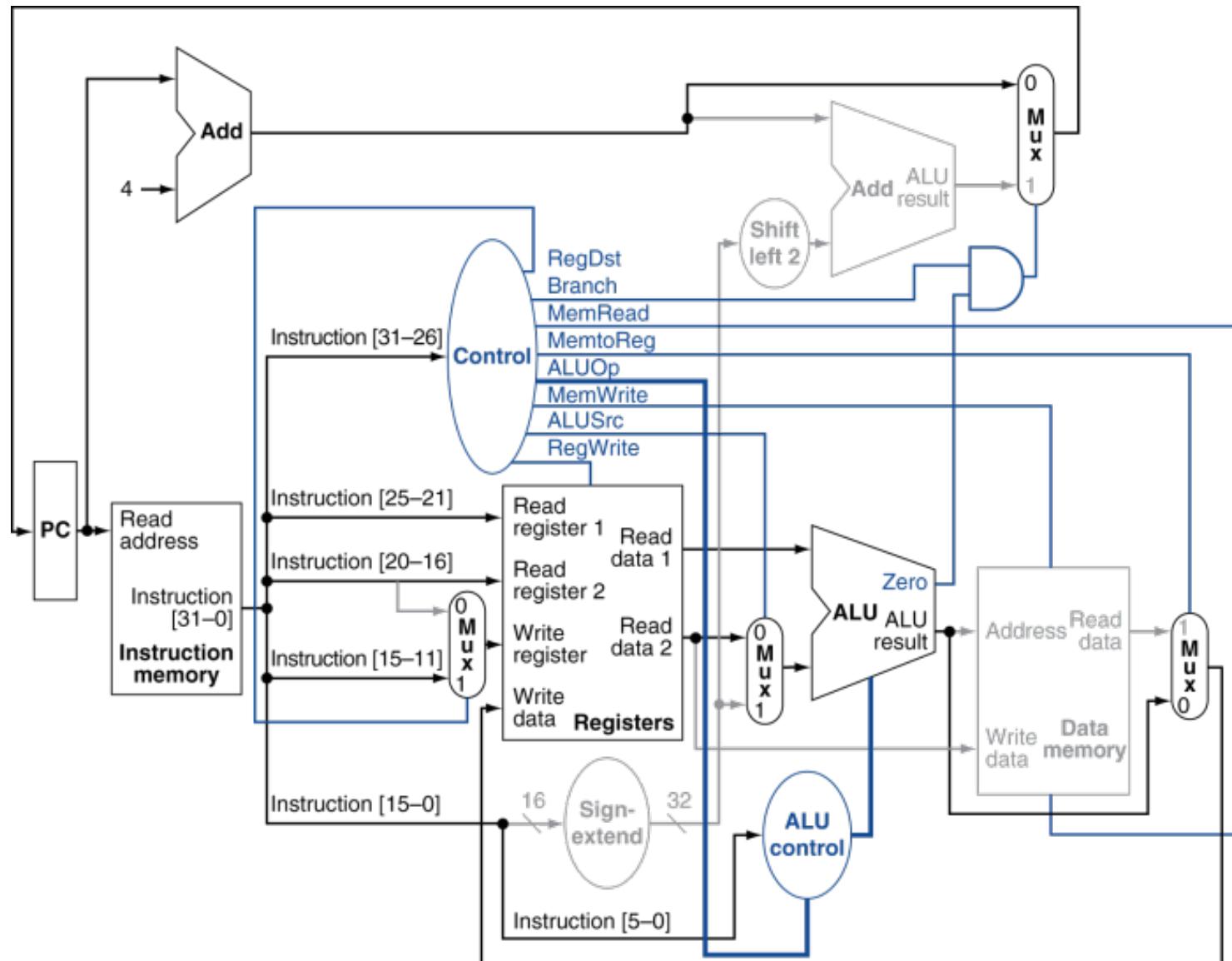
Tên tín hiệu	Hiệu ứng khi tín hiệu = 0	Hiệu ứng khi tín hiệu = 1
RegDst	Số hiệu thanh ghi đích là các bit 20:16 (rt)	Số hiệu thanh ghi đích là các bit 15:11 (rd)
Branch	Không có lệnh rẽ nhánh beq	Có lệnh rẽ nhánh beq (Branch =1) & (Zero=1): rẽ nhánh xảy ra (Branch =1) & (Zero=0): rẽ nhánh không xảy ra
RegWrite	Không làm gì cả	Ghi dữ liệu trên đầu vào <i>Write Data</i> ở tập thanh ghi đến thanh ghi đích
ALUSrc	Toán hạng thứ hai của ALU lấy từ thanh ghi nguồn thứ hai (Read data 2)	Toán hạng thứ hai của ALU là giá trị 16 bit thấp của lệnh (bits 15:0) được mở rộng có dấu thành 32-bit
PCSrc	PC \leftarrow PC+4	PC \leftarrow địa chỉ đích



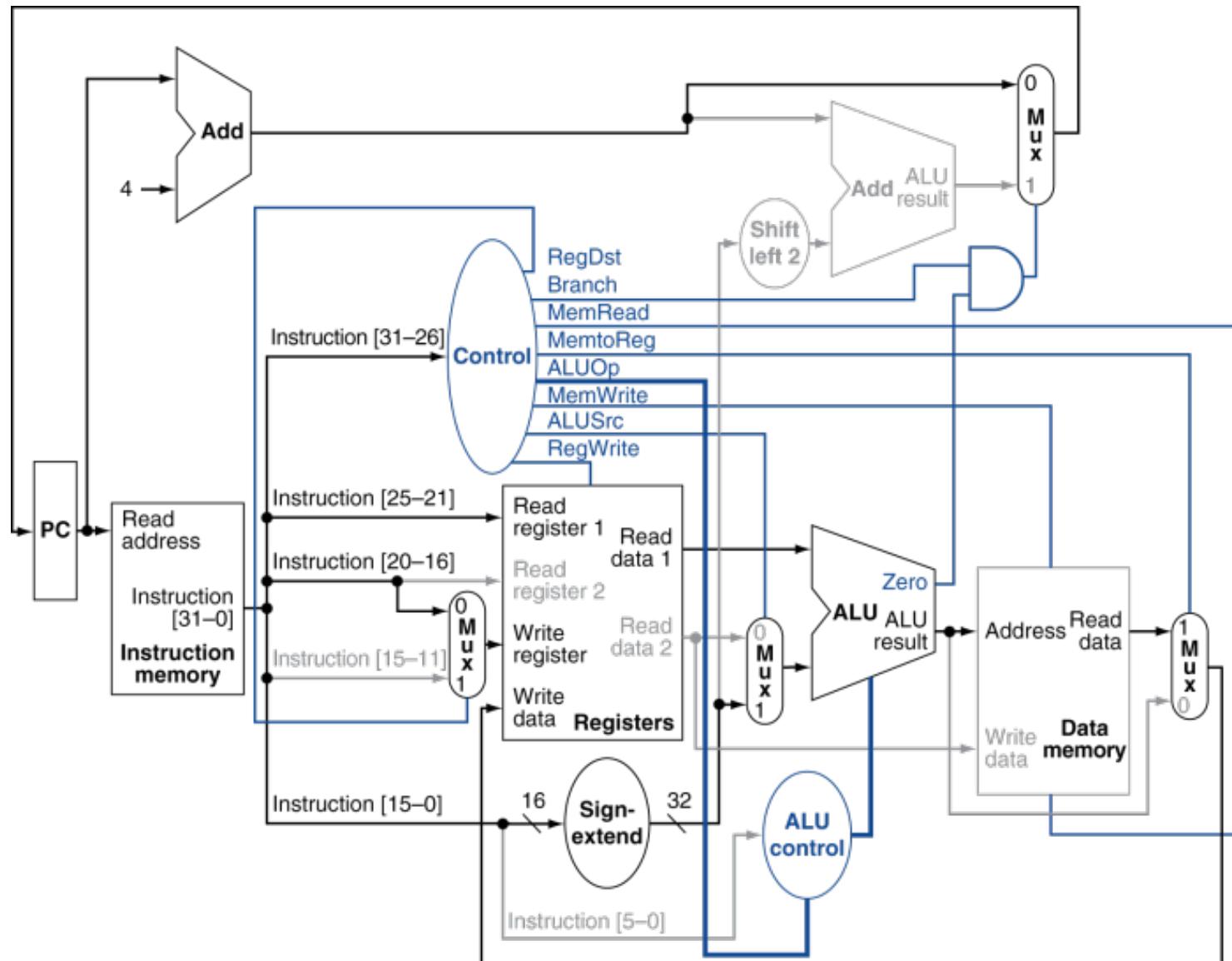
Các tín hiệu điều khiển (tiếp)

Tên tín hiệu	Hiệu ứng khi tín hiệu = 0	Hiệu ứng khi tín hiệu = 1
MemRead	Không làm gì cả	Nội dung ngăn nhớ dữ liệu, được xác định bởi địa chỉ do ALU tính, được đưa ra đầu ra <i>Read data</i> của bộ nhớ dữ liệu
MemWrite	Không làm gì cả	Dữ liệu trên đầu vào <i>Write Data</i> của bộ nhớ dữ liệu được ghi vào ngăn nhớ có địa chỉ do ALU tính
MemtoReg	Giá trị được đưa đến đầu vào <i>Write data</i> của tập thanh ghi là từ <i>ALU result</i>	Giá trị được đưa đến đầu vào <i>Write data</i> của tập thanh ghi là từ bộ nhớ dữ liệu

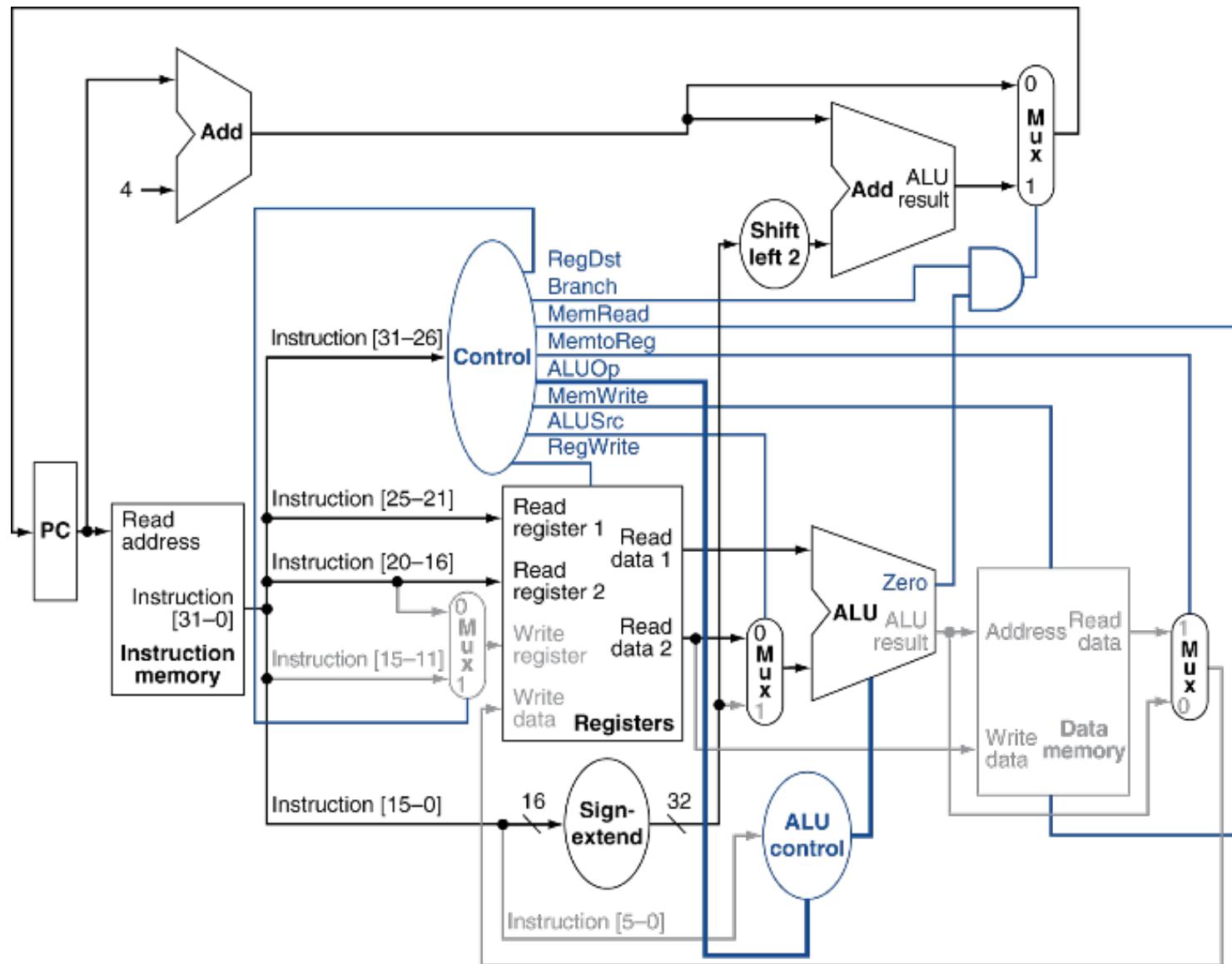
Thực hiện lệnh số học/logic kiểu R



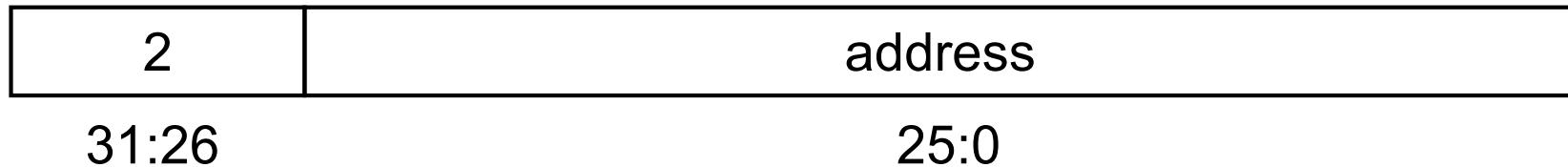
Thực hiện lệnh Load



Thực hiện lệnh beq

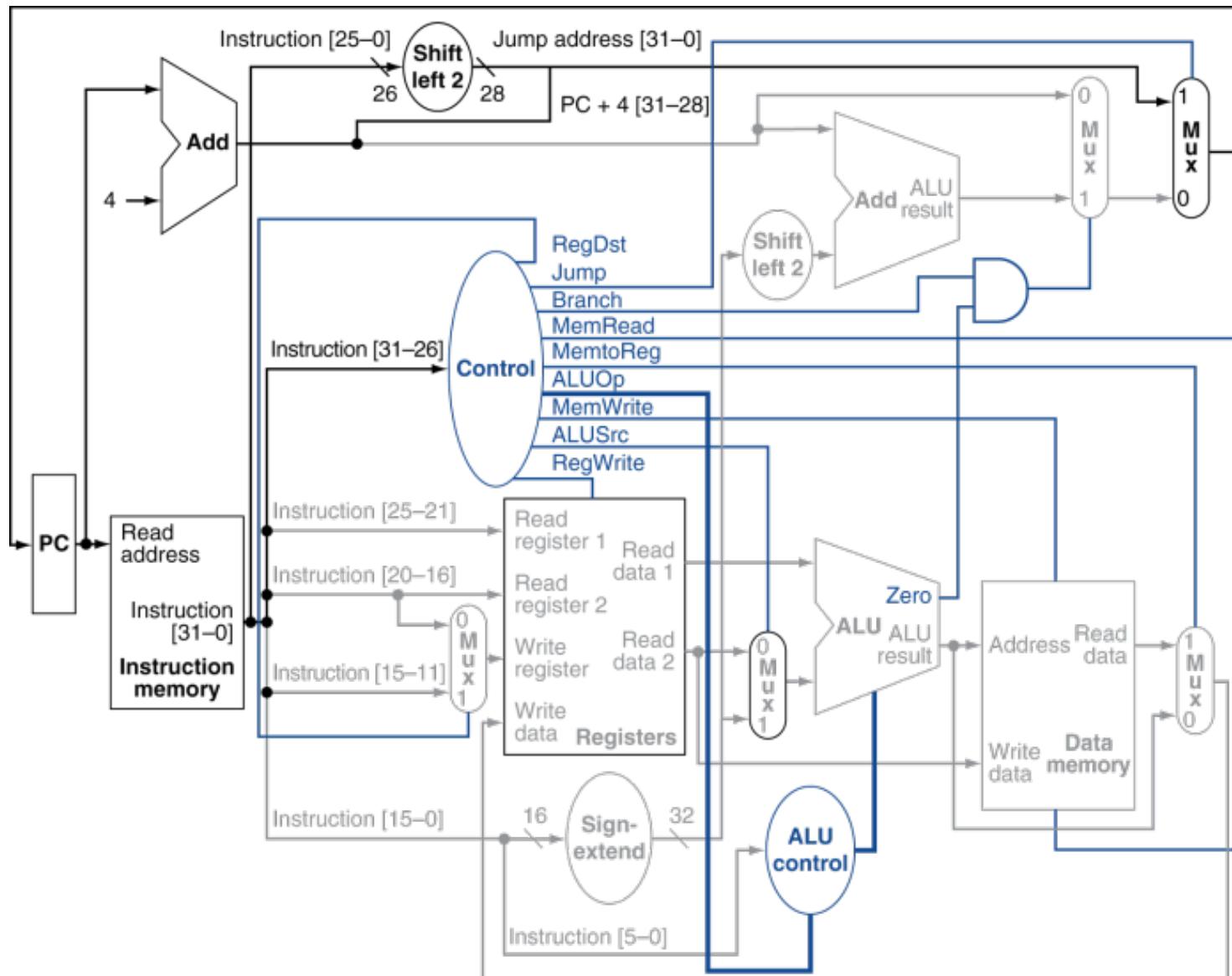


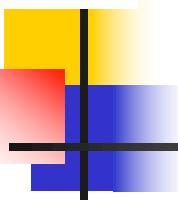
Thực hiện lệnh Jump



- Bits 31:26 là mã thao tác = 000010
- Bits 25:0: phần địa chỉ
- PC nhận giá trị sau:
 - Địa chỉ đích = $PC_{31\dots 28} : (address \ll 2)$
 - 4 bit bên trái là của PC cũ
 - 26-bit của lệnh jump (bits 25:0)
 - 2 bit cuối là 00
 - Cần thêm tín hiệu điều khiển được giải mã từ opcode

Datapath thêm cho lệnh jump



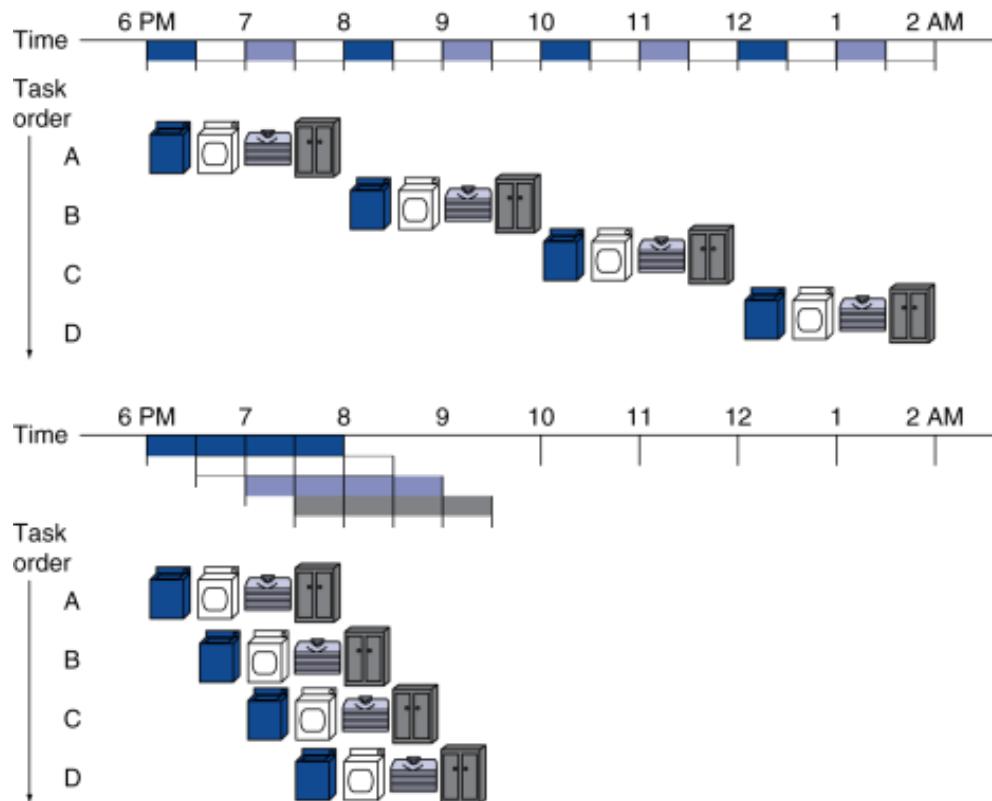


Thiết kế đơn chu kỳ (single-cycle)

- Chu kỳ xung nhịp có độ dài bằng nhau với tất cả các lệnh → chu kỳ xung nhịp được xác định bởi thời gian thực thi lệnh lâu nhất
- Ví dụ: Lệnh load sử dụng 5 đơn vị chức năng:
Bộ nhớ lệnh → tập thanh ghi → ALU → bộ nhớ dữ liệu → tập thanh ghi
- Thời gian thực hiện chương trình tăng → hiệu năng giảm
- Chúng ta sẽ tăng hiệu năng bằng kỹ thuật đường ống lệnh (pipelining)

4.4. Kỹ thuật đường ống lệnh (Pipelining)

- Chia quá trình thực hiện thành các công đoạn và cho thực hiện gối lên nhau.
- Ví dụ:



- 4 lô:
- Speedup
 $= 8/3.5 = 2.3$
- Liên tục:
- Speedup
 $= 2n/(0.5n + 1.5) \approx 4$
= số công đoạn

Đường ống lệnh ở MIPS

5 công đoạn:

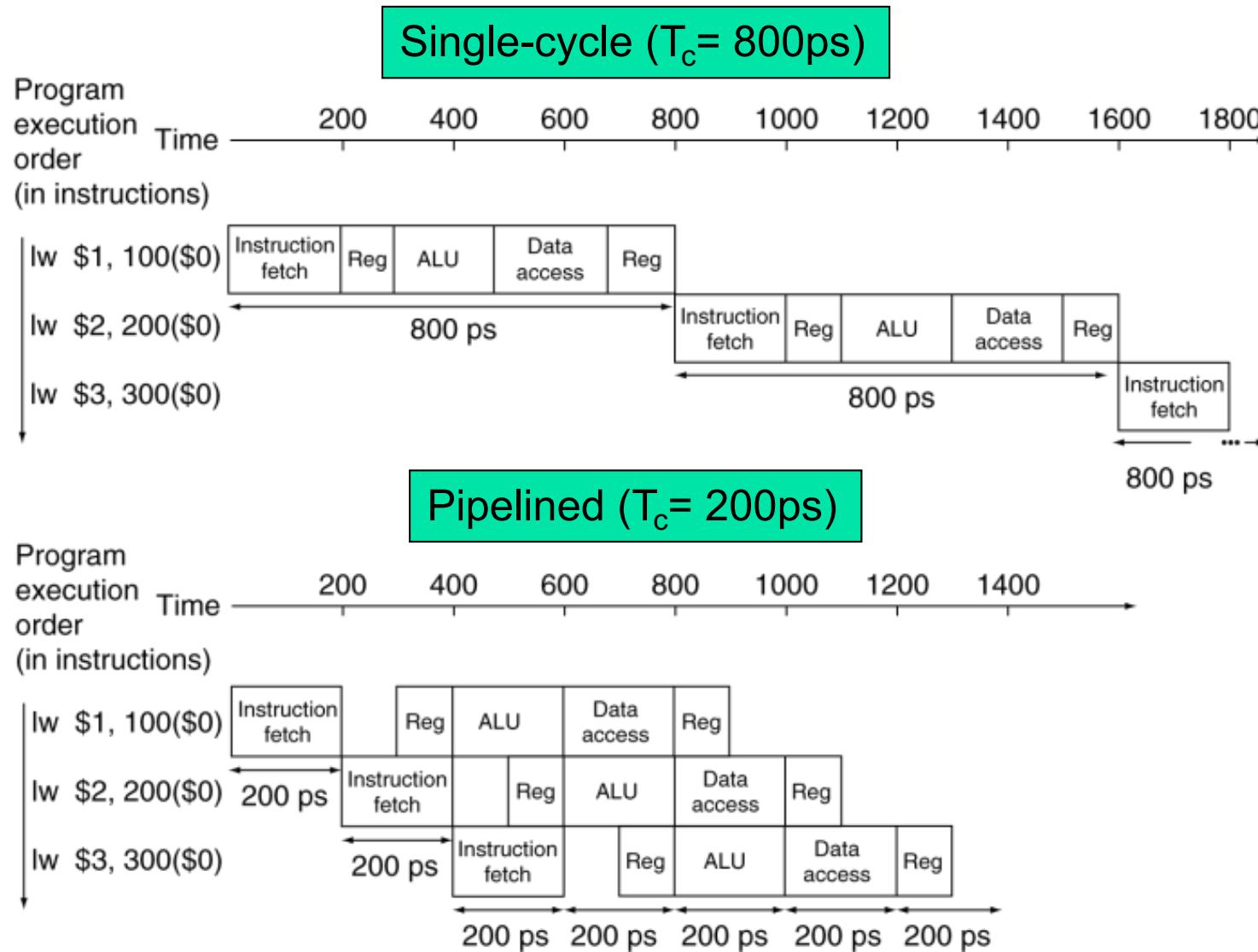
1. IF: Instruction fetch from memory – Nhận lệnh từ bộ nhớ
2. ID: Instruction decode & register read – Giải mã lệnh và đọc thanh ghi
3. EX: Execute operation or calculate address – Thực hiện thao tác hoặc tính toán địa chỉ
4. MEM: Access memory operand – Truy nhập toán hạng bộ nhớ
5. WB: Write result back to register – Ghi kết quả trả về thanh ghi

Hiệu năng của đường ống

- Giả thiết thời gian cho các công đoạn:
 - 100ps với đọc hoặc ghi thanh ghi
 - 200ps cho các công đoạn khác
- Thời gian của datapath đơn chu kỳ với một số lệnh:

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
Iw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Hiệu năng của đường ống



Độ tăng tốc của đường ống

- Nếu tất cả các công đoạn có thời gian thực hiện như nhau và số lệnh của chương trình là lớn:

Thời gian thực hiện pipeline =

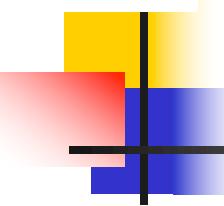
Thời gian thực hiện tuần tự

Số công đoạn

Thiết kế đường ống lệnh

■ Kiến trúc tập lệnh MIPS được thiết kế phù hợp với kỹ thuật đường ống

- Tất cả các lệnh là 32-bits
 - Dễ dàng để nhận và giải mã lệnh trong một chu kỳ
 - Intel x86: lệnh từ 1 đến 17 bytes
- Có ít dạng lệnh và thông dụng
 - Có thể giải mã và đọc thanh ghi trong một bước
- Địa chỉ hóa cho các lệnh load/store
 - Có thể tính địa chỉ trong công đoạn thứ 3, truy cập bộ nhớ công đoạn thứ 4
- Toán hạng bộ nhớ nằm thẳng hàng trên các băng nhớ
 - Truy cập bộ nhớ chỉ mất một chu kỳ



Các mối trở ngại (Hazard) của đường ống lệnh

- Hazard: Tình huống ngăn cản bắt đầu của lệnh tiếp theo ở chu kỳ tiếp theo
 - Hazard cấu trúc: do tài nguyên được yêu cầu đang bận
 - Hazard dữ liệu: cần phải đợi để lệnh trước hoàn thành việc đọc/ghi dữ liệu
 - Hazard điều khiển: do rẽ nhánh gây ra

Hazard cấu trúc

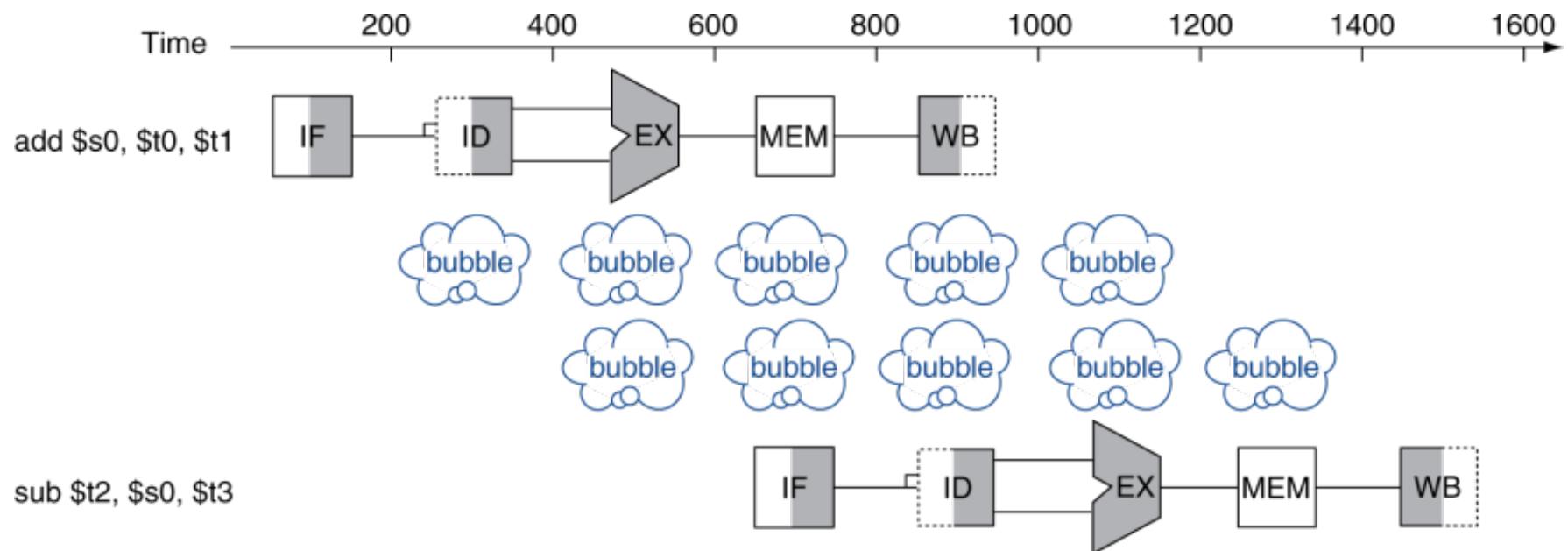
- Xung đột khi sử dụng tài nguyên
- Trong đường ống của MIPS với một bộ nhớ dùng chung
 - Lệnh Load/store yêu cầu truy cập dữ liệu
 - Nhận lệnh cần trì hoãn cho chu kỳ đó
- Bởi vậy, datapath kiểu đường ống yêu cầu bộ nhớ lệnh và bộ nhớ dữ liệu tách rời (hoặc cache lệnh/cache dữ liệu tách rời)

Hazard dữ liệu

- Lệnh phụ thuộc vào việc hoàn thành truy cập dữ liệu của lệnh trước đó

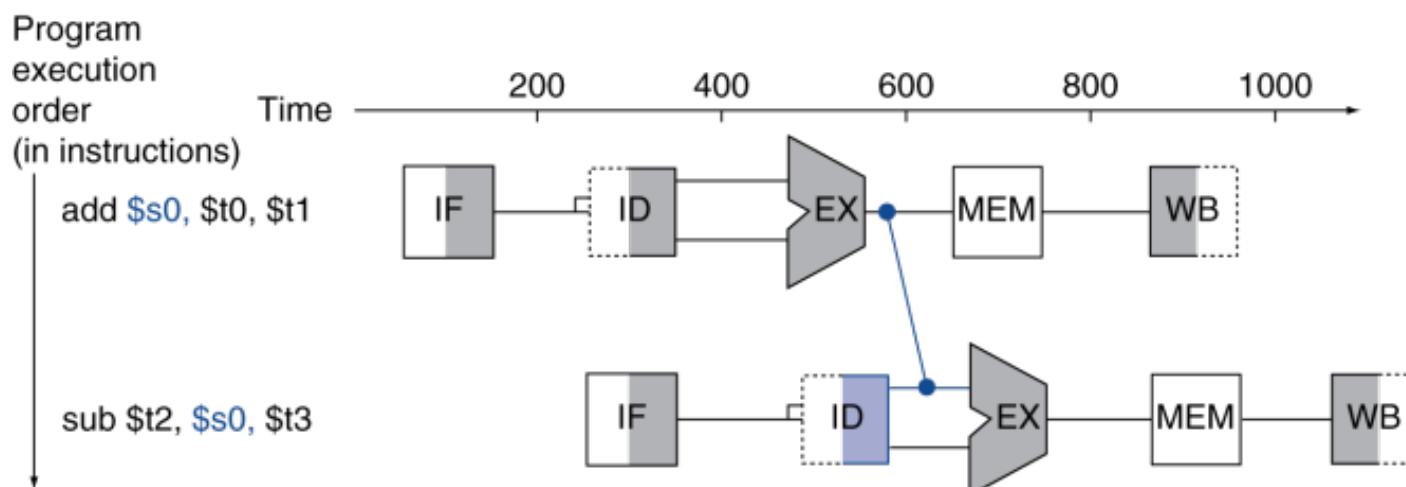
add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3



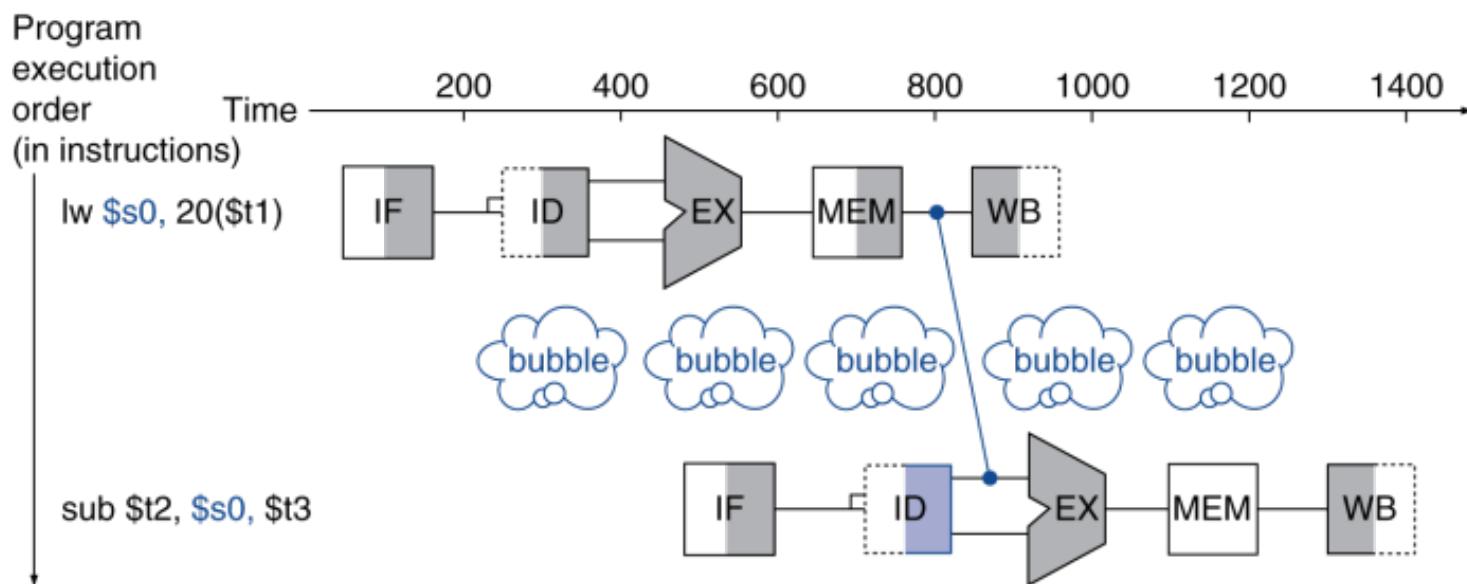
Forwarding (gửi vượt trước)

- Sử dụng kết quả ngay sau khi nó được tính
 - Không đợi đến khi kết quả được lưu đến thanh ghi
 - Yêu cầu có đường kết nối thêm trong datapath



Hazard dữ liệu với lệnh load

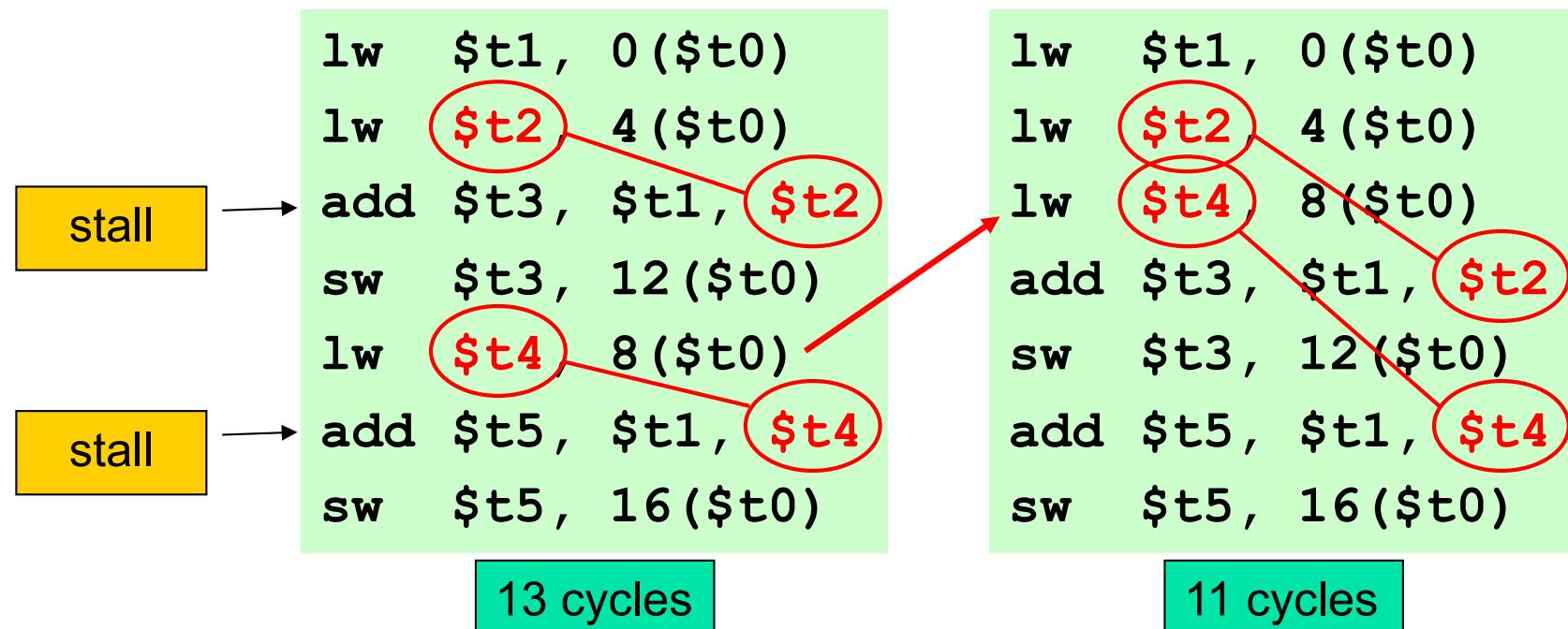
- Không phải luôn luôn có thể tránh trì hoãn bằng cách forwarding
 - Nếu giá trị chưa được tính khi cần thiết
 - Không thể chuyển ngược thời gian
 - Cần chèn bước trì hoãn (stall hay bubble)

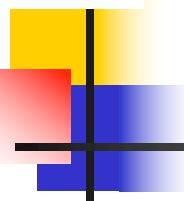


Lập lịch mã để tránh trì hoãn

- Thay đổi trình tự mã để tránh sử dụng kết quả load ở lệnh tiếp theo
- Mã C:

$$a = b + e; \quad c = b + f;$$



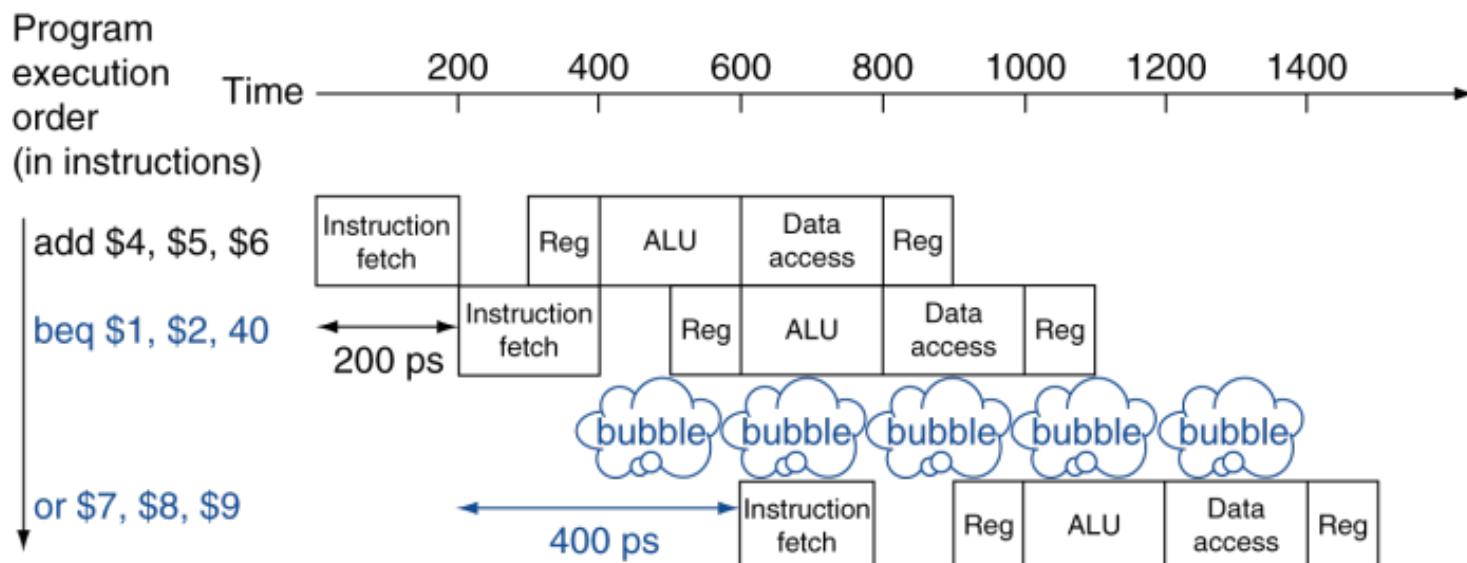


Hazard điều khiển

- Rẽ nhánh xác định luồng điều khiển
 - Nhận lệnh tiếp theo phụ thuộc vào kết quả rẽ nhánh
 - Đường ống không thể luôn nhận đúng lệnh
 - Vẫn đang làm ở công đoạn giải mã lệnh (ID) của lệnh rẽ nhánh
- Với đường ống của MIPS
 - Cần so sánh thanh ghi và tính địa chỉ đích sớm trong đường ống
 - Thêm phần cứng để thực hiện việc đó trong công đoạn ID

Trì hoãn khi rẽ nhánh

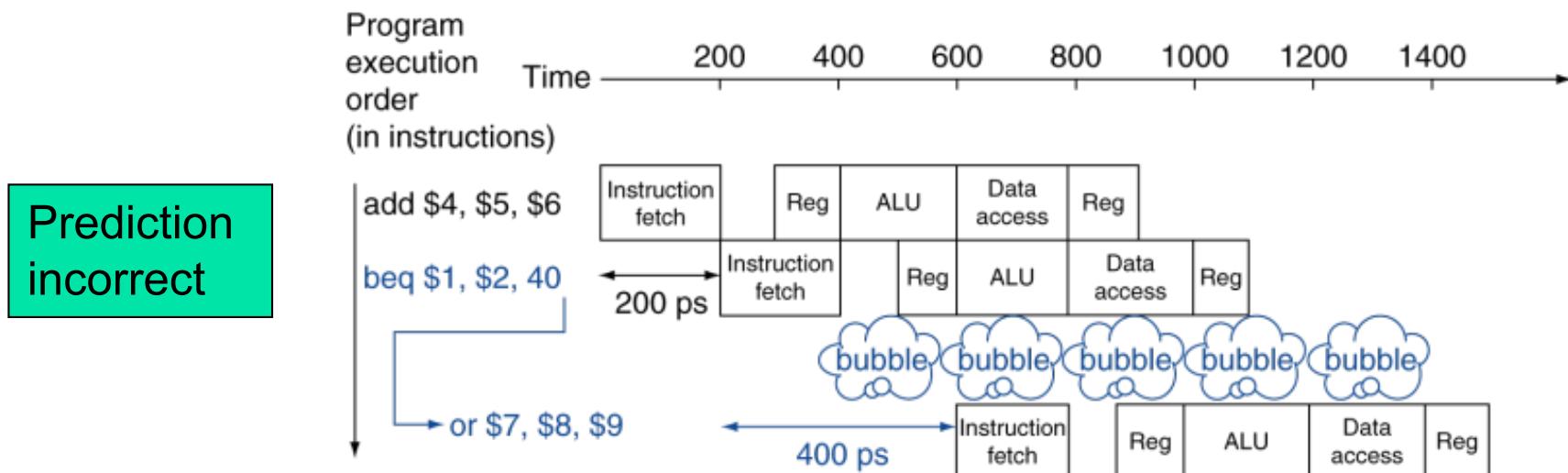
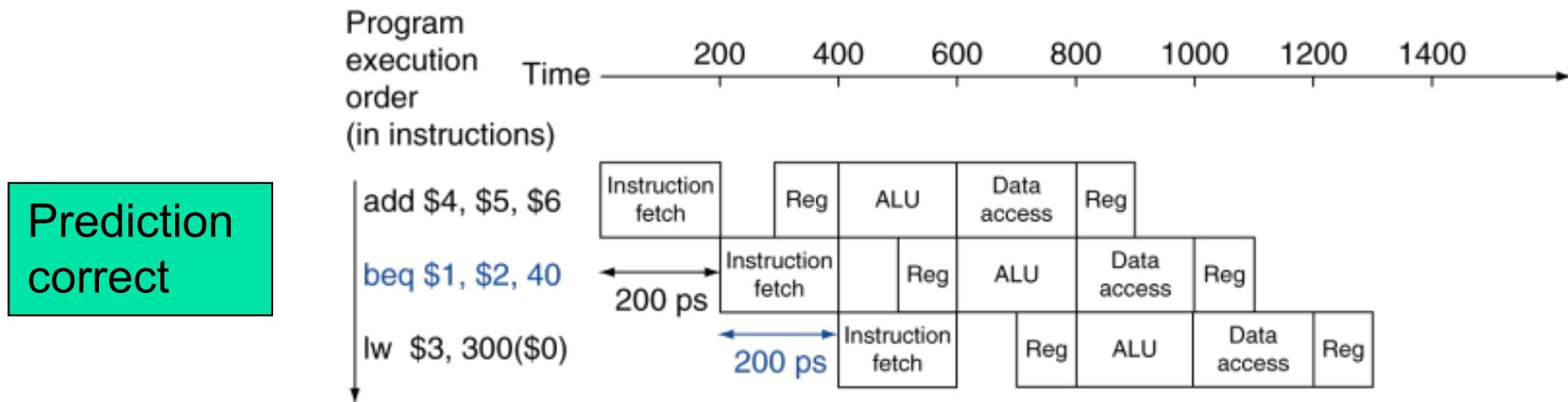
- Đợi cho đến khi kết quả rẽ nhánh đã được xác định trước khi nhận lệnh tiếp theo

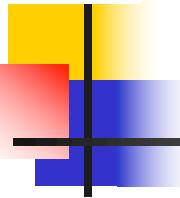


Dự đoán rẽ nhánh

- Những đường ống dài hơn không thể sớm xác định dễ dàng kết quả rẽ nhánh
 - Cách trì hoãn không đáp ứng được
- Dự đoán kết quả rẽ nhánh
 - Chỉ trì hoãn khi dự đoán là sai
- Với MIPS
 - Có thể dự đoán rẽ nhánh không xảy ra
 - Nhận lệnh ngay sau lệnh rẽ nhánh (không làm trễ)

MIPS với dự đoán rẽ nhánh không xảy ra

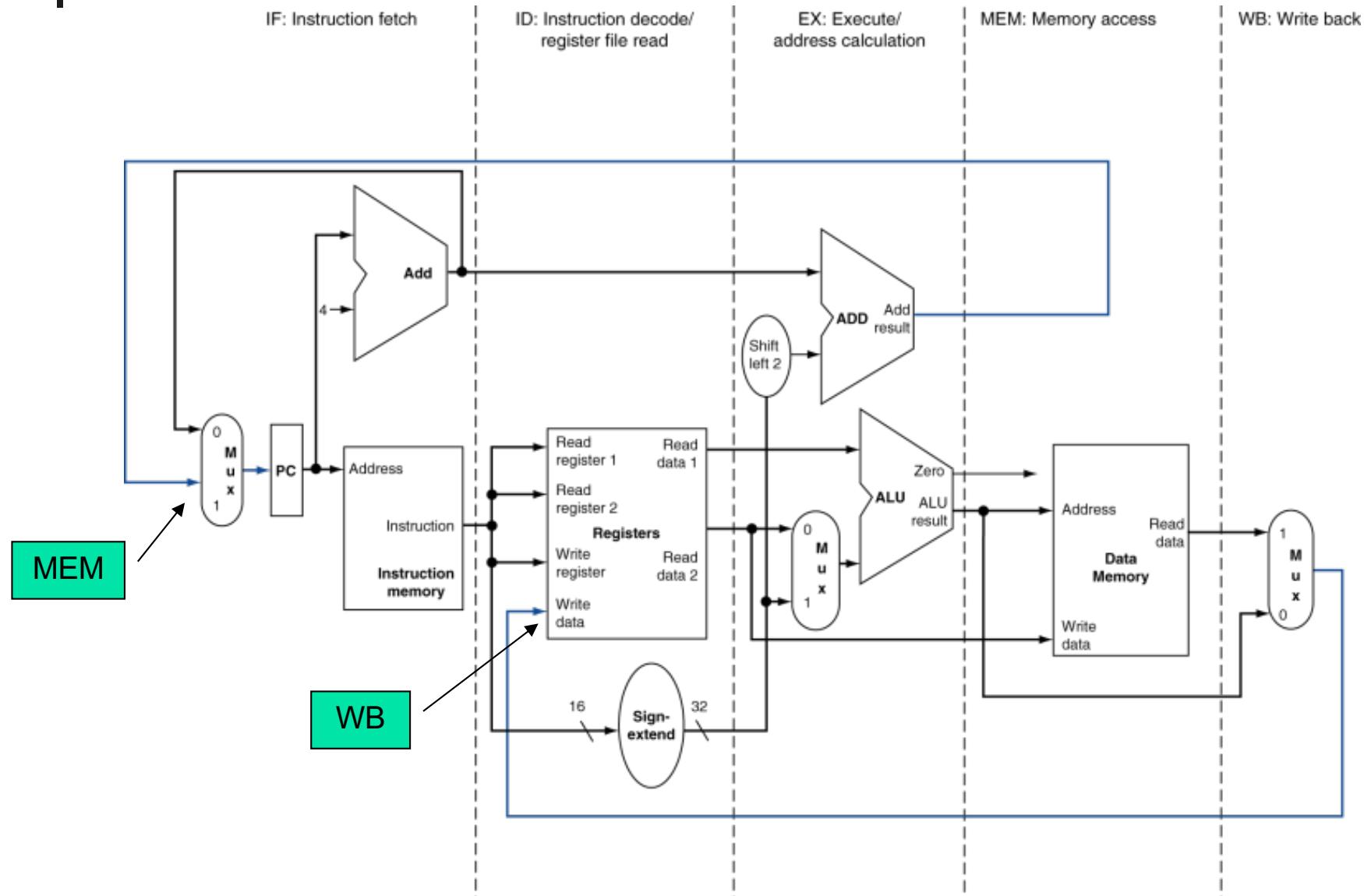




Đặc điểm của đường ống

- Kỹ thuật đường ống cải thiện hiệu năng bằng cách tăng số lệnh thực hiện
 - Thực hiện nhiều lệnh đồng thời
 - Mỗi lệnh có cùng thời gian thực hiện
- Các dạng hazard:
 - Cấu trúc, dữ liệu, điều khiển
- Thiết kế tập lệnh ảnh hưởng đến độ phức tạp của việc thực hiện đường ống

MIPS Datapath được ống hóa theo đơn chu kỳ

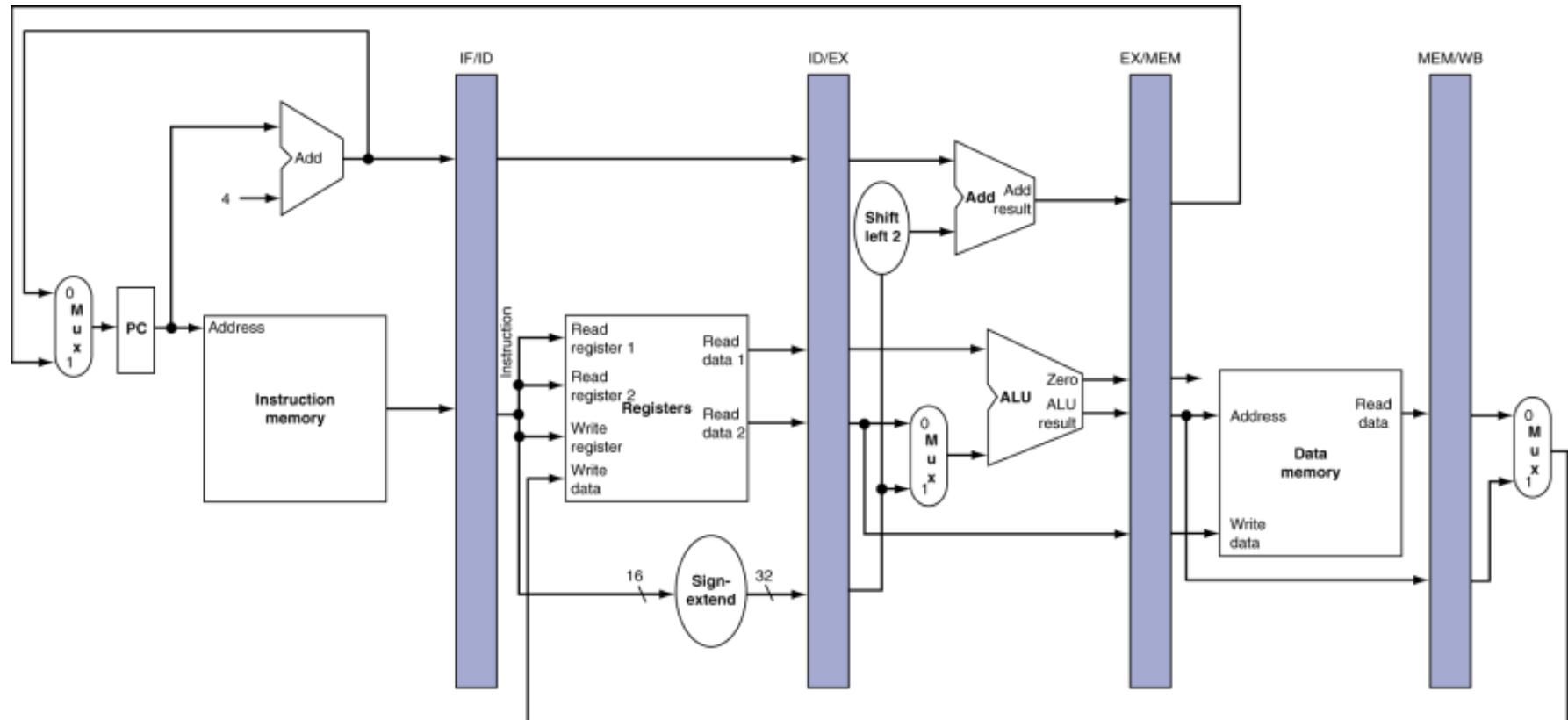


Nhận xét

- Các lệnh và các dữ liệu được chuyển từ trái sang phải qua 5 công đoạn.
- Có hai ngoại lệ từ phải sang trái:
 - Công đoạn write-back đặt kết quả về thanh ghi ở giữa datapath → dẫn đến data hazard
 - Chọn giá trị tiếp theo của PC là PC+4 hay địa chỉ đích rẽ nhánh từ công đoạn MEM → dẫn đến control hazard

Các thanh ghi đường ống

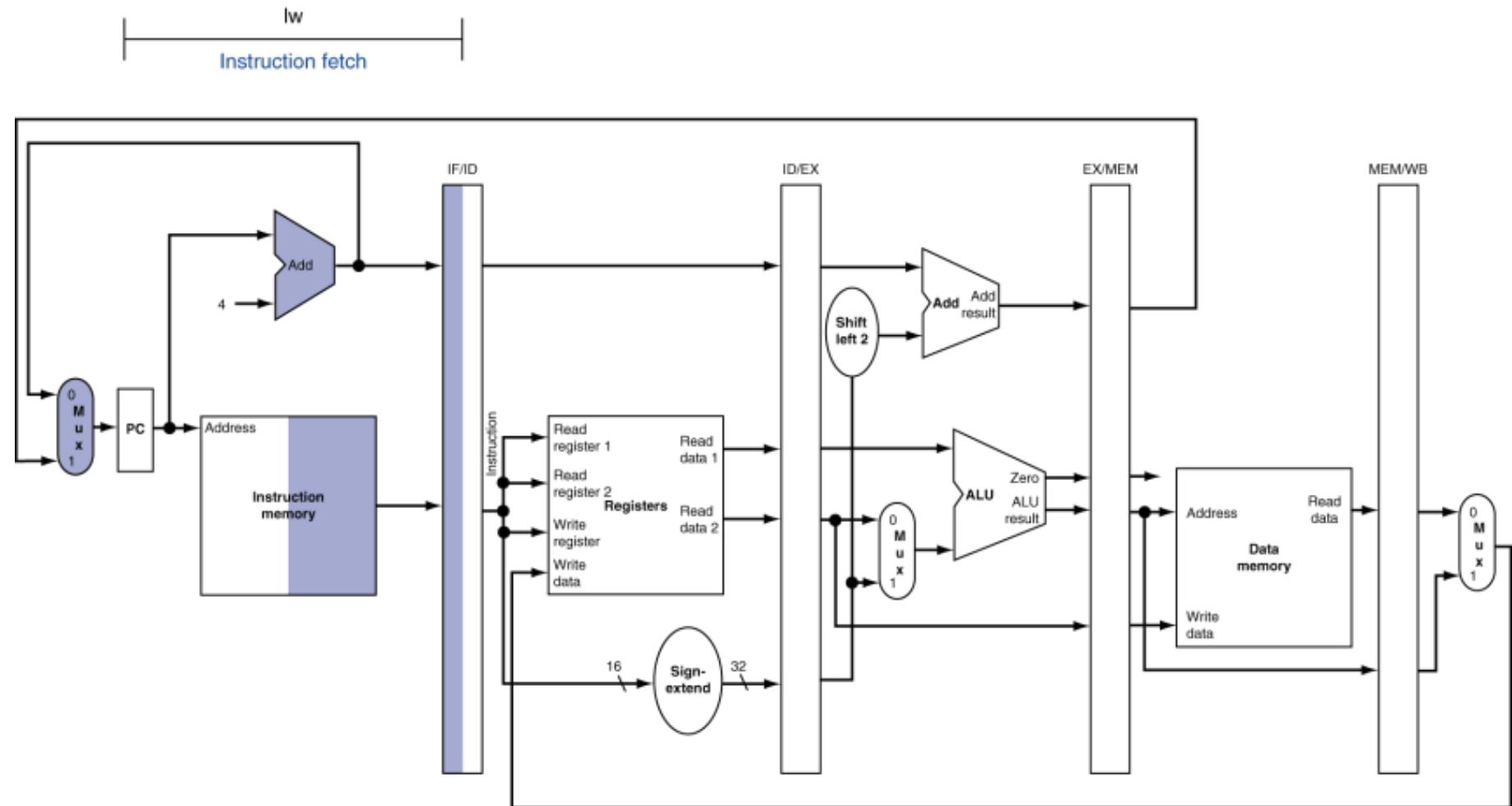
- Cần các thanh ghi đặt giữa các công đoạn
 - Để giữ thông tin được tạo ra bởi chu kỳ trước



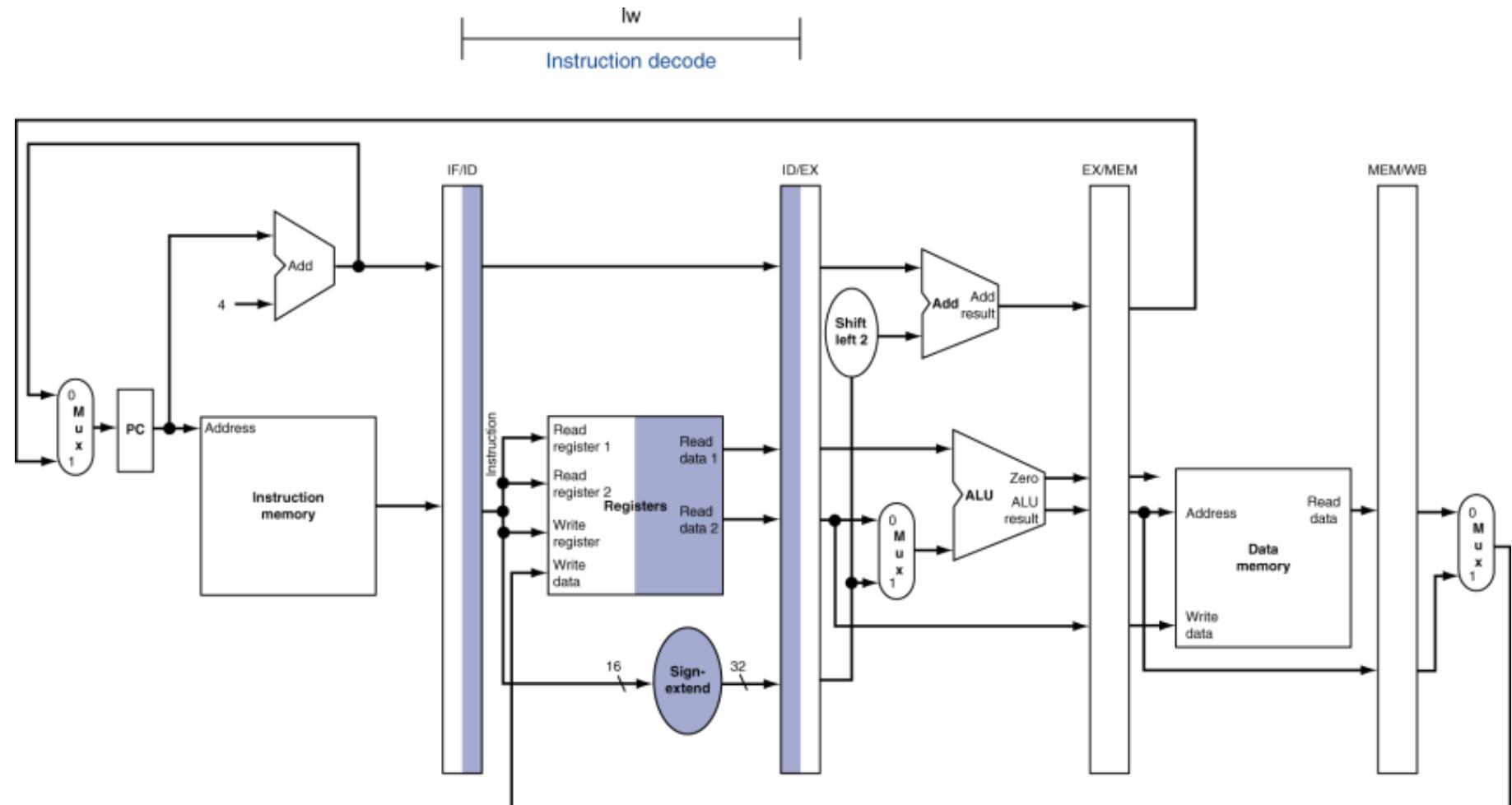
Hoạt động của đường ống

- Dòng lệnh được đưa qua datapath đường ống theo từng chu kỳ.
- Có hai cách thực hiện:
 - Đơn chu kỳ (Single-clock-cycle)
 - Đa chu kỳ (Multi-clock-cycle)
- Xem xét đường ống đơn chu kỳ với load & store

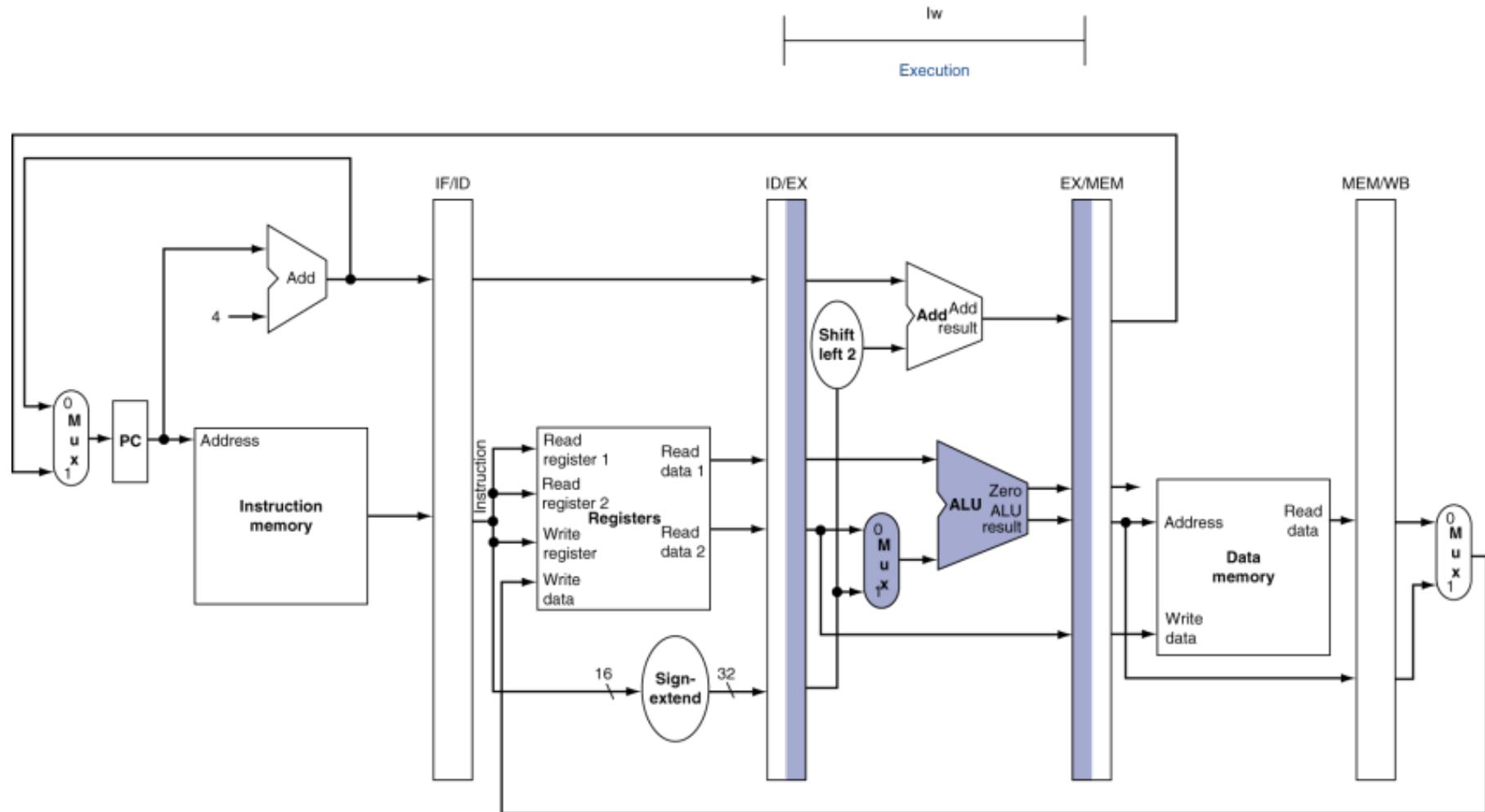
IF cho lệnh Load, Store



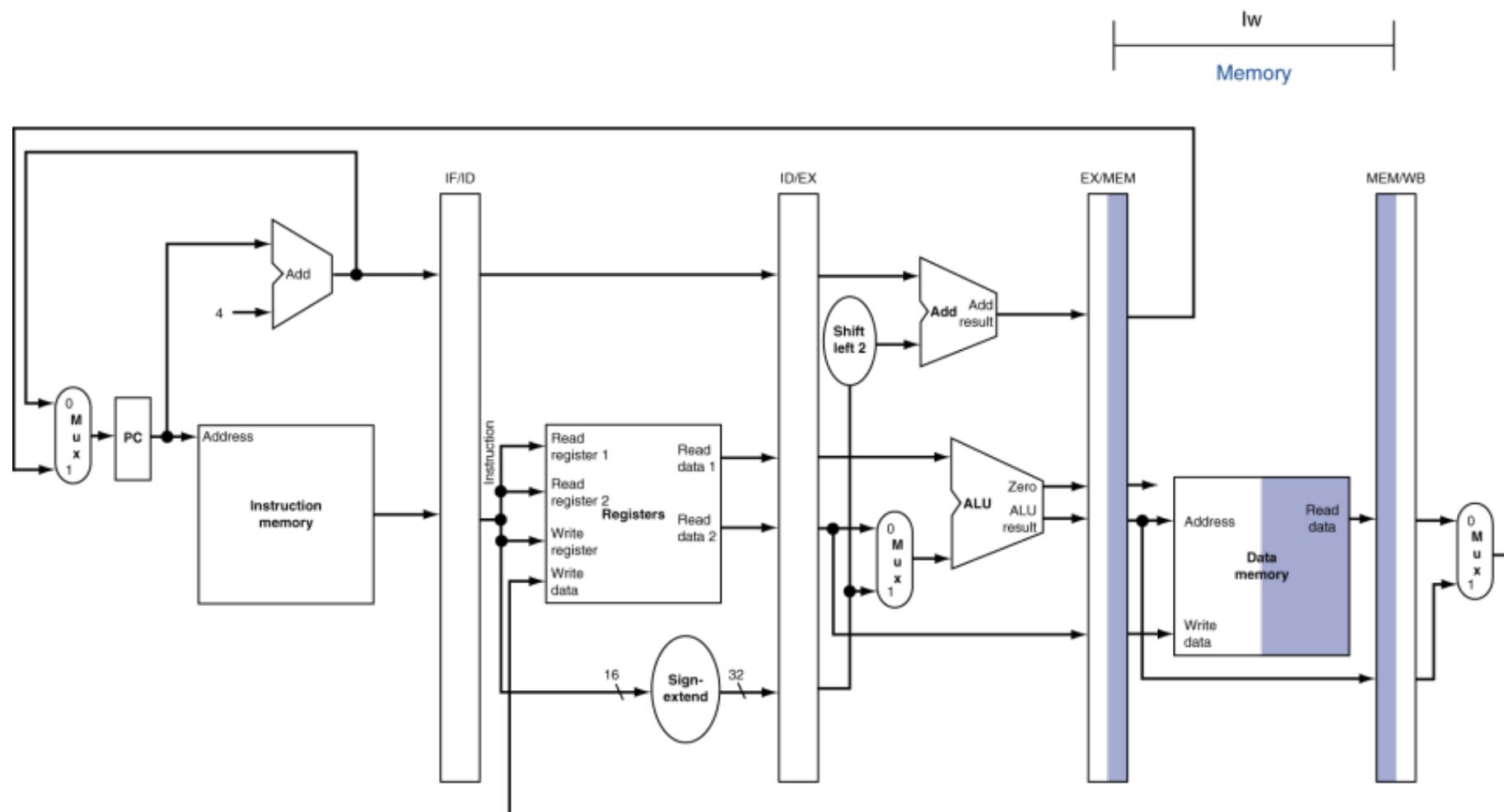
ID cho lệnh Load, Store



EX cho lệnh Load

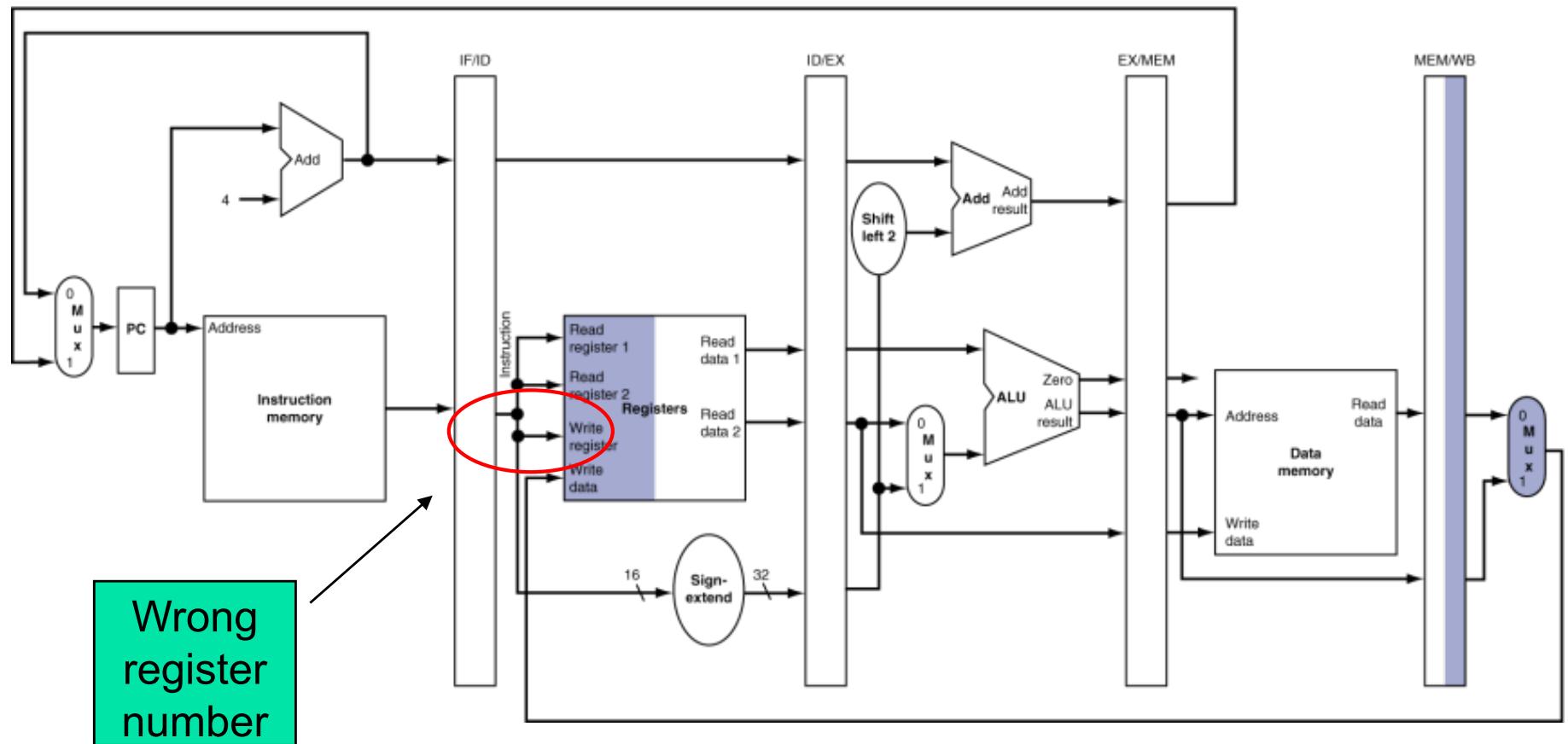


MEM cho lệnh Load

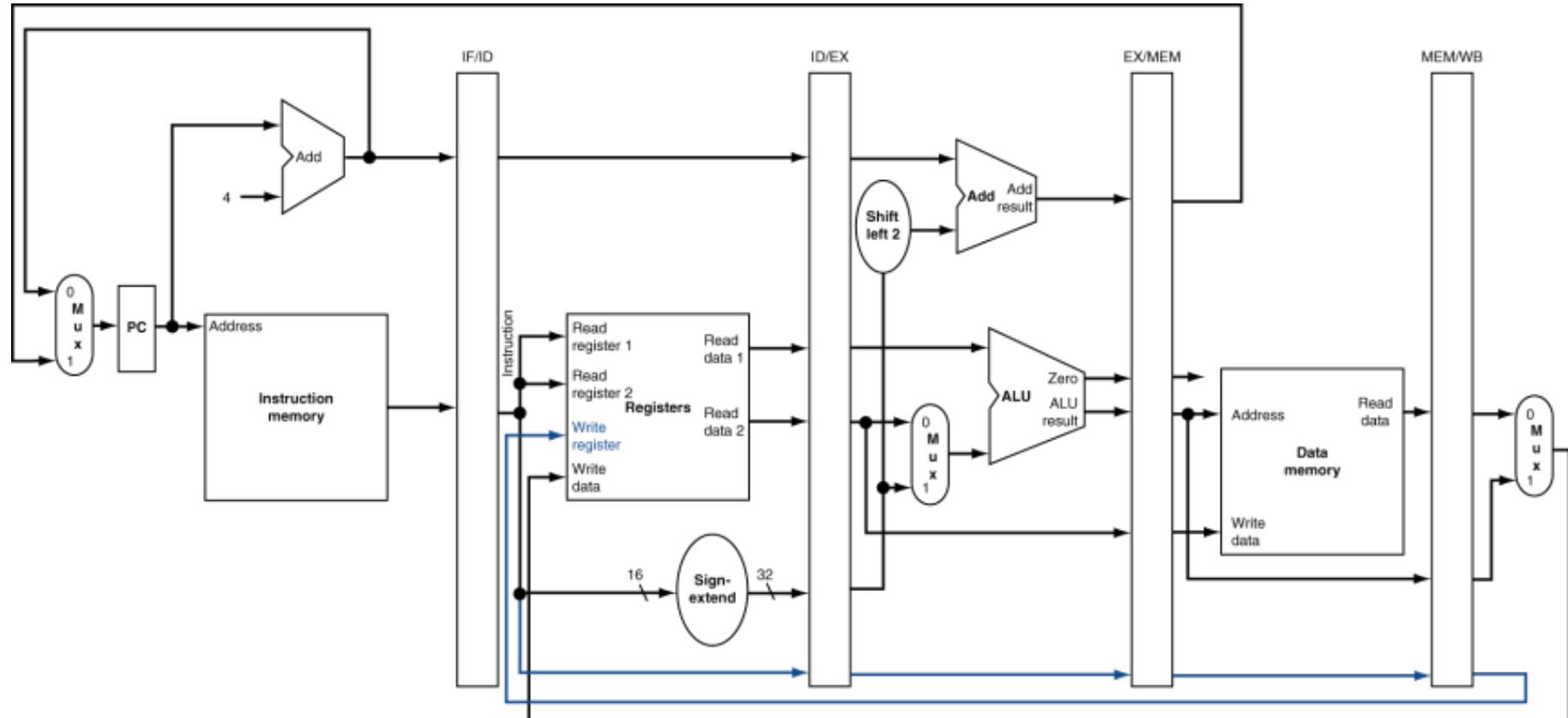


WB cho lệnh Load

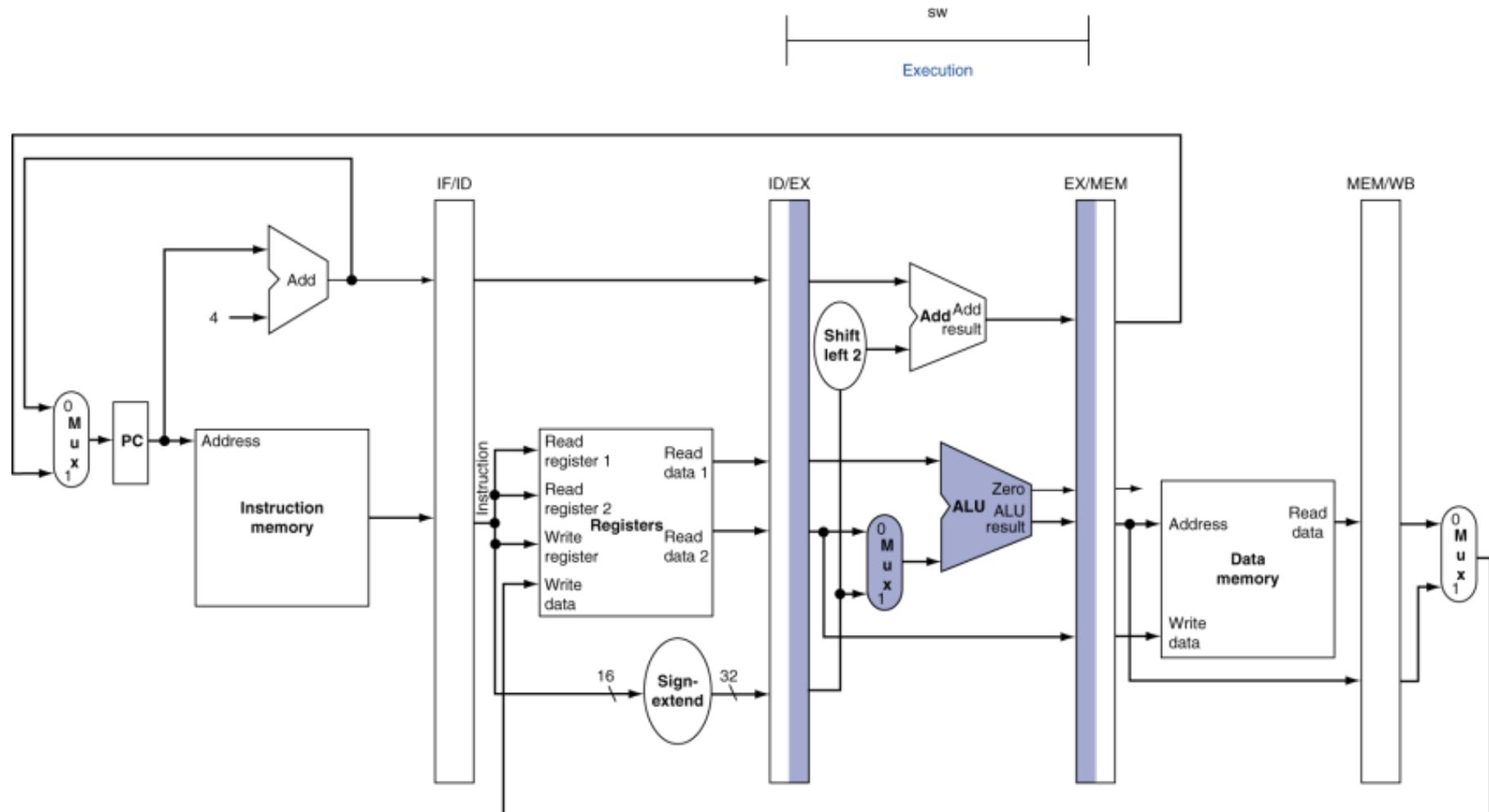
Iw
Write back



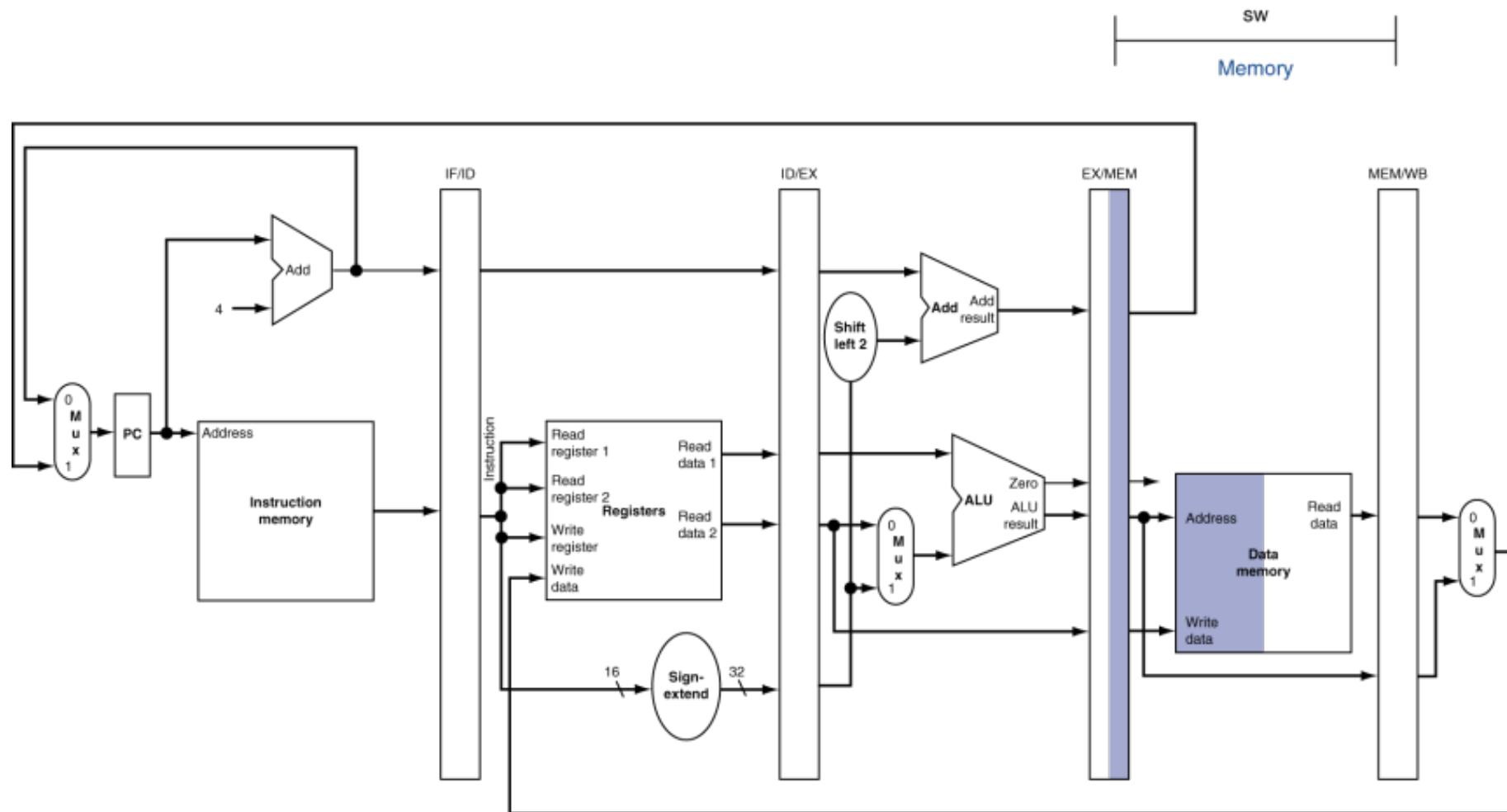
Datapath được hiệu chỉnh cho lệnh Load



EX cho lệnh Store

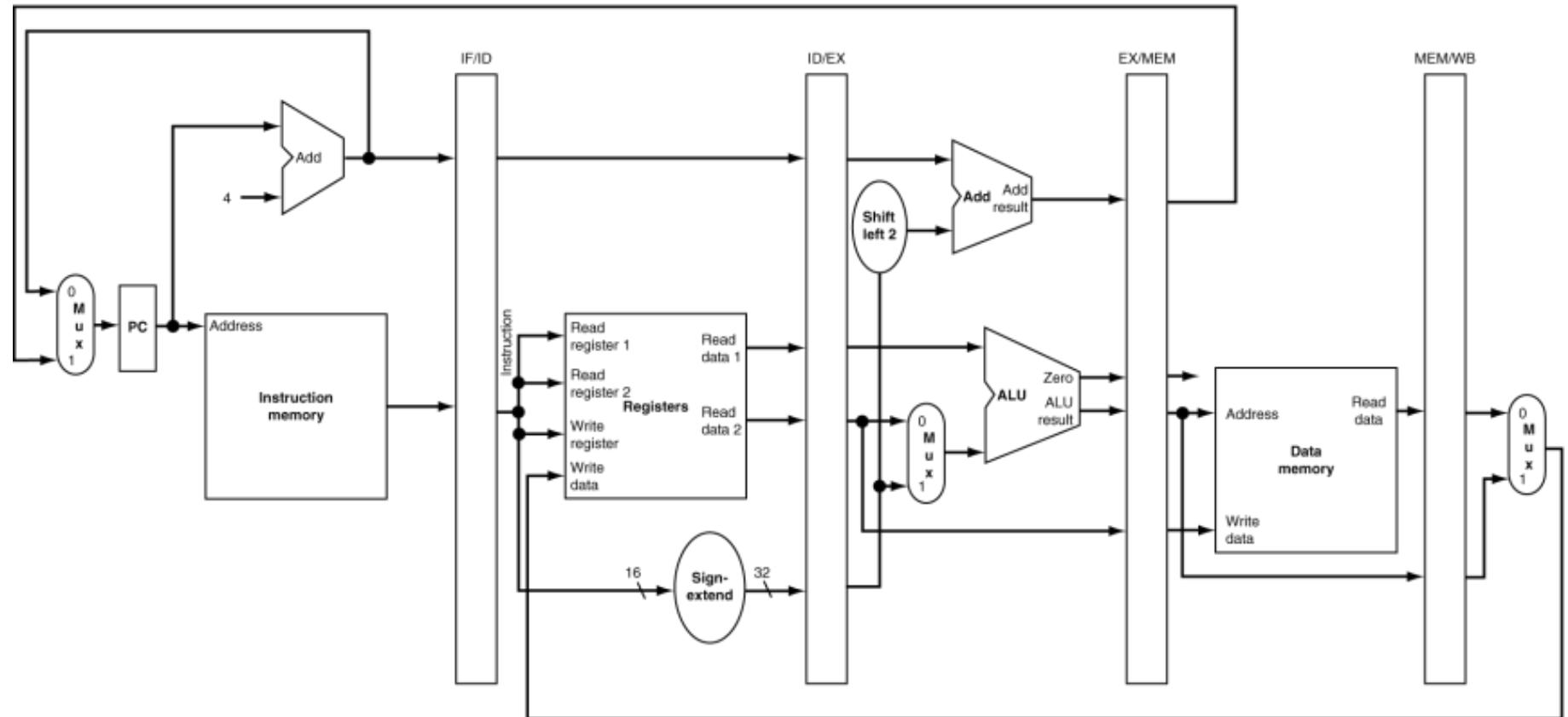


MEM cho lệnh Store



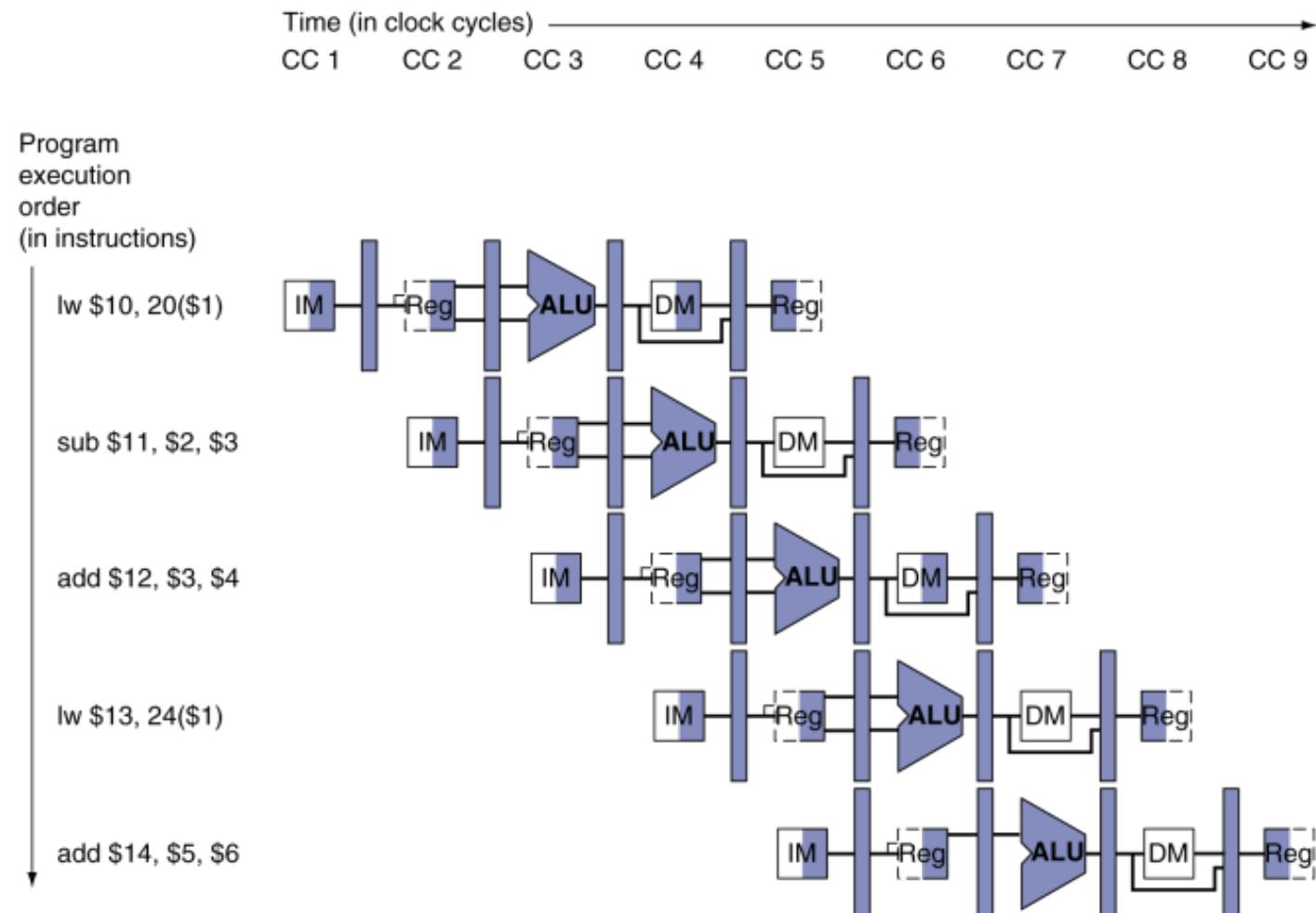
WB cho lệnh Store

SW
Write-back



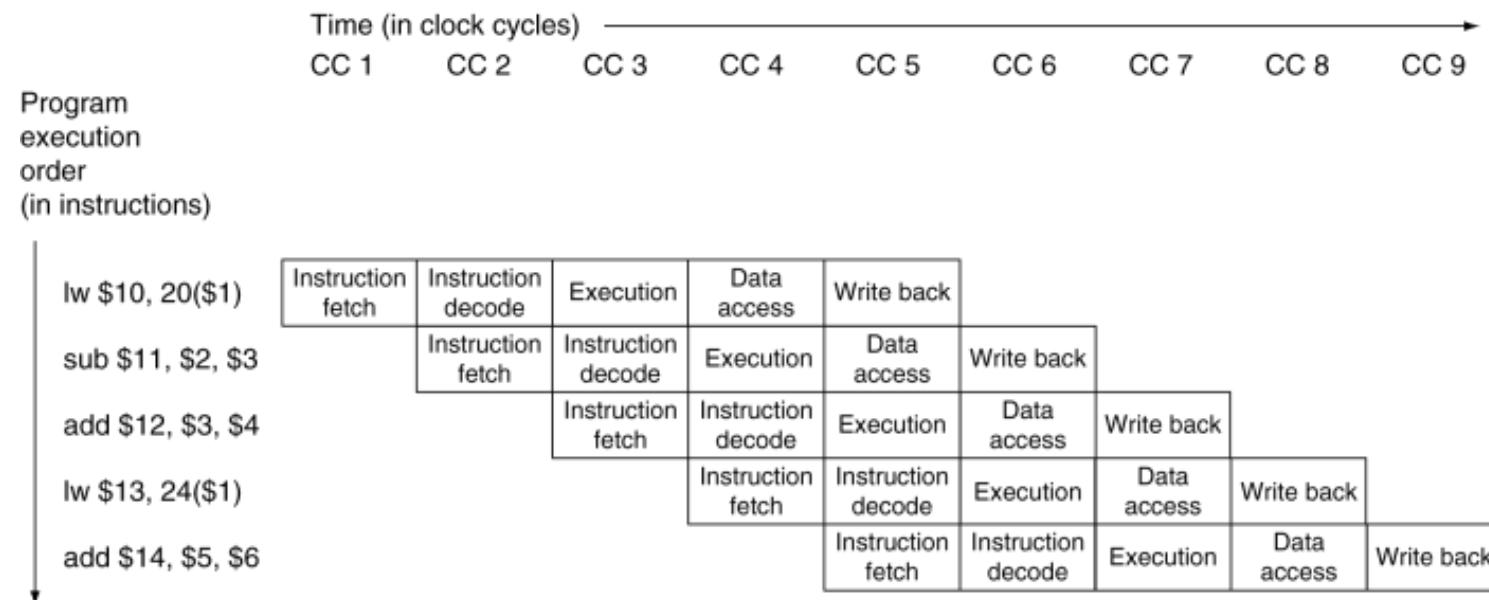
Giản đồ đường ống đa chu kỳ

■ Dạng tài nguyên được sử dụng



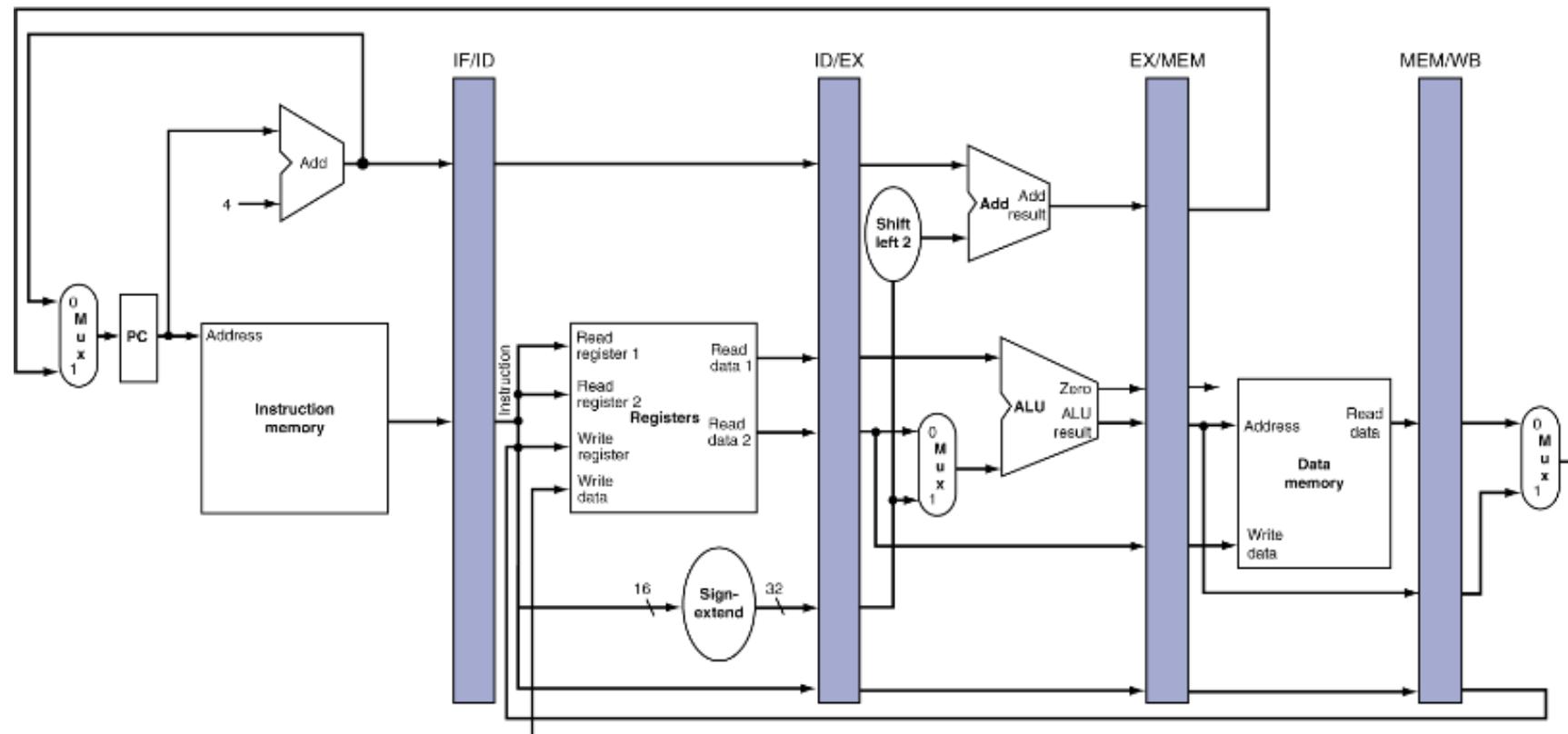
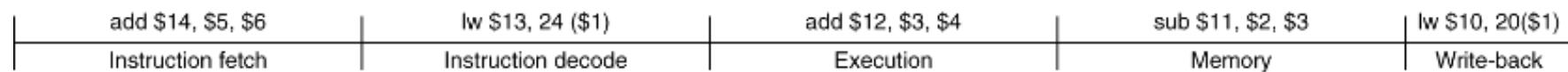
Giản đồ đường ống đa chu kỳ

■ Dạng truyền thống

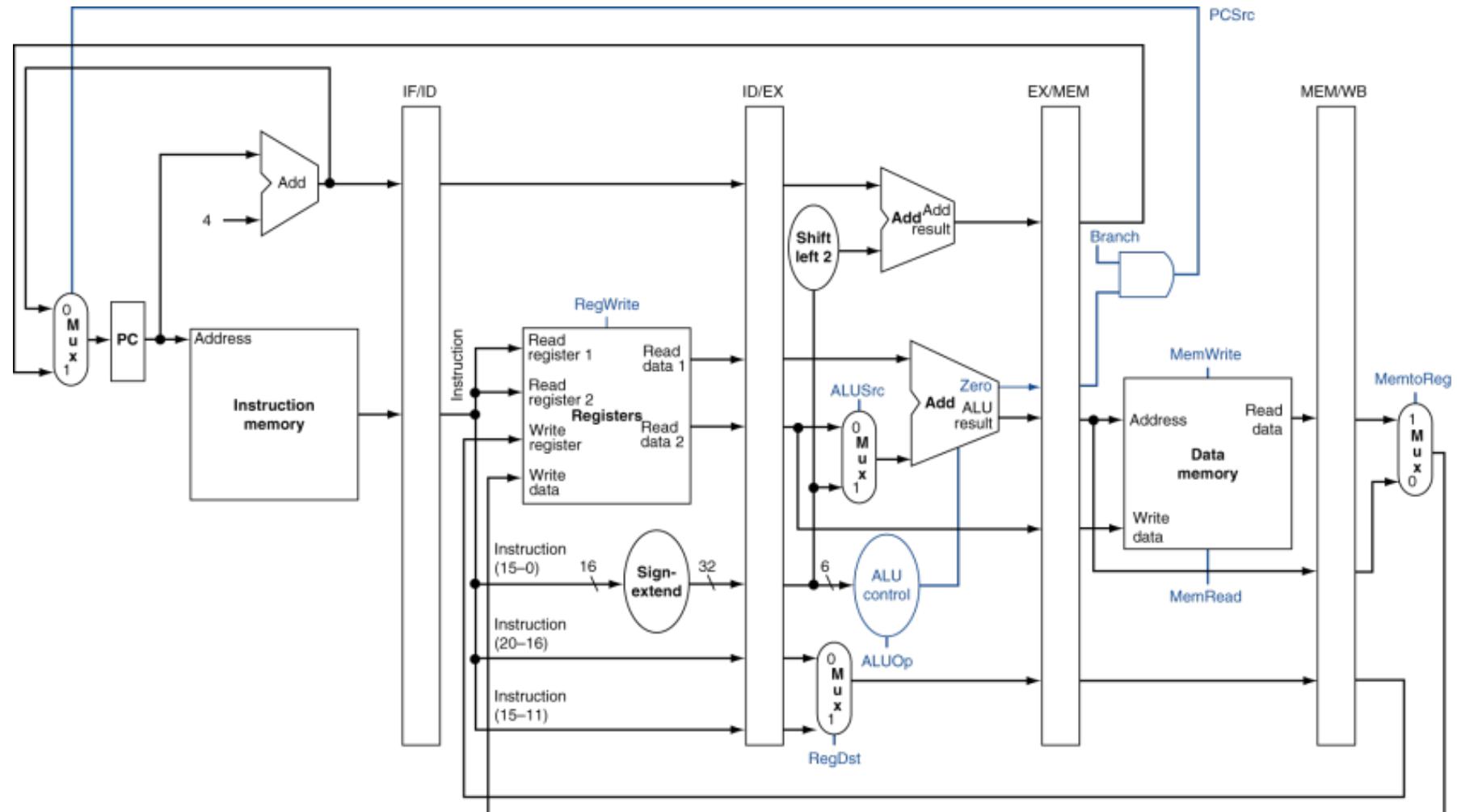


Giản đồ đường ống ở chu kỳ thứ 5

- Công đoạn của đường ống trong chu kỳ đã cho

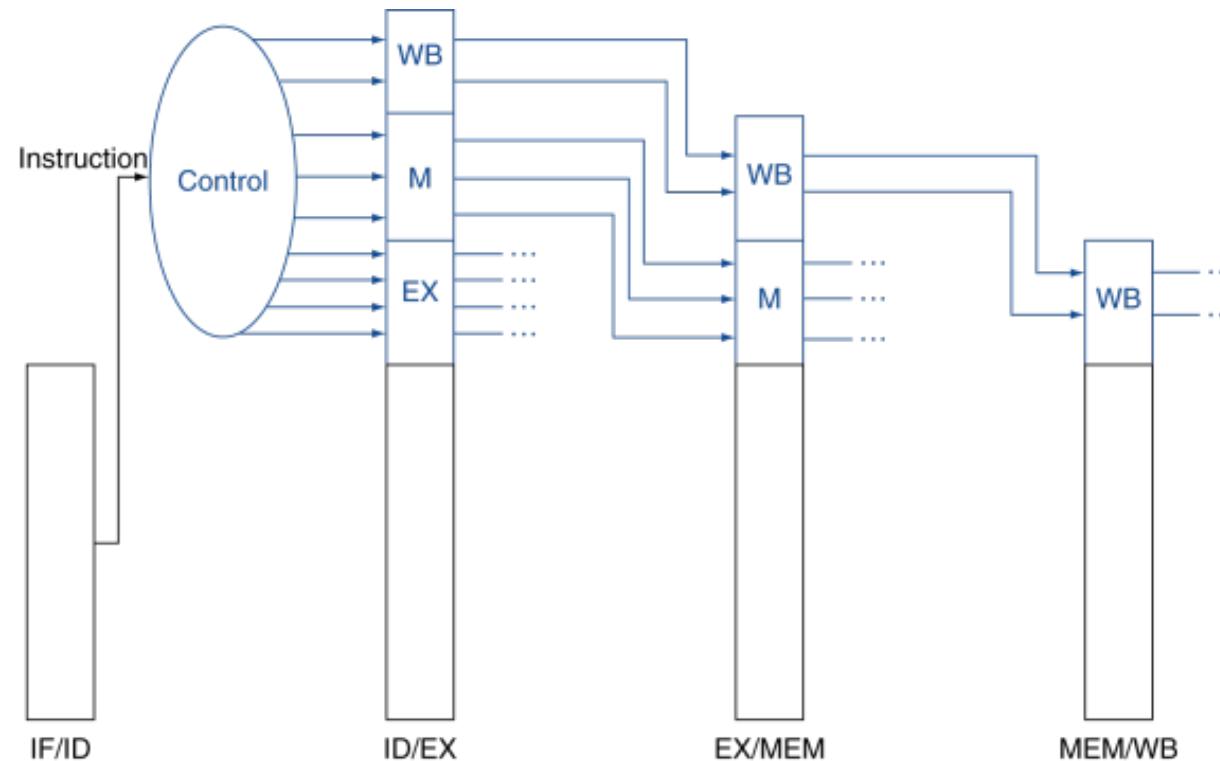


Điều khiển đường ống (dạng đơn giản)

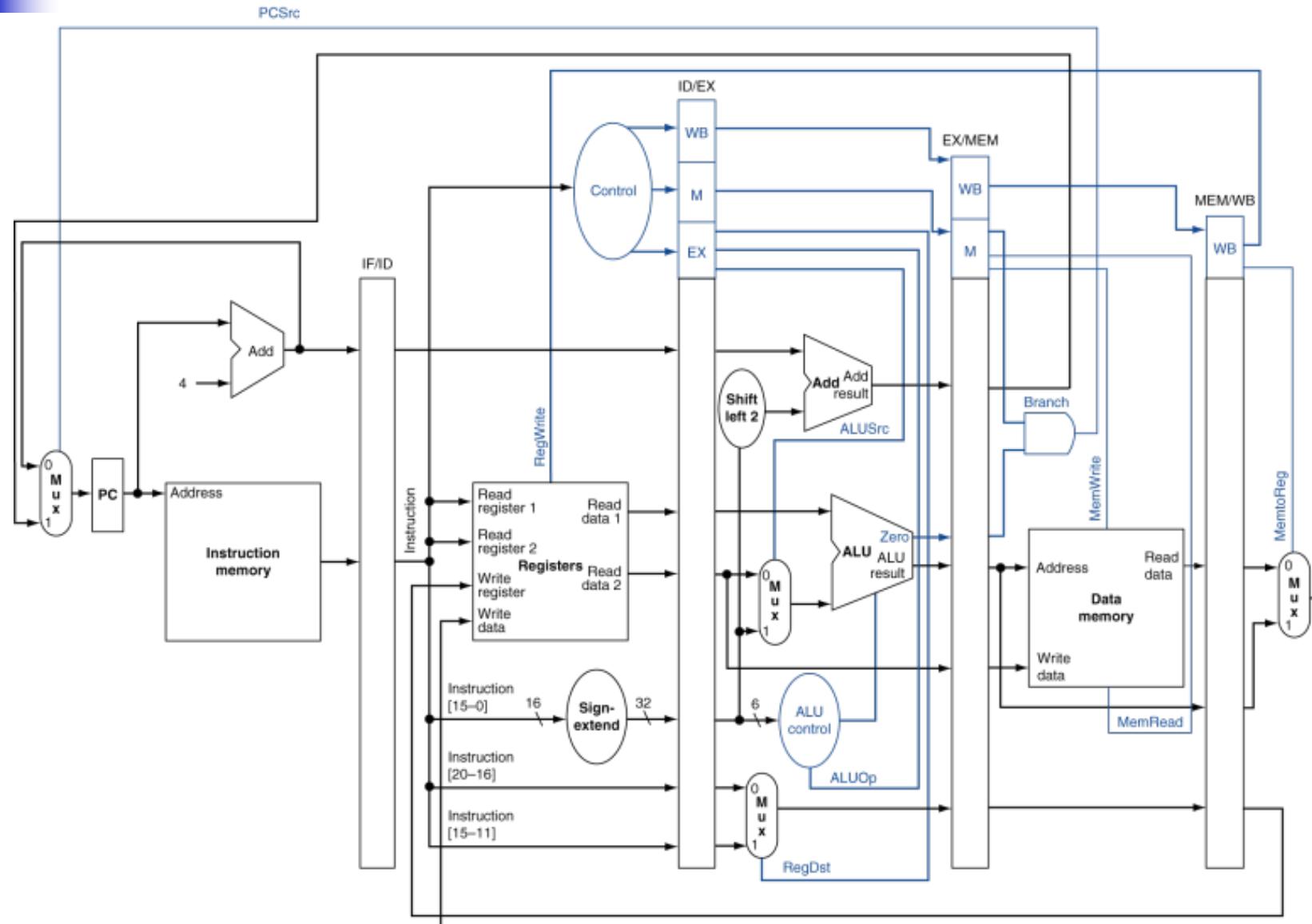


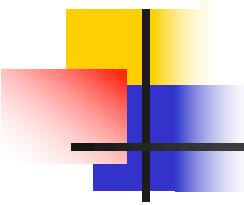
Điều khiển đường ống

- Các tín hiệu điều khiển được tạo ra từ lệnh
 - Như thực hiện đơn chu kỳ



Điều khiển đường ống





Hết chương 4

Kết thúc học phần