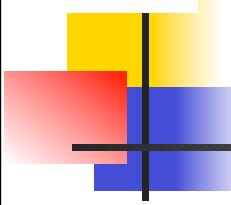


# Thiết kế bộ xử lý theo kiến trúc MIPS

Nguyễn Kim Khánh

Trường Đại học Bách khoa Hà Nội



# Nội dung

1. Thực hiện bộ xử lý MIPS cơ bản
2. Thiết kế khối datapath
3. Thiết kế control unit
4. Kỹ thuật đường ống lệnh

# 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

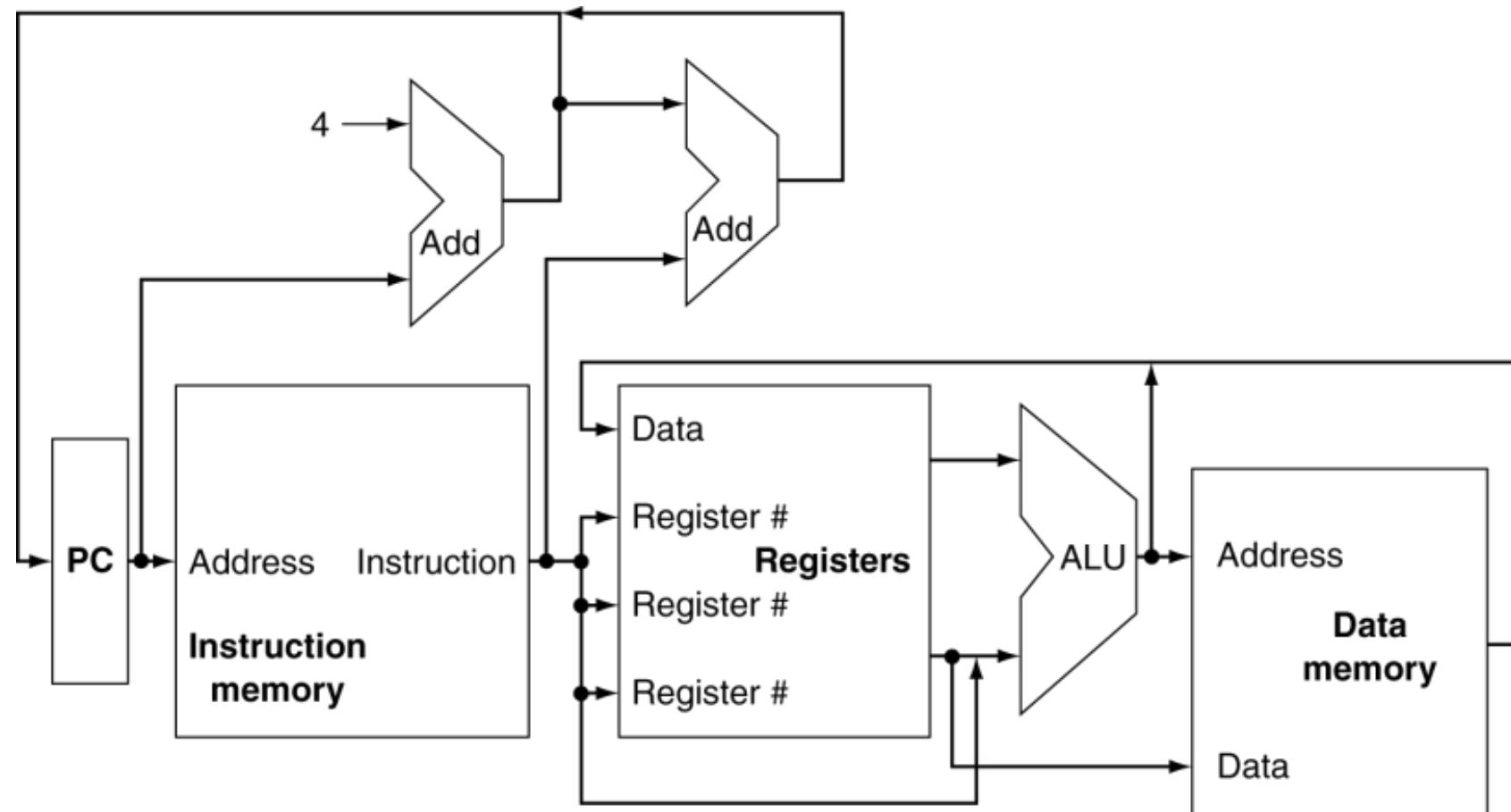
## Tổng quan quá trình thực hiện các lệnh (tiếp)

- 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

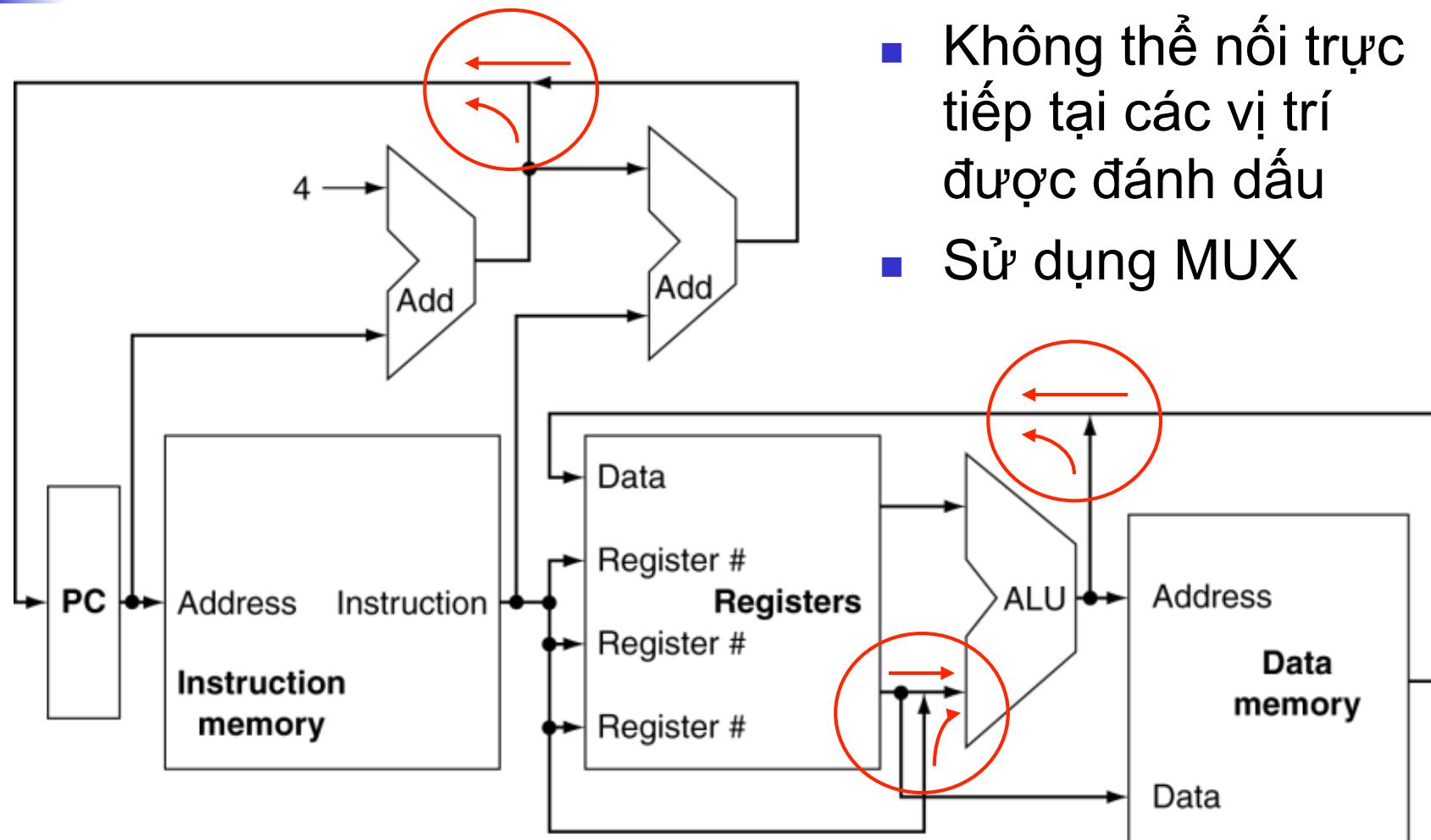
## Tổng quan quá trình thực hiện các lệnh (tiếp)

- 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)

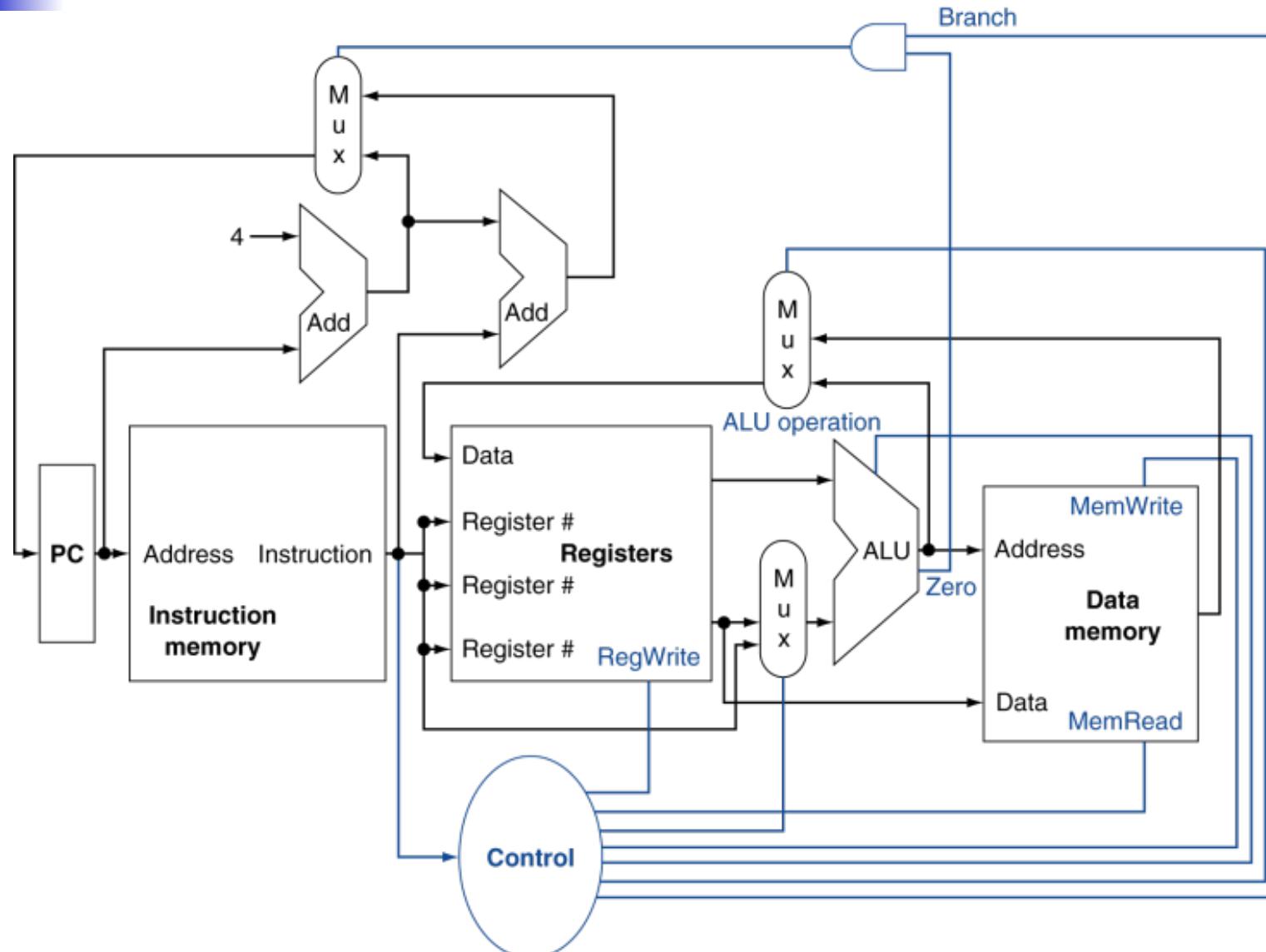
# Sơ đồ khái quát của bộ xử lý MIPS



# Sử dụng bộ chọn kênh (MUX)



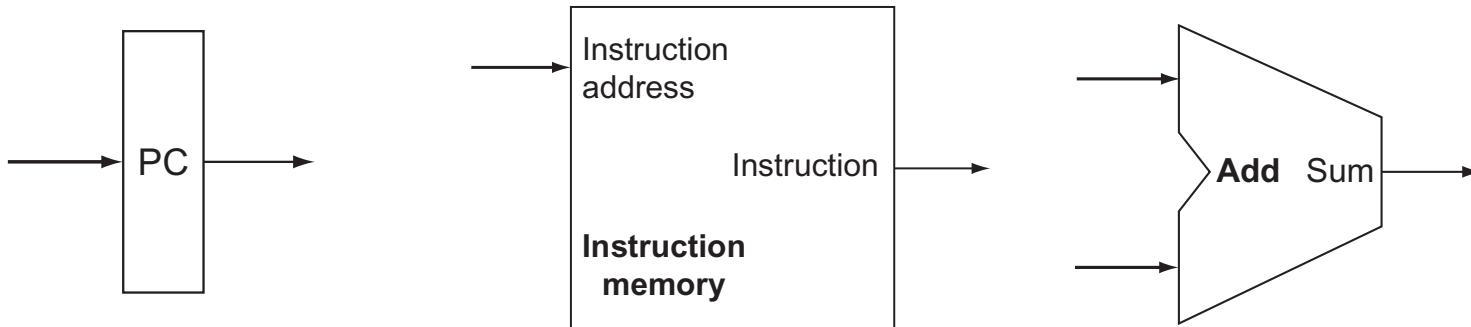
# Bộ xử lý với các đường điều khiển chính



## 2. Thiết kế khối Datapath

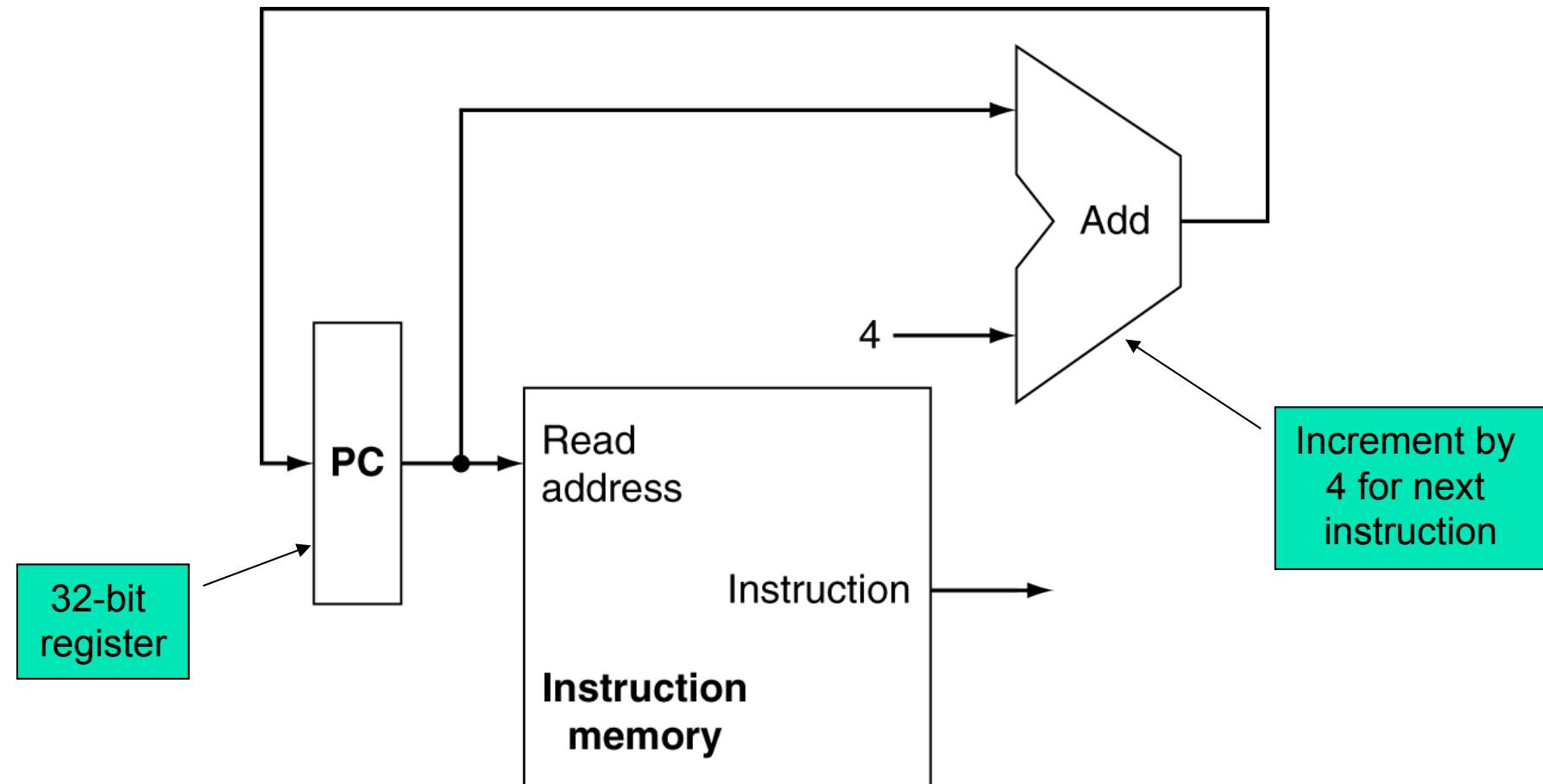
- Datapath: gồm các thành phần để xử lý dữ liệu và địa chỉ
  - Tập thanh ghi, ALUs, MUX's, bộ nhớ, ...
- Sẽ xây dựng tăng dần khối datapath cho MIPS

# Các thành phần để thực hiện nhận lệnh

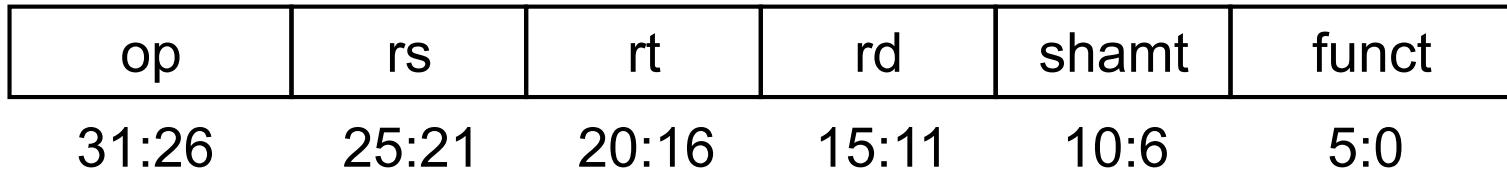


- Bộ đếm chương trình *PC*:
  - Thanh ghi 32-bit chứa địa chỉ của lệnh hiện tại
  - Địa chỉ khởi động = 0xBFC0 0000
- Bộ nhớ lệnh (*Instruction memory*):
  - Chứa các lệnh của chương trình
  - Khi có địa chỉ lệnh từ PC đưa đến thì lệnh được đọc ra
- Bộ cộng (*Add*): được sử dụng tăng nội dung PC thêm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

# Thực hiện phần nhận lệnh

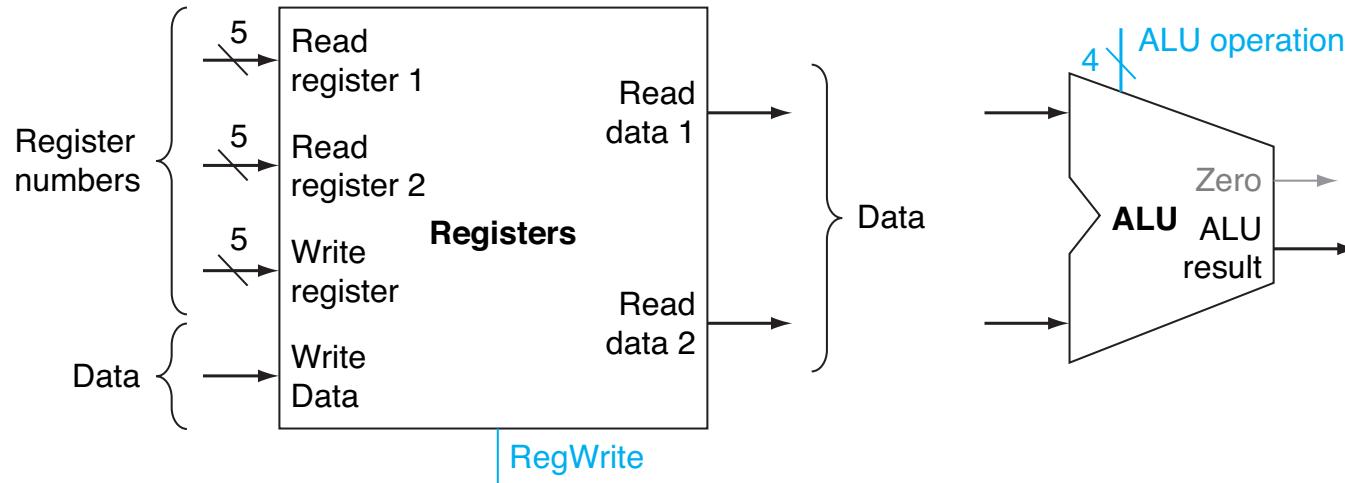


# Thực hiện lệnh số học/logic kiểu R



- 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

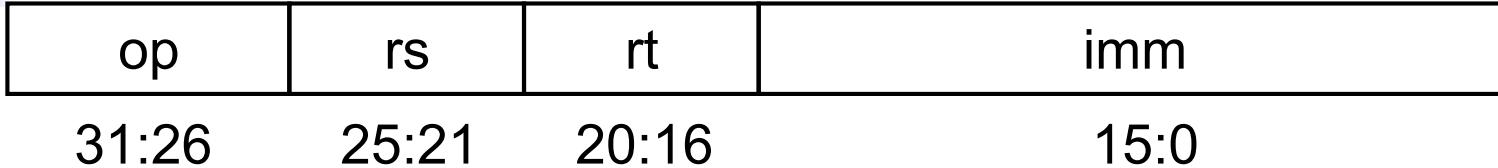


- Tập thanh ghi (*Registers*): có 32 thanh ghi 32-bit, mỗi thanh ghi được xác định bởi số hiệu 5-bit
  - *Read register 1, Read register 2*: các đầu vào để chọn các thanh ghi cần đọc
  - *Write register*: đầu vào để chọn thanh ghi cần ghi
  - *Read data 1, Read data 2*: hai đầu ra dữ liệu đọc từ thanh 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

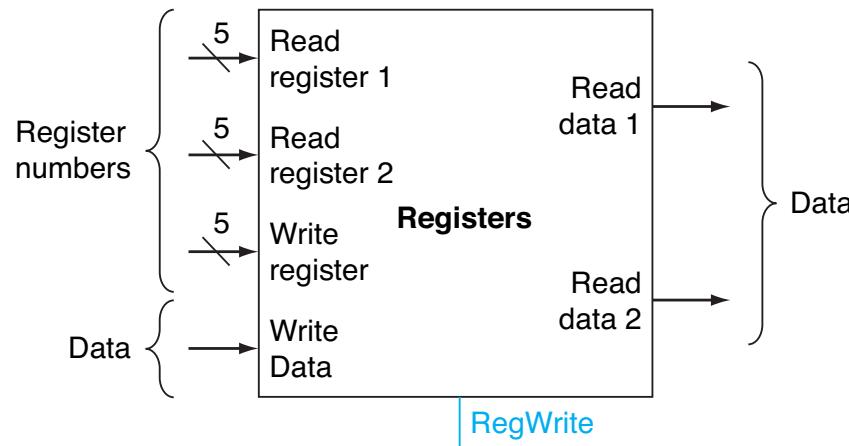
# Thực hiện các lệnh lw/sw



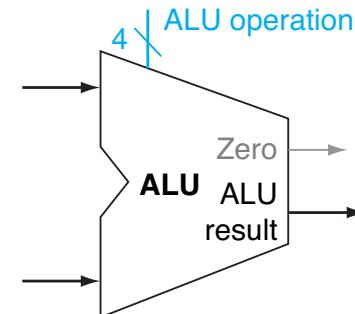
- bits 31:26 là mã thao tác
  - 100011 (35) với lệnh lw
  - 101011 (43) với lệnh sw
- bits 25:21: số hiệu thanh ghi cơ sở rs
- bits 20:16: số hiệu thanh ghi rt
  - thanh ghi đích với lệnh lw
  - thanh ghi nguồn với lệnh sw
- bits 15:0: hằng số imm có giá trị trong dải  $[-2^{15}, +2^{15} - 1]$
- Địa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

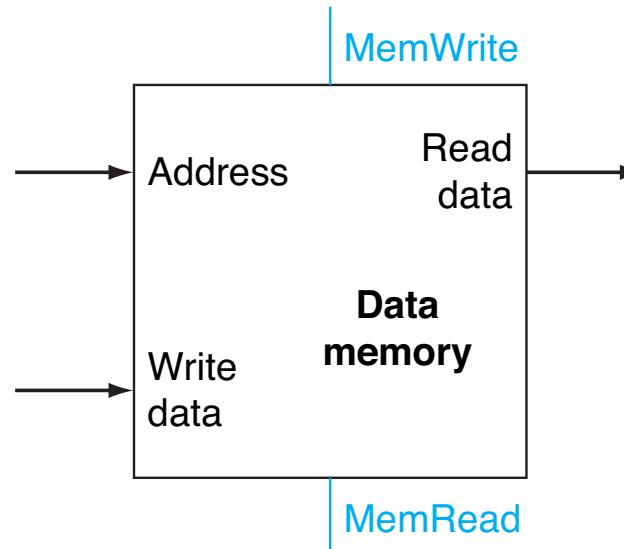
Tập thanh ghi



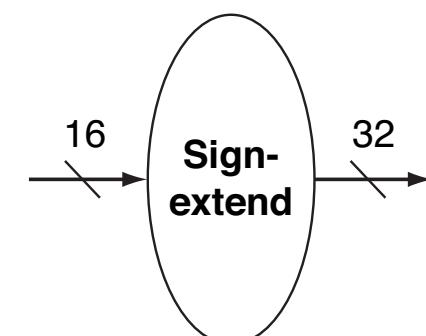
Bộ số học/logic



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**
- Hằng 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

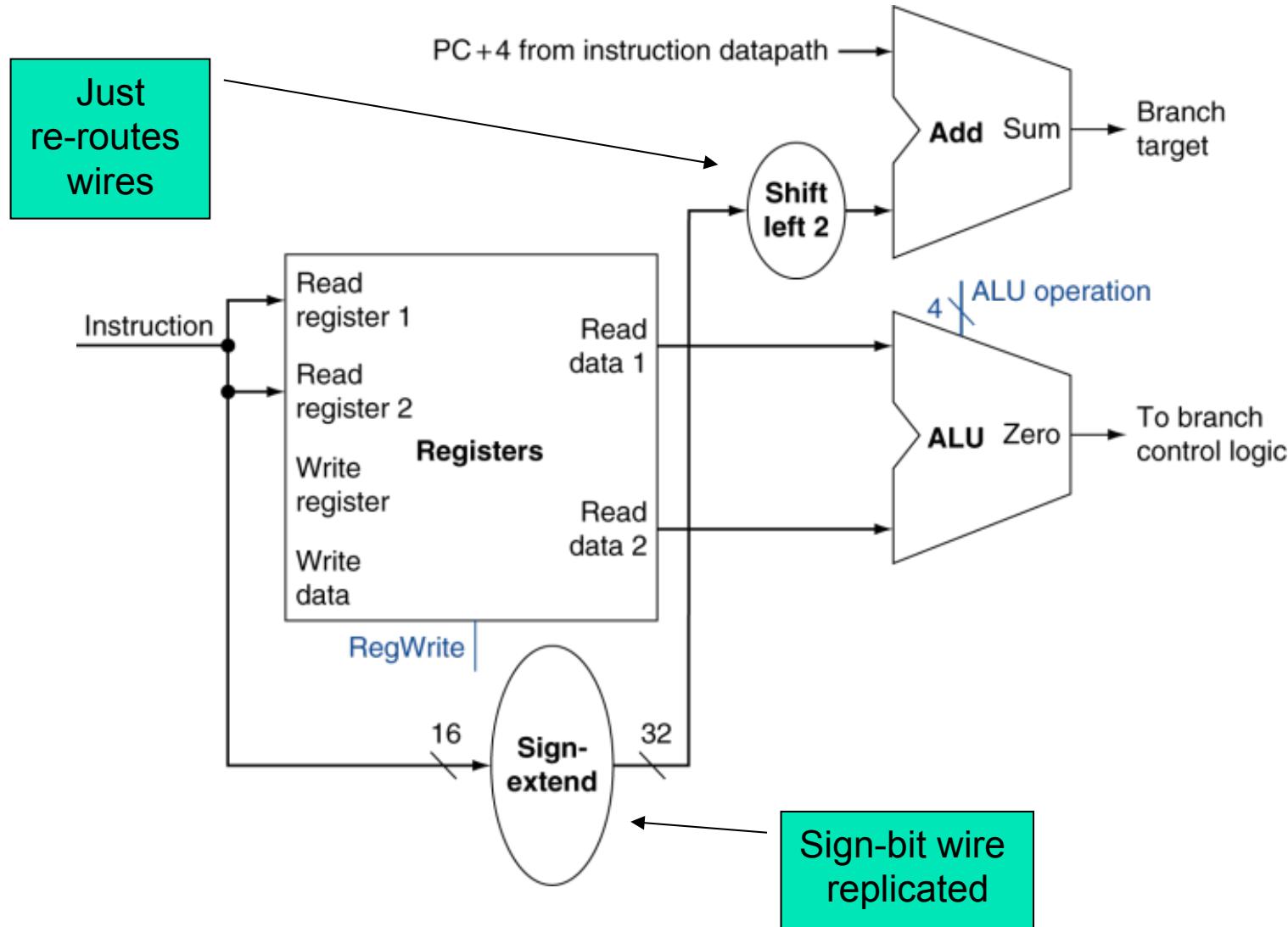
- Số hiệu thanh ghi **rs** đưa đến đầu vào **Read register 1** để chọn thanh ghi cơ sở **rs**, nội dung **rs** được đưa ra đầu ra **Read Data 1**, rồi chuyển đến **ALU**
- Hằng số imm 16-bit đưa đến bộ **Sign-extend** để mở rộng thành 32-bit, rồi chuyển đến **ALU**
- **ALU** cộng hai giá trị trên đưa 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



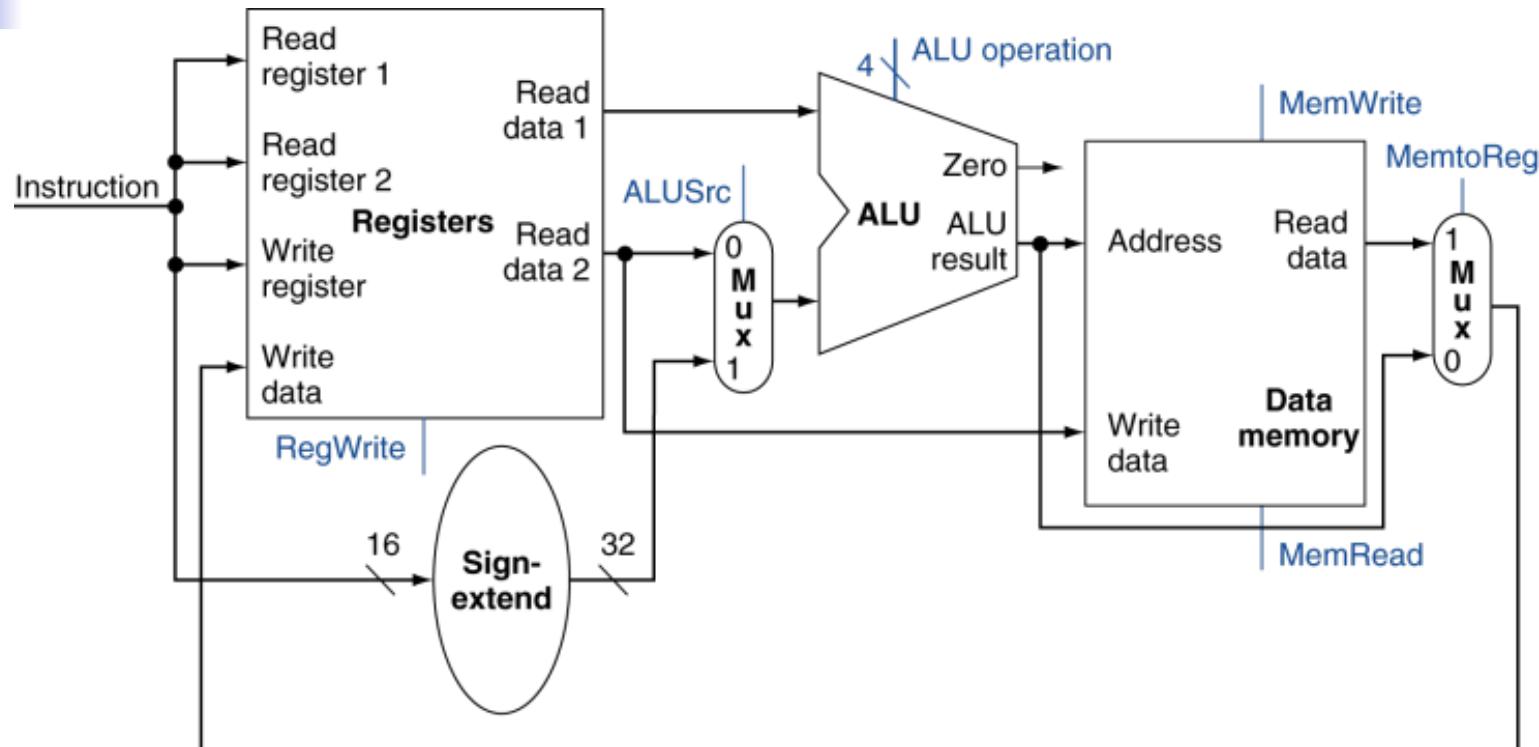
# Thực hiện lệnh Branch (beq, bne)

- 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

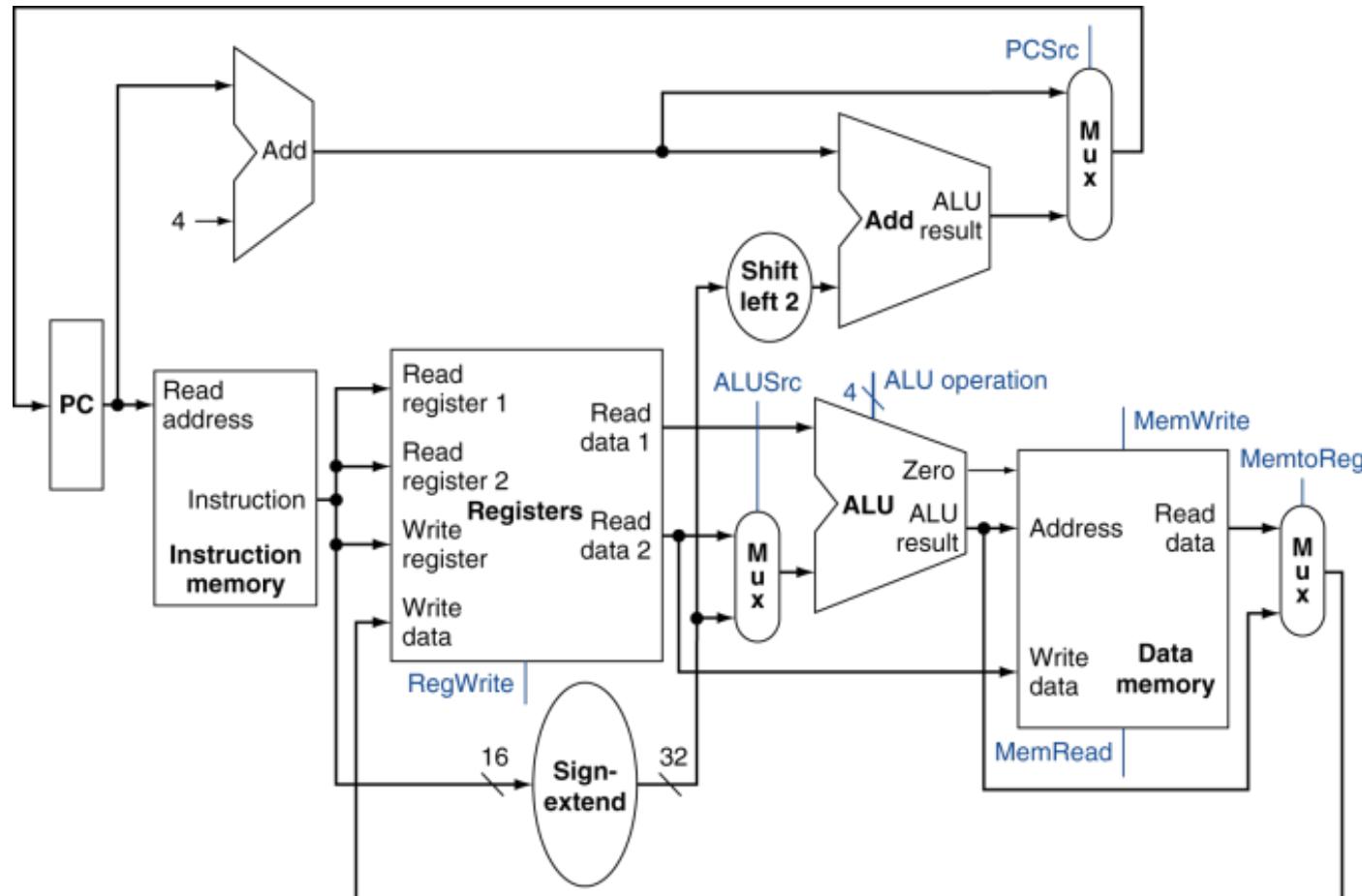
- Datapath cho các lệnh thực hiện trong 1 chu kỳ
  - Mỗi phần tử của datapath chỉ có thể làm một chức năng trong mỗi chu kỳ
  - Do đó, cần tách rời bộ nhớ lệnh và bộ nhớ dữ liệu
- Sử dụng các bộ chọn kênh để chọn dữ liệu nguồn cho các lệnh khác nhau

# 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:  $PC \leftarrow (PC + 4) + (\text{hằng số imm đã mở rộng thành 32-bit} \ll 2)$

### 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

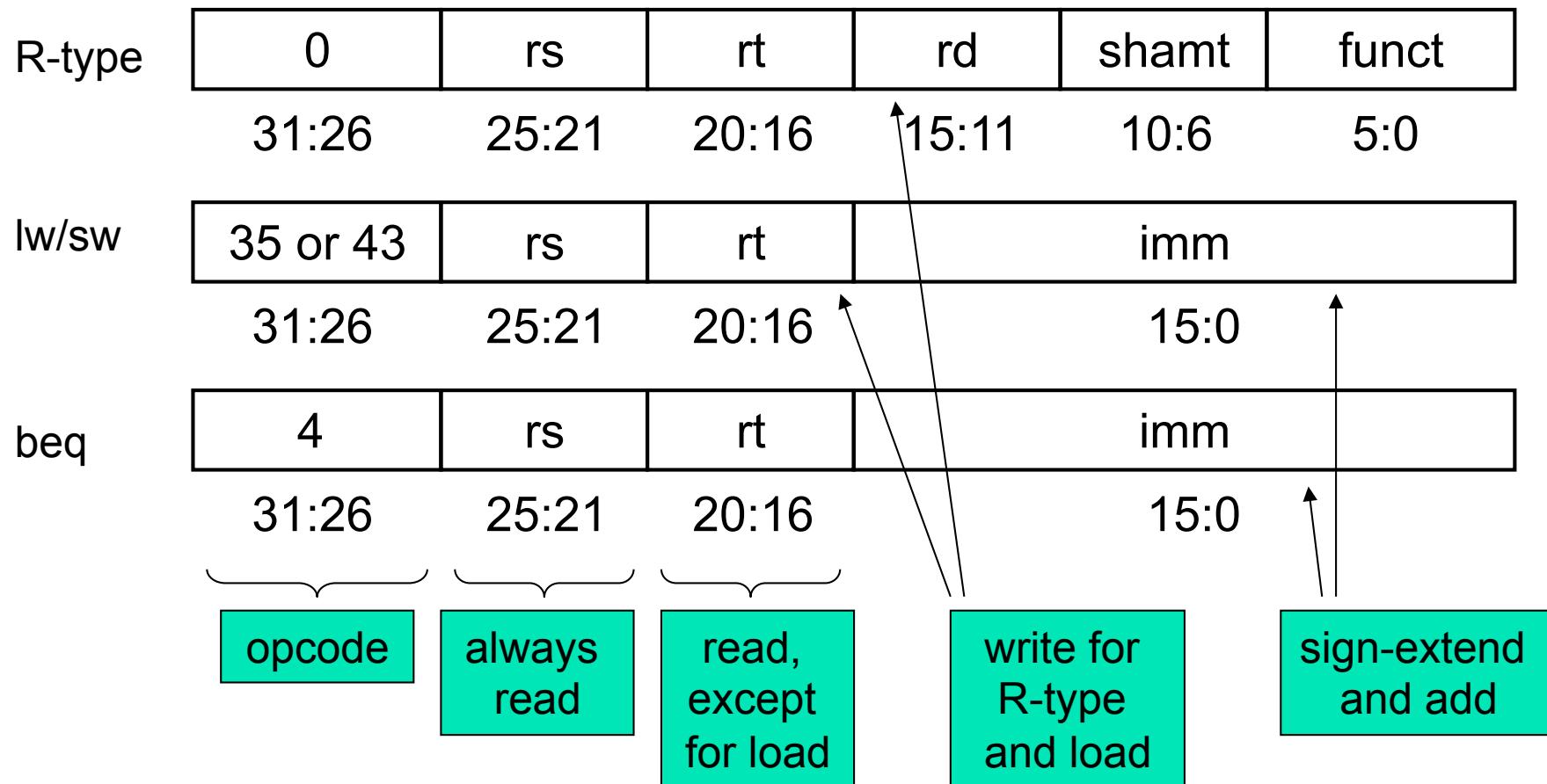
# Tín hiệu điều khiển ALU

- 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

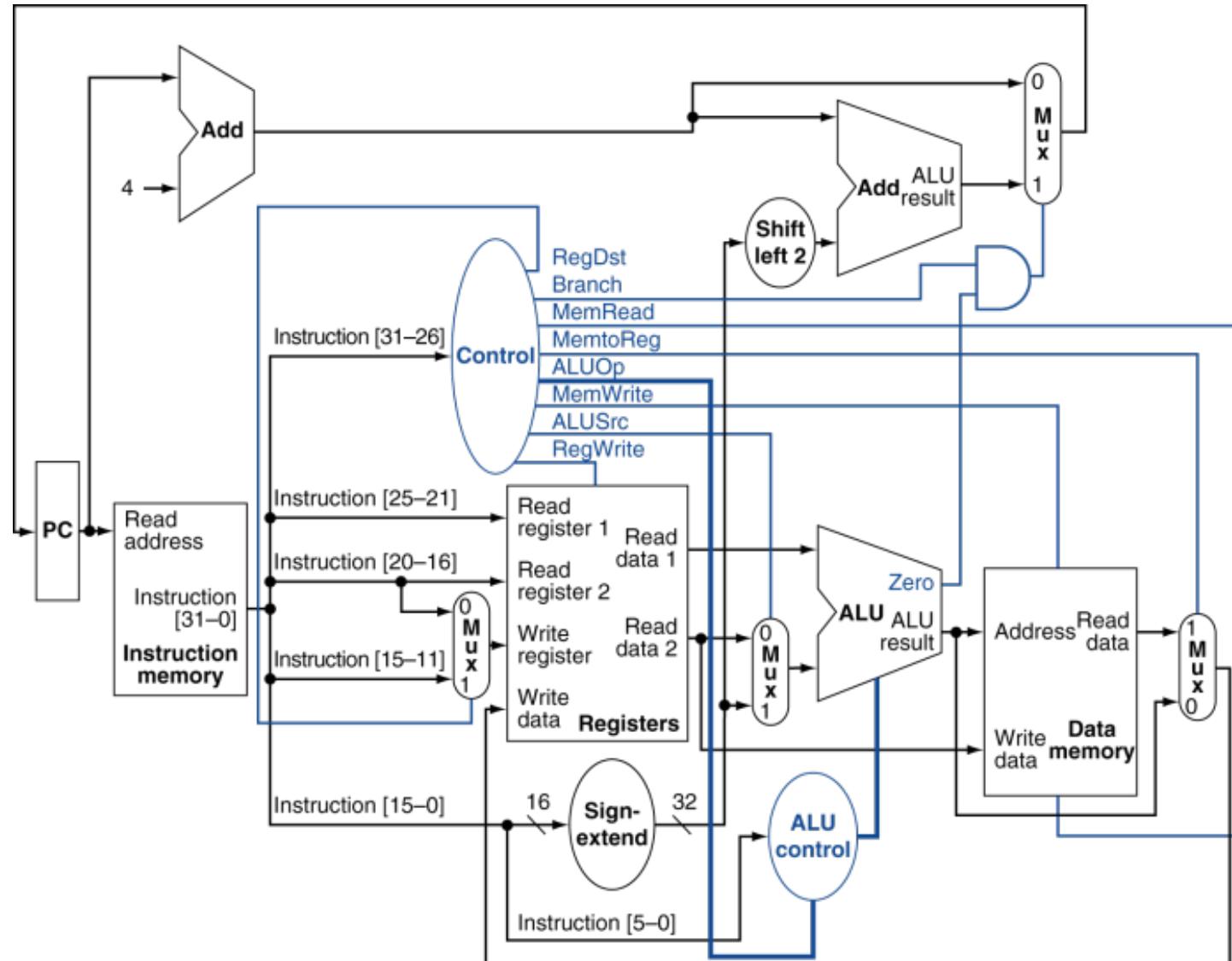
Opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than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

# Thiết kế bộ điều khiển chính

## ■ Các tín hiệu điều khiển được tạo ra từ lệnh



# Datapath và Control Unit



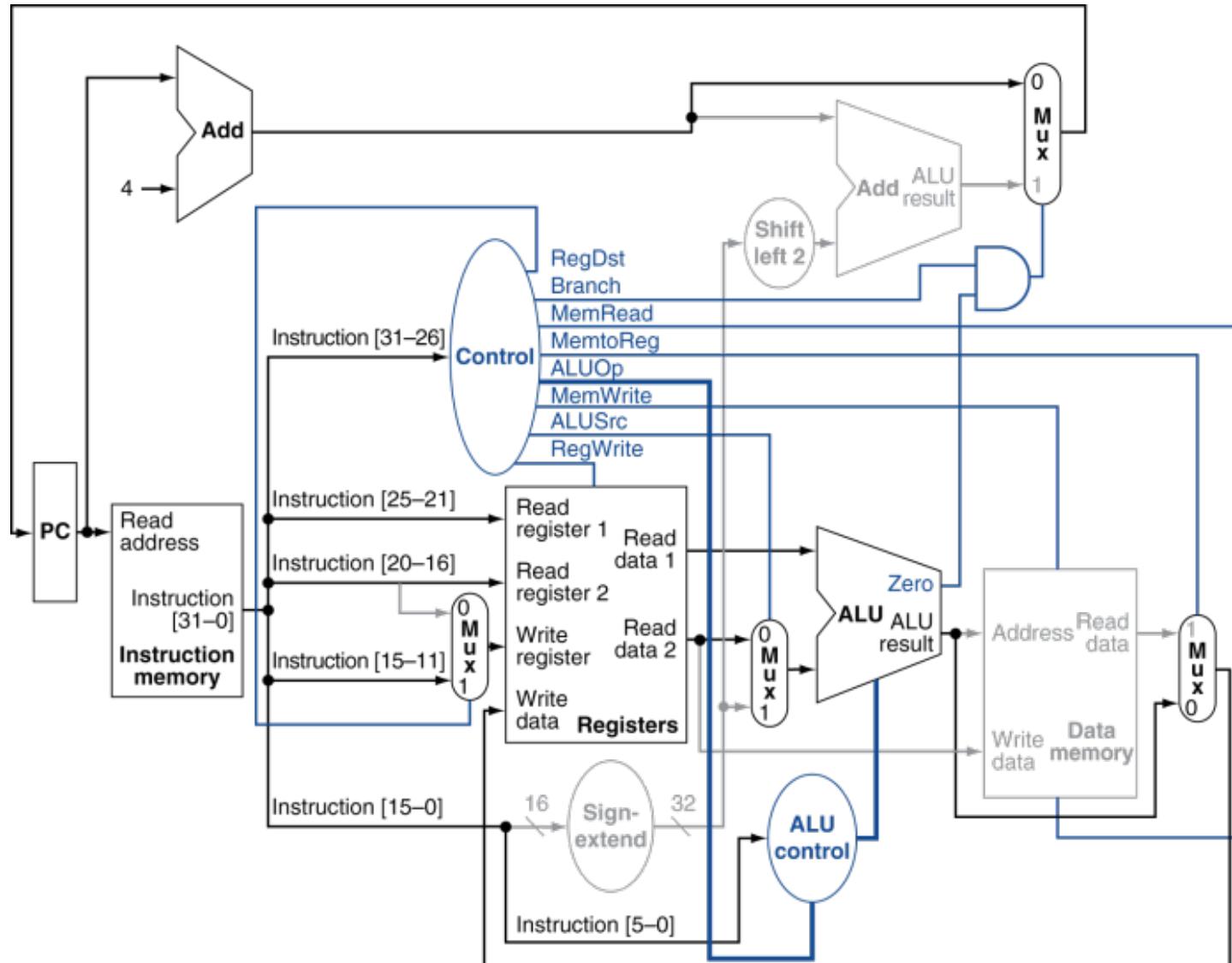
# Các tín hiệu điều khiển

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	$PC \leftarrow PC + 4$	$PC \leftarrow \text{địa chỉ đích}$

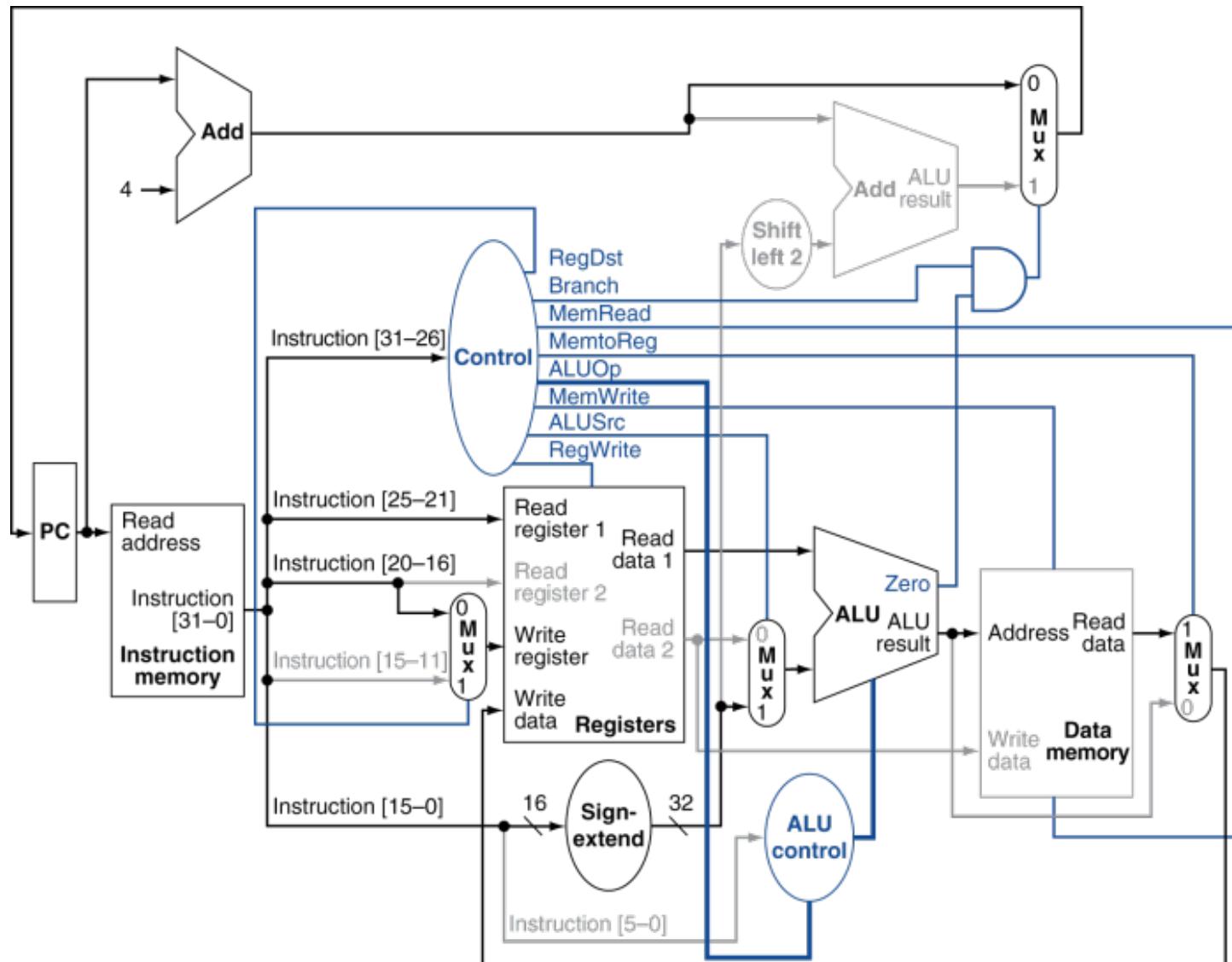
# Các tín hiệu điều khiển (tiếp)

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

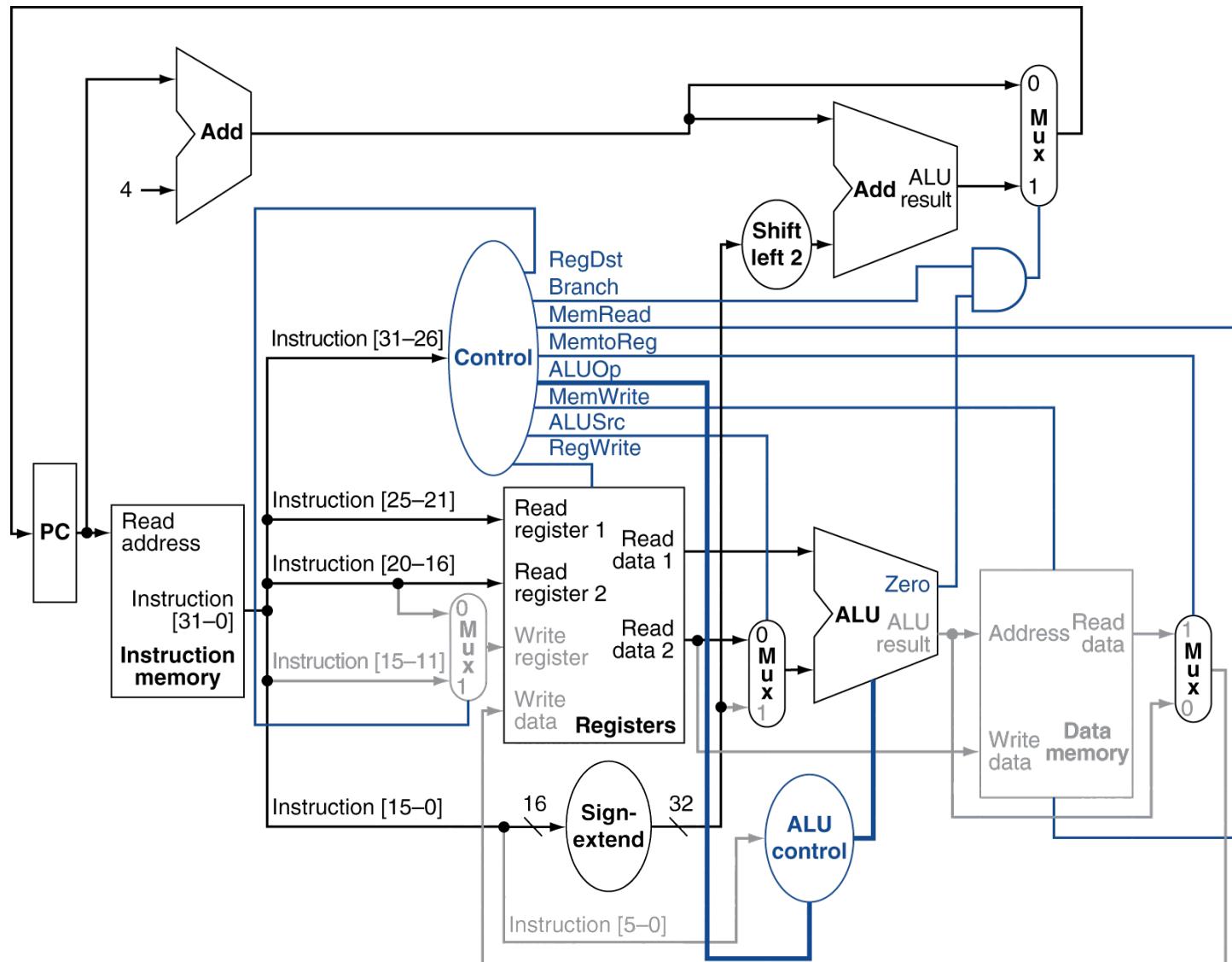
# Thực hiện lệnh số học/logic kiểu R



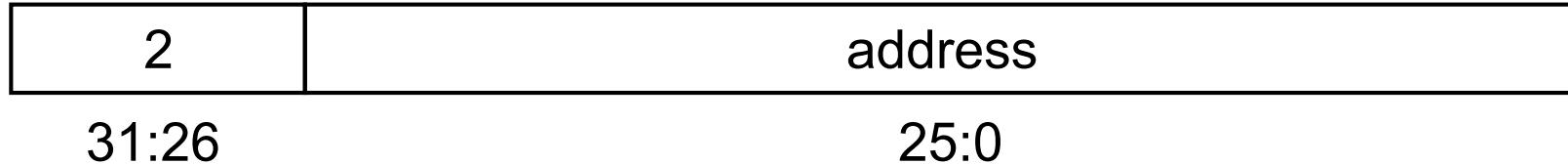
# Thực hiện lệnh Load



# Thực hiện lệnh beq

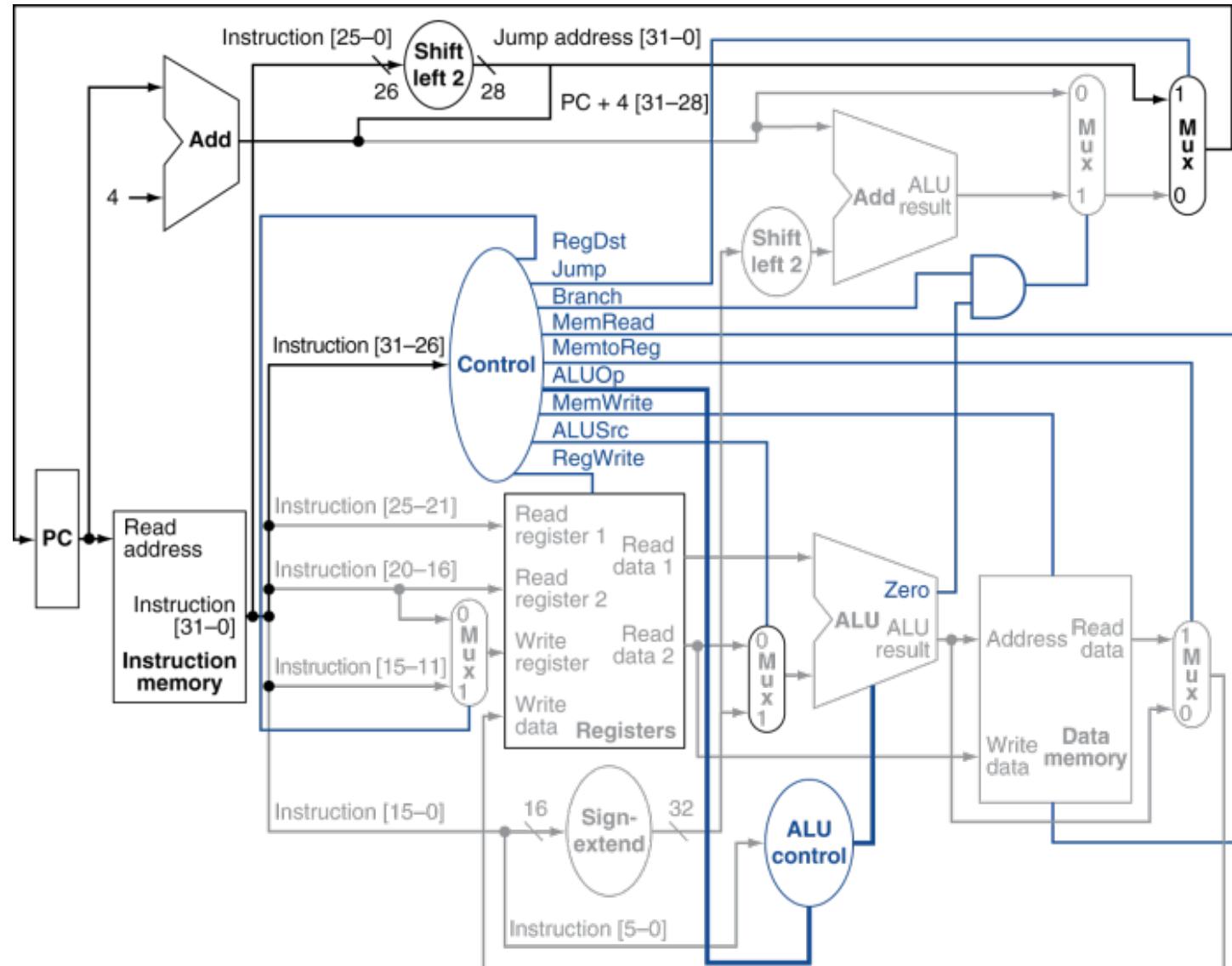


# Thực hiện lệnh Jump



- 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. Đường ống lệnh ở MIPS

5 công đoạn:

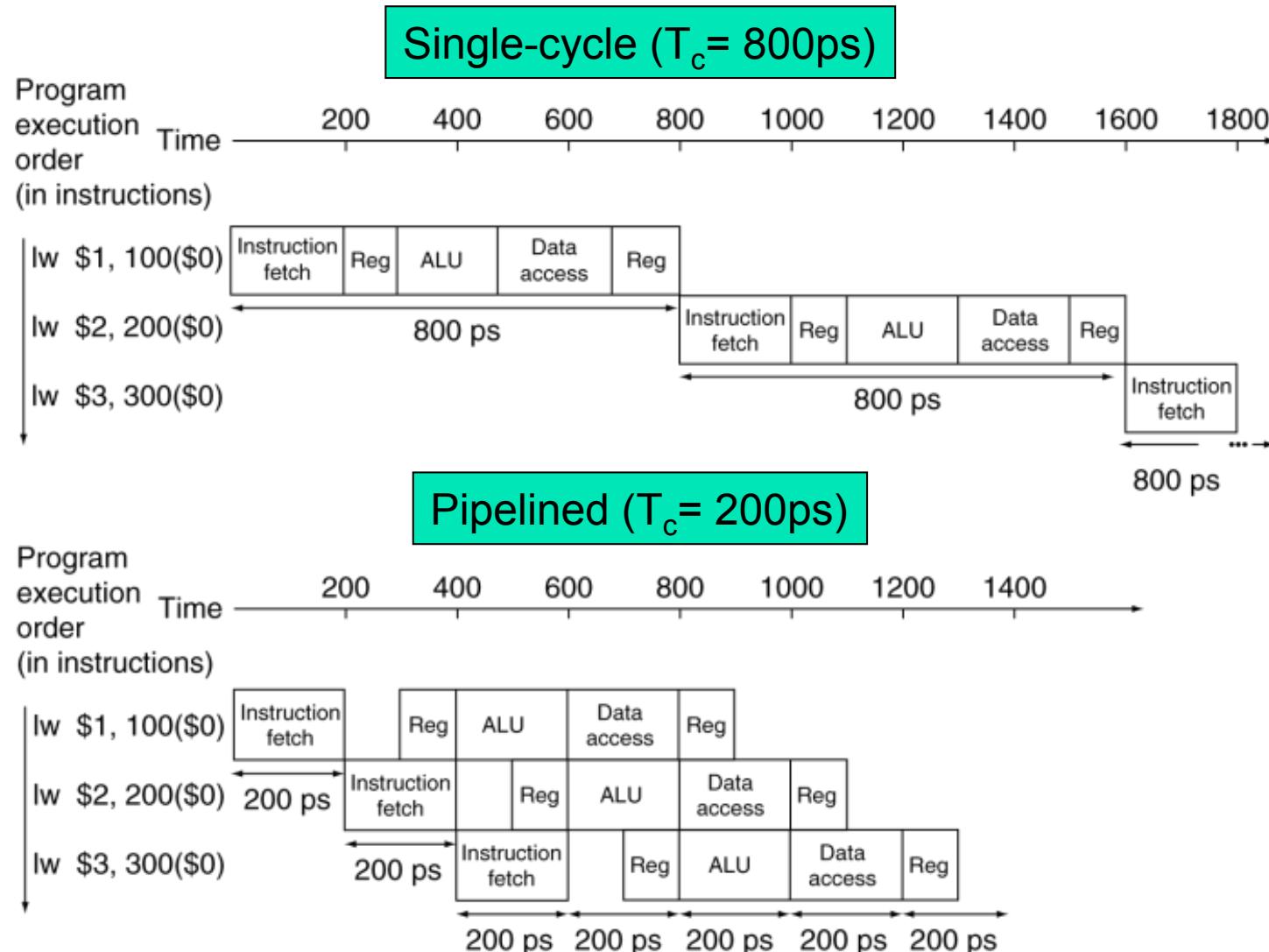
1. IF: Instruction fetch from memory – Nhận lệnh từ bộ nhớ
2. ID: Instruction decode & register read – Giải mã lệnh và đọc thanh ghi
3. EX: Execute operation or calculate address – Thực hiện thao tác hoặc tính toán địa chỉ
4. MEM: Access memory operand – Truy nhập toán hạng bộ nhớ
5. WB: Write result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

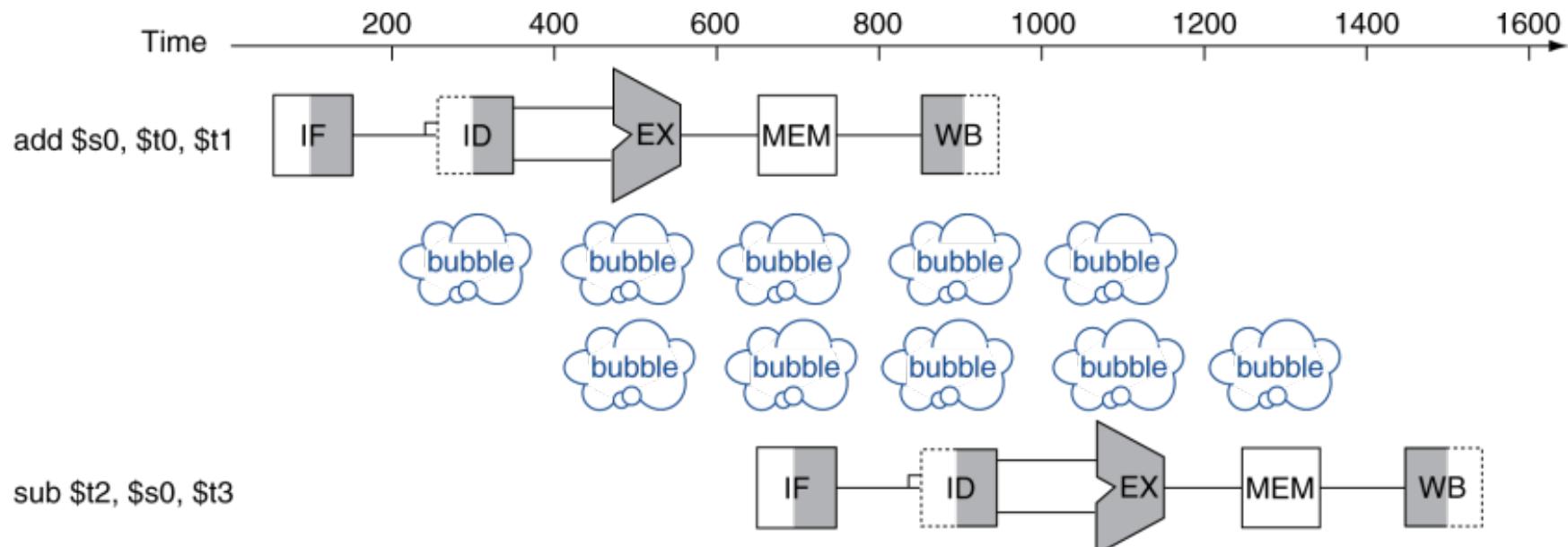
# Hazard cấu trúc

- 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

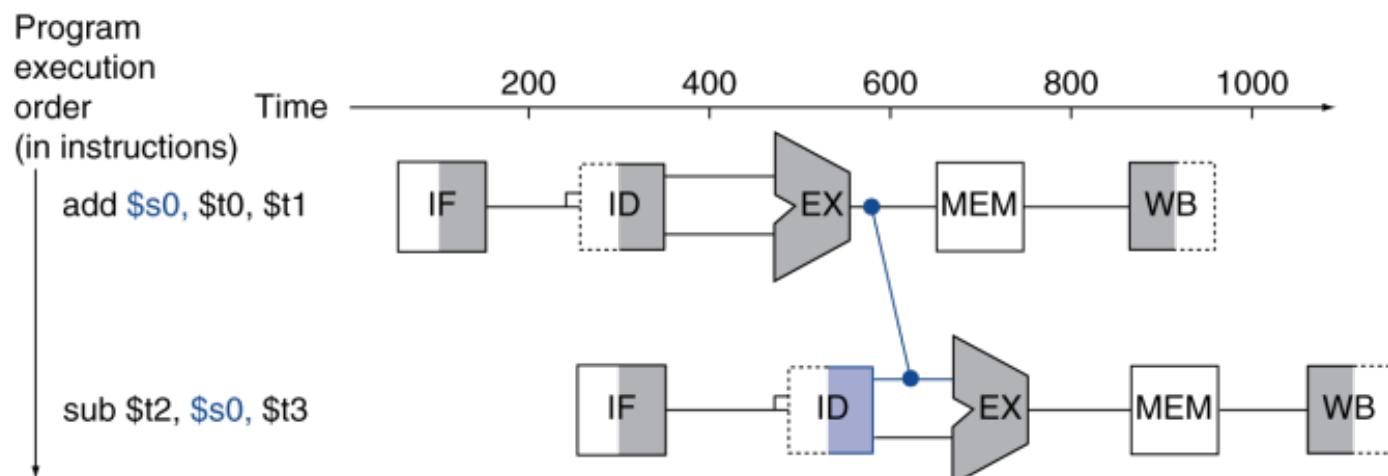
- Lệnh phụ thuộc vào việc hoàn thành truy cập dữ liệu của lệnh trước đó

add \$s0, \$t0, \$t1  
sub \$t2, \$s0, \$t3



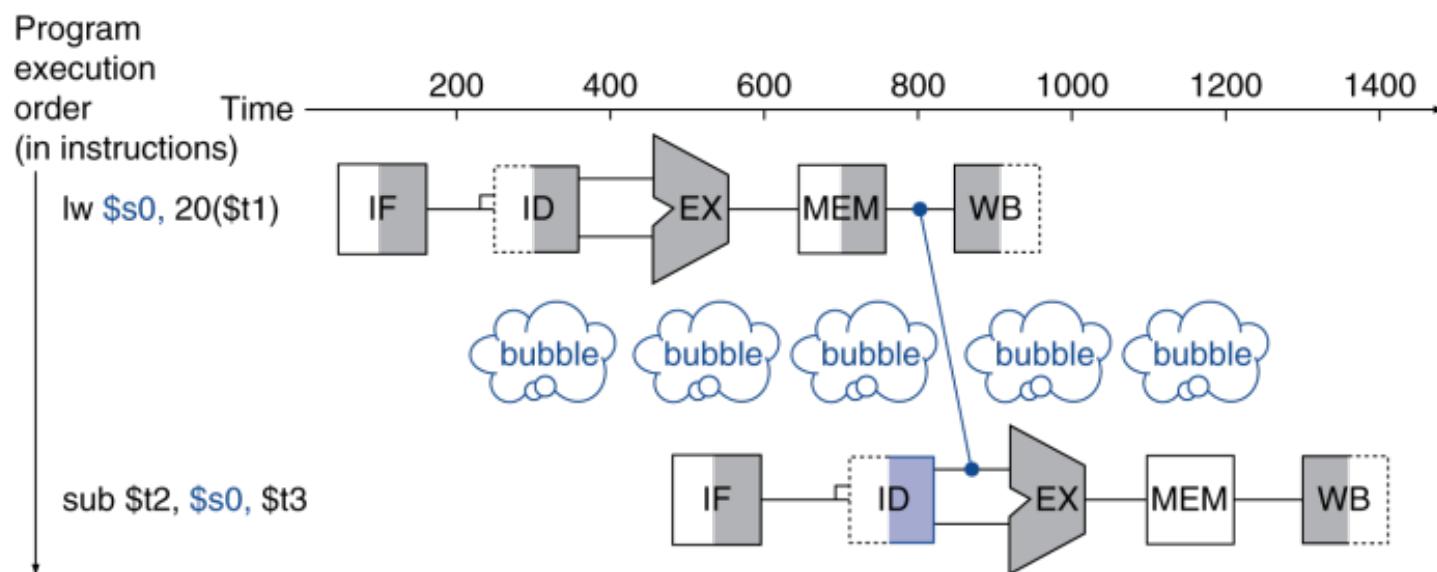
# Forwarding (gửi vượt trước)

- 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

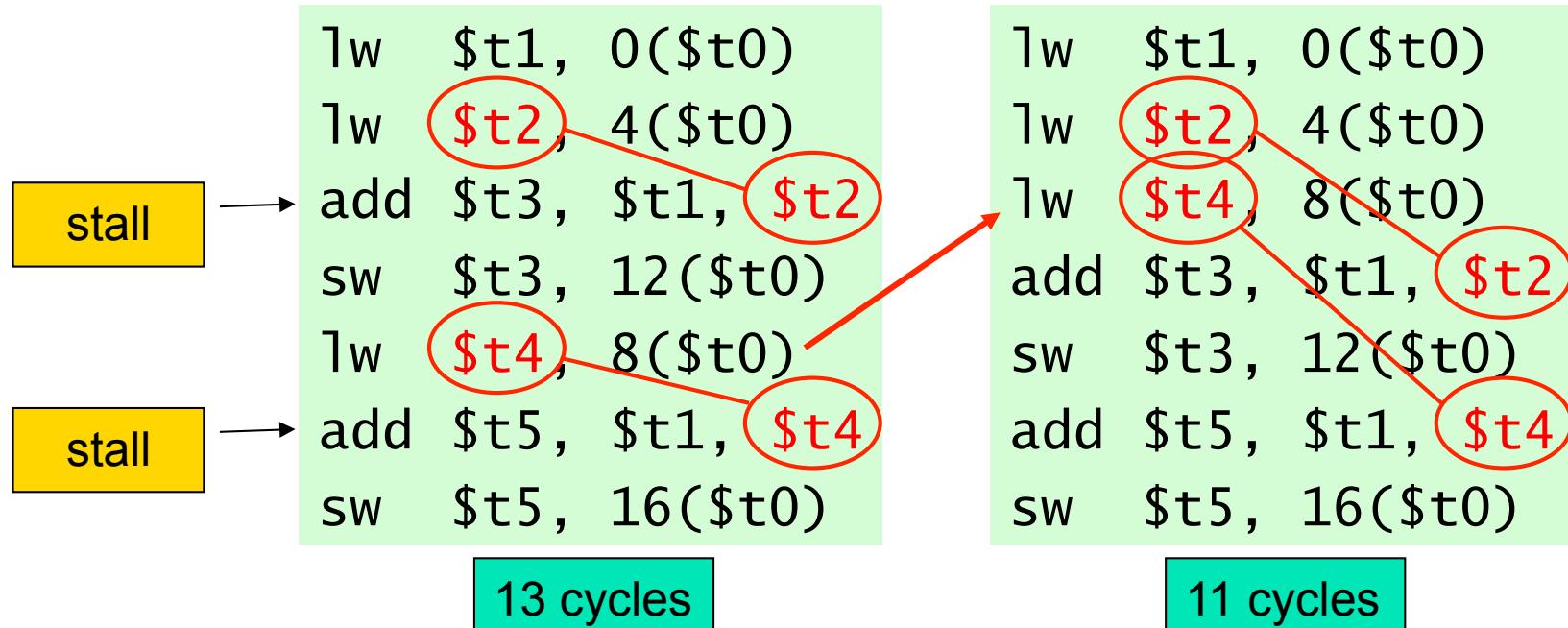
- Không phải luôn luôn có thể tránh trì hoãn bằng cách forwarding
  - Nếu giá trị chưa được tính khi cần thiết
  - Không thể chuyển ngược thời gian
  - Cần chèn bước trì hoãn (stall hay bubble)



# 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:

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

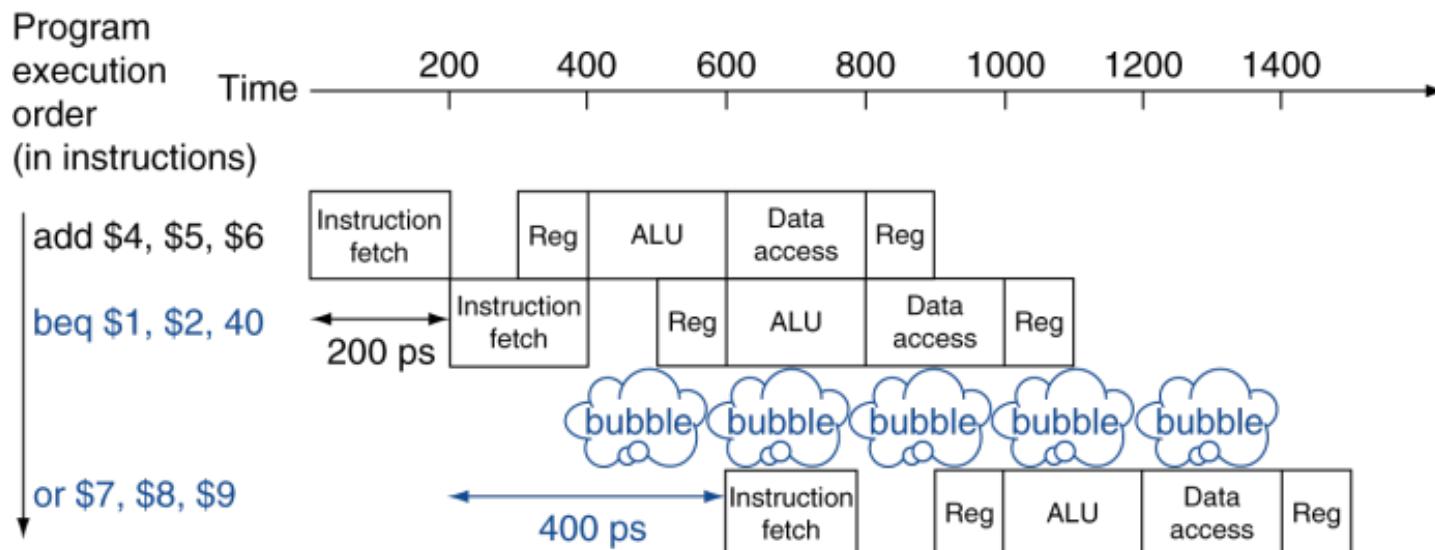


# Hazard điều khiển

- 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

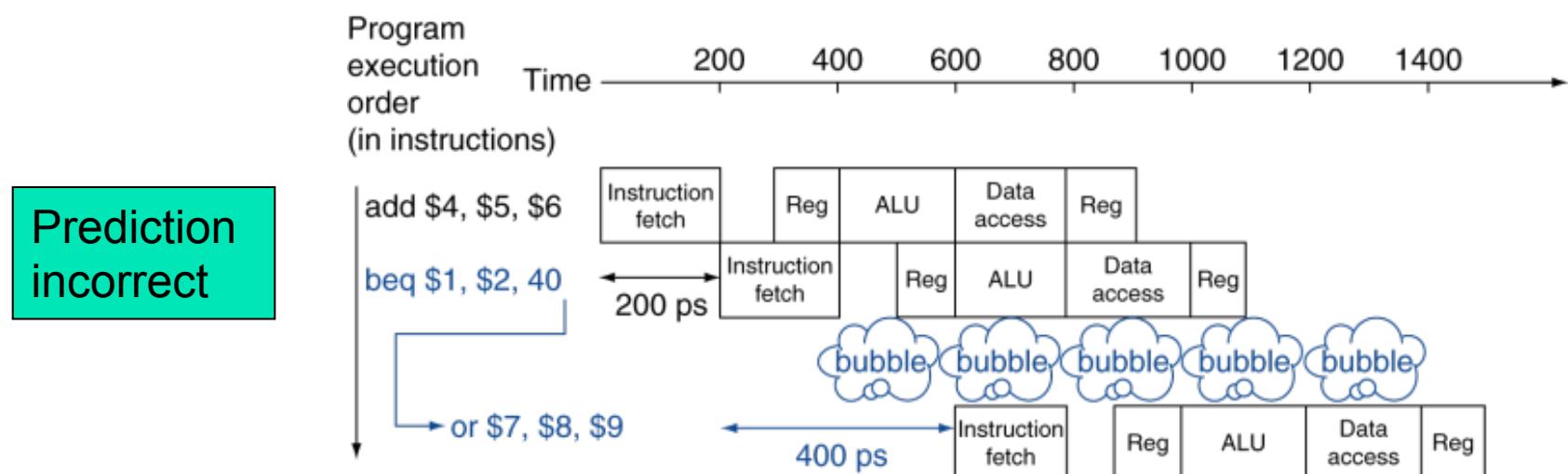
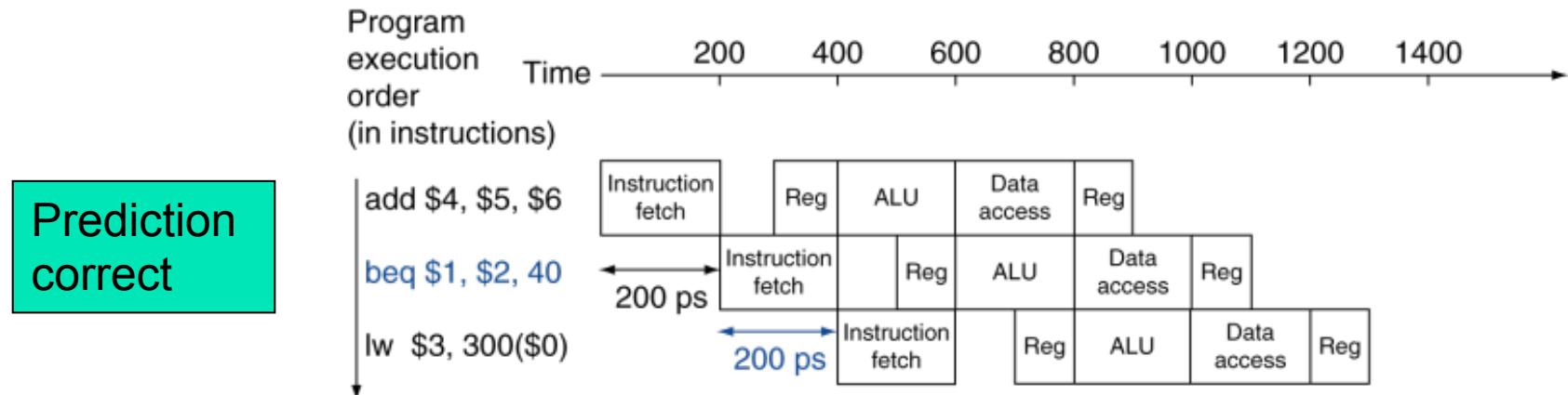
- Đợi cho đến khi kết quả rẽ nhánh đã được xác định trước khi nhận lệnh tiếp theo



# 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)

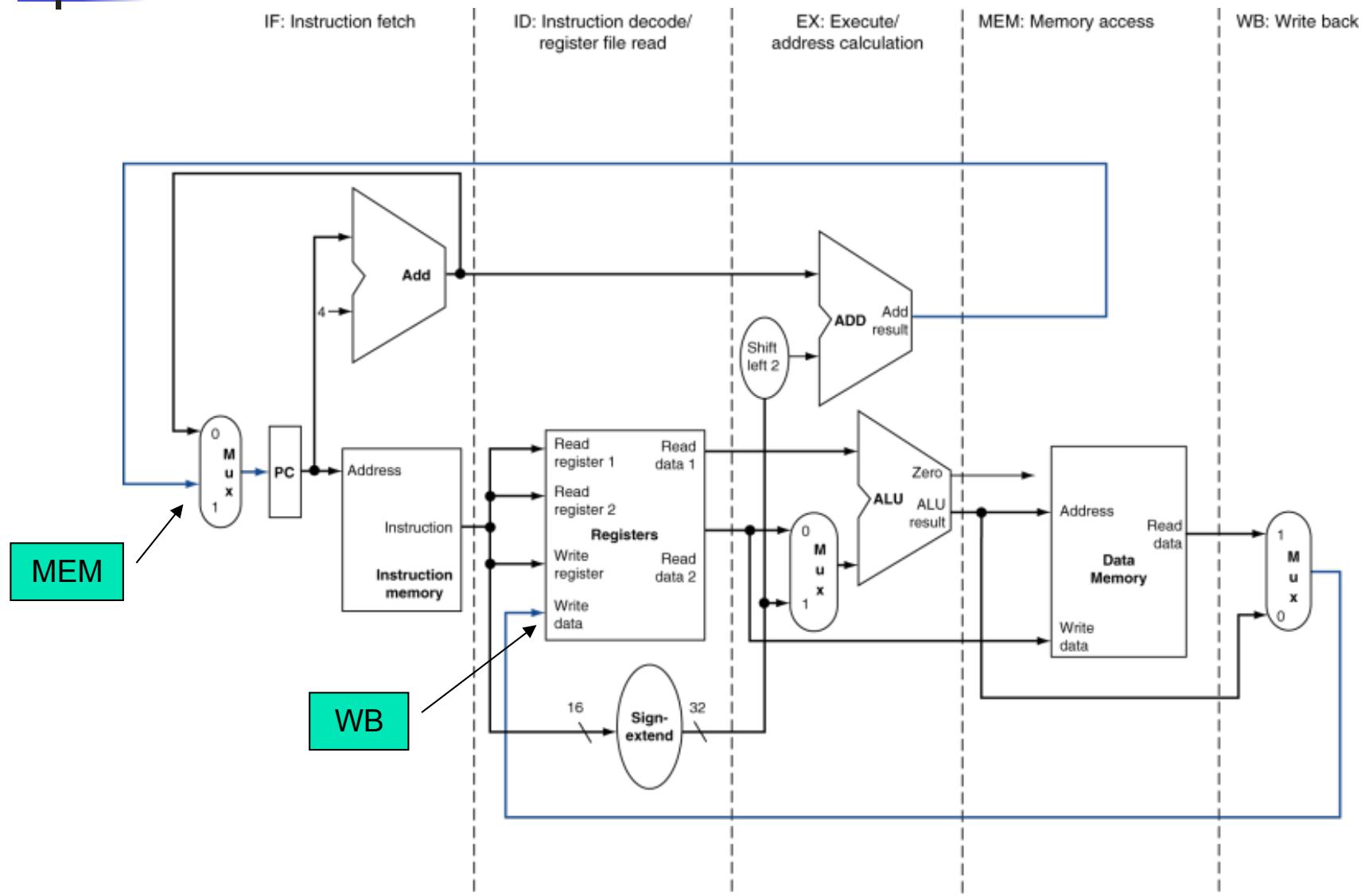
# MIPS với dự đoán rẽ nhánh không xảy ra



# 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

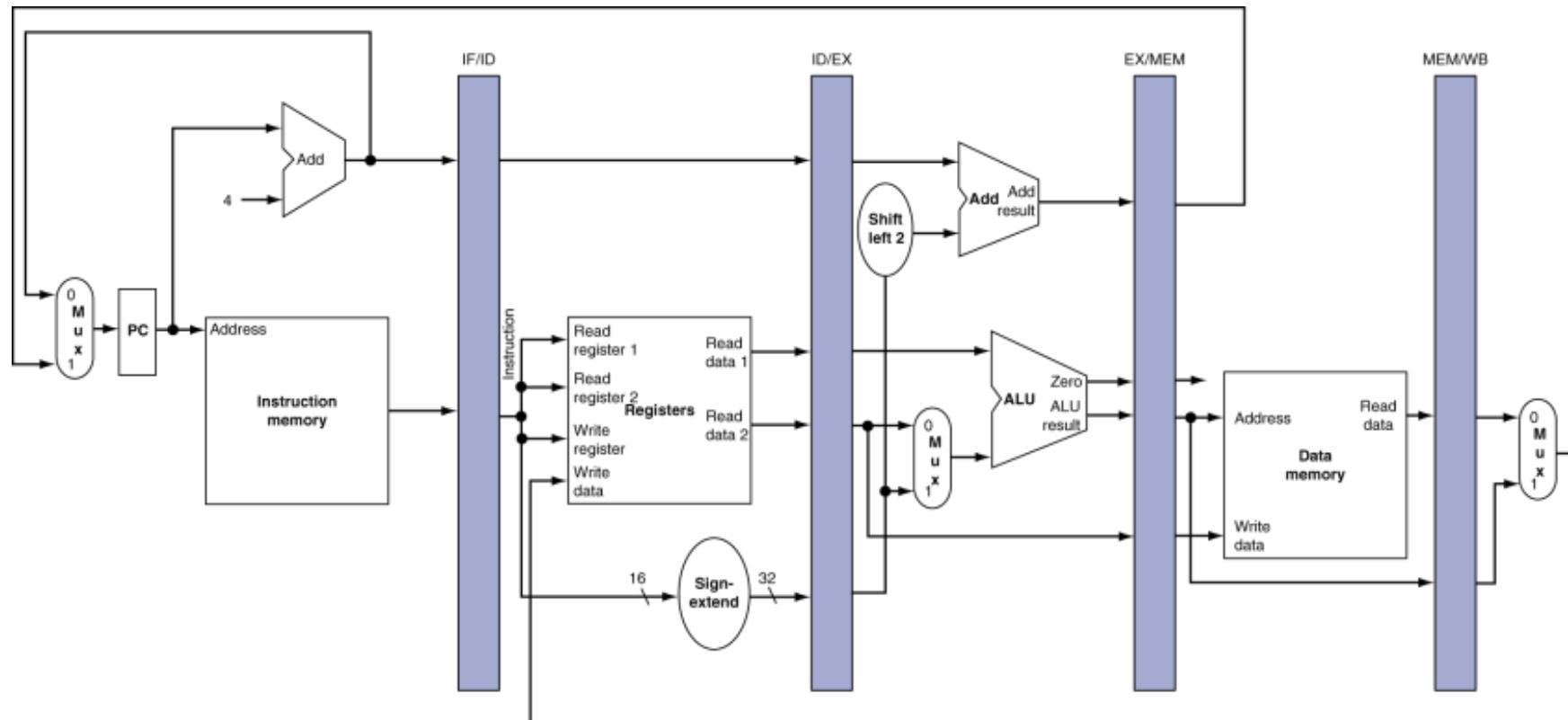


## Nhận xét

- 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

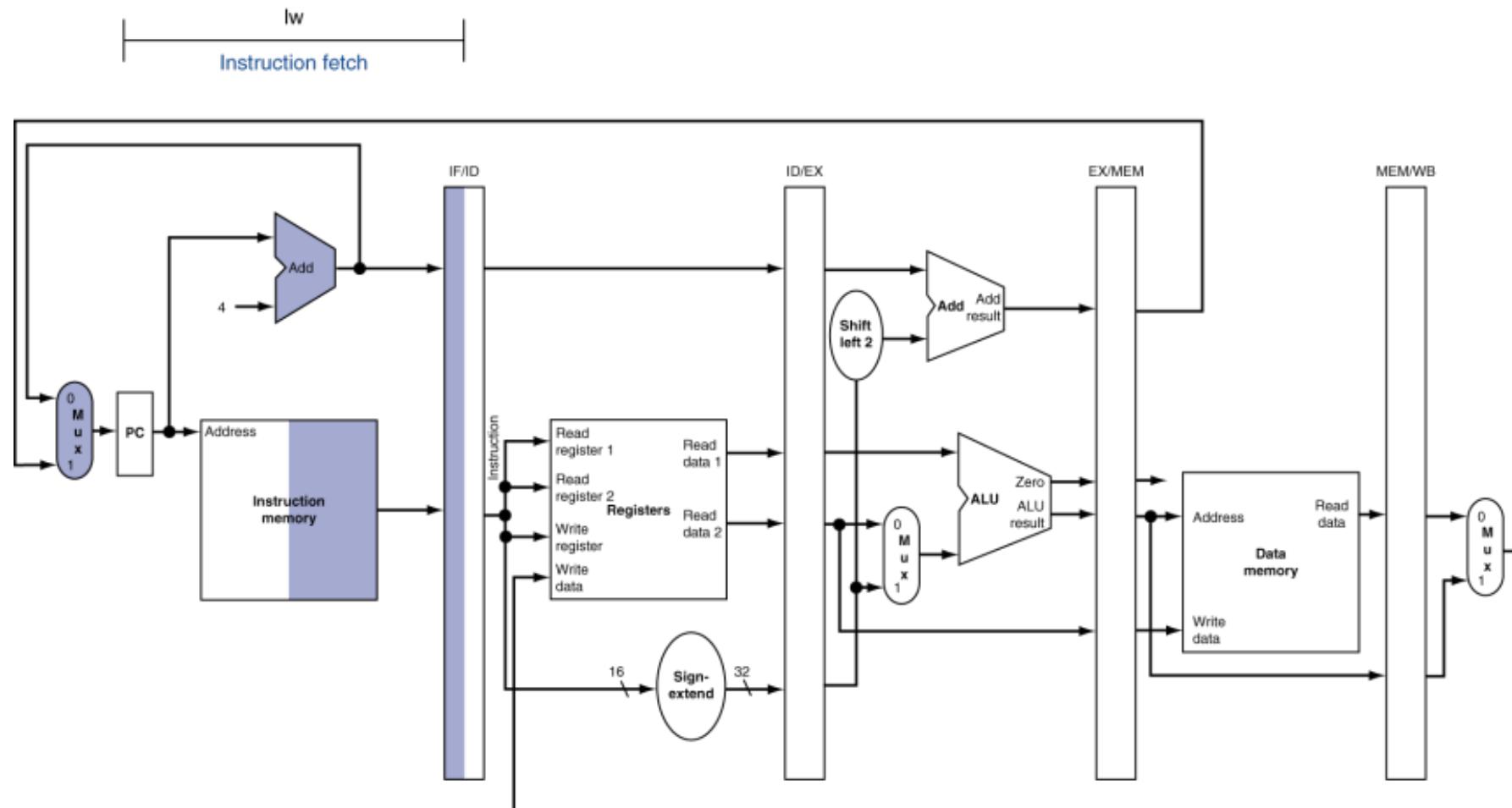
- Cần các thanh ghi đặt giữa các công đoạn
  - Để giữ thông tin được tạo ra bởi chu kỳ trước



# Hoạt động của đường ống

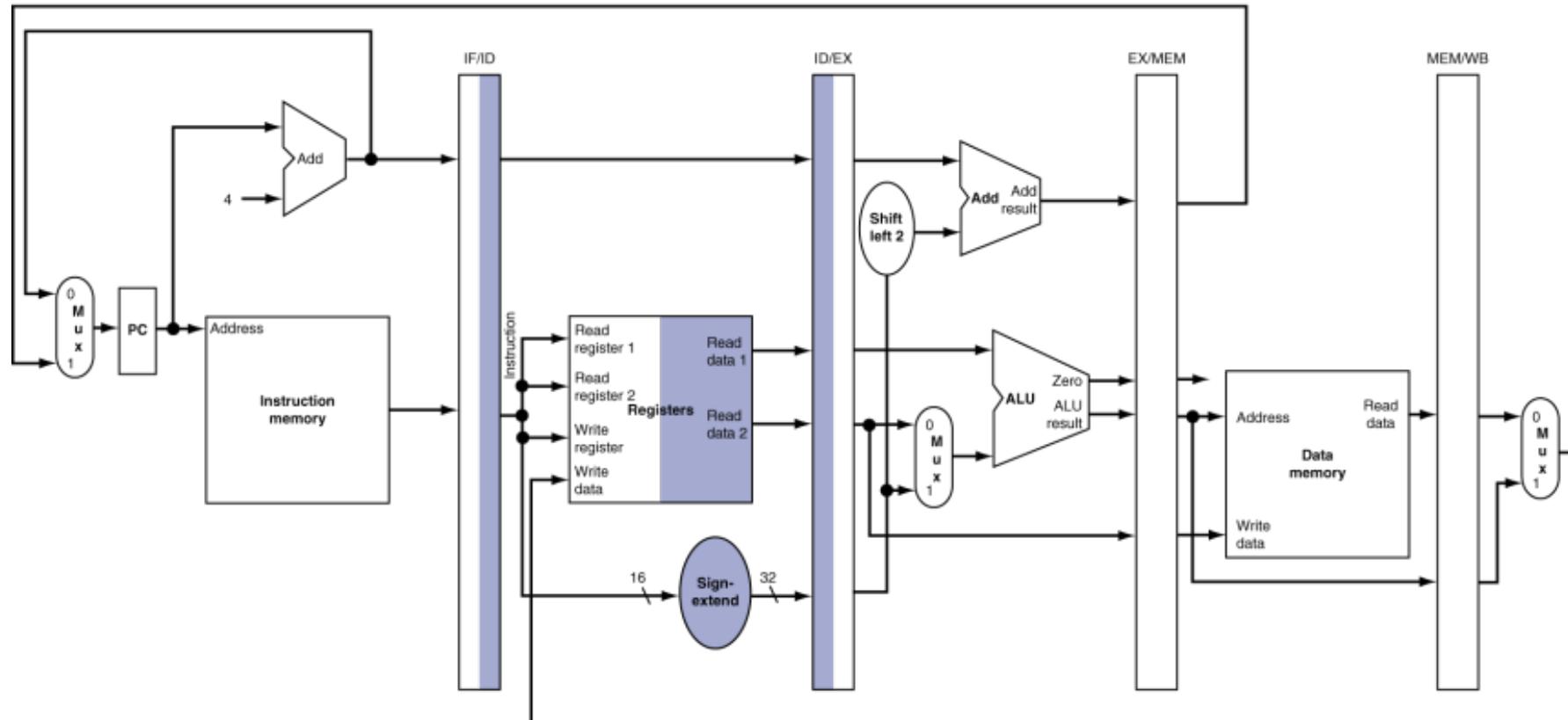
- Dòng lệnh được đưa qua datapath đường ống theo từng chu kỳ.
- Có hai cách thực hiện:
  - Đơn chu kỳ (Single-clock-cycle)
  - Đa chu kỳ (Multi-clock-cycle)
- Xem xét đường ống đơn chu kỳ với load & store

# IF cho lệnh Load, Store

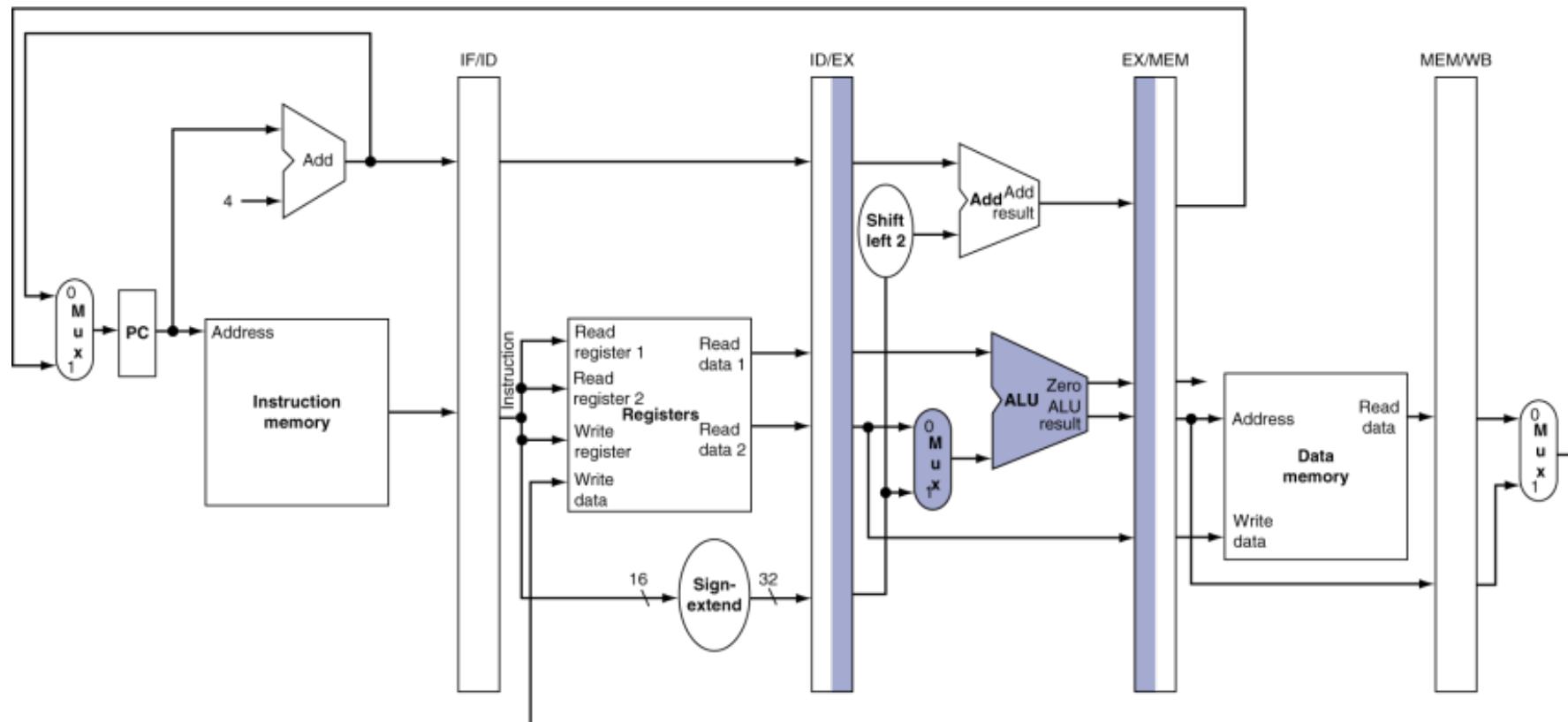


# ID cho lệnh Load, Store

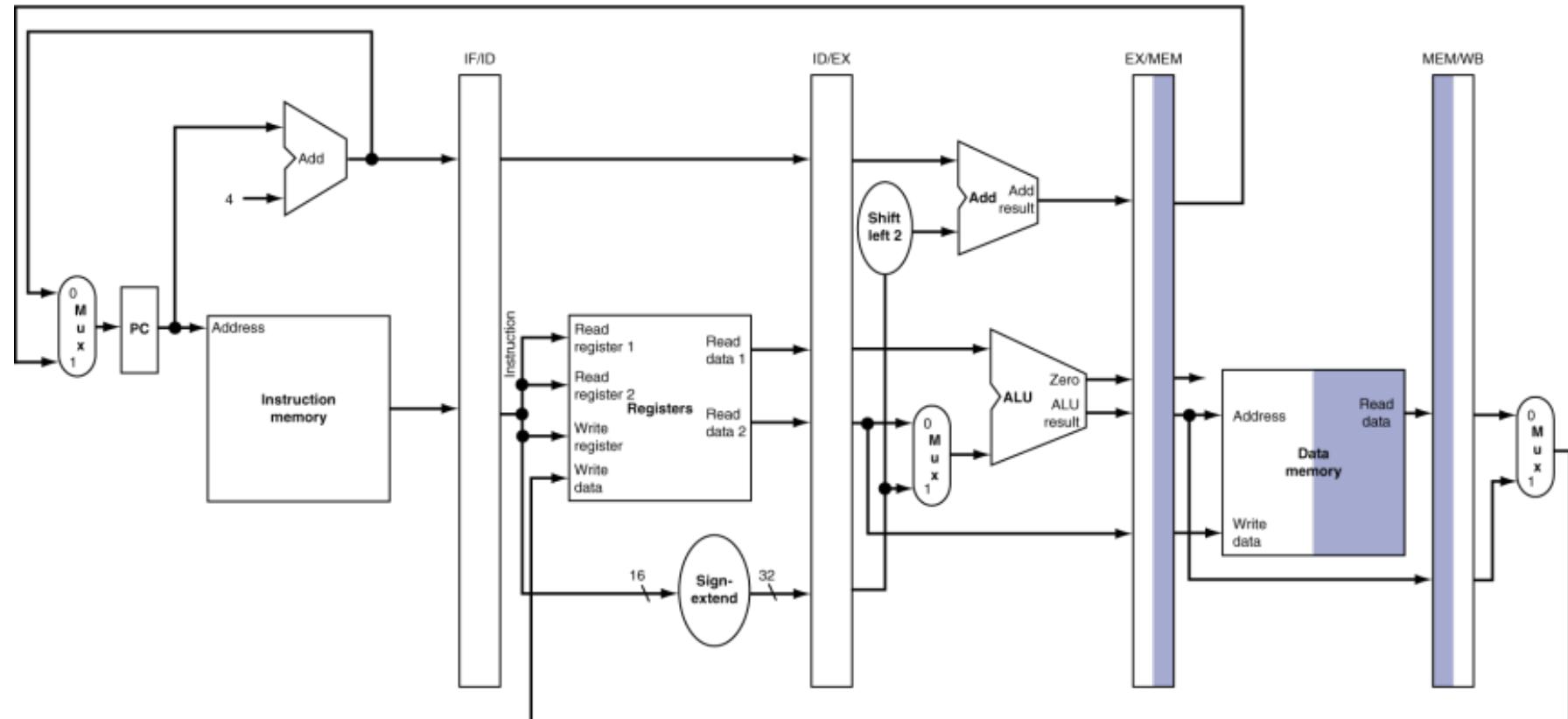
$\overbrace{\hspace{20em}}$   
 Instruction decode



# EX cho lệnh Load

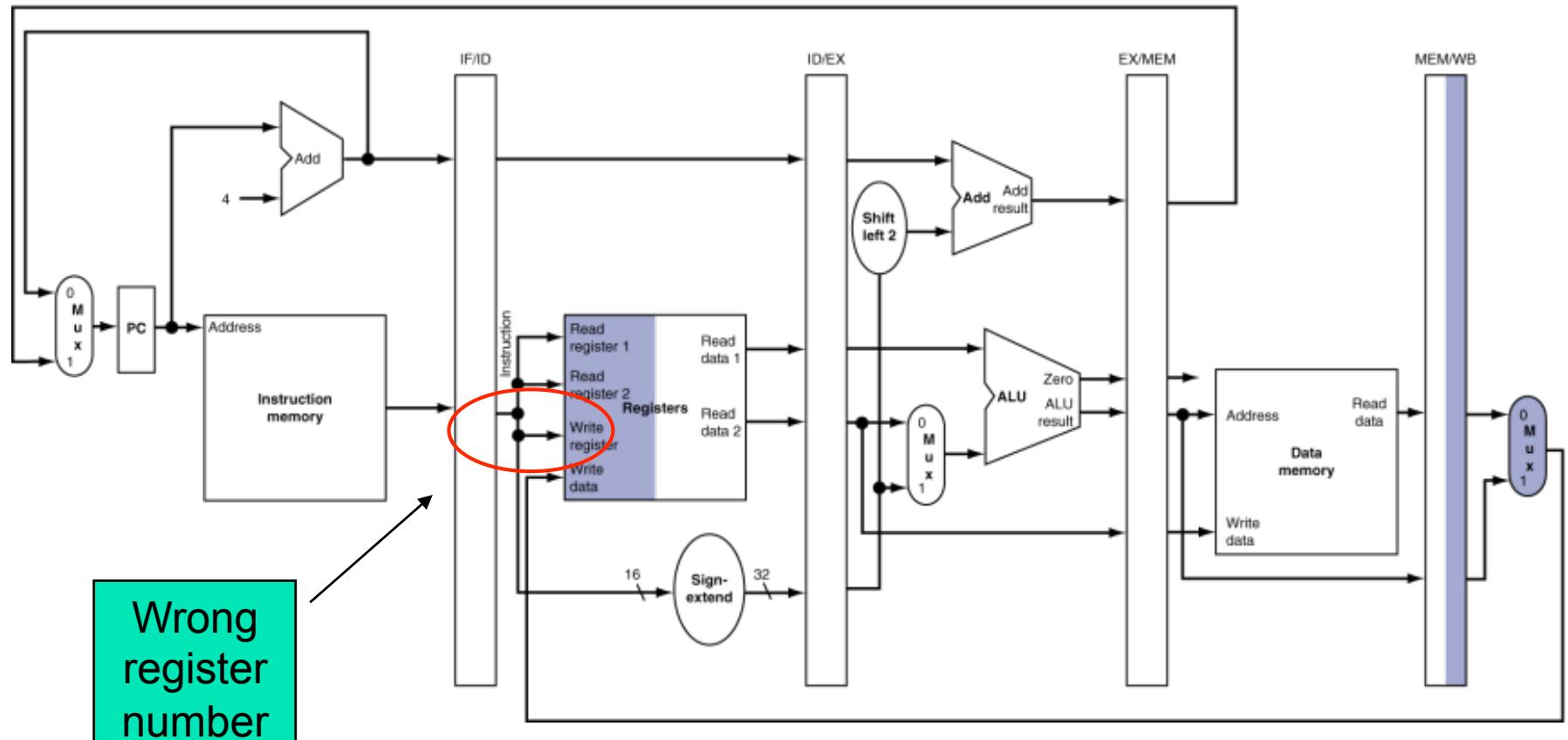


# MEM cho lệnh Load

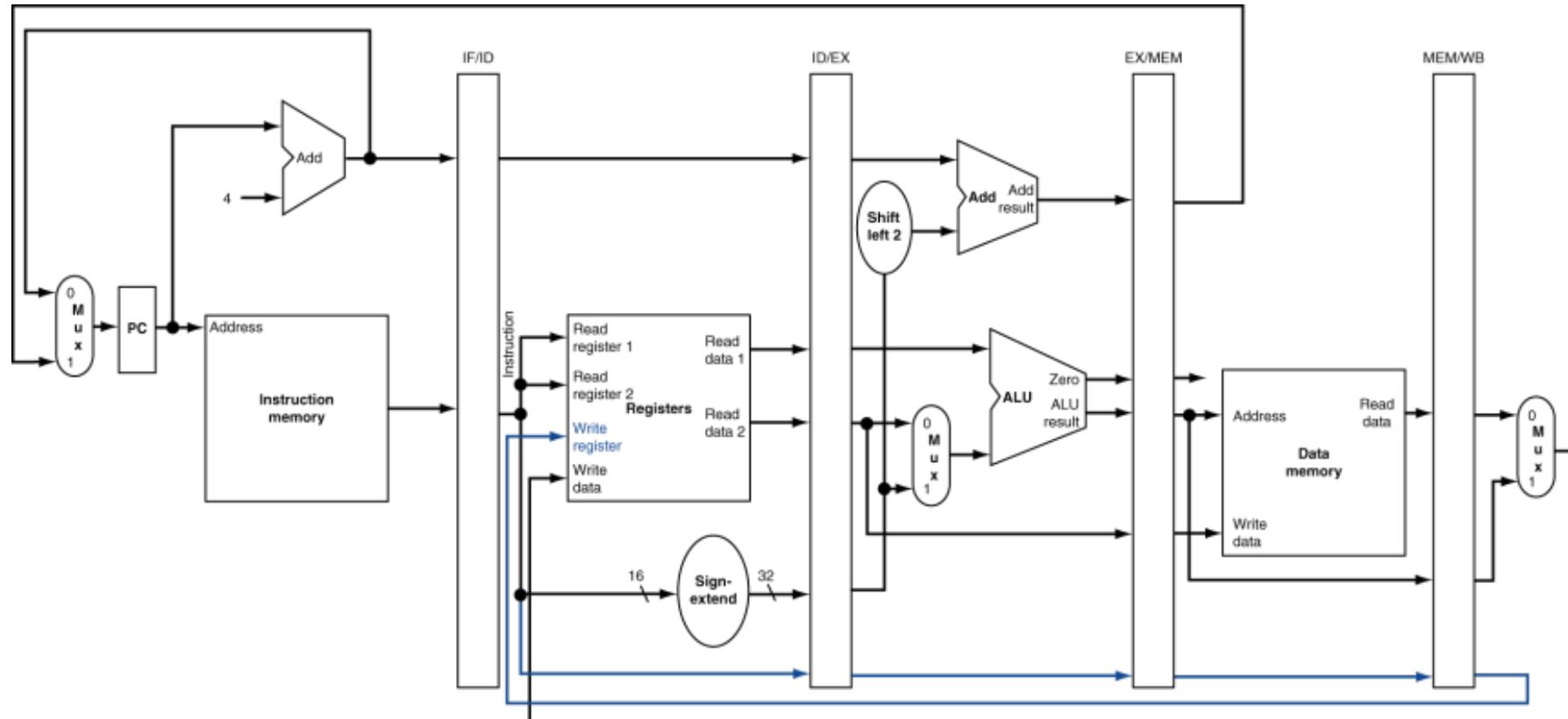


# WB cho lệnh Load

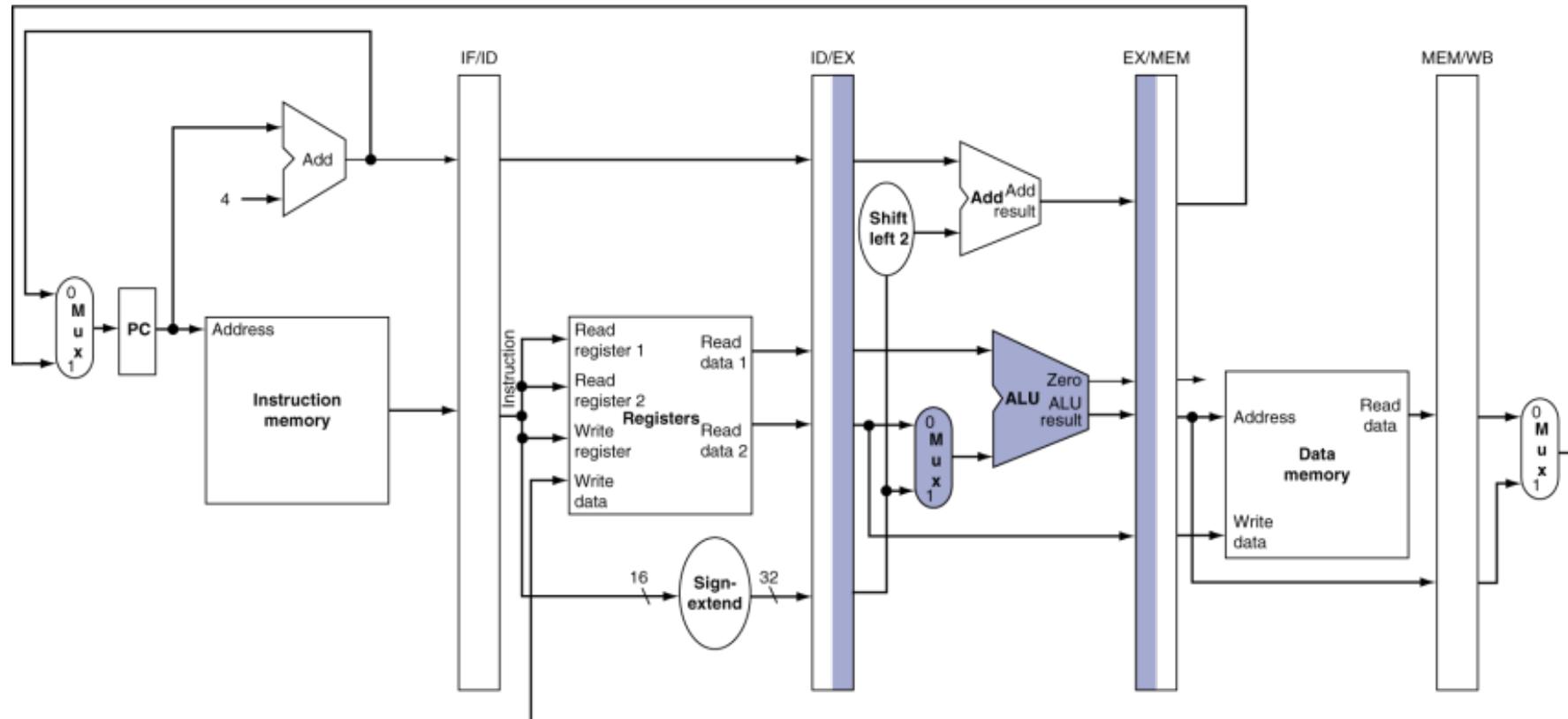
$lw$   
Write back



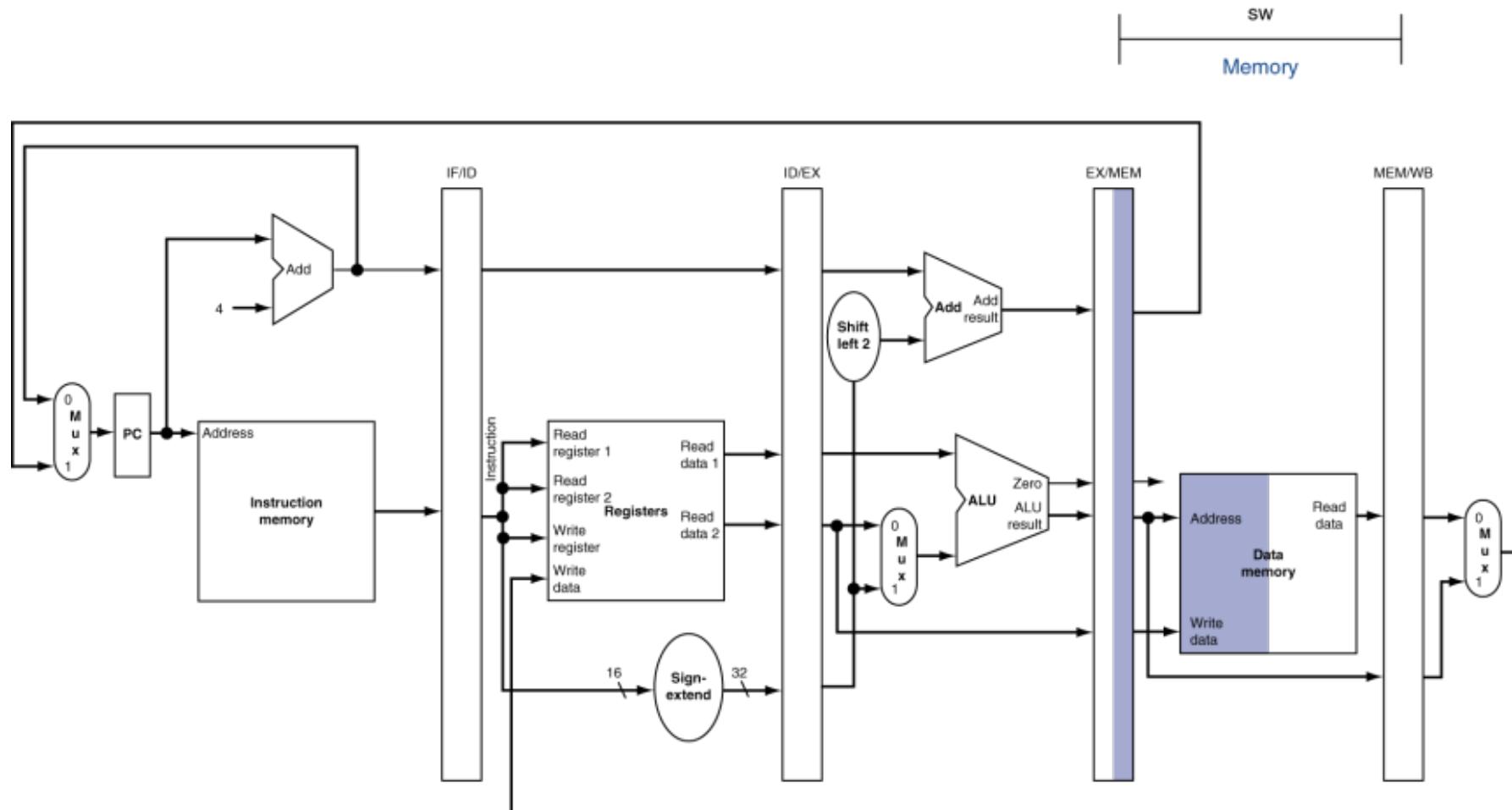
# Datapath được hiệu chỉnh cho lệnh Load



# EX cho lệnh Store

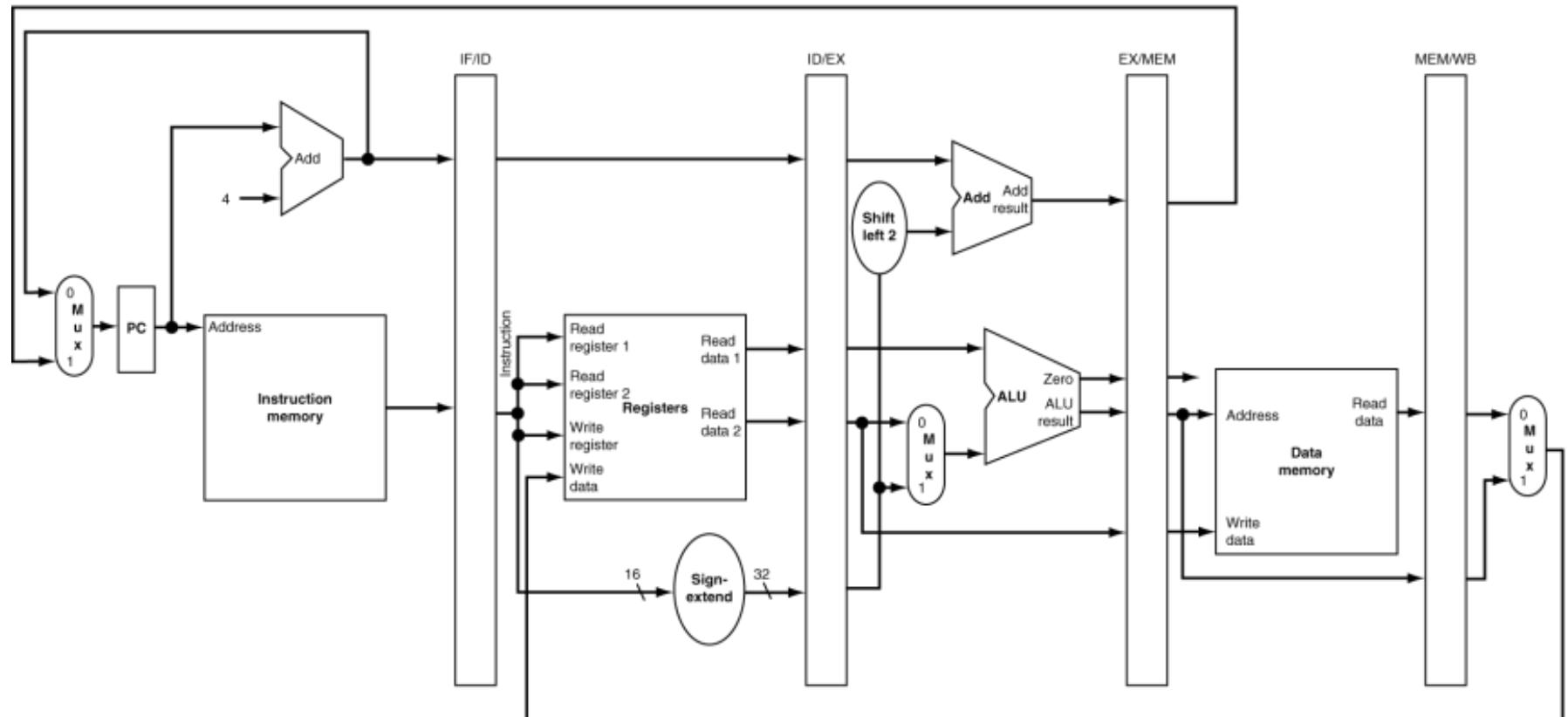


# MEM cho lệnh Store



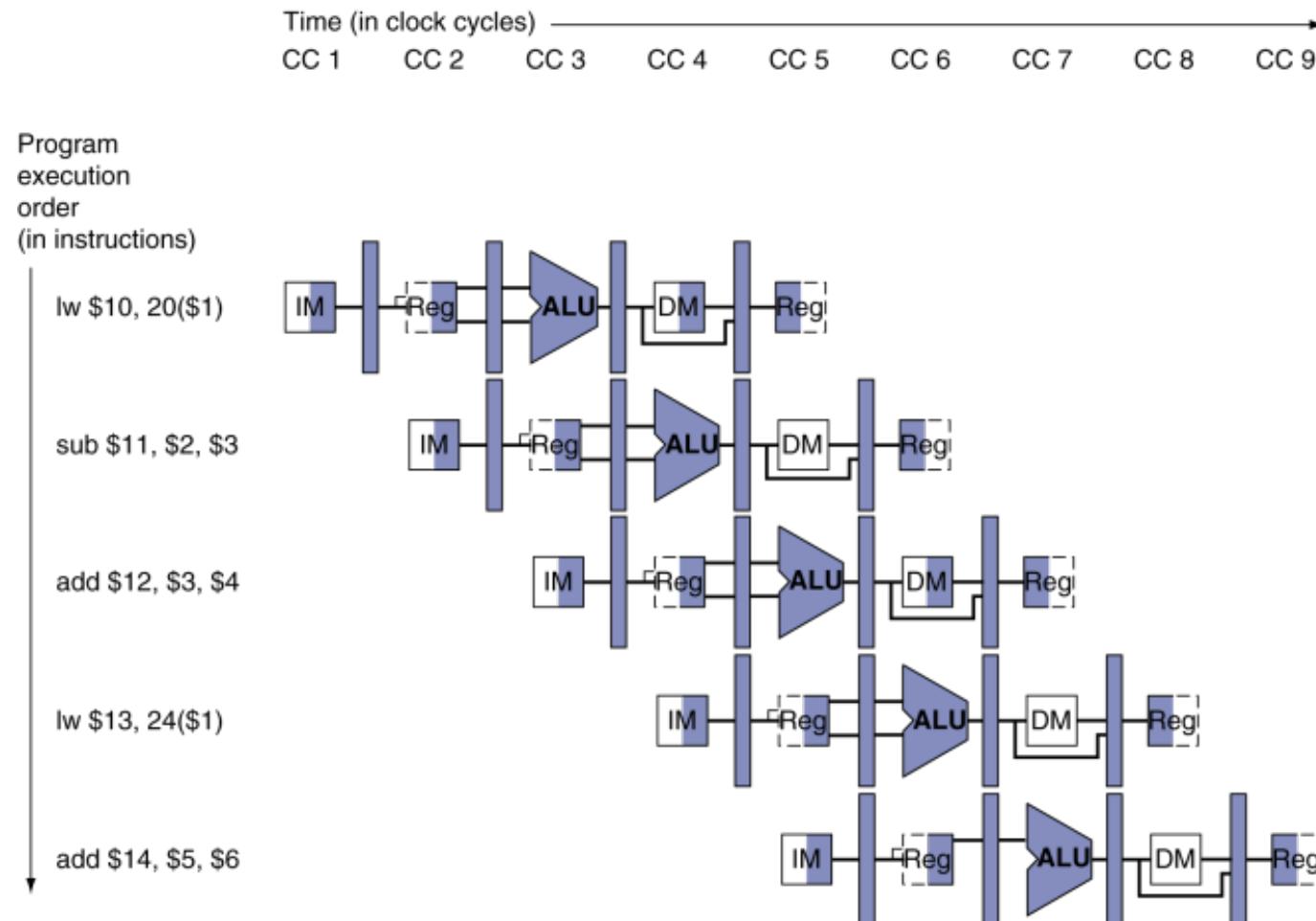
# WB cho lệnh Store

SW  
Write-back



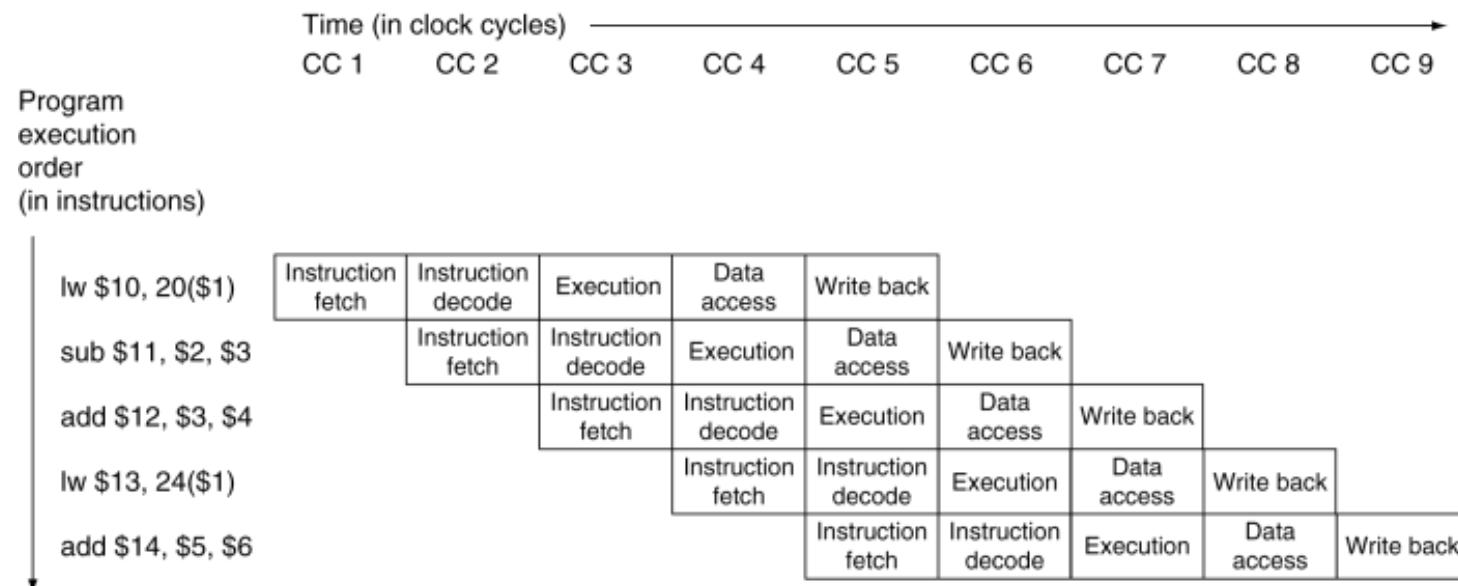
# Giản đồ đường ống đa chu kỳ

## ■ Dạng tài nguyên được sử dụng



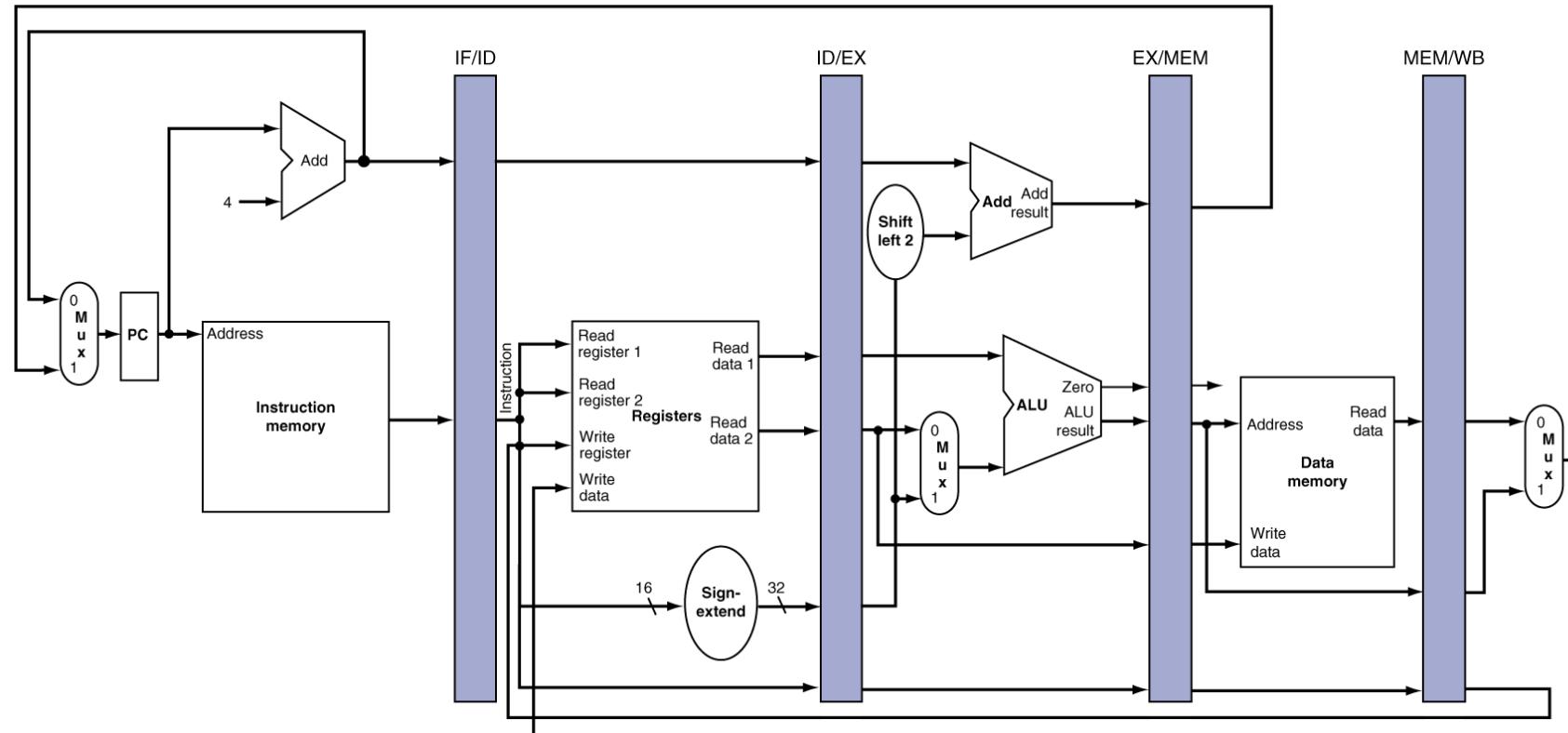
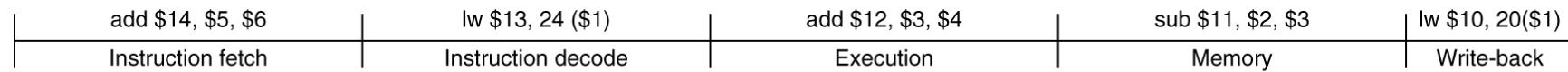
# Giản đồ đường ống đa chu kỳ

## ■ Dạng truyền thống

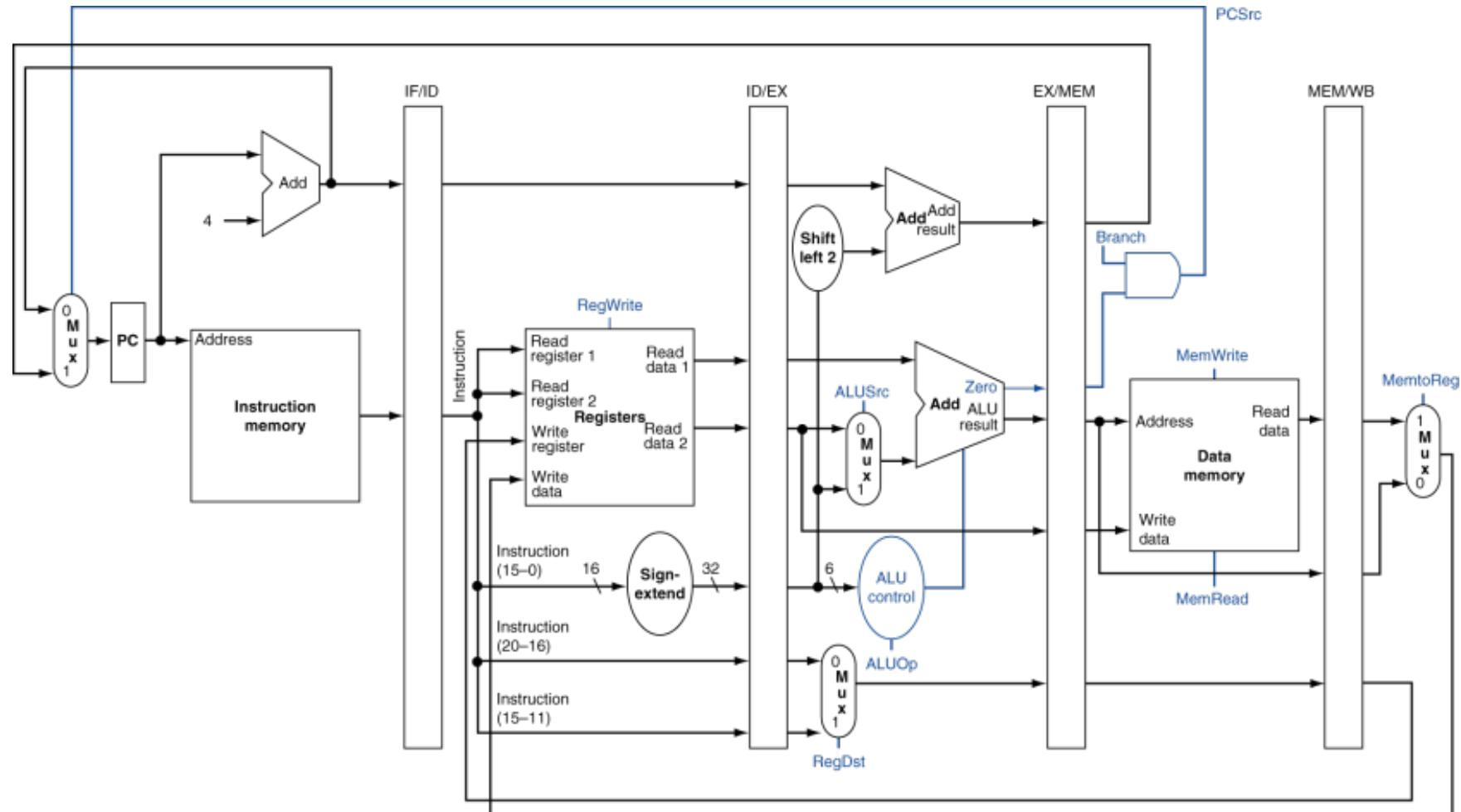


# Giản đồ đường ống đơn chu kỳ

- Công đoạn của đường ống trong chu kỳ đã cho

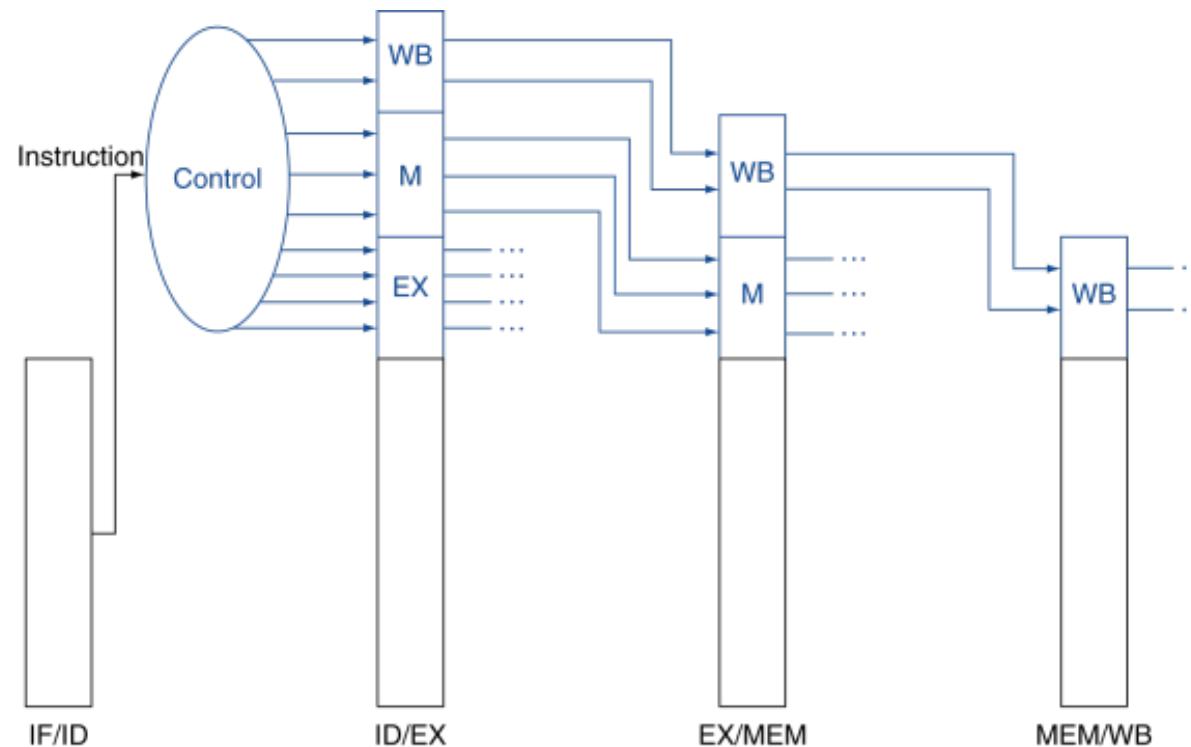


# Điều khiển đường ống (dạng đơn giản)

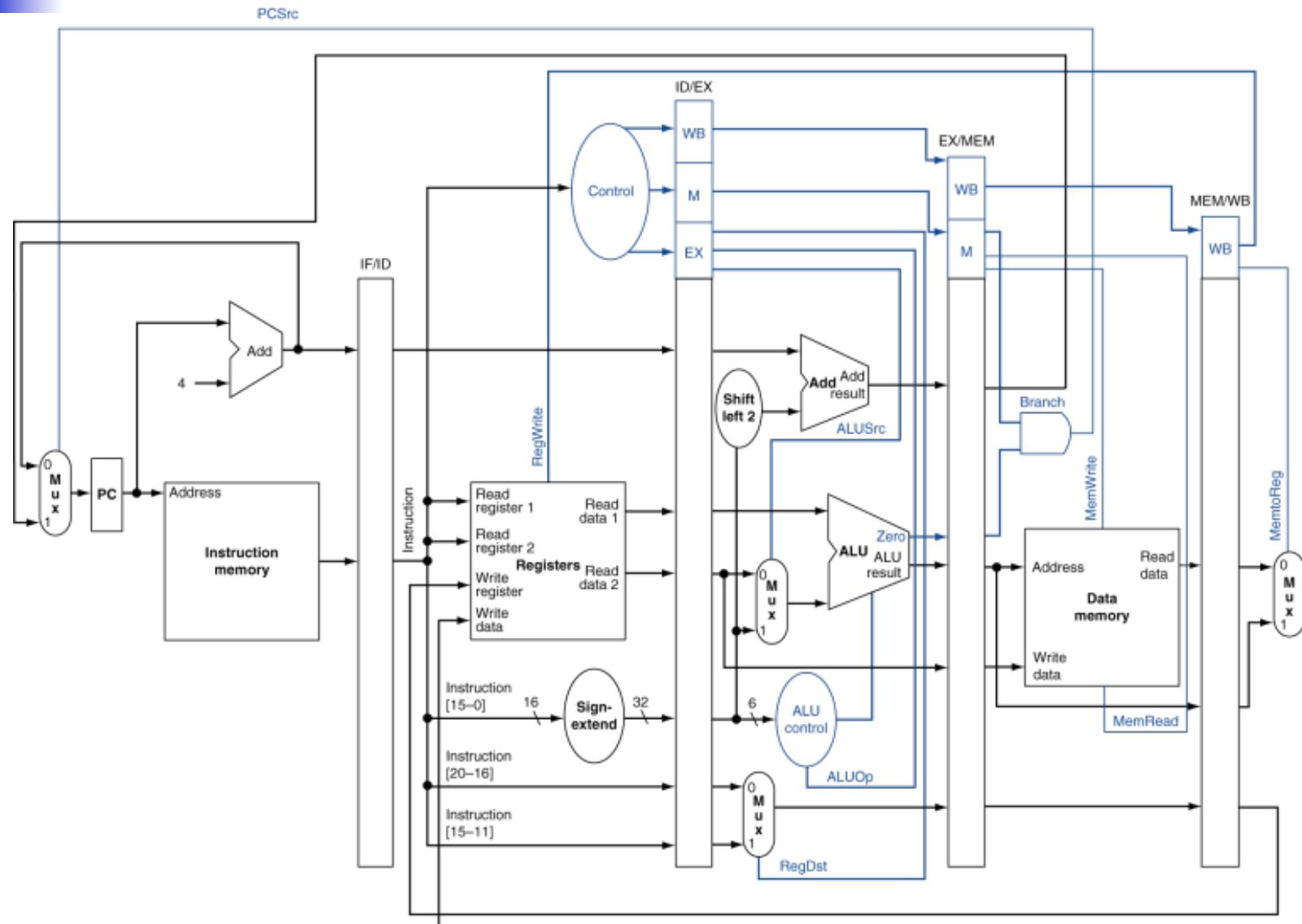


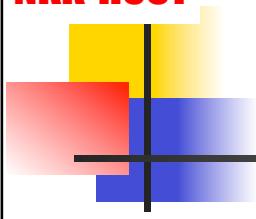
# Điều khiển đường ống

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



# Điều khiển đường ống





Hết