# CHƯƠNG 4

# BỘ XỬ LÝ (THE PROCESSOR)

# CHƯƠNG 4

# BỘ XỬ LÝ (THE PROCESSOR)

- **Phần 1. Xây dựng đường truyền dữ liệu (Datapath)**
- **Phần 2. Kỹ thuật ống dẫn (Pipeline)**

# BỘ XỬ LÝ

## Nội dung phần 1

1. Giới thiệu
2. Nhắc lại các quy ước thiết kế logic
3. Xây dựng đường truyền dữ liệu (datapath) đơn giản
4. Hiện thực datapath đơn chu kỳ

# BỘ XỬ LÝ

## Nội dung

1. Giới thiệu
2. Nhắc lại các quy ước thiết kế logic
3. Xây dựng đường truyền dữ liệu (datapath) đơn giản
4. Hiện thực datapath đơn chu kỳ

# Giới thiệu

❖ Hiệu suất của một máy tính được xác định bởi ba yếu tố:

- Tổng số câu lệnh — Được xác định bởi trình biên dịch và kiến trúc tập lệnh
- Chu kỳ xung clock
- Số chu kỳ xung clock trên một lệnh (Clock cycles per instruction − CPI)

} Được xác định bởi quá trình hiện thực bộ xử lý

❖ Mục đích chính của chương này:

- Giải thích quy tắc hoạt động và hướng dẫn xây dựng datapath cho một bộ xử lý chứa một số lệnh đơn giản (giống kiến trúc tập lệnh dạng MIPS), gồm hai ý chính:
  - Thiết kế datapath
  - Hiện thực datapath đã thiết kế

MIPS (bắt nguồn từ chữ viết tắt của 'Microprocessor without Interlocked Pipeline Stages') là một kiến trúc tập tập lệnh dạng RISC, được phát triển bởi MIPS Technologies (trước đây là MIPS Computer Systems, Inc.)

# Giới thiệu

**Chương này chỉ xem xét 8 lệnh trong 3 nhóm chính của tập lệnh MIPS:**

- Nhóm lệnh tham khảo bộ nhớ (**lw** và **sw**)
- Nhóm lệnh liên quan đến logic và số học (**add, sub, AND, OR**, và **slt)**
- Nhóm lệnh nhảy (Lệnh nhảy với điều kiện bằng **beq**)

# Giới thiệu

## Tổng quan các lệnh cần xem xét:

**Nhóm lệnh tham khảo bộ nhớ:**

Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Truy xuất bộ nhớ để đọc/ghi dữ liệu

**Nhóm lệnh logic và số học:**

Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Ghi dữ liệu vào thanh ghi

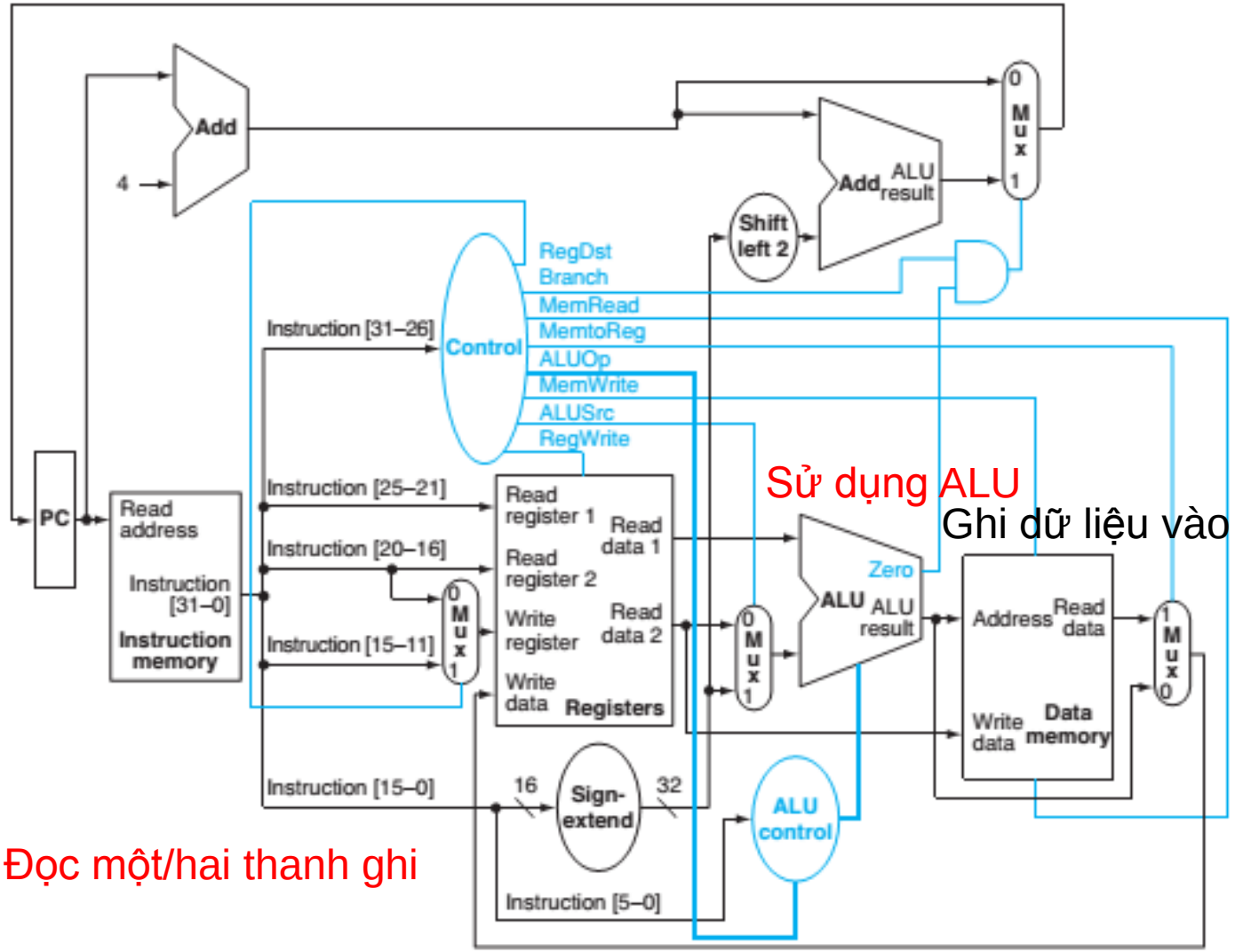
**Nhóm lệnh nhảy:**

Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Chuyển đến địa chỉ lệnh tiếp theo dựa trên kết quả so sánh

# Giới thiệu

**Hình ảnh datapath của một bộ xử lý với 8 lệnh MIPS: *add, sub, AND, OR, slt, lw, sw và beq***

Nạp lệnh




Sử dụng ALU

Ghi dữ liệu vào thanh ghi

Đọc một/hai thanh ghi

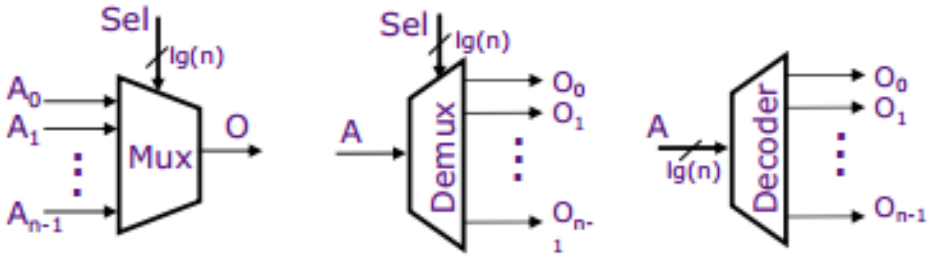
# BỘ XỬ LÝ

## Nội dung

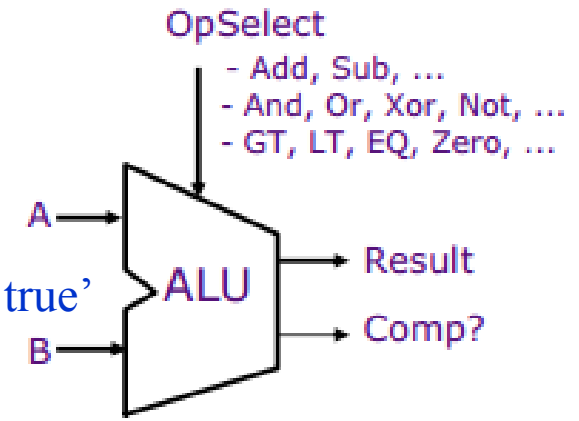
1. Giới thiệu
2. Nhắc lại các quy ước thiết kế logic
3. Xây dựng đường truyền dữ liệu (datapath) đơn giản
4. Hiện thực datapath đơn chu kỳ

# Quy ước thiết kế

Phần này nhắc lại các khái niệm:




- **Mạch tổ hợp (Combinational): ALU**
- **Mạch tuần tự (Sequential): instruction/data memories và thanh ghi**
- **Tín hiệu điều khiển (Control signal)**
- **Tín hiệu dữ liệu (Data signal)**
  - **Asserted (assert)**: Khi tín hiệu ở mức cao hoặc 'true'
  - **Deasserted (deassert):** Khi tín hiệu ở mức thấp hoặc 'false'




- **Bus**

# BỘ XỬ LÝ

## Nội dung

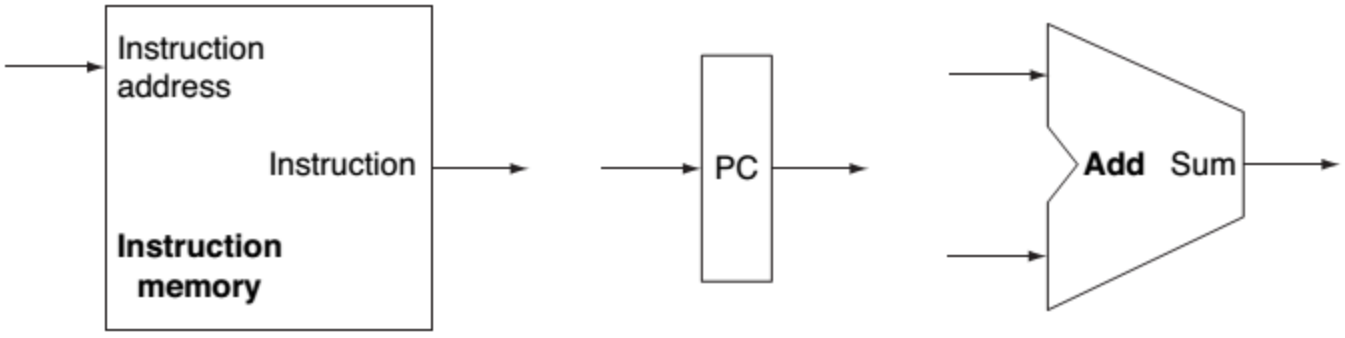
1. Giới thiệu
2. Nhắc lại các quy ước thiết kế logic
3. Xây dựng đường truyền dữ liệu (datapath) đơn giản
4. Hiện thực datapath đơn chu kỳ

# Xây dựng Datapath

- **Các thành phần trong một datapath bao gồm:** *bộ nhớ lệnh và dữ liệu (instruction and data memories), tập các thanh ghi (register file/registers), ALU và bộ cộng (Adders).*

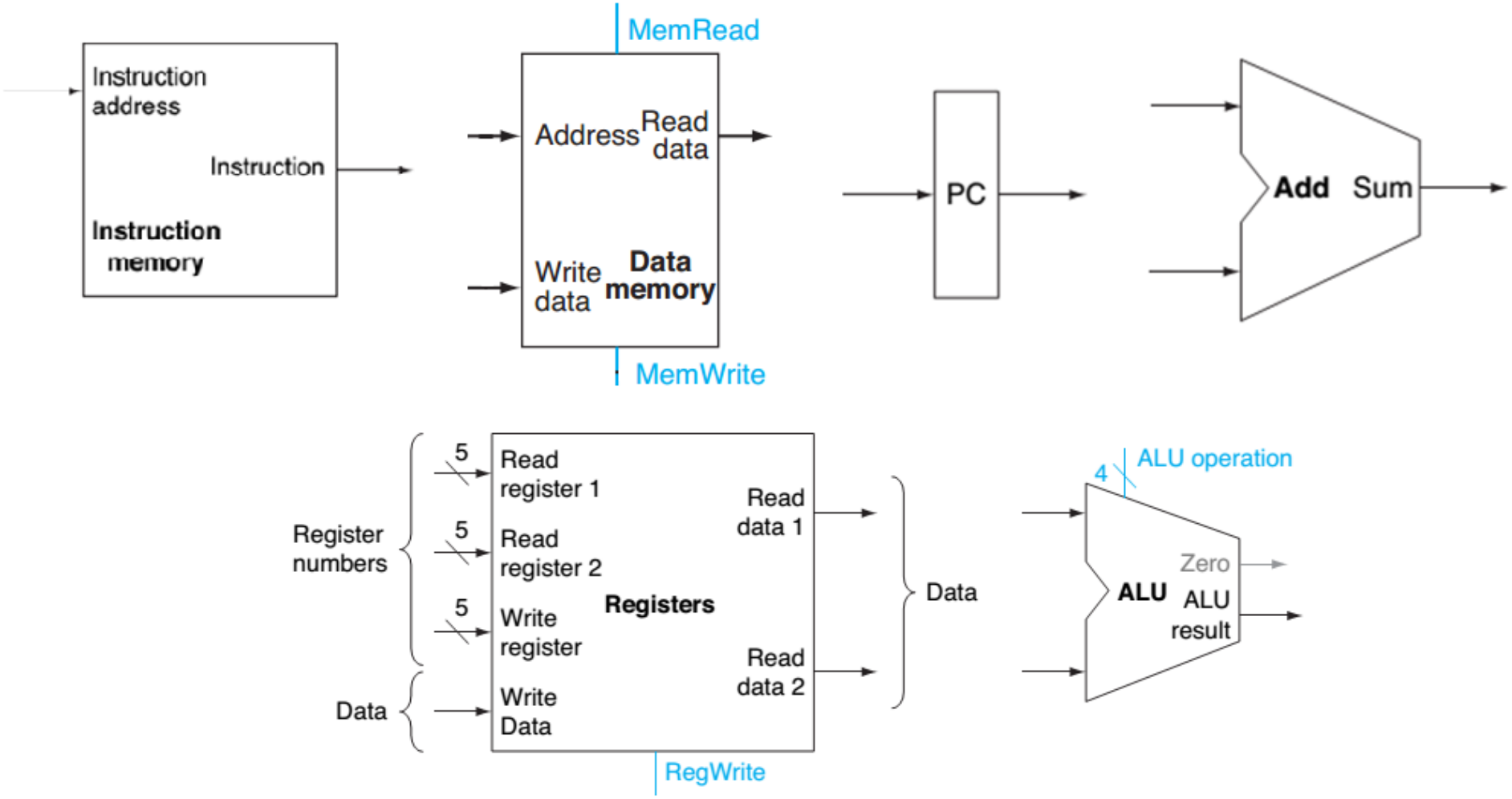
- **Program Counter (PC):** là thanh ghi chứa địa chỉ của lệnh đang được thực thi trong chương trình

- **Tập các thanh ghi:** Là một mạch tuần tự chứa tập hợp các thanh ghi. Việc truy xuất đọc/ghi lên mỗi thanh ghi được thực hiện bằng cách cung cấp chỉ số thanh ghi đó (mỗi thanh ghi có một chỉ số riêng, cũng được xem là địa chỉ thanh ghi)




a. Instruction memory  b. Program counter  c. Adder

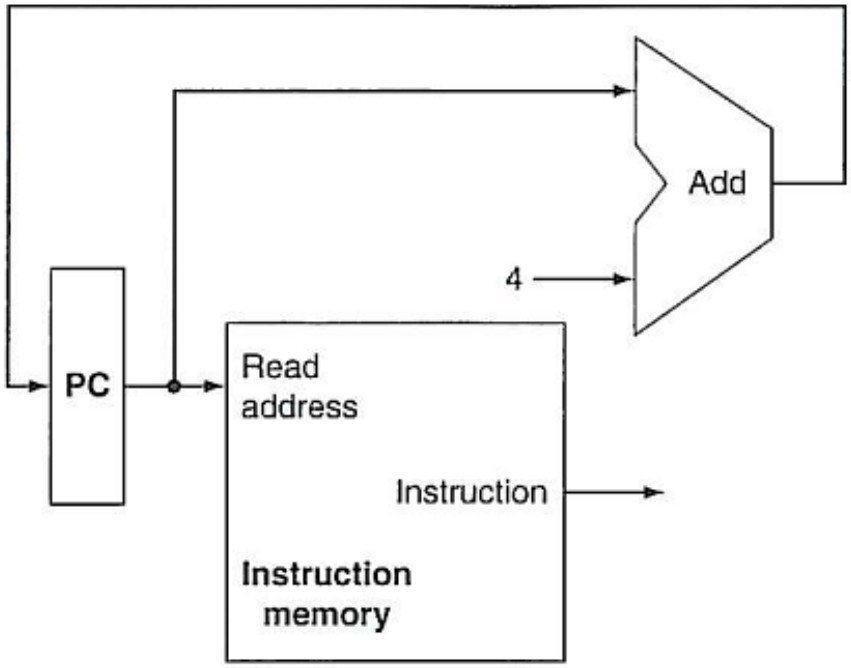
# Xây dựng Datapath




**Tập các thanh ghi**

# Xây dựng Datapath

## 1. Nạp lệnh




*Một phần hình ảnh của datapath, được sử dụng cho quá trình nạp lệnh và sau đó tăng con lên 4 của con trỏ PC*

# Xây dựng Datapath

## 2. Nhóm lệnh logic và số học (*add, sub, AND, OR and slt*)

**Ví dụ:** ***add*** $\$t_1, \$t_2, \$t_3$

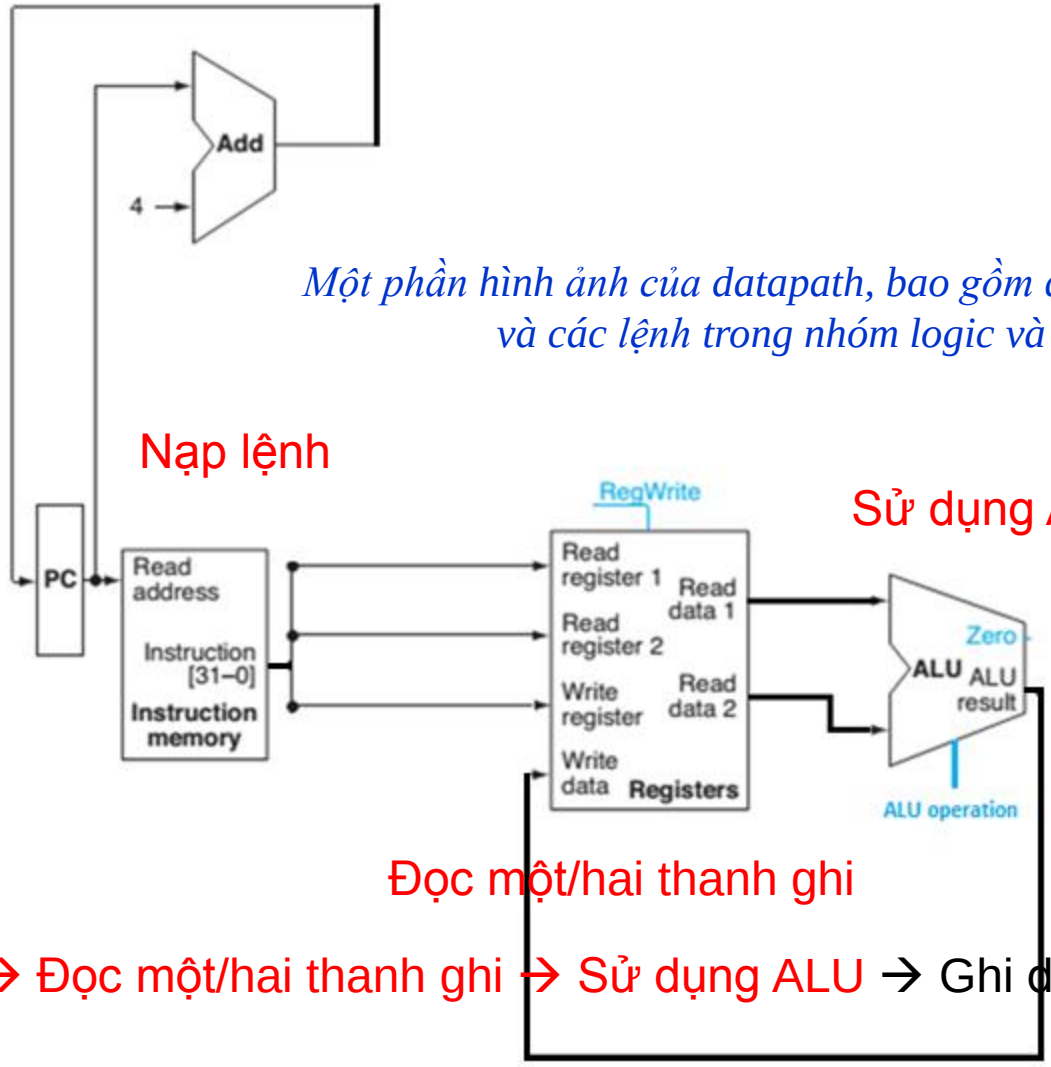
| Field | 0 | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:11 | 10:6 | 5:0 |


a. Registers b. ALU

*Hai thành phần cần để hiện thực các lệnh thuộc nhóm logic và số học là tập thanh ghi và ALU*

# Xây dựng Datapath

## 2. Nhóm lệnh logic và số học (*add, sub, AND, OR and slt*)




*Một phần hình ảnh của datapath, bao gồm quá trình nạp lệnh và các lệnh trong nhóm logic và số học*

Nạp lệnh

Sử dụng ALU

Đọc một/hai thanh ghi

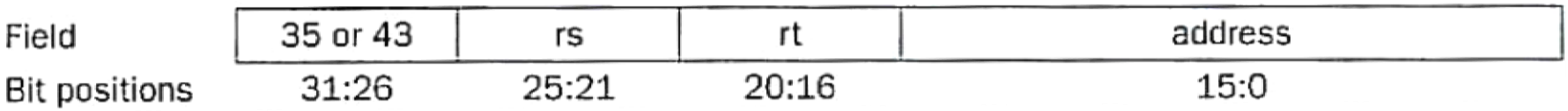
Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Ghi dữ liệu vào thanh ghi

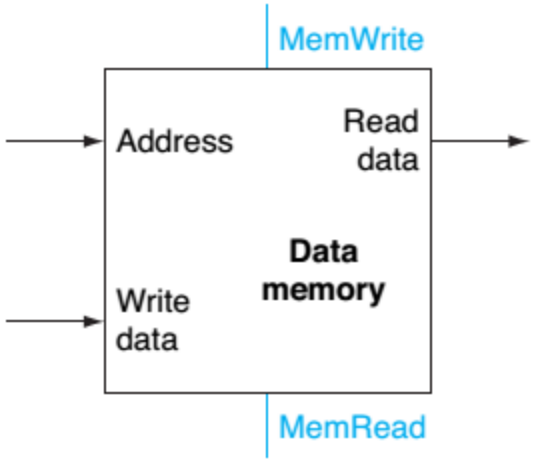
# Xây dựng Datapath

## 3. Nhóm lệnh tham khảo bộ nhớ (*lw và sw*)

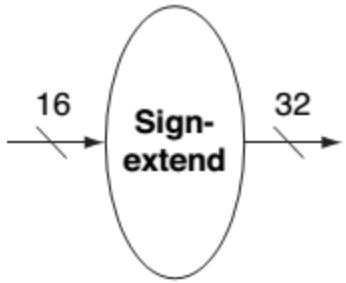
**Ví dụ:** ***lw*** *$s1, offset_value($t2)*

***sw*** *$s1, offset_value($t2)*




a. Data memory unit




b. Sign extension unit

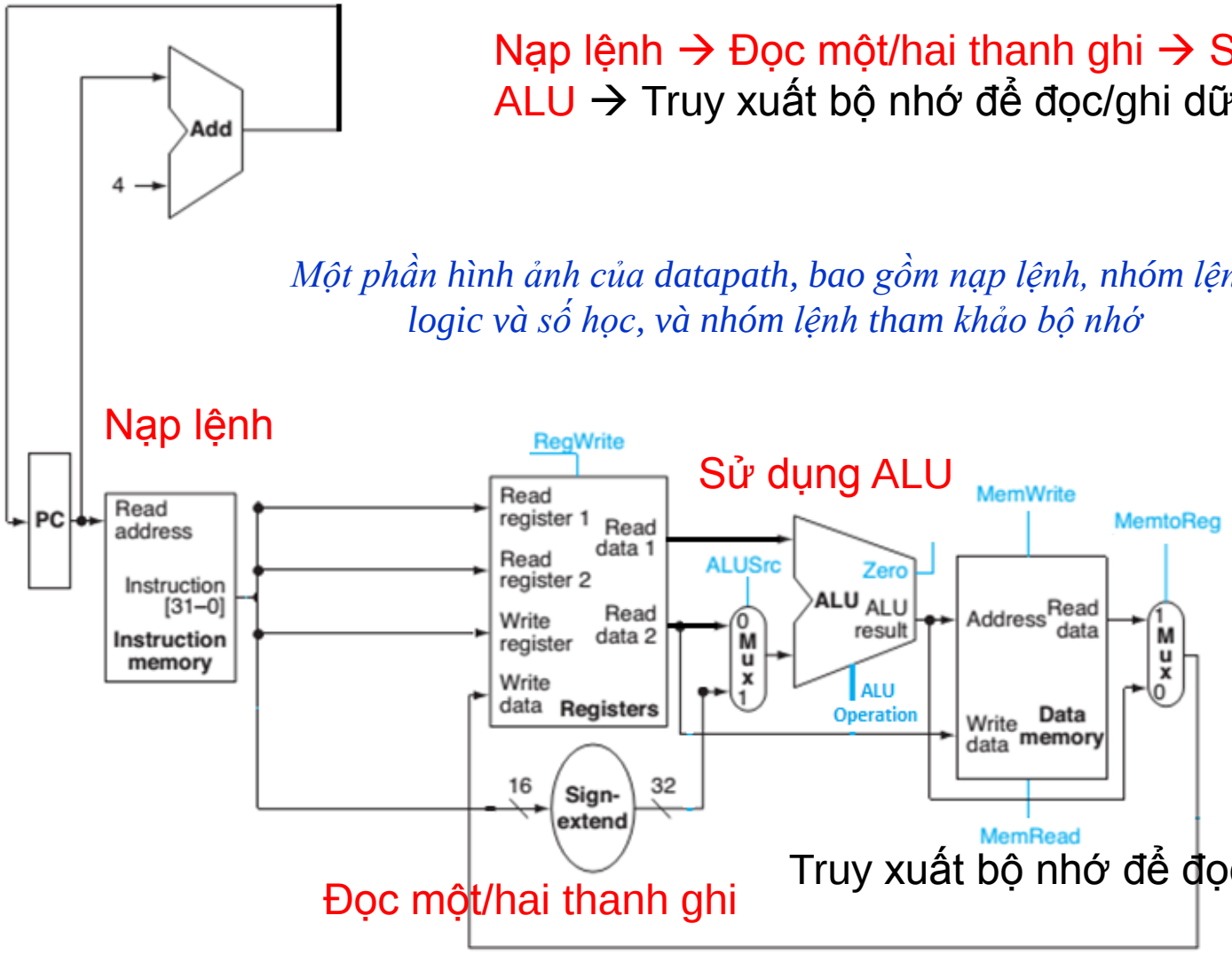
*Ngoài register file và ALU, hai thành phần cần thêm vào trong datapath khi có nhóm lệnh tham khảo bộ nhớ là bộ nhớ dữ liệu và khối mở rộng dấu (Sign-extend)*

# Xây dựng Datapath

## 3. Nhóm lệnh tham khảo bộ nhớ (*lw và sw*)

Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Truy xuất bộ nhớ để đọc/ghi dữ liệu

*Một phần hình ảnh của datapath, bao gồm nạp lệnh, nhóm lệnh logic và số học, và nhóm lệnh tham khảo bộ nhớ*



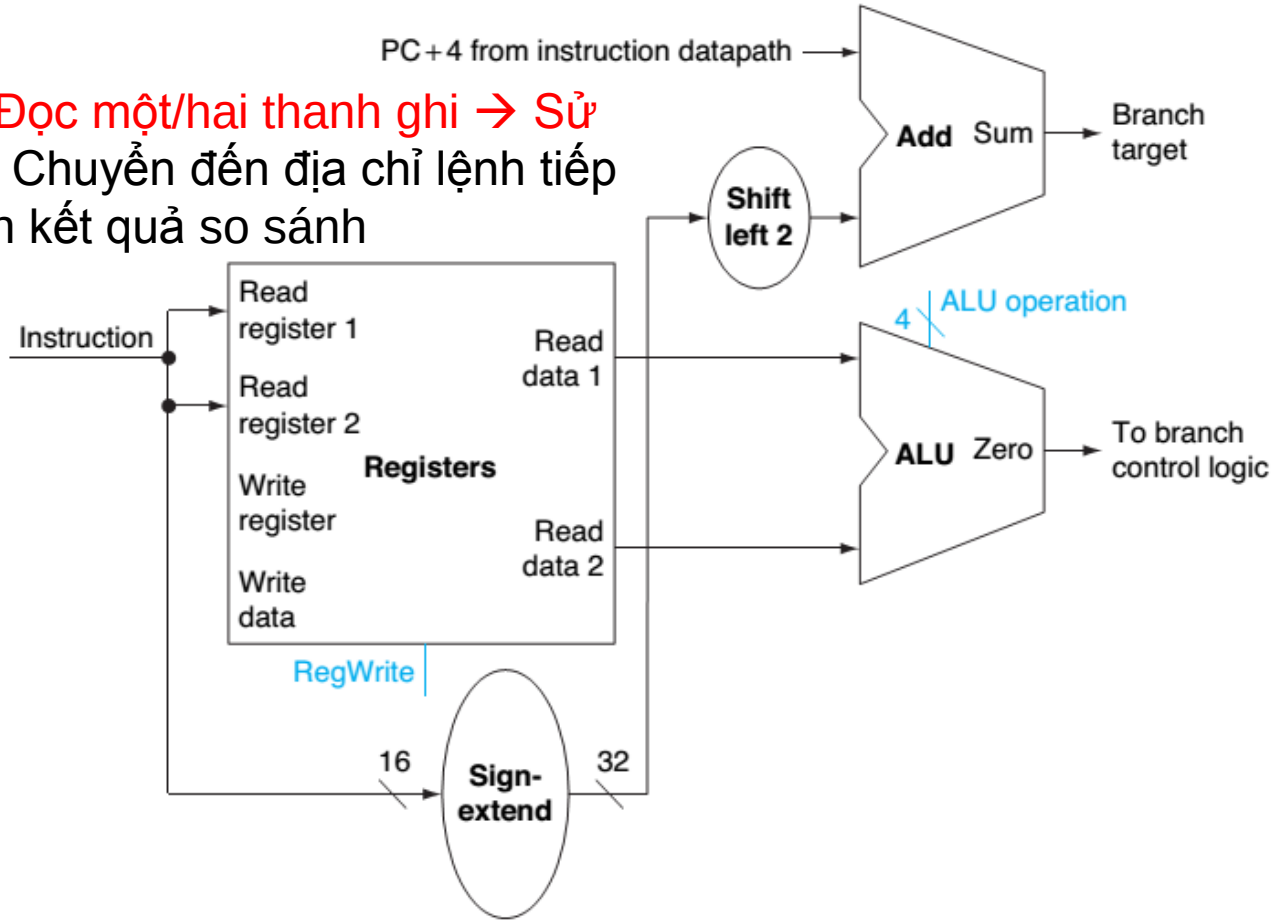
# Xây dựng Datapath

## 4. Nhóm lệnh nhảy (*beq*)

**Example:** **beq** $t_1$, $t_2$, offset

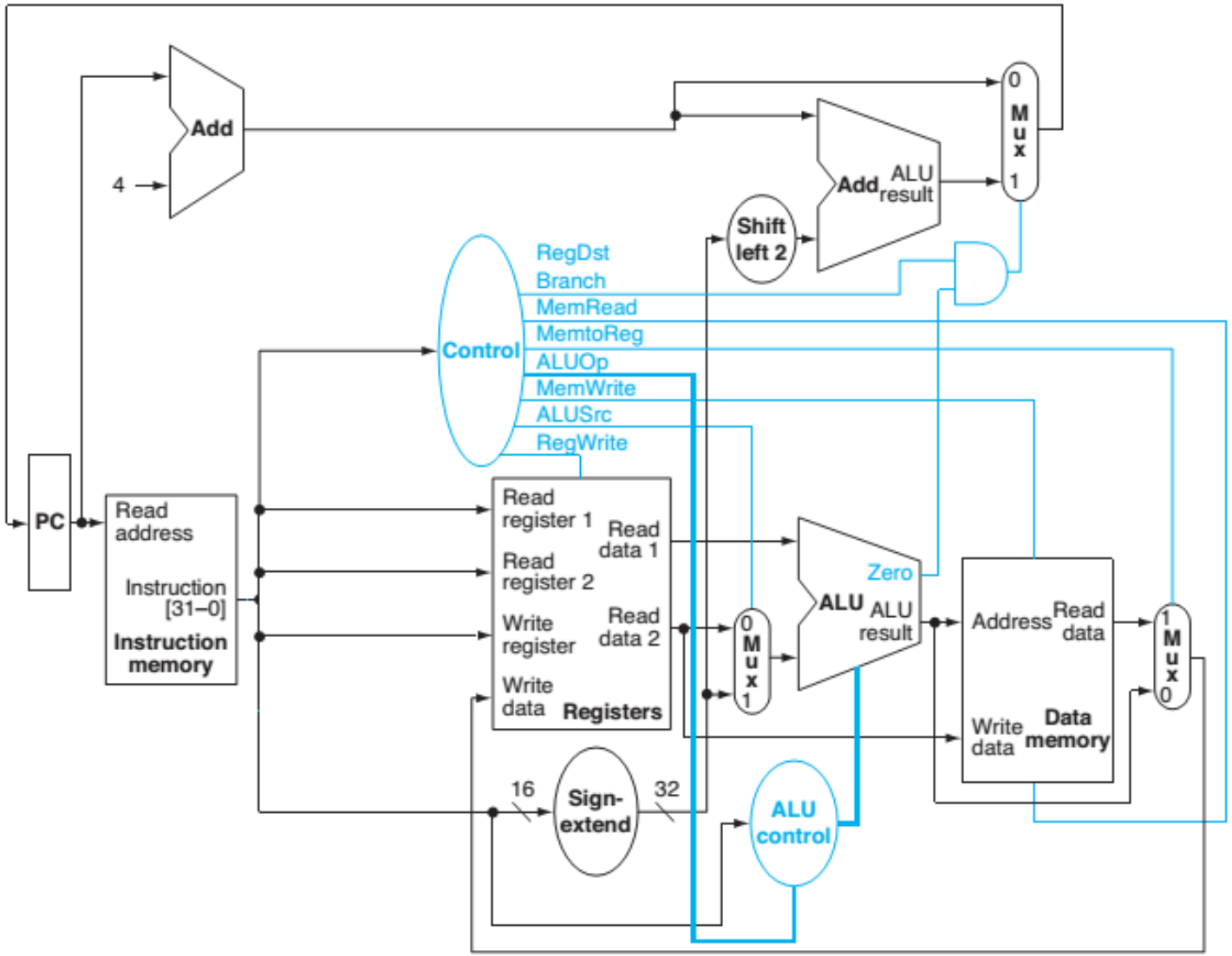
| Field | 4 | rs | rt | address |
|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:0 |

Nạp lệnh → Đọc một/hai thanh ghi → Sử dụng ALU → Chuyển đến địa chỉ lệnh tiếp theo dựa trên kết quả so sánh




*Một phần hình ảnh của datapath cho lệnh beq. Trong đó, ALU được sử dụng để tính toán điều kiện nhảy và bộ cộng dùng để tính địa chỉ của lệnh sẽ nhảy đến.*

# Xây dựng Datapath




*Datapath hoàn chỉnh cho nhóm 8 lệnh: add, sub, and, or, slt, lw, sw và beq*

# Bài tập

- Cho các lệnh sau, dựa vào sơ đồ khối hình 4.2:
  - AND Rd, Rs, Rt  Reg[Rd] = Reg[Rs] AND Reg[Rt]
  - SW Rt,Offs(Rs)  Mem[Reg[Rs] + Offs] = Reg[Rt]
- Xác định các giá trị tín hiệu cho từng lệnh trên
- Các khối sử dụng thực hiện cho lệnh trên.

# Bài tập

- Different execution units and blocks of digital logic have different latencies (time needed to do their work). In Figure 4.2 there are seven kinds of major blocks. Latencies of blocks along the critical (longest-latency) path for an instruction determine the minimum latency of that instruction. For the remaining three problems in this exercise, assume the following resource latencies:

| | I-Mem | Add | Mux | ALU | Regs | D-Mem | Control |
|---|---|---|---|---|---|---|---|
| a. | 200ps | 70ps | 20ps | 90ps | 90ps | 250ps | 40ps |
| b. | 750ps | 200ps | 50ps | 250ps | 300ps | 500ps | 300ps |

- What is the critical path for an MIPS AND instruction?
- What is the critical path for an MIPS load (LW) instruction?
- What is the critical path for an MIPS BEQ instruction?

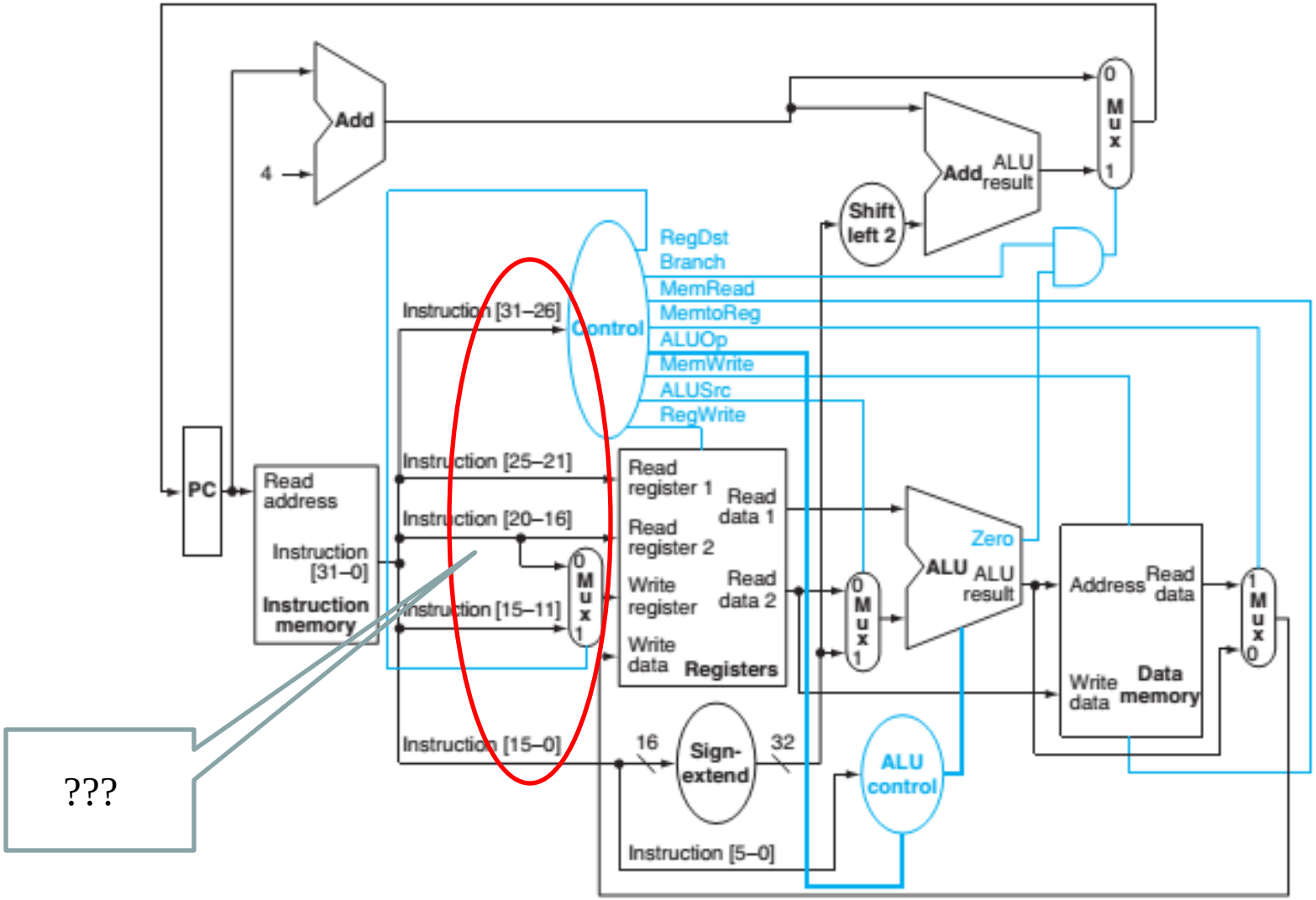
# BỘ XỬ LÝ

## Nội dung

1. Giới thiệu
2. Nhắc lại các quy ước thiết kế logic
3. Xây dựng đường truyền dữ liệu (datapath) đơn giản
4. Hiện thực datapath đơn chu kỳ

# Hiện thực datapath

## 1. Inputs của khối "Registers", "Control" và "Sign-extend"




Datapath với đầy đủ dữ liệu input cho từng khối

# Hiện thực datapath

| Field | 0 | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:11 | 10:6 | 5:0 |

a. R-type instruction

| Field | 35 or 43 | rs | rt | address |
|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:0 |

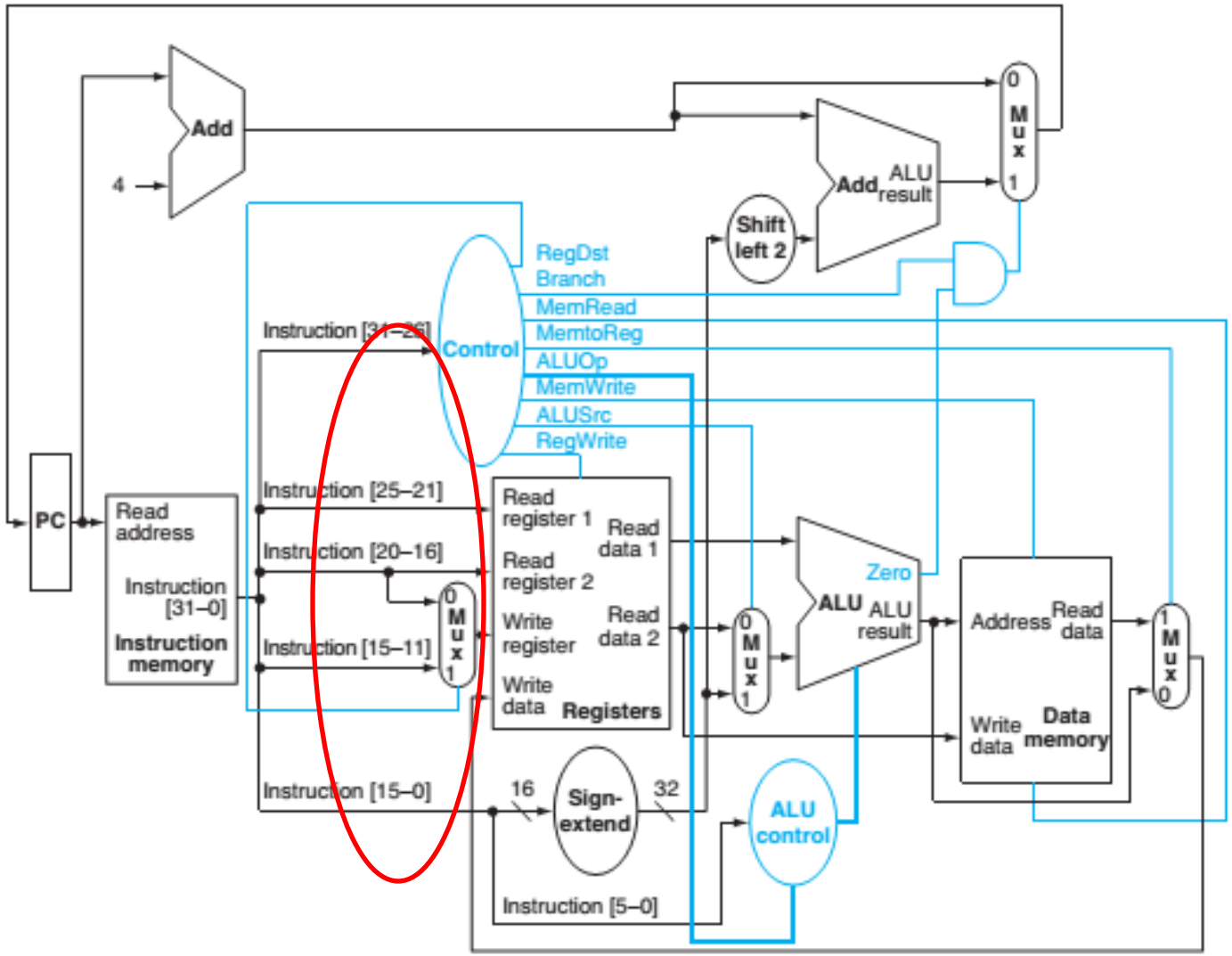
b. Load or store instruction

| Field | 4 | rs | rt | address |
|---|---|---|---|---|
| Bit positions | 31:26 | 25:21 | 20:16 | 15:0 |

c. Branch instruction

- Trường op (hay opcode) luôn chứa bits từ 31:26.
- Hai thanh ghi dùng để đọc trong tất cả các lệnh luôn luôn là rs và rt, tại vị trí bits từ 25:21 và 20:26.
- Thanh ghi nền cho lệnh load và store luôn là *rs* và tại vị trí bits 25:21.
- 16 bits offset cho *beq*, *lw* và *sw* thì luôn tại vị trí 15:0.
- Các thanh ghi đích dùng để ghi kết quả vào ở hai vị trí: Với *lw*, thanh ghi đích tại vị trí bits từ **20:16 (*rt*),** trong khi với nhóm lệnh logic và số học, thanh ghi đích ở vị trí **15:11 (*rd*).** Vì vậy, một multiplexor cần sử dụng ở đây để lựa chọn thanh ghi nào sẽ được ghi.

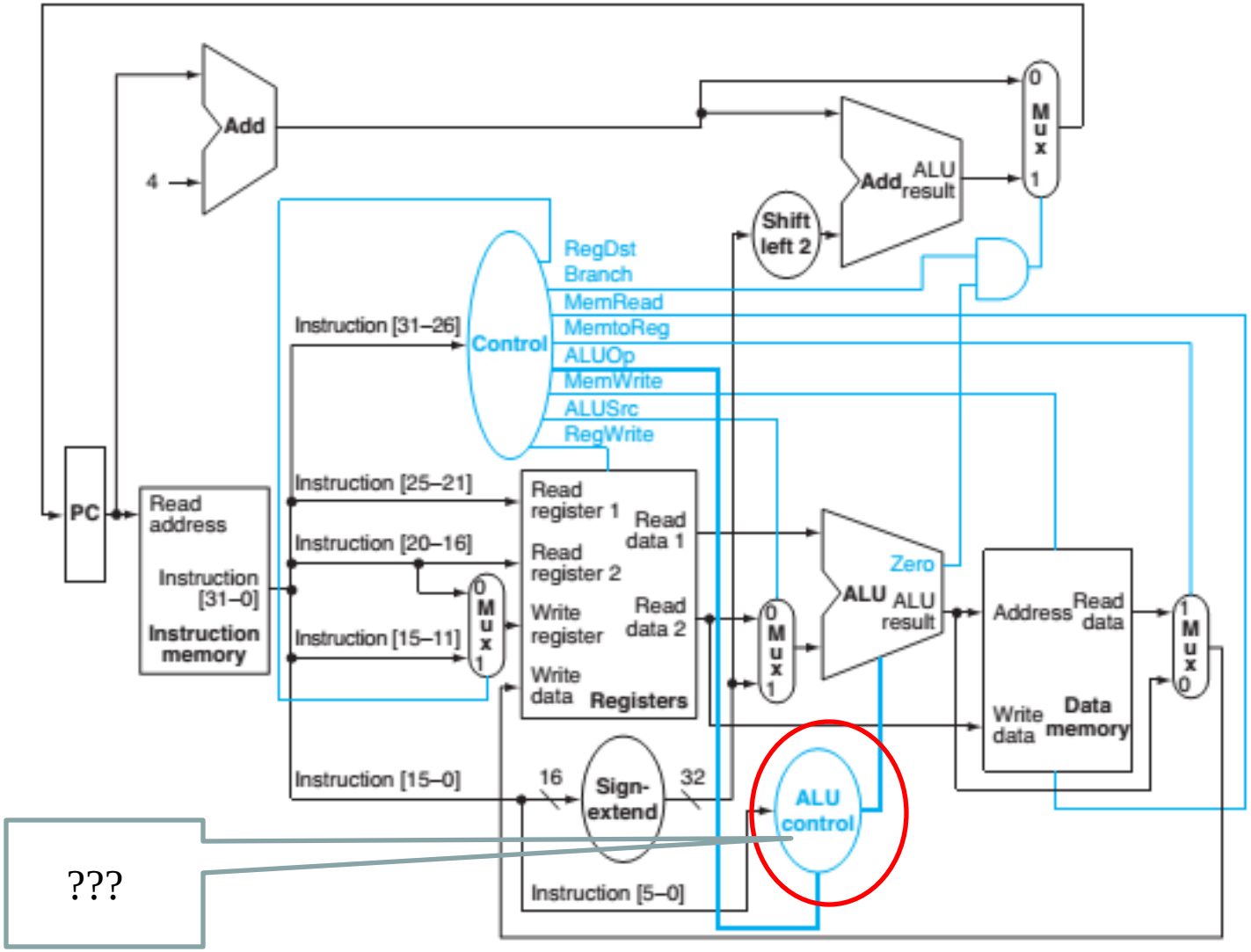
# Hiện thực datapath




*Datapath với đầy đủ dữ liệu input cho từng khối*

# Hiện thực datapath

## 2. Khối "ALU Control"



# Hiện thực datapath

Bộ ALU của MIPS gồm 6 chức năng tính toán dựa trên 4 bits điều khiển đầu vào:

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

Tùy thuộc vào từng nhóm lệnh mà ALU sẽ thực hiện 1 trong 5 chức năng đầu (NOR sẽ được dùng cho các phần khác)

- Với các lệnh **load word** và **store word**, ALU sử dụng chức năng **'add'** để tính toán địa chỉ của bộ nhớ
- Với các lệnh thuộc **nhóm logic và số học**, ALU thực hiện 1 trong 5 chức năng (***AND, OR, subtract, add,*** *và* ***set on less than***), tùy thuộc vào giá trị của trường funct (6 bits) trong mã máy lệnh.
- Với lệnh **nhảy nếu bằng**, ALU thực hiện chức năng **'subtract'** để xem điều khiện bằng có đúng không.

# Hiện thực datapath

Như vậy, để sinh ra 4 bits điều khiển ALU, một trong số các cách hiện thực có thể là sử dụng thêm một khối điều khiển "ALU Control"

"ALU Control" nhận input là 6 bits từ trường *funct* của mã máy, đồng thời dựa vào 2 bits "ALUOp" được sinh ra từ khối "Control" để sinh ra output là 4 bits điều khiển ALU, theo quy tắc như bảng sau:

| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

*Một gợi ý để sinh ra 4 bits điều khiển ALU dựa vào trường "opcode" và trường "funct" của mã máy.*

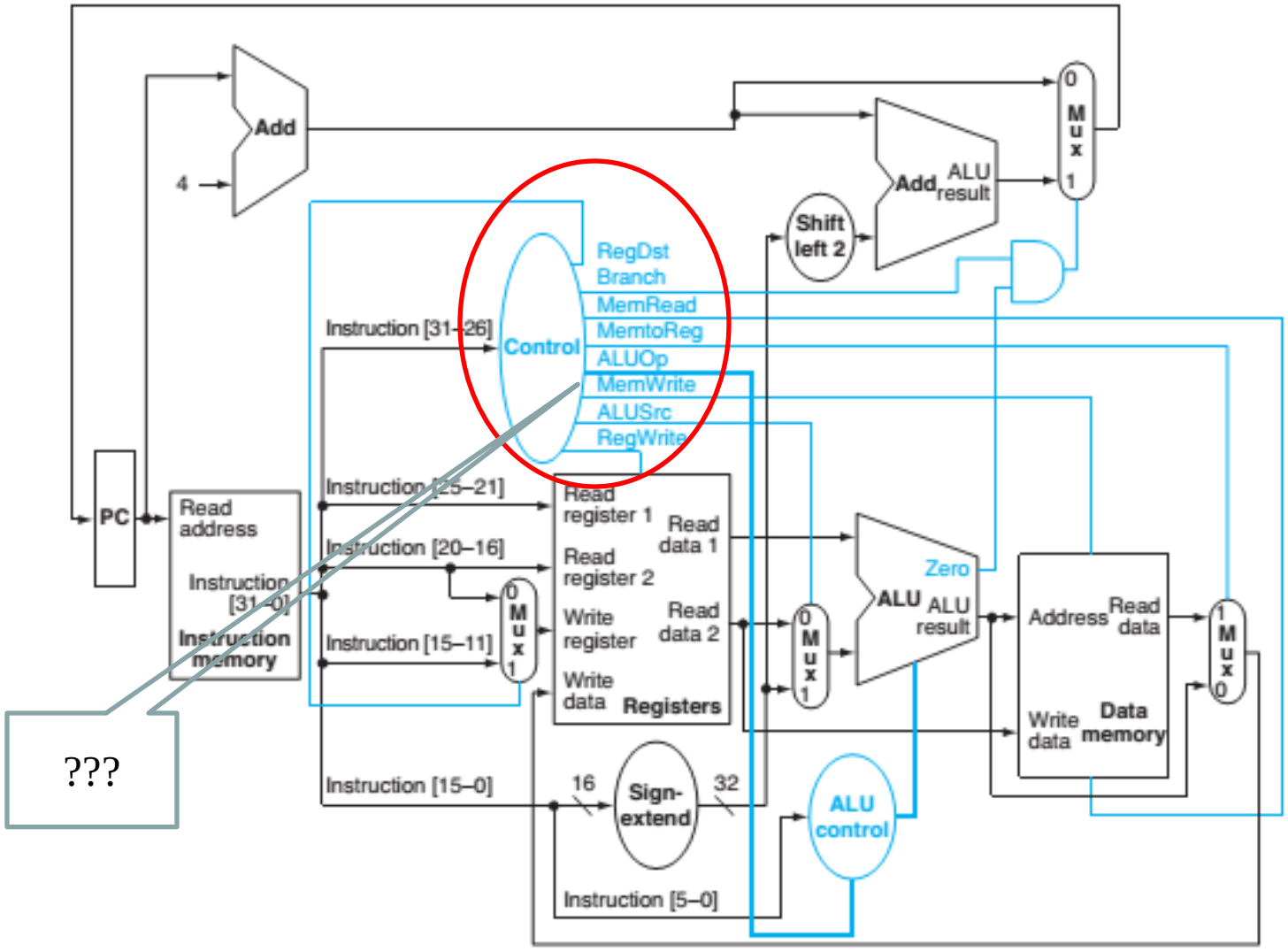
# Hiện thực datapath

**Bảng sự thật: Từ quy tắc hoạt động, bảng sự thật gợi ý cho khối "ALU Control" như sau**

| ALUOp | | Funct field | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | Operation |
| 0 | 0 | X | X | X | X | X | X | 0010 |
| 0 | 1 | X | X | X | X | X | X | 0110 |
| 1 | 0 | X | X | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 |
| 1 | 0 | X | X | 0 | 1 | 0 | 0 | 0000 |
| 1 | 0 | X | X | 0 | 1 | 0 | 1 | 0001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 |

# Hiện thực datapath

## 3. Khối điều khiển chính "Control"



# Hiện thực datapath

| Tên tín hiệu điều khiển | Tác động khi ở mức thấp | Tác động khi ở mức cao |
|---|---|---|
| RegDst | Thanh ghi đích cho thao tác ghi sẽ từ thanh ghi *rt* (bits từ 20:16) | Thanh ghi đích cho thao tác ghi sẽ từ thanh ghi *rd* (bits từ 15:11) |
| RegWrite | Khối "Registers" chỉ thực hiện mỗi chức năng đọc thanh ghi | Ngoài chức năng đọc, khối "Register" sẽ thực hiện thêm chức năng ghi. Thanh ghi được ghi là thanh ghi có chỉ số được đưa vào từ ngõ "Write register" và dữ liệu dùng ghi vào thanh ghi này được lấy từ ngõ "Write data" |
| ALUSrc | Input thứ hai cho ALU đến từ "Read data 2" của khối "Registers" | Input thứ hai cho ALU đến từ output của khối "Sign-extend" |
| Branch | Cho biết lệnh nạp vào không phải "beq". Thanh ghi PC nhận giá trị là PC + 4 | Lệnh nạp vào là lệnh "beq", kết hợp với điều kiện bằng thông qua cổng AND nhằm xác định xem lệnh tiếp theo có nhảy đến địa chỉ mới hay không. Nếu điệu kiện bằng đúng, PC nhận giá trị mới từ kết quả của bộ cộng "Add" |
| MemRead | (Không) | Khối "Data register" thực hiện chức năng đọc dữ liệu. Địa chỉ dữ liệu cần đọc được đưa vào từ ngõ "Address" và nội dung đọc được xuất ra ngõ "Read data" |
| MemWrite | (Không) | Khối "Data register" thực hiện chức năng ghi dữ liệu. Địa chỉ dữ liệu cần ghi được đưa vào từ ngõ "Address" và nội dung ghi vào lấy từ ngõ "Write data" |
| MemtoReg | Giá trị đưa vào ngõ "Write data" đến từ ALU | Giá trị đưa vào ngõ "Write data" đến từ khối "Data memory" |

*Tác động của các tín hiệu điều khiển*

# Hiện thực datapath

## Giá trị các tín hiệu điều khiển tương ứng với mỗi lệnh như sau:

| Instruction | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

*Khối "Control" trong datapath nhận input là 6 bits từ trường "opcode" của mã máy, dựa vào đó các tín hiệu điều khiển được sinh ra tương ứng như bảng.*

# Hiện thực datapath

## Bảng sự thật khối "Control":

| Input or output | Signal name | R-format | lw | sw | beq |
|---|---|---|---|---|---|
| Inputs | Op5 | 0 | 1 | 1 | 0 |
| | Op4 | 0 | 0 | 0 | 0 |
| | Op3 | 0 | 0 | 1 | 0 |
| | Op2 | 0 | 0 | 0 | 1 |
| | Op1 | 0 | 1 | 1 | 0 |
| | Op0 | 0 | 1 | 1 | 0 |
| Outputs | RegDst | 1 | 0 | X | X |
| | ALUSrc | 0 | 1 | 1 | 0 |
| | MemtoReg | 0 | 1 | X | X |
| | RegWrite | 1 | 1 | 0 | 0 |
| | MemRead | 0 | 1 | 0 | 0 |
| | MemWrite | 0 | 0 | 1 | 0 |
| | Branch | 0 | 0 | 0 | 1 |
| | ALUOp1 | 1 | 0 | 0 | 0 |
| | ALUOp0 | 0 | 0 | 0 | 1 |

*Bảng sự thật khối "Control"*

# Hiện thực datapath

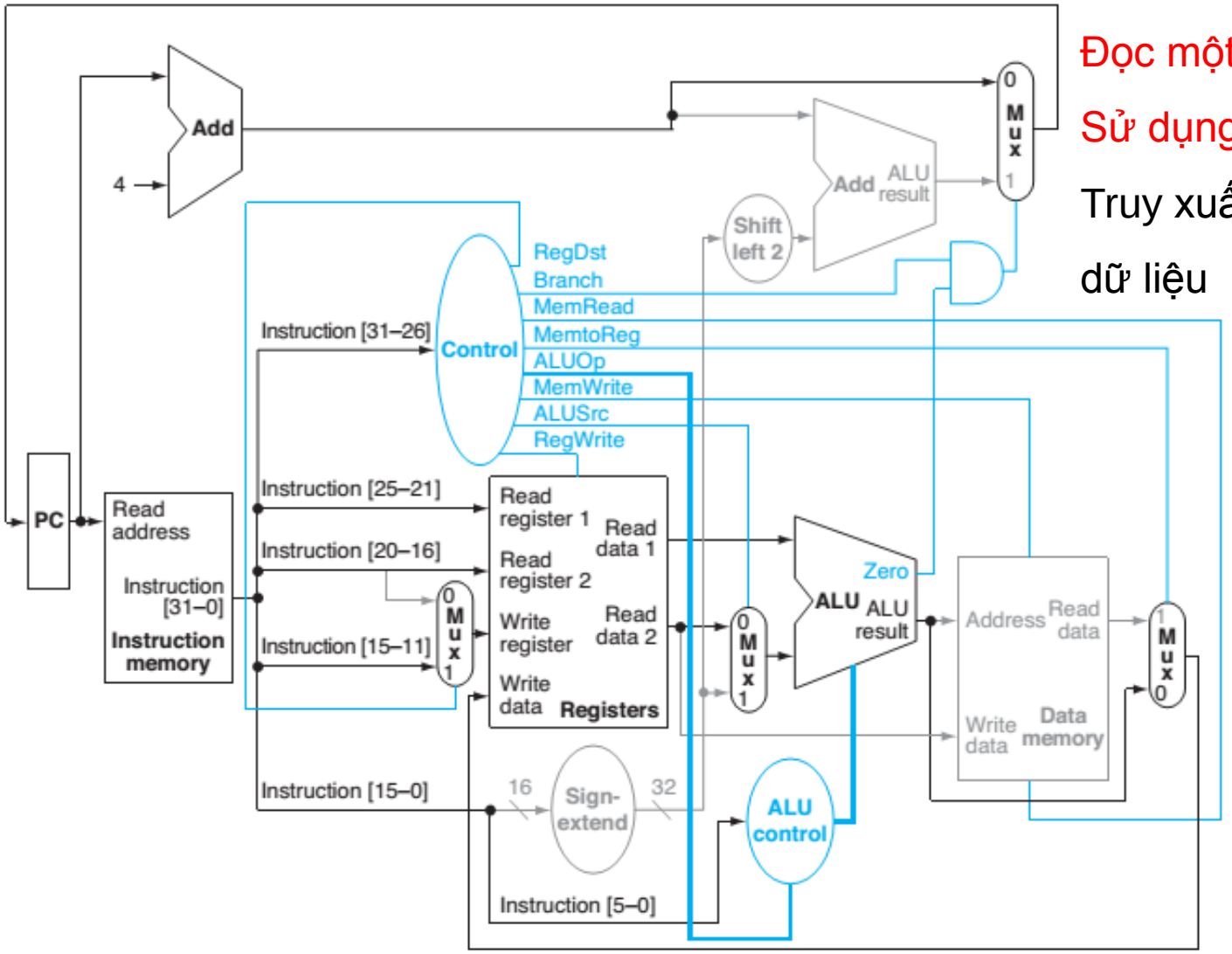
- **Hiện thực bộ xử lý đơn chu kỳ** (Single-cycle implementation hay single clock cycle implementation): là cách hiện thực sao cho bộ xử lý đáp ứng thực thi mỗi câu lệnh chỉ trong 1 chu kỳ xung clock ➔ đòi hỏi chu kỳ xung clock phải bằng thời gian của lệnh dài nhất.

- Cách hiện thực bộ xử lý như đã trình bày trên là cách hiện thực **đơn chu kỳ**:

  Lệnh dài nhất là *lw*, gồm truy xuất vào "Instruction memory", "Registers", "ALU", "Data memory" và quay trở lại "Registers", trong khi các lệnh khác không đòi hỏi tất cả các công đoạn trên → chu kỳ xung clock thiết kế phải bằng thời gian thực thi lệnh *lw*.

- Mặc dù hiện thực bộ xử lý đơn chu kỳ có CPI = 1 nhưng hiệu suất rất kém, vì một chu kỳ xung clock quá dài, các lệnh ngắn đều phải thực thi cùng thời gian với lệnh dài nhất.
  Vì vậy, **Hiện thực đơn chu kỳ hiện tại không còn được sử dụng (hoặc chỉ có thể chấp nhận cho các tập lệnh nhỏ)**

# Xem lại Datapath với từng nhóm lệnh




Nạp lệnh →

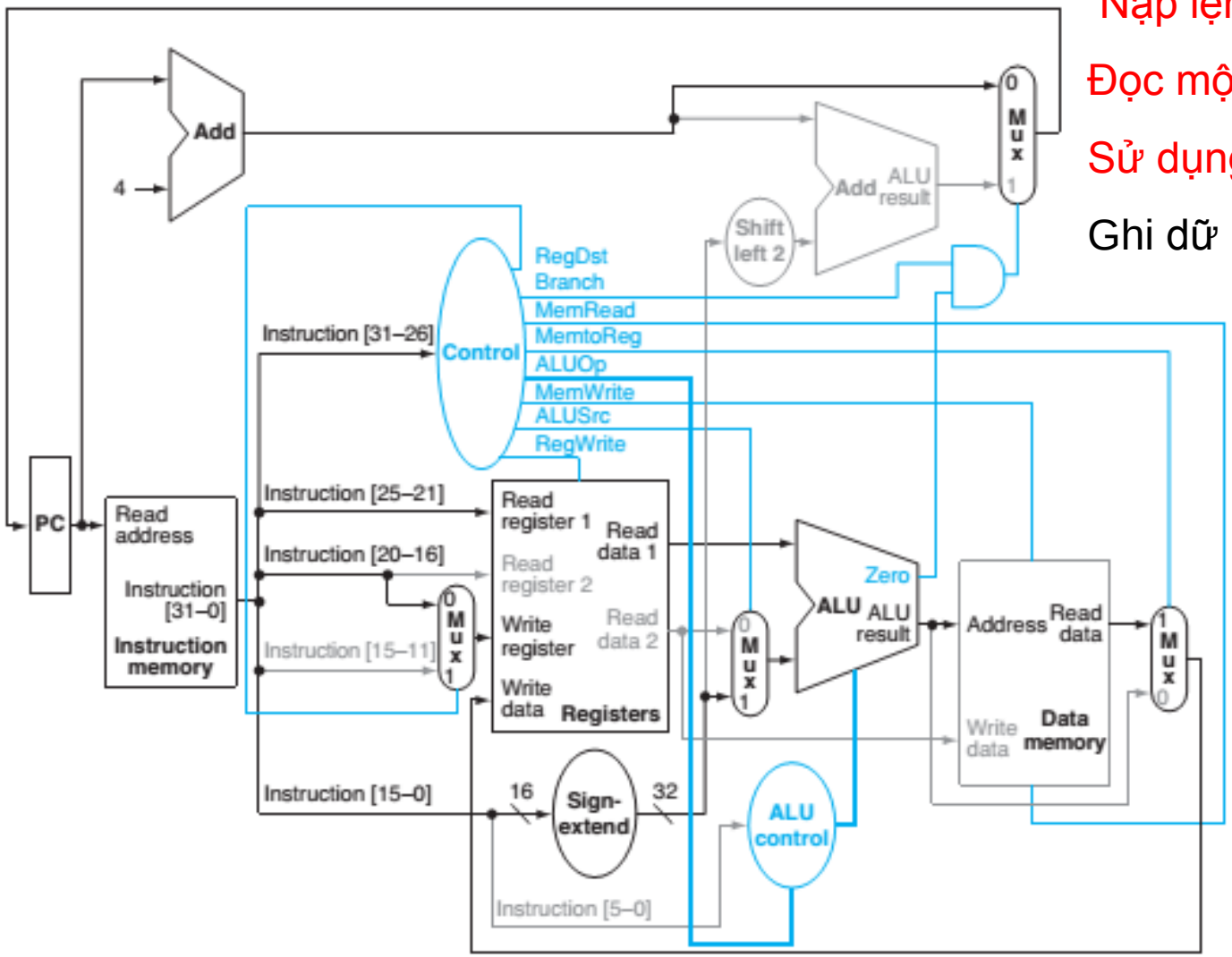
Đọc một/hai thanh ghi →

Sử dụng ALU →

Truy xuất bộ nhớ để đọc/ghi dữ liệu

*Các đường đậm nét là các đường hoạt động khi lệnh thuộc* ***nhóm logic và số học*** *thực thi*

# Xem lại Datapath với từng nhóm lệnh




Nạp lệnh →

Đọc một/hai thanh ghi →

Sử dụng ALU →

Ghi dữ liệu vào thanh ghi

*Các đường đậm nét là các đường hoạt động khi lệnh **lw** thực thi*

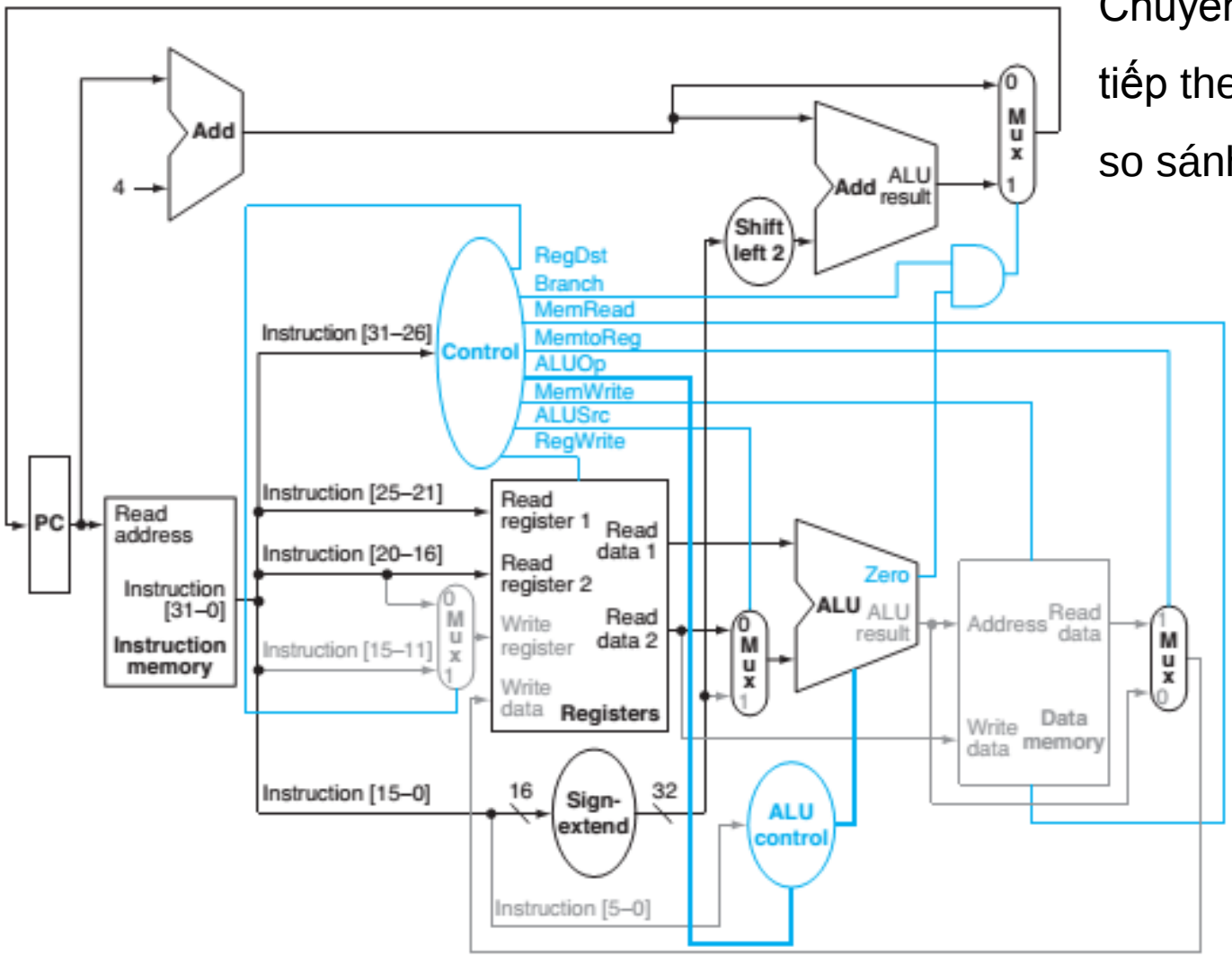
Nạp lệnh →

Đọc một/hai thanh ghi →

Sử dụng ALU →

Chuyển đến địa chỉ lệnh tiếp theo dựa trên kết quả so sánh

# Xem lại Datapath với từng nhóm lệnh




*Các đường đậm nét là các đường hoạt động khi lệnh **beq** thực thi*

# Exercise

- Different instructions require different control signals to be asserted in the datapath. The remaining problems in this exercise refer to the following two control signals:

| | Control Signal 1 | Control Signal 2 |
|---|---|---|
| **a.** | ALUSrc | Branch |
| **b.** | Jump | RegDst |

- What is the value of these two signals for this instruction?

# Exercise

- In this exercise we examine how the clock cycle time of the processor affects the design of the control unit, and vice versa. Problems in this exercise assume that the logic blocks used to implement the datapath have the following latencies:

- To avoid lengthening the critical path of the datapath shown in Figure 4.24, how much time can the control unit take to generate the MemWrite signal?
- Which control signal in Figure 4.24 has the most slack and how much time does the control unit have to generate it if it wants to avoid being on the critical path?
- Which control signal in Figure 4.24 is the most critical to generate quickly and how much time does the control unit have to generate it if it wants to avoid being on the critical path?

# Exercise

- The remaining problems in this exercise assume that the time needed by the control unit to generate individual control signals is as follows

| | RegDst | Jump | Branch | MemRead | MemtoReg | ALUOp | MemWrite | ALUSrc | RegWrite |
|---|---|---|---|---|---|---|---|---|---|
| **a.** | 500ps | 500ps | 450ps | 200ps | 450ps | 200ps | 500ps | 100ps | 500ps |
| **b.** | 1100ps | 1000ps | 1100ps | 800ps | 1200ps | 300ps | 1300ps | 400ps | 1200ps |

- What is the clock cycle time of the processor?
- If you can speed up the generation of control signals, but the cost of the entire processor increases by $1 for each 5ps improvement of a single control signal, which control signals would you speed up and by how much to maximize performance? What is the cost (per processor) of this performance improvement?

# Exercise

- The remaining problems in this exercise assume that the time needed by the control unit to generate individual control signals is as follows

| | RegDst | Jump | Branch | MemRead | MemtoReg | ALUOp | MemWrite | ALUSrc | RegWrite |
|---|---|---|---|---|---|---|---|---|---|
| **a.** | 500ps | 500ps | 450ps | 200ps | 450ps | 200ps | 500ps | 100ps | 500ps |
| **b.** | 1100ps | 1000ps | 1100ps | 800ps | 1200ps | 300ps | 1300ps | 400ps | 1200ps |

- If the processor is already too expensive, instead of paying to speed it up as we did in 4.10.5, we want to minimize its cost without further slowing it down. If you can use slower logic to implement control signals, saving $1 of the processor cost for each 5ps you add to the latency of a single control signal, which control signals would you slow down and by how much to reduce the processor's cost without slowing it down?

# Exercise

- In this exercise we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word:

| | Instruction word |
|---|---|
| a. | 10101100011000100000000000010100 |
| b. | 00000000100000100000100000101010 |

- What are the outputs of the sign-extend and the jump "Shift left 2" unit (near the top of Figure 4.24) for this instruction word?
- What are the values of the ALU control unit's inputs for this instruction?
- What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

# CHƯƠNG 4

# BỘ XỬ LÝ (THE PROCESSOR)

- Phần 1. Xây dựng đường truyền dữ liệu (Datapath)
- Phần 2. Kỹ thuật ống dẫn (Pipeline)

# Kỹ thuật ống dẫn (pipeline)

**Pipeline** là một kỹ thuật mà trong đó các lệnh được thực thi theo kiểu chồng lắp lên nhau.

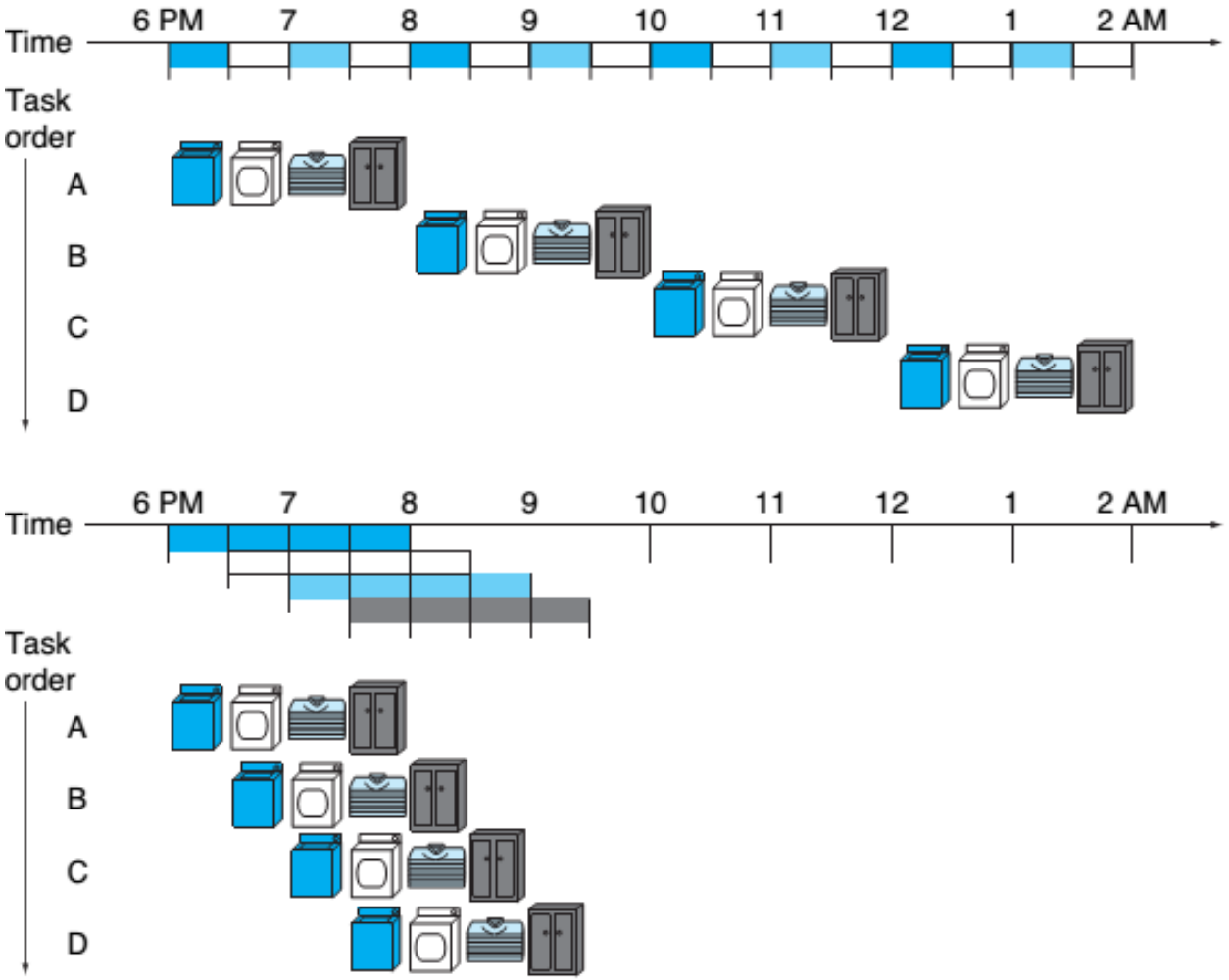
Ví dụ minh họa hoạt động như thế nào là **không pipeline** hay **pipeline**:

- Giả sử một phòng có nhiều người, mỗi người đều cần giặt quần áo bẩn của mình. Quá trình giặt quần áo bao gồm 4 công đoạn:
  1. Đặt quần áo bẩn vào máy giặt để giặt
  2. Khi máy giặt hoàn thành, đưa quần áo ướt vào máy sấy
  3. Khi máy sấy hoàn thành, đặt quần áo khô lên bàn và ủi
  4. Khi ủi hoàn tất, xếp quần áo vào tủ

✓ Nếu một người hoàn tất tất cả các công đoạn giặt quần áo (xong công đoạn ủi, xếp quần áo vào tủ) thì người khác mới bắt đầu (bắt đầu đặt quần áo bẩn vào máy giặt), quá trình thực hiện này gọi là **không pipeline**.

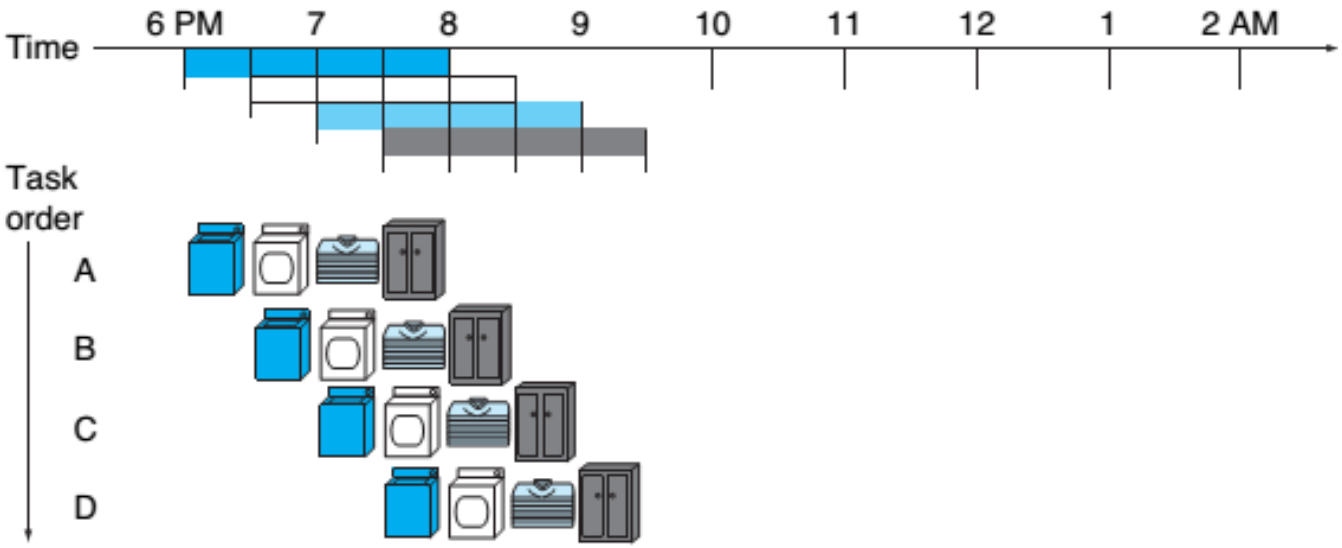
✓ Tuy nhiên, rõ ràng rằng khi người trước hoàn thành công đoạn 1, sang công đoạn 2 thì máy giặt đã trống, lúc này người tiếp theo có thể đưa quần áo bẩn vào giặt. Như vậy, người tiếp theo không cần phải chờ người trước xong công đoạn thứ 4 mới có thể bắt đầu, mà ngay khi người trước đến công đoạn thứ 2 thì người tiếp theo đã có thể bắt đầu công đoạn thứ nhất và cứ tiếp tục như vậy. Quá trình thực hiện chồng lắp này gọi là **pipeline.**

# Kỹ thuật ống dẫn (pipeline)




*Hình ảnh 4 người A, B, C, D giặt quần áo theo kiểu tiếp cận không pipeline (hình trên) và pipeline (hình dưới)*

# Kỹ thuật ống dẫn (pipeline)




- Cách tiếp cận dùng kỹ thuật pipeline tiêu tốn ít thời gian hơn cho tất cả các công việc hoàn tất bởi vì các công việc được thực hiện song song, vì vậy số công việc hoàn thành trong một giờ sẽ nhiều hơn so với không pipeline.

- Chú ý, pipeline không làm giảm thời gian hoàn thành một công việc mà làm giảm thời gian hoàn thành tổng số công việc (như trong ví dụ trên, thời gian cho người A hoàn thành việc giặt khi áp dụng pipeline hay không pipeline đều là 2 giờ, nhưng tổng số giờ cho 4 người A, B, C và D hoàn thành dùng pipeline giảm rất nhiều so với không pipeline)

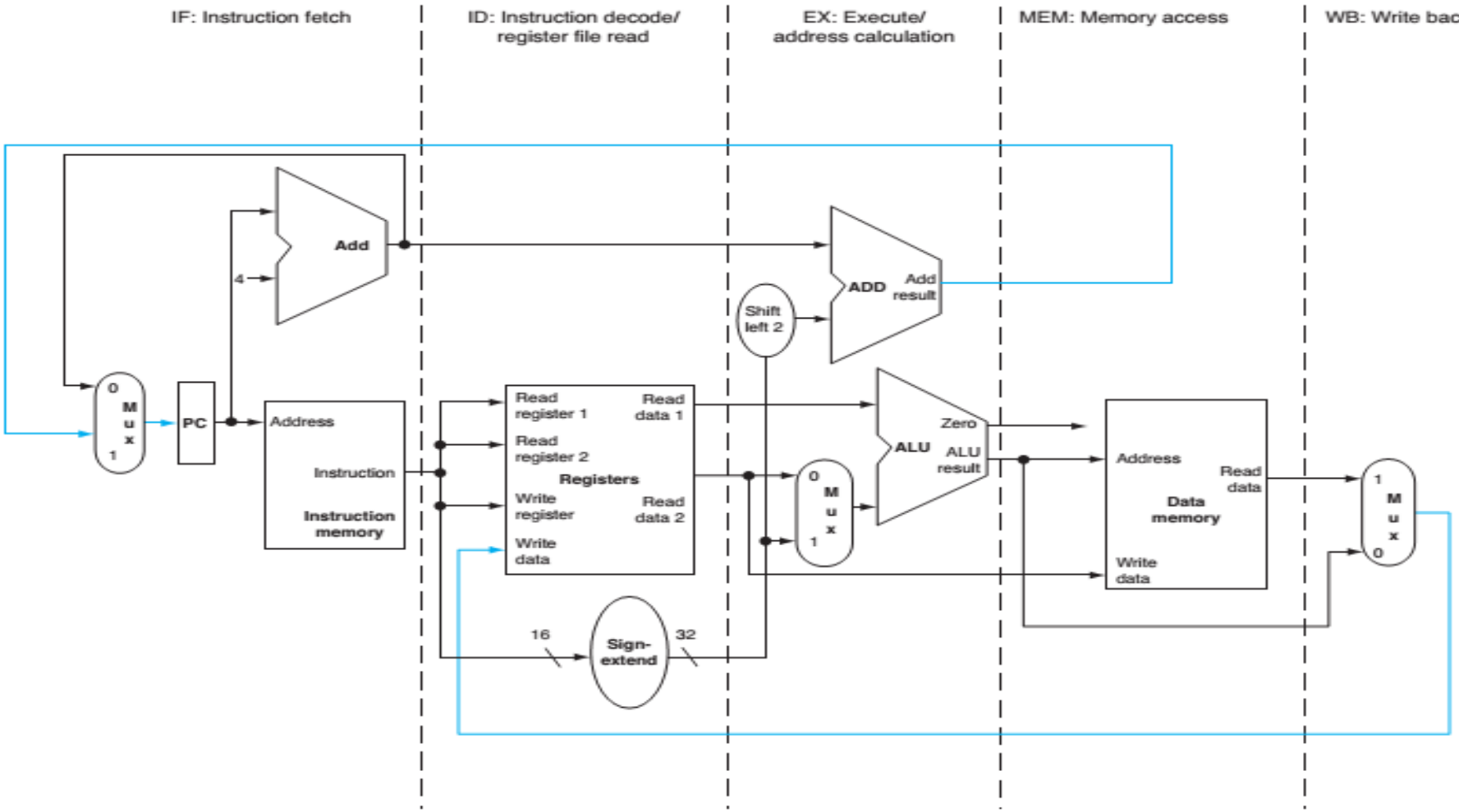
# Kỹ thuật ống dẫn (pipeline)

Tương tự việc giặt quần áo, thay vì một lệnh phải chờ lệnh trước đó hoàn thành mới được thực thi thì các lệnh trong một chương trình của bộ xử lý có thể thực thi theo kiểu pipeline.

Khi thực thi, các lệnh MIPS được chia làm 5 công đoạn:

1. Nạp lệnh từ bộ nhớ
2. Giải mã lệnh và đọc các thanh ghi cần thiết (MIPS cho phép đọc và giải mã đồng thời)
3. Thực thi các phép tính hoặc tính toán địa chỉ
4. Truy xuất các toán hạng trong bộ nhớ
5. Ghi kết quả cuối vào thanh ghi

Vì vậy, MIPS pipeline trong chương này xem như có 5 công đoạn (còn gọi là pipeline 5 tầng)



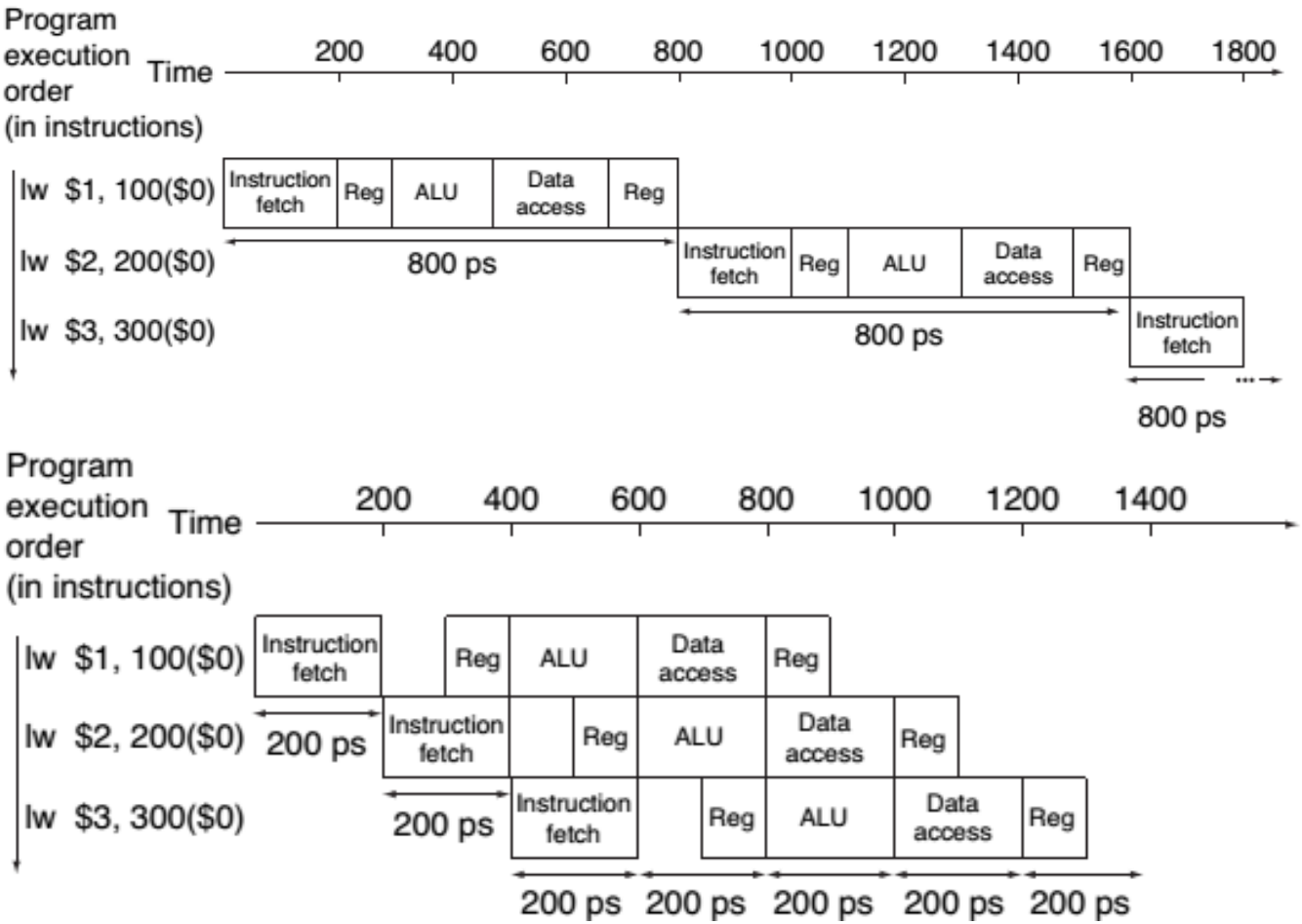

1. Nạp lệnh từ bộ nhớ – **IF**
2. Giải mã lệnh và đọc các thanh ghi – **ID**
3. Thực thi – **EX**
4. Truy xuất bộ nhớ – **MEM**
5. Ghi kết quả vào thanh ghi – **WB**

# Kỹ thuật ống dẫn (pipeline)

- Xét một bộ xử lý với 8 lệnh cơ bản: load word (***lw***), store word (***sw***), add (***add***), subtract (***sub***), AND (***and***), OR (***or***), set less than (***slt***), và nhảy với điều kiện bằng (***beq***).
- Giả sử thời gian hoạt động các công đoạn như sau: 200 ps cho truy xuất bộ nhớ, 200 ps cho tính toán của ALU, 100 ps cho thao tác đọc/ghi thanh ghi
- So sánh thời gian trung bình giữa các lệnh của hiện thực đơn chu kỳ và pipeline.

| Instruction class | Instruction fetch | Register read | ALU operation | Data access | Register write | Total time |
|---|---|---|---|---|---|---|
| Load word (`lw`) | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| Store word (`sw`) | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format (`add`, `sub`, `AND`, `OR`, `slt`) | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| Branch (`beq`) | 200 ps | 100 ps | 200 ps | | | 500 ps |

# Kỹ thuật ống dẫn (pipeline)




*Ví dụ hình ảnh 3 lệnh lw thực hiện theo kiểu không pipeline, đơn chu kỳ (hình trên) và có pipeline (hình dưới)*

→ Thời gian giữa lệnh thứ nhất và thứ tư trong không pipeline là 3 x 800 = 2400 ps, nhưng trong pipeline là 3 x 200 = 600 ps

# Kỹ thuật ống dẫn (pipeline)

## Sự tăng tốc của pipeline

❖ **Trong trường hợp lý tưởng:** khi mà các công đoạn pipeline hoàn toàn bằng nhau thì thời gian giữa hai lệnh liên tiếp được thực thi trong pipeline bằng:

$$\text{Thời gian giữa hai lệnh liên tiếp pipeline} = \frac{\text{thời gian giữa hai lệnh liên tiếp không pipeline}}{\text{số tầng pipeline}}$$

*Như vậy, trong ví dụ trên, thời gian giữa hai lệnh liên tiếp có pipeline bằng 160 ps (800:5 = 160)*

→ **Trong trường hợp lý tưởng, pipeline sẽ tăng tốc so với không pipeline với số lần đúng bằng số tầng của pipeline.**

❖ **Trong thực tế:** Các công đoạn thực tế không bằng nhau, việc áp dụng pipeline phải chọn công đoạn dài nhất để làm một chu kỳ pipeline.

Vì vậy, trong ví dụ trên, thời gian liên tiếp giữa hai lệnh pipeline là 200 ps. Và áp dụng pipeline tăng tốc gấp 4 lần so với không pipeline.

*Speed-up ≈ Thời gian giữa hai lệnh liên tiếp không pipeline : Thời gian giữa hai lệnh liên tiếp pipeline*
*≈ 800 : 200 = **4** < **5** (number pipeline stages)*

→ **Trong thực tế, pipeline sẽ tăng tốc so với không pipeline với số lần nhỏ hơn số tầng của pipeline.**

# Kỹ thuật ống dẫn (pipeline)

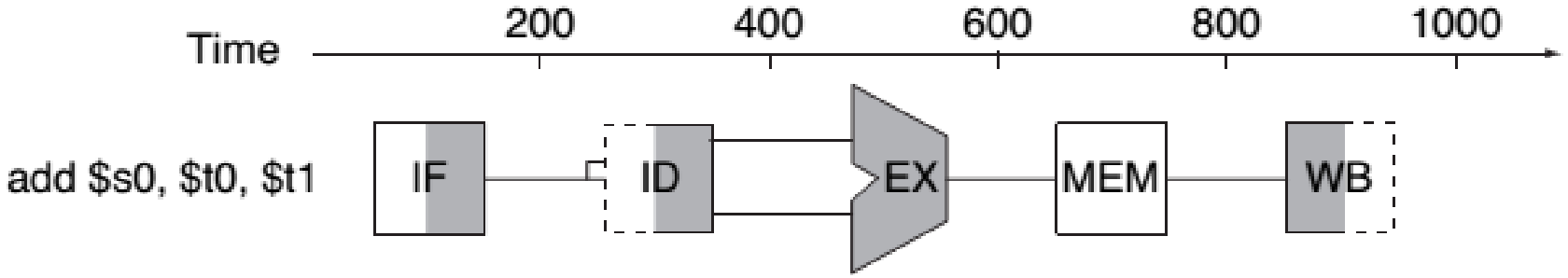
**Lưu ý, pipeline tăng tốc so với không pipeline:**

- Kỹ thuật pipeline không giúp giảm thời gian thực thi của từng lệnh riêng lẽ mà giúp giảm tổng thời gian thực thi của đoạn lệnh/chương trình chứa nhiều lệnh (từ đó giúp thời gian trung bình của mỗi lệnh giảm)

- Việc giúp giảm thời gian thực thi cho nhiều lệnh vô cùng quan trọng, vì các chương trình chạy trong thực tế thông thường lên đến hàng tỉ lệnh.

# Kỹ thuật ống dẫn (pipeline)

## Quy ước trình bày 5 công đoạn thực thi một lệnh của pipeline:




Lưu ý cách vẽ hình các công đoạn pipeline như sau:

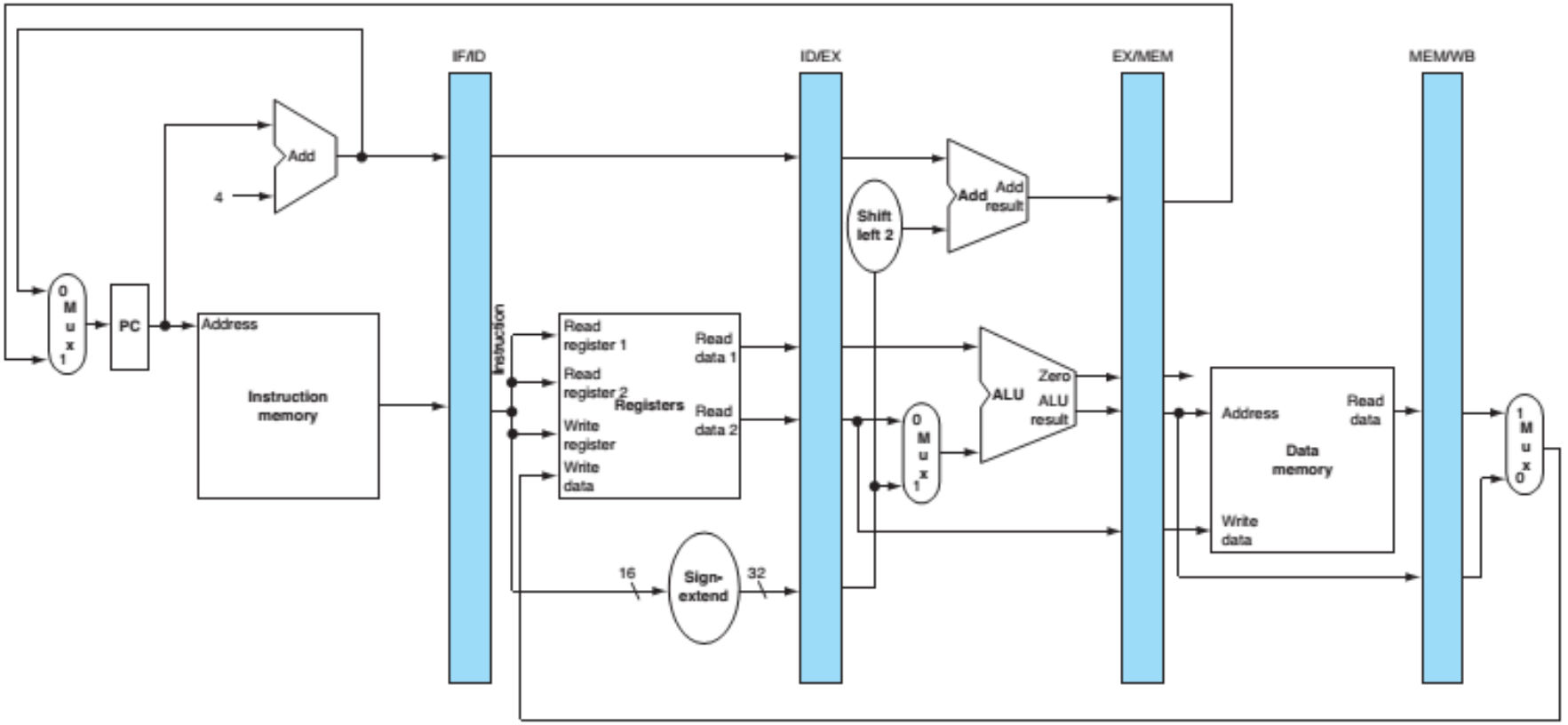
- ✓ Khối tô đen hoàn toàn hoặc để trắng hoàn toàn: Trong mỗi công đoạn pipeline, nếu lệnh thực thi không làm gì trong công đoạn này sẽ được tô trắng, ngược lại sẽ được tô đen.

  *Ví dụ lệnh "add" có EX đen và MEM trắng tức lệnh này có tính toán trong công đoạn EX và không truy xuất bộ nhớ dữ liệu trong công đoạn MEM.*

- ✓ Các công đoạn liên qua đến bộ nhớ và thanh ghi có thể tô nữa trái hoặc nữa phải đen: Nếu nữa phải tô đen, tức công đoạn đó đang thực hiện thao tác đọc; ngược lại nếu nữa trái tô đen, công đoạn đó đang thực hiện thao tác ghi.

# Kỹ thuật ống dẫn (pipeline)

## Hình ảnh datapath có hỗ trợ pipeline



# Kỹ thuật ống dẫn (pipeline)

**Các xung đột có thể xảy ra khi áp dụng kỹ thuật pipeline (Pipeline Hazards):**

**Xung đột** là trạng thái mà lệnh tiếp theo không thể thực thi trong chu kỳ pipeline ngay sau đó (hoặc thực thi nhưng sẽ cho ra kết quả sai), thường do một trong ba nguyên nhân sau:

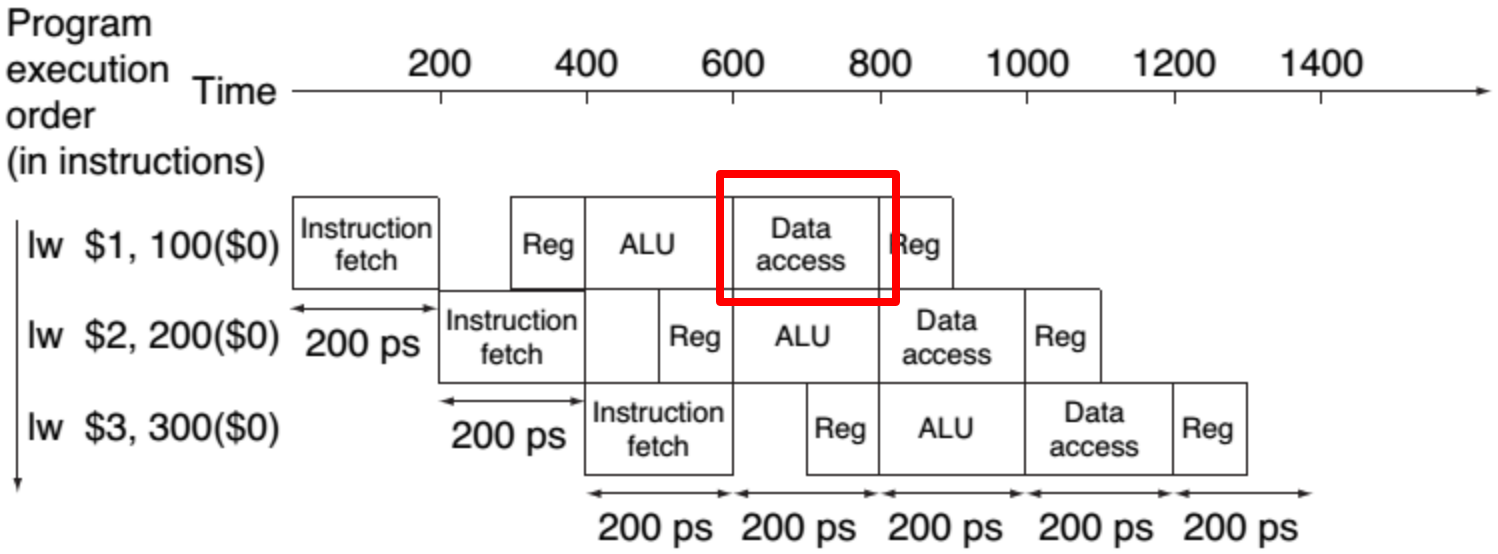
- **Xung đột cấu trúc (Structural hazard):** là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do phần cứng cần không thể hỗ trợ. Nói cách khác, xung đột cấu trúc xảy ra khi có hai lệnh cùng truy xuất vào một tài nguyên phần cứng nào đó cùng một lúc.

- **Xung đột dữ liệu (Data hazard):** là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do dữ liệu mà lệnh này cần vẫn chưa sẵn sàng.

- **Xung đột điều khiển (Control/Branch hazard):** là khi một lệnh dự kiến không thể thực thi trong đúng chu kỳ pipeline của nó do lệnh nạp vào không phải là lệnh được cần. Xung đột này xảy ra trong trường hợp luồng thực thi chứa các lệnh nhảy.

# Kỹ thuật ống dẫn (pipeline)

## Xung đột cấu trúc




Ví dụ về xung đột cấu trúc:

Giả sử rằng chúng ta có một bộ nhớ đơn duy nhất thay vì hai bộ nhớ lệnh và dữ liệu rời rạc nhau. Nếu pipeline trong ví dụ ở hình trên có thêm lệnh thứ tư thì trong chu kỳ pipeline từ 600 tới 800 khi lệnh thứ nhất thực hiện truy xuất bộ nhớ lấy dữ liệu thì lệnh thứ tư sẽ thực hiện truy xuất bộ nhớ lấy lệnh. Do không có bộ nhớ lệnh và dữ liệu riêng lẽ, trong trường hợp này sẽ có xung đột cấu trúc xảy ra.

# Kỹ thuật ống dẫn (pipeline)

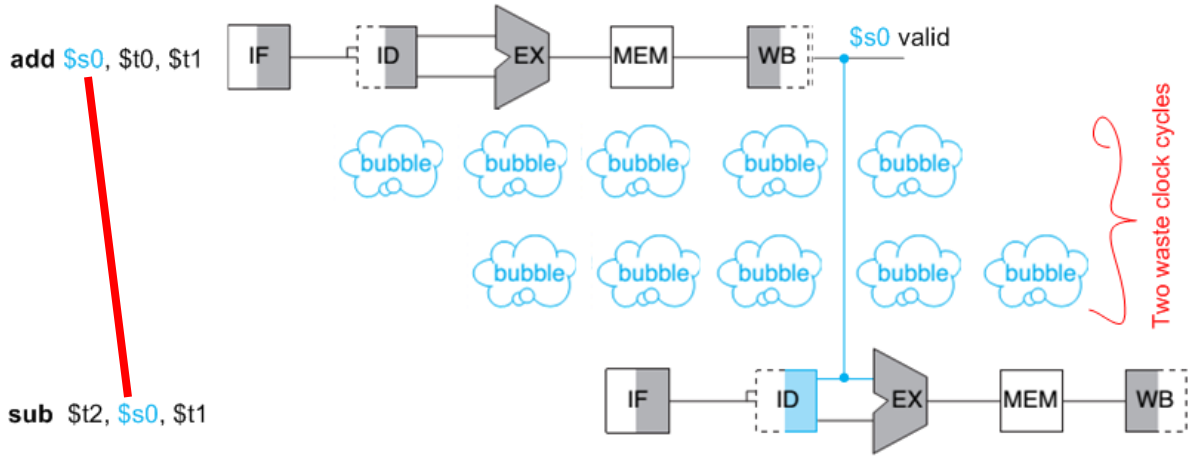
## Xung đột dữ liệu

Ví dụ cho đoạn lệnh sau: *add $s0, $t0, $t1*

*sub $t2, $s0, $t3*

Trong ví dụ trên, nếu áp dụng pipeline bình thường thì công đoạn ID của lệnh *sub* sẽ thực hiện cùng lúc với cộng đoạn EX của lệnh *add*. Trong công đoạn ID, lệnh *sub* sẽ cần đọc giá trị của thanh ghi *$s0*, trong khi đó giá trị mới của thanh ghi $s0 phải tới công đoạn WB của lệnh *add* mới sẵn sàng. Vì vậy, nếu thực hiện pipeline thông thường, trường hợp này sẽ xảy ra xung đột dữ liệu

Một cách giải quyết có thể trong trường hợp này là chờ thêm hai chu kỳ xung xung clock thì lệnh add mới được nạp vào



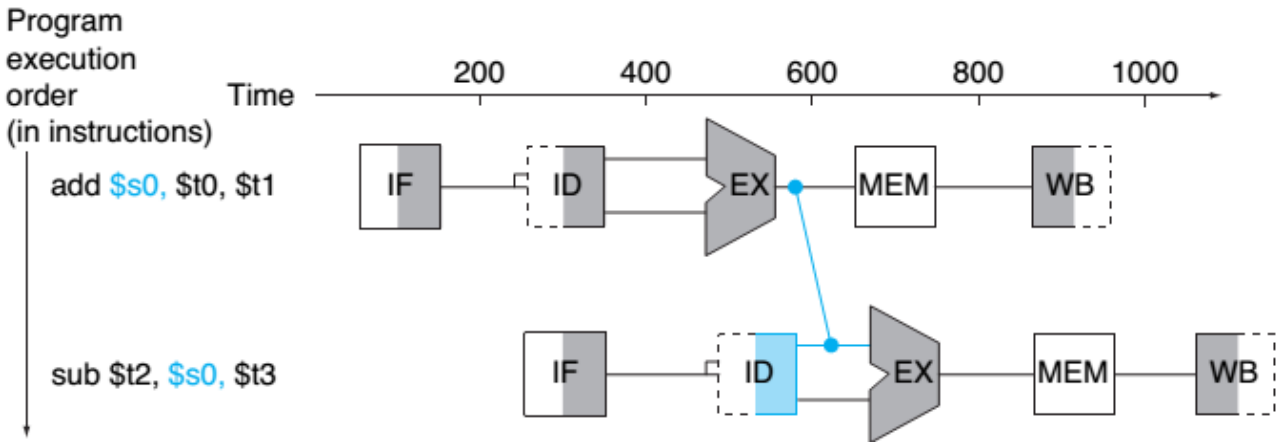
# Kỹ thuật ống dẫn (pipeline)

## Xung đột dữ liệu

❖ Thay vì chờ một số chu kỳ đến khi dữ liệu cần sẵn sàng, một kỹ thuật có thể được áp dụng để rút ngắn số chu kỳ rỗi, gọi là **kỹ thuật nhìn trước (forwarding** hay **bypassing).**

Như trong ví dụ trước, thay vì chờ sau hai chu kỳ rỗi mới nạp lệnh *add* vào, ngay khi ALU hoàn thành tính toán tổng cho lệnh *add* thì tổng này cũng được cung cấp ngay cho công đoạn EX của lệnh *sub* (thông qua một bộ đệm  dữ liệu gắn thêm bên trong) để ALU tính toán kết quả cho *sub* nhanh.

❖ Kỹ thuật nhìn trước: một phương pháp giải quyết xung đột dữ liệu bằng đưa thêm vào các bộ đệm phụ bên trong, các dữ liệu cần có thể được truy xuất từ bộ đệm này hơn là chờ đợi đến khi nó sẵn sàng trong bộ nhớ hay trong thanh ghi.



# Kỹ thuật ống dẫn (pipeline)

## Xung đột dữ liệu

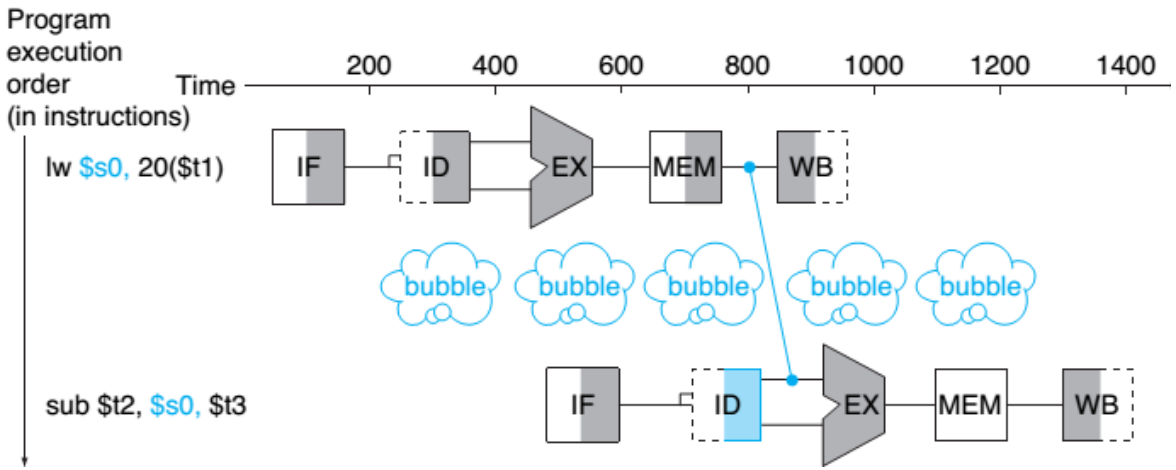
❖ Lưu ý, với lệnh ***lw*** và các lệnh có chức năng tương tự, thông thường kết quả của nó không phải khi hoàn tất công đoạn EX mà là khi hoàn tất công đoạn MEM.

Xét ví dụ sau: *lw $s0, 20($t1)*

*sub $t2, $s0, $t3*

Với lệnh *lw*, dữ liệu mong muốn sẽ chỉ sẵn sàng sau 4 chu kỳ pipeline (tức sau khi công đoạn MEM hoàn tất). Vì vậy, giả sử dữ liệu đầu ra của công đoạn MEM của lệnh *lw* được truyền tới đầu vào công đoạn EX của lệnh sub theo sau, thì lệnh *sub* vẫn phải chờ sau một chu kỳ rỗi mới được nạp vào.

❖ Kỹ thuật forwarding có thể hỗ trợ giải quyết xung đột dữ liệu hiệu quả, tuy nhiên nó không thể ngăn chặn tất cả các trường hợp chu kỳ rỗi

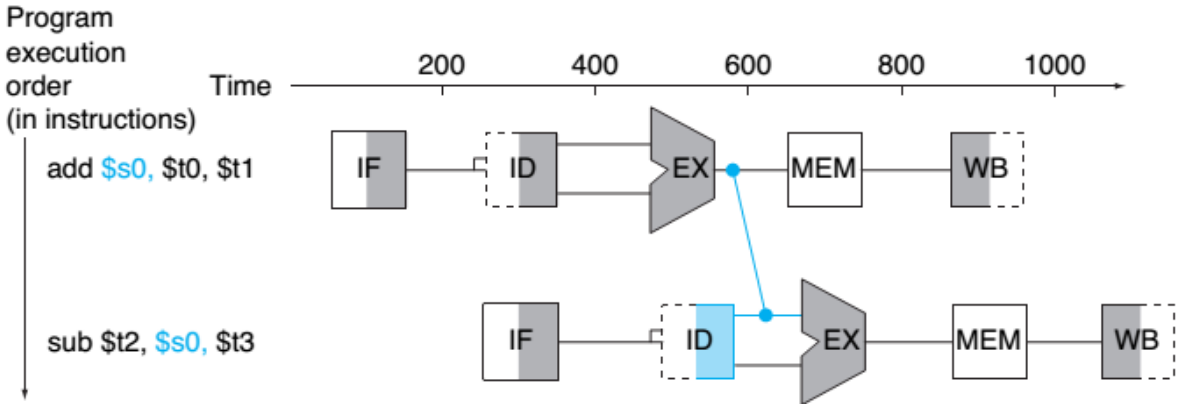


# Kỹ thuật ống dẫn (pipeline)

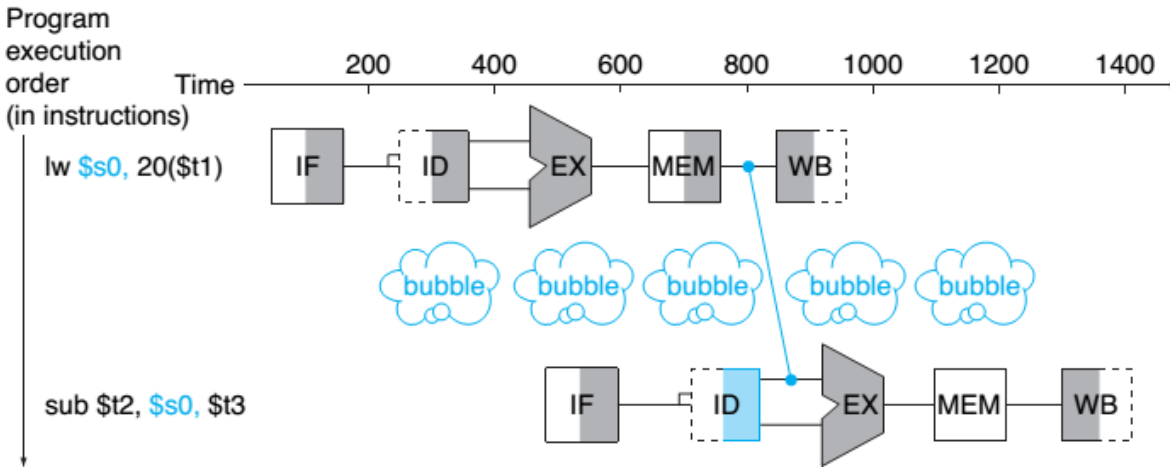
## Xung đột dữ liệu

Tóm lại, với kỹ thuật forwarding có:

- ✓ **ALU-ALU forwarding** hay **EX-EX forwarding** (hình 1)
- ✓ **MEM-ALU forwarding** hay **MEM-EX forwarding** (hình 2)




*Hình 1.*




*Hình 2.*

# Kỹ thuật ống dẫn (pipeline)

## Xung đột điều khiển

- Một số lệnh nhảy có điều kiện và không điều kiện trong MIPS (branches, jump) tạo ra xung đột điều kiển này

  Ví dụ xét đoạn chương trình sau:

  *add $4, $5, $6*

  *beq $1, $2, label*

  *lw $3, 300($s0)*

  Nếu áp dụng pipeline thông thường, tại chu kỳ thứ ba của pipeline, khi *beq* đang thực thi công đoạn ID thì lệnh *lw* sẽ được nạp vào. Nhưng nếu điều kiện bằng của lệnh *beq* xảy ra thì lệnh thực hiệp tiếp sau đó không phải là *lw* mà là lệnh được gán nhãn '*label*', lúc này xảy ra xung đột điều khiên.

- Các giải pháp giải quyết xung đột điều khiển (tham khảo thêm mục 4.8, sách tham khảo chính)

# Example

- we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| **a.** | 250ps | 350ps | 150ps | 300ps | 200ps |
| **b.** | 200ps | 170ps | 220ps | 210ps | 150ps |

- What is the clock cycle time in a pipelined and non-pipelined processor?
- What is the total latency of an LW instruction in a pipelined and non-pipelined processor?
- How many time does it take processor to execute 3 statement? (pipeline – not pipeline), if hazard, not?

  OR R1,R2,R3

  OR R2,R1,R4

  OR R1,R1,R2

# Example

- Hazards?

  SW R16,–100(R6)

  LW R4,8(R16)

  ADD R5,R4,R4

  SUB R1, R3, R2

- Hazards?

  OR R1,R2,R3

  OR R2,R1,R4

  OR R1,R1,R2