

Computer Architecture

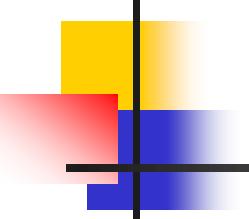
Computer Science & Engineering

Chương 4

Bộ Xử lý

Lộ trình dữ liệu – Điều khiển

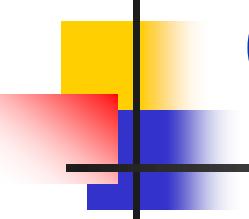




Dẫn nhập

- Các yếu tố xác định hiệu xuất Bộ Xử lý
 - Số lệnh (Instruction Count)
 - Xác định bởi “Kiến trúc tập lệnh” ISA và Trình biên dịch
 - Số chu kỳ cho mỗi lệnh và thời gian chu kỳ đ/hồ
 - Xác định bằng phần cứng CPU
- Đề cập 2 mô hình thực hiện MIPS
 - Phiên bản đơn giản
 - Phiên bản thực (cơ chế đường ống)
- Nhóm các lệnh đơn giản, nhưng đặc trưng:
 - Truy cập bộ nhớ: lw, sw
 - Số học/luận lý: add, sub, and, or, slt
 - Nhảy, rẽ nhánh (chuyển điều khiển): beq, j



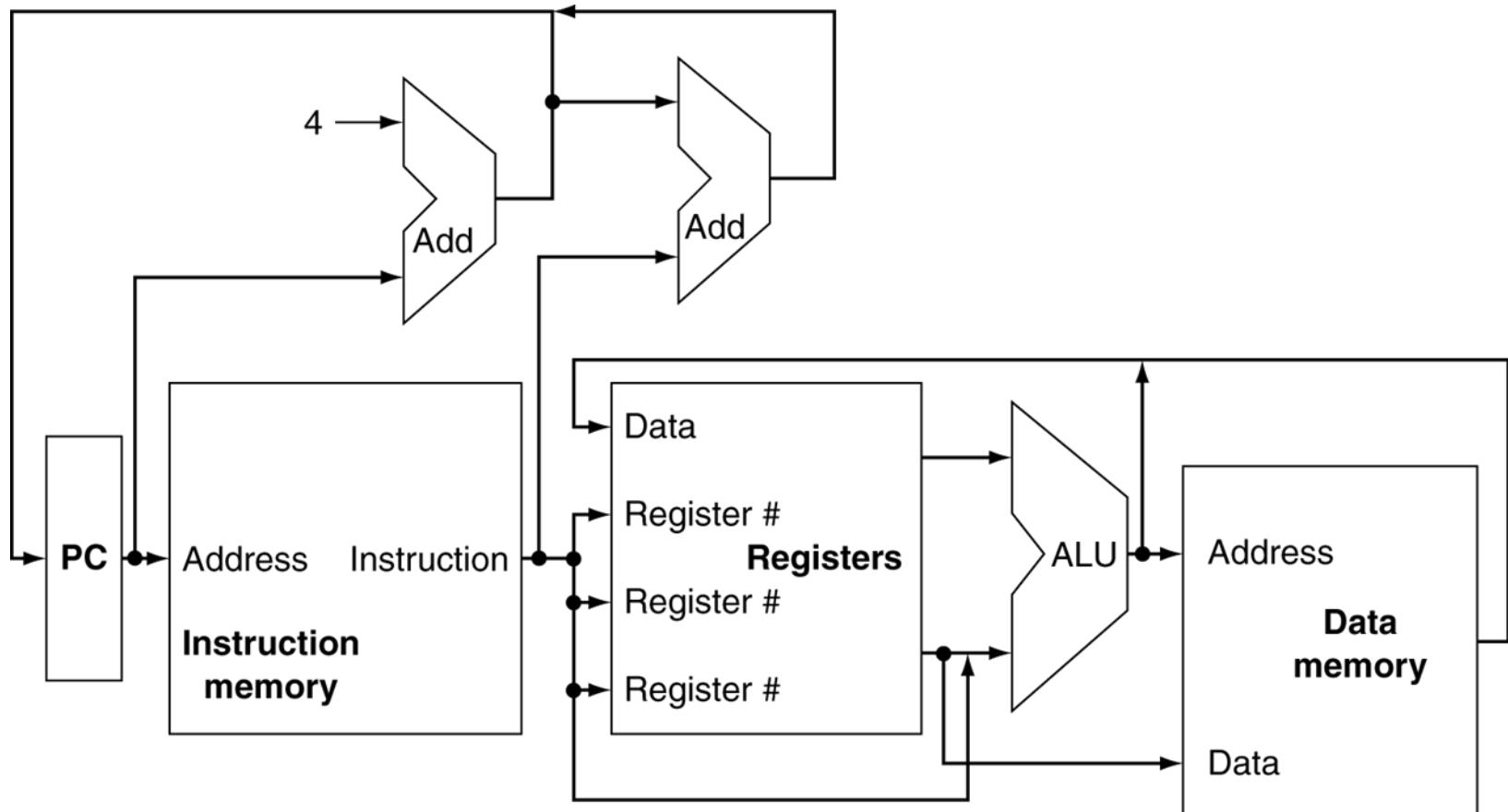


Các bước thực hiện lệnh

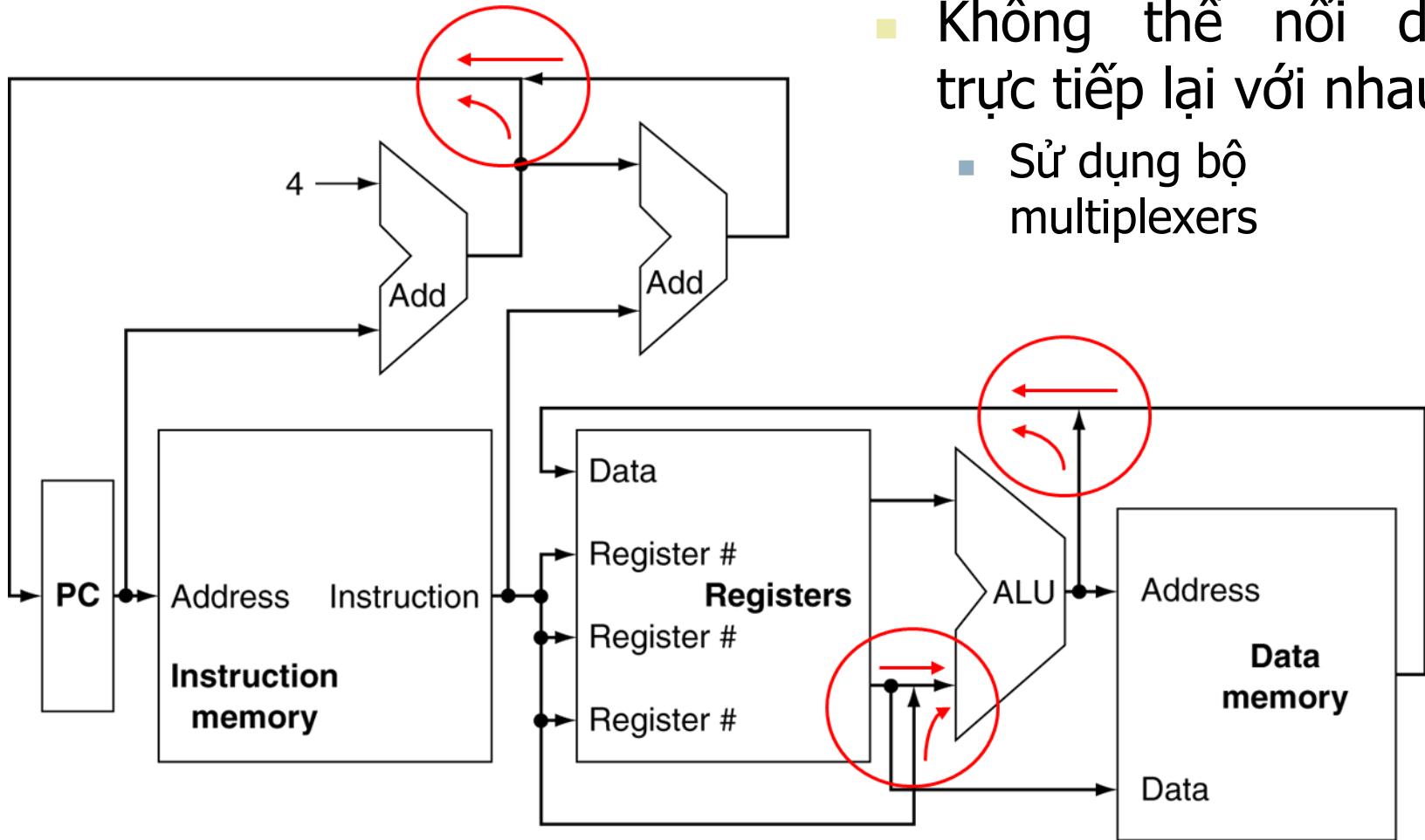
- PC → Bộ nhớ chứa lệnh, Nạp lệnh
- Đọc nội dung thanh ghi (Register numbers[rs, rt, rd] → register file)
- Tùy thuộc vào loại lệnh mà
 - Sử dụng ALU để tính
 - Phép số học → Kết quả
 - Xác định địa chỉ bộ nhớ (load/store)
 - Xác định địa chỉ rẽ nhánh
 - Truy cập dữ liệu bộ nhớ cho lệnh for load/store
 - PC ← Địa chỉ lệnh kế or PC + 4



Lược đồ thực hiện (CPU)

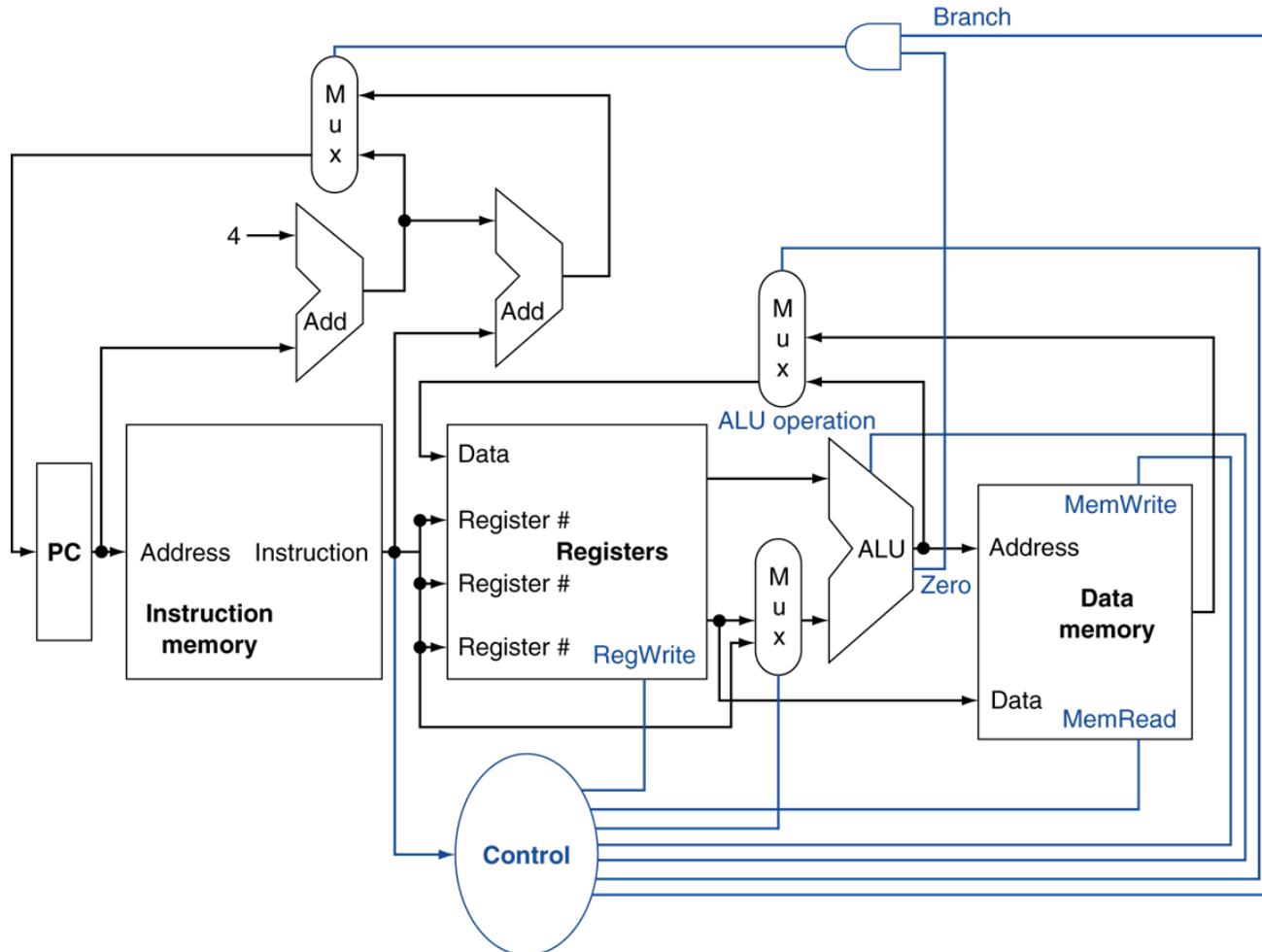


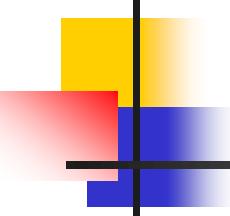
Bộ Multiplexer



- Không thể nối dây trực tiếp lại với nhau
 - Sử dụng bộ multiplexers

Bộ phận Điều khiển





Nguyên lý thiết kế luận lý

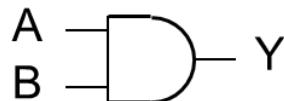
- Biểu diễn thông tin nhị phân
 - Áp mức thấp = 0, Áp mức cao = 1
 - Một đường dây cho mỗi bit
 - Dữ liệu gồm nhiều bit sẽ biểu diễn một tuyến nhiều đường dây
- Phần tử tổ hợp
 - Thực hiện trên dữ liệu
 - Kết quả đầu ra = hàm(đầu vào)
- Phần tử trạng tái (mạch tuần tự)
 - Lưu được dữ liệu



Ví dụ: các phần tử tổ hợp

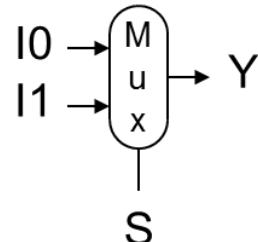
■ AND-gate

- $Y = A \& B$



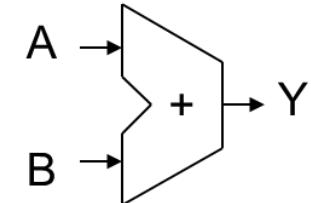
■ Multiplexer

- $Y = S ? I1 : I0$



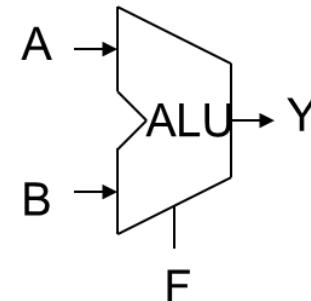
■ Adder

- $Y = A + B$



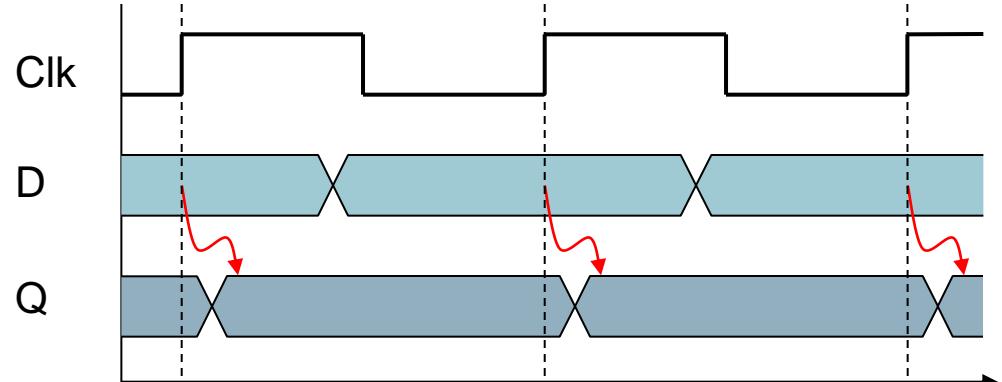
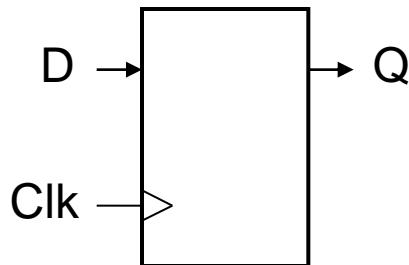
■ Arithmetic/Logic Unit

- $Y = F(A, B)$



Phần tử tuần tự

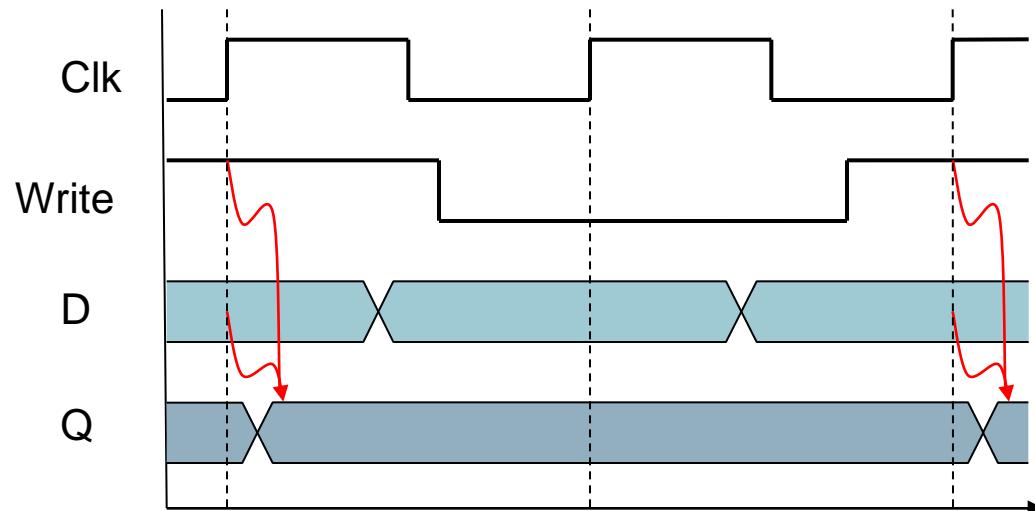
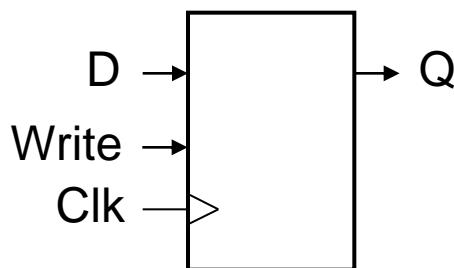
- Thanh ghi: lưu dữ liệu trong bộ mạch
 - Sử dụng tín hiệu xung đồng hồ để xác định khi nào cập nhật giá trị lưu trữ
 - Kích cạnh: đầu ra cập nhật khi xung đồng hồ thay đổi từ 0 lên 1



Phần tử tuần tự (tt.)

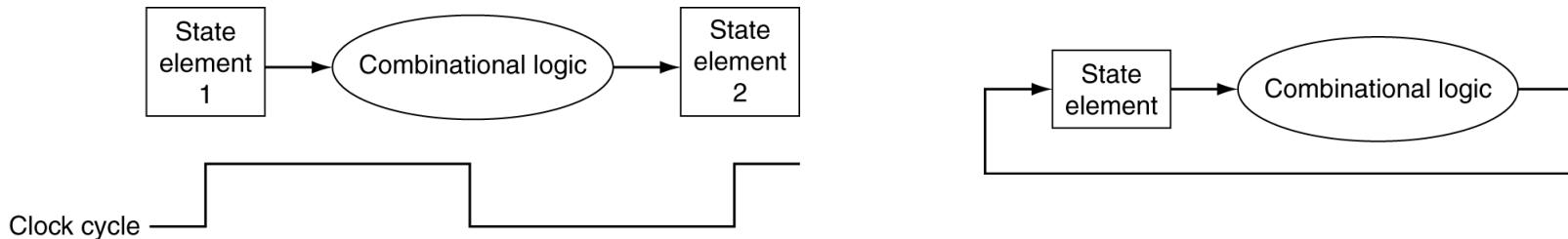
Thanh ghi với tín hiệu đ/khiển write

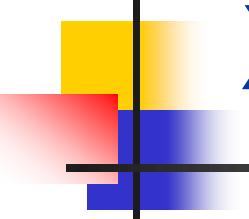
- Chỉ cập nhật theo cạnh xung khi mức điều khiển write ở mức 1
- Sử dụng trong trường hợp lưu cho chu kỳ sau



Phương thức làm việc dựa trên xung đồng hồ (Clocking Methodology)

- Mạch tổ hợp sẽ thay đổi giá trị dữ liệu trong chu kỳ đồng hồ
 - Giữa các cạnh của xung
 - Trạng thái của phần tử trước → Đầu vào của phần tử sau (tức thời)
 - Độ trễ dài nhất quyết định độ dài chu kỳ



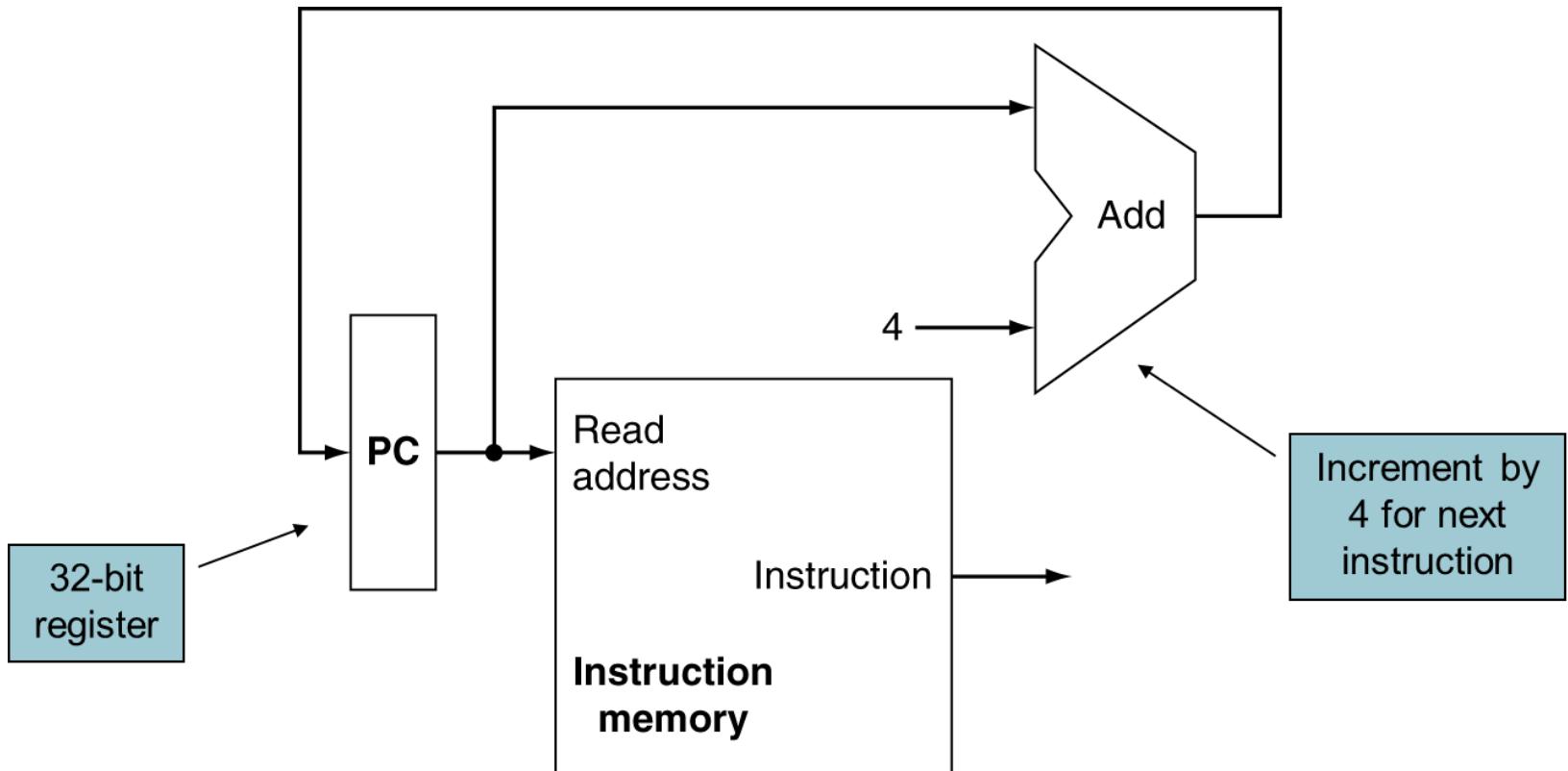


Xây dựng lộ trình dữ liệu

- Lộ trình xử lý Datapath
 - Các phần tử chức năng xử lý dữ liệu và địa chỉ trong CPU
 - Registers, ALUs, mux's, memories, ...
- Lộ trình sẽ được xây dựng từng bước từ thấp đến cao (đơn giản đến chi tiết)
 - Chi tiết và cụ thể hóa từng phần, bắt đầu từ Nạp lệnh (Instruction Fetch)

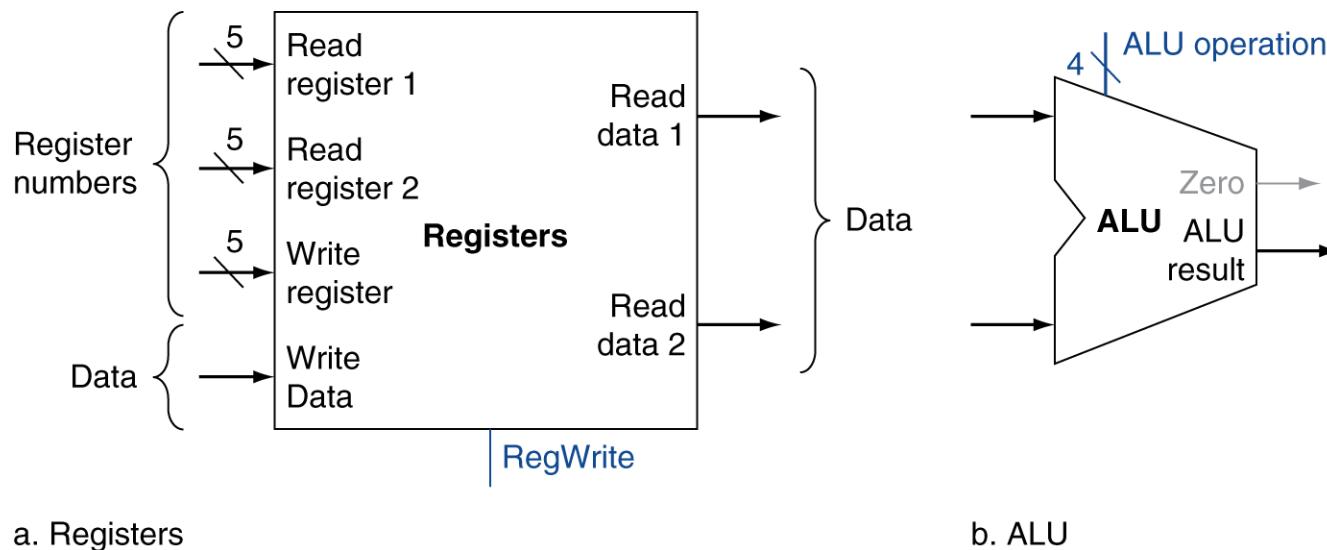


Nạp lệnh (Inst. Fetch)



Lệnh dạng R (R-Format)

- Đọc 2 toán hạng là thanh ghi
- Thực hiện phép Số học/Luận lý
- Ghi kết quả vào thanh ghi



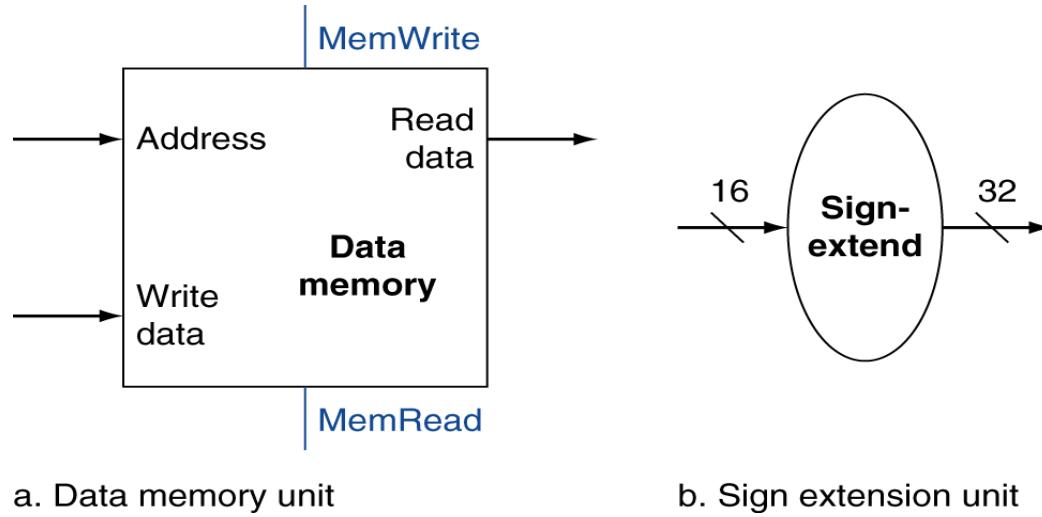
a. Registers

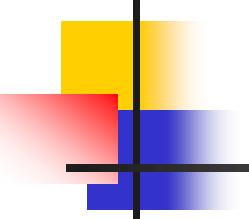
b. ALU



Lệnh Load/Store

- Đọc toán hạng thanh ghi
- Tính địa chỉ của bộ nhớ (16-bit độ dời)
 - Sử dụng ALU, nhưng độ dời phát triển ra 32-bit có dấu
- Nạp (Load): Đọc bộ nhớ & cập nhật thanh ghi
- Cất (Store): Ghi giá trị (register) → Bộ nhớ



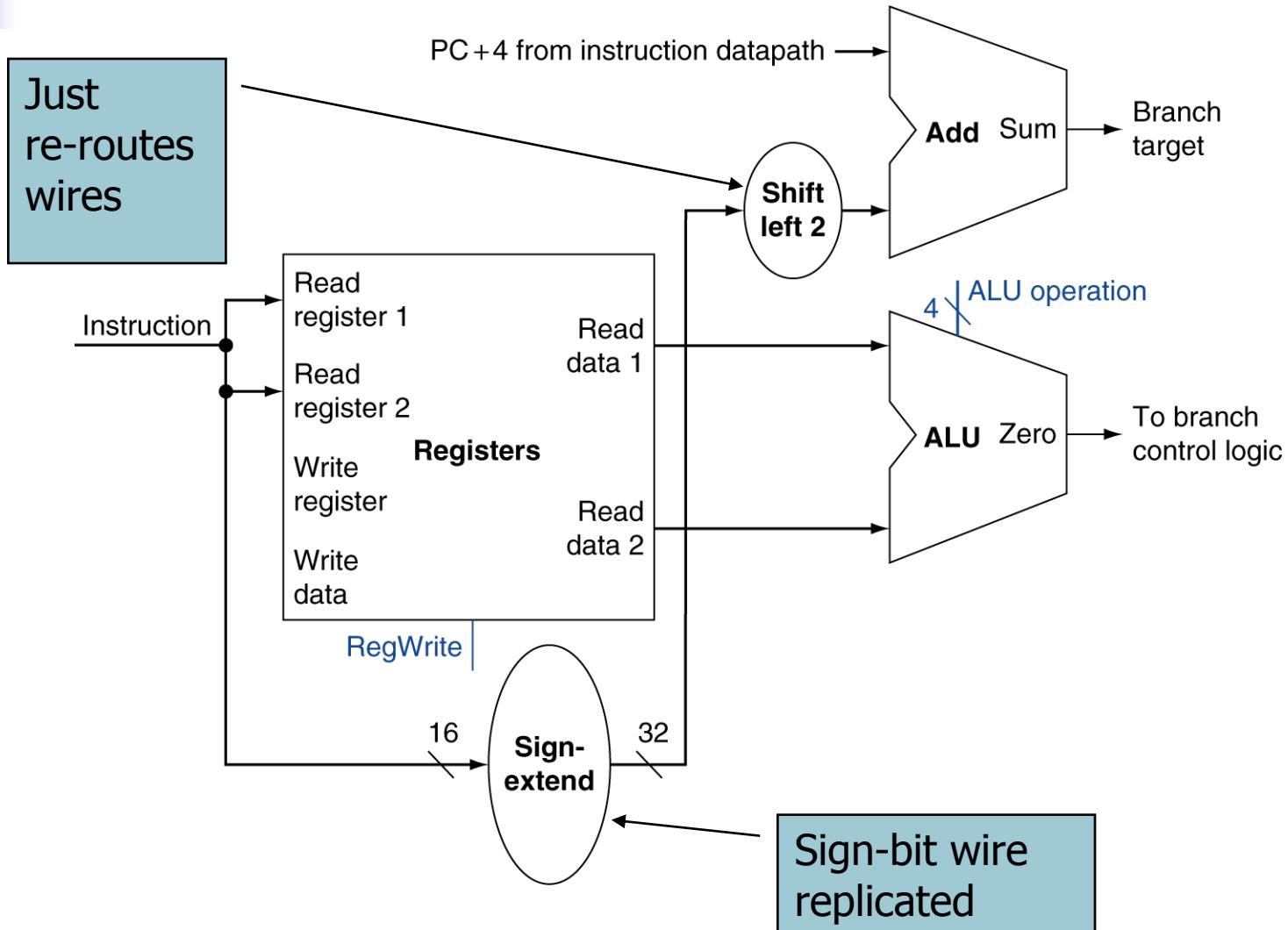


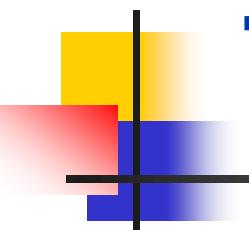
Lệnh rẽ nhánh

- Đọc toán hạng (thanh ghi)
- So sánh toán hạng
 - Sử dụng ALU, subtract and check Zero
- Tính toán địa chỉ đích
 - Mở rộng 16 sang 32 bit có dấu (địa chỉ)
 - Dịch trái 2 vị trí (1 word = 4 bytes)
 - Cộng $PC=PC + 4$
 - Đã được tính tự động khi nạp lệnh



Lệnh rẽ nhánh



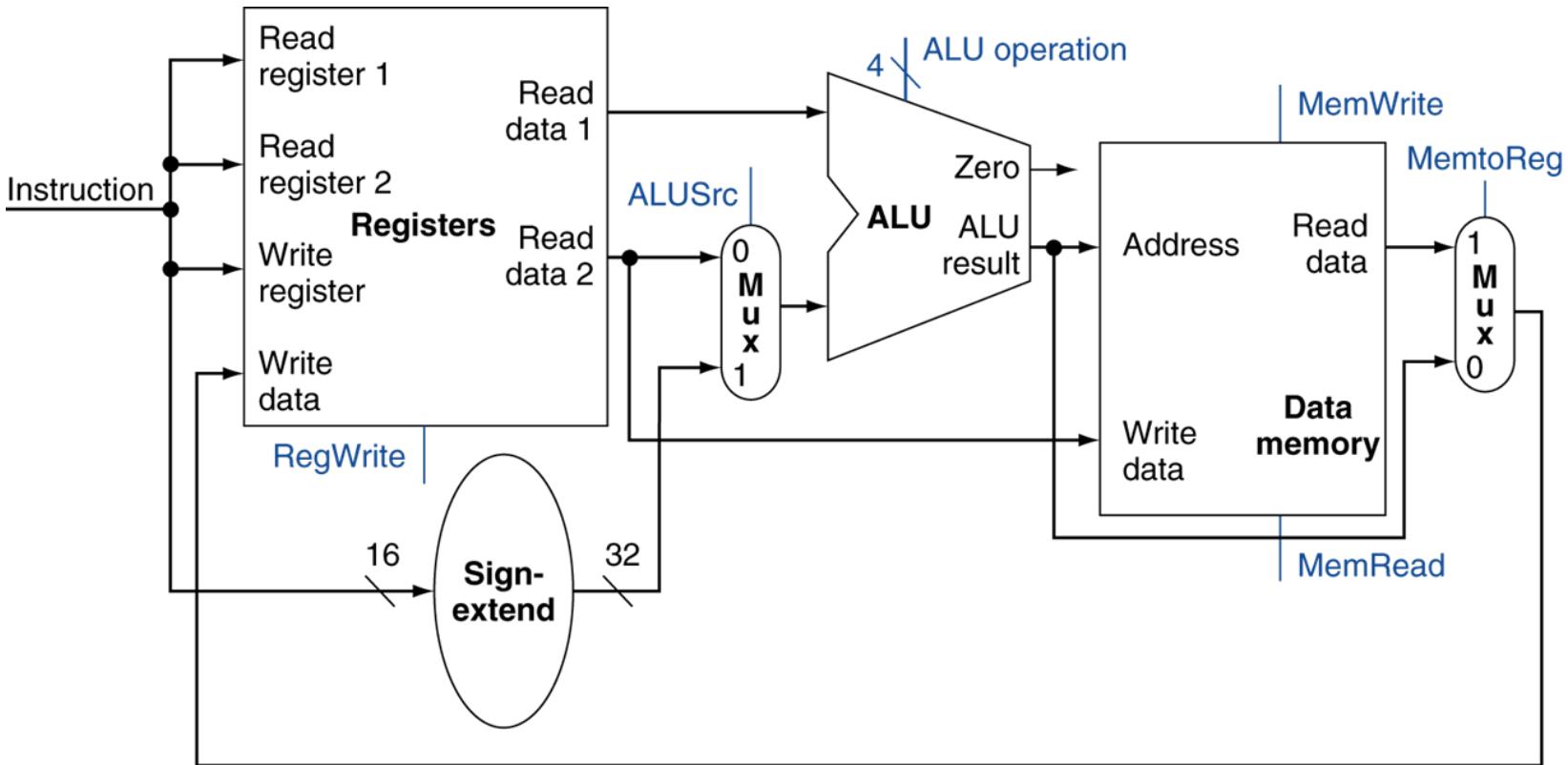


Tổng hợp các phần tử

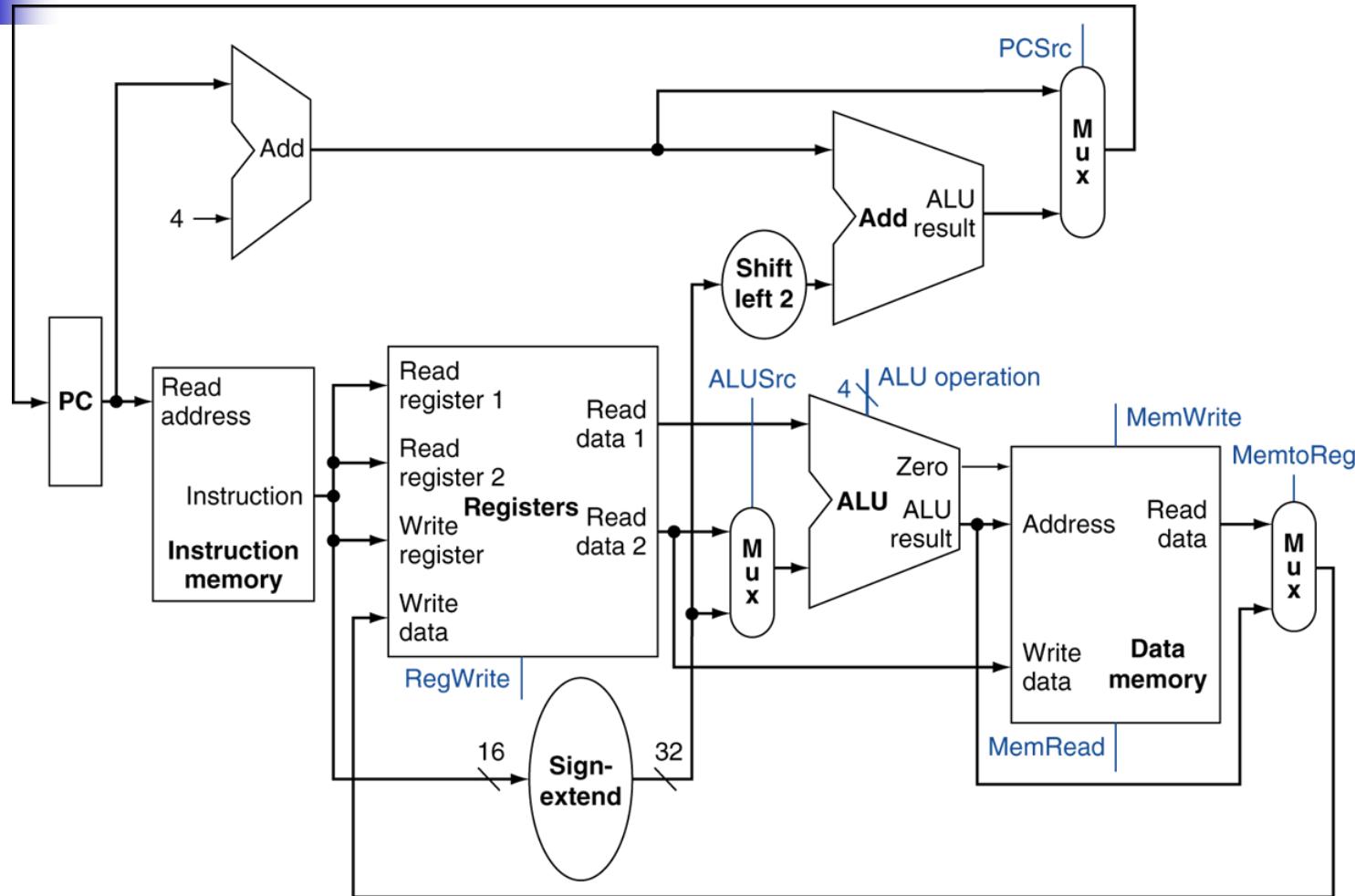
- Lệnh được thực hiện trong 1 chu kỳ xung Clock
 - Mỗi bộ phận chỉ có thể thực hiện 1 chức năng tại mỗi thời điểm
 - Vì vậy, phải tách biệt giữa bộ nhớ lệnh và bộ nhớ dữ liệu
- Multiplexer được sử dụng tại những nơi mà nguồn dữ liệu khác nhau ứng với lệnh khác nhau

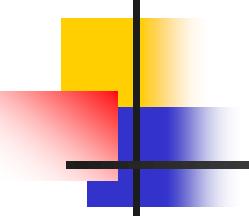


Lộ trình tổng hợp (R-Type/Load/Store)



Lộ trình toàn phần





Bộ điều khiển tín hiệu ALU

- ALU dùng trong những lệnh
 - Load/Store: F(unction) = add
 - Branch: F(unction) = subtract
 - R-type: F phụ thuộc vào hàm (funct)

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



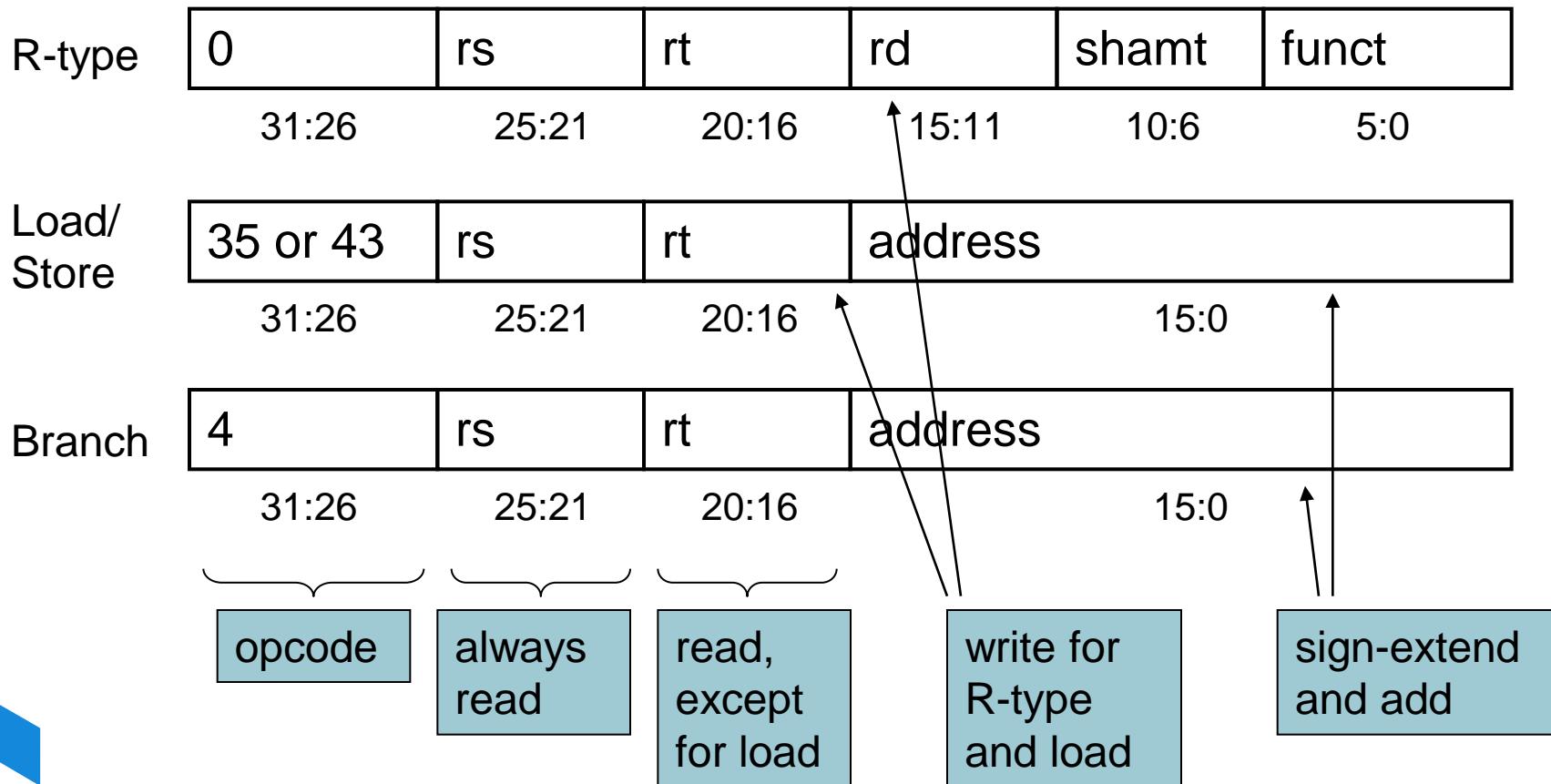
Bộ điều khiển tín hiệu ALU (tt.)

- Giả sử 2-bit ALUOp từ opcode của lệnh
 - Tín hiệu đ/khiển ALU từ mạch tổ hợp như sau:

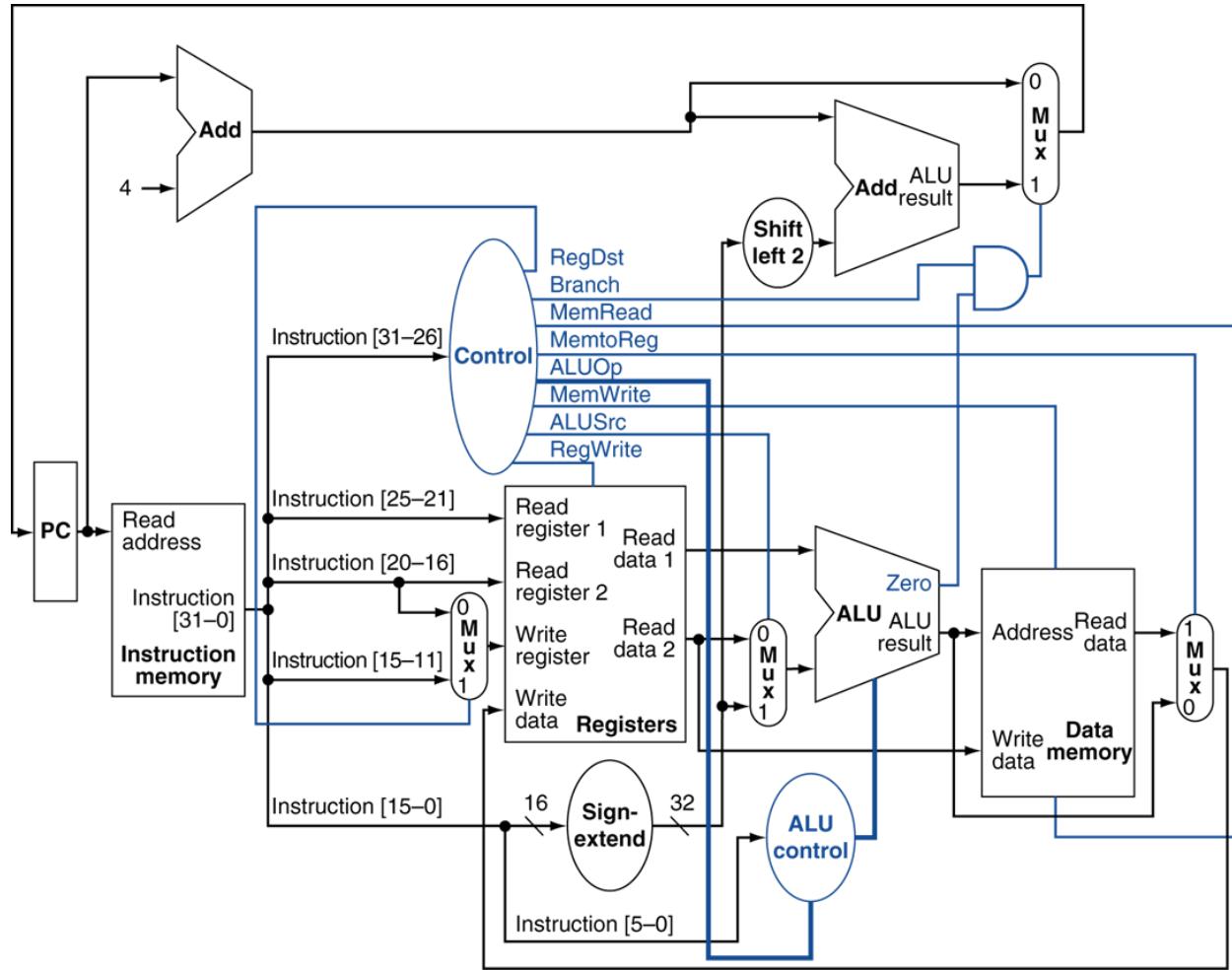
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|--------|-------|------------------|--------|------------------|-------------|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 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 |

Bộ phận điều khiển chính

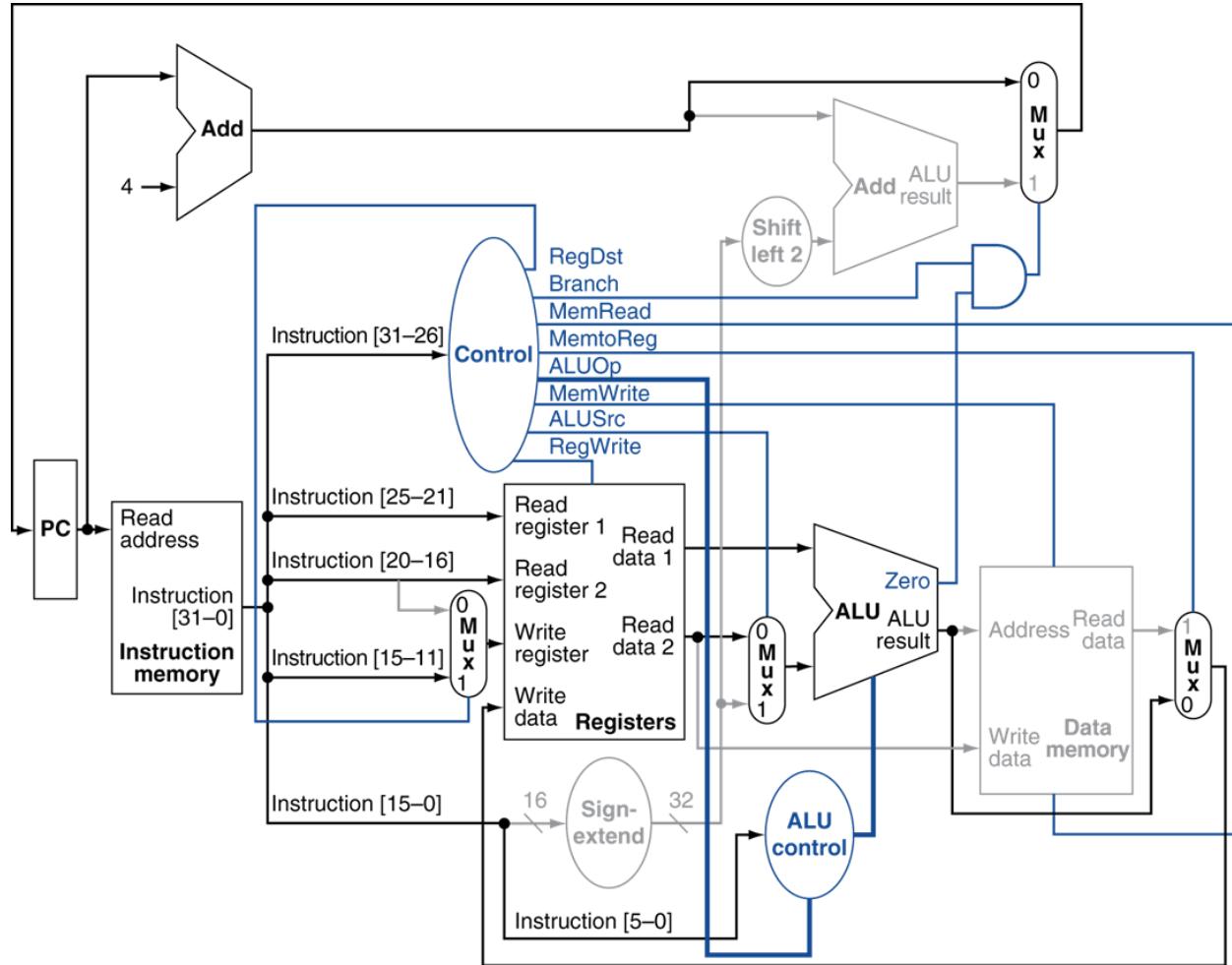
■ Các tín hiệu đ/khiển giải mã từ lệnh



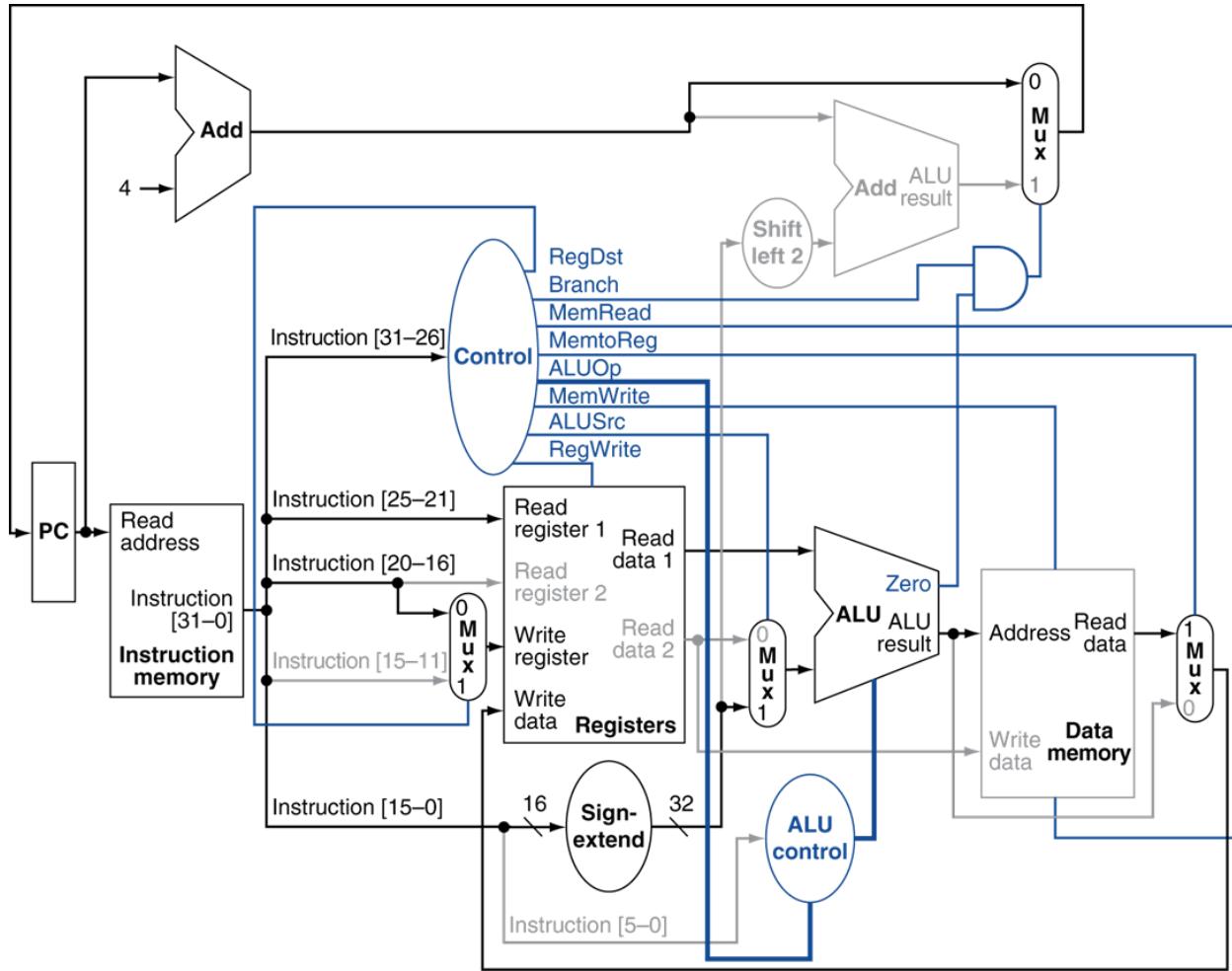
Lộ trình với tín hiệu đ/khiển



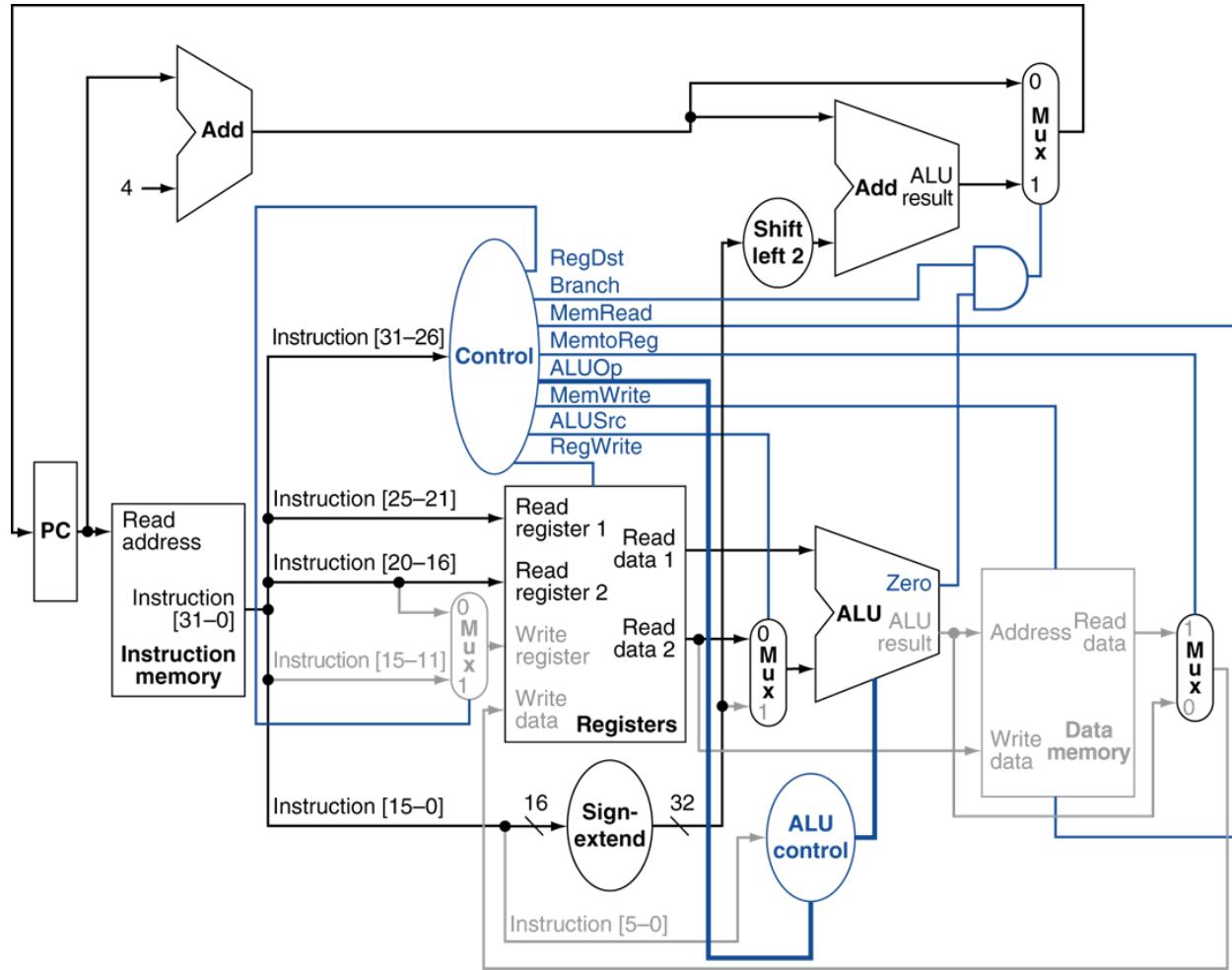
Lệnh dạng R-Type



Lệnh nạp (Load)



Rẽ nhánh với đ/kiện (=)



Thực hiện lệnh Jumps

Jump

2

address

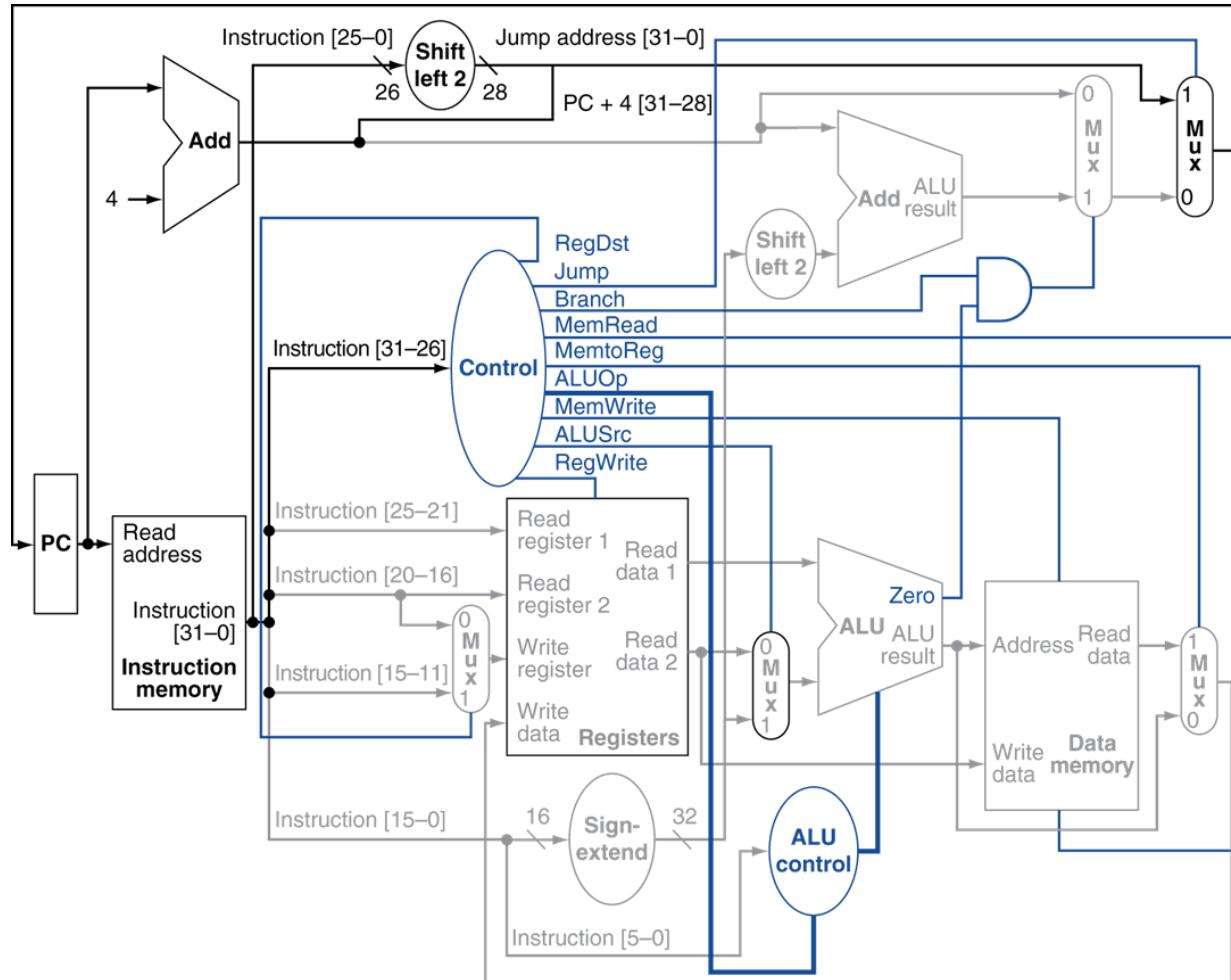
31:26

25:0

- Jump sử dụng địa chỉ trong 1 từ (word)
- Cập nhật PC bằng cách tổng hợp từ
 - 4 bits cao của thanh ghi cũ PC
 - 26-bit jump address
 - 00
- Yêu cầu thêm các tín hiệu đ/khiển giải mã từ opcode



Lộ trình với lệnh Jumps



Vấn đề hiệu xuất

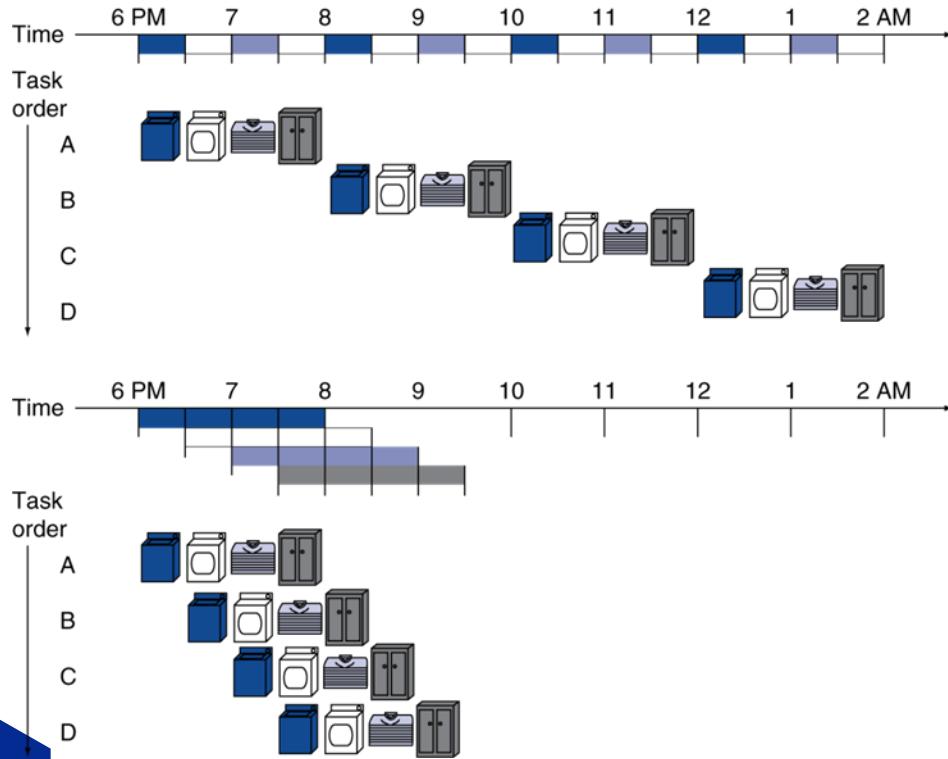
- Trễ tối đa sẽ xác định độ dài chu kỳ đồng hồ
 - Lộ trình dài nhất: lệnh load
 - Instruction memory → register file → ALU → data memory → register file
- Không khả thi nếu thay đổi chu kỳ xung theo lệnh khác nhau
- Phá vỡ nguyên tắc thiết kế
 - Cái gì phổ biến nhất thực hiện nhanh nhất
- Chúng ta sẽ cải thiện hiệu xuất theo cơ chế ống



Giới thiệu: Cơ chế ống

■ Ví dụ thực tế: Quy trình giặt đồ (các bước thực hiện phủ lấp)

■ Các bước thực hiện đồng thời: cải thiện HS

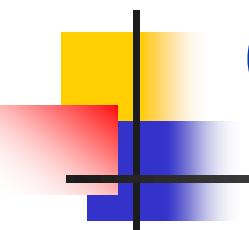


■ 4 mẻ:

■ Tăng tốc
 $= 8/3.5 = 2.3$

■ Giặt không ngừng:

■ Tăng tốc
 $= 2n/0.5n + 1.5 \approx 4$
= số bước/mẻ giặt



Cơ chế ống trong MIPS

- Mỗi lệnh: 5 công đoạn (mỗi bước/công đoạn), đó là
 1. IF: Nạp lệnh (Inst. Fetch) từ bộ nhớ
 2. ID: Giải mã (Inst. Decode) & đọc th/ghi
 3. EX: Thực thi (Ex.) hay tính địa chỉ
 4. MEM: Truy cập bộ nhớ
 5. WB: Cất kết trả lại th/ghi

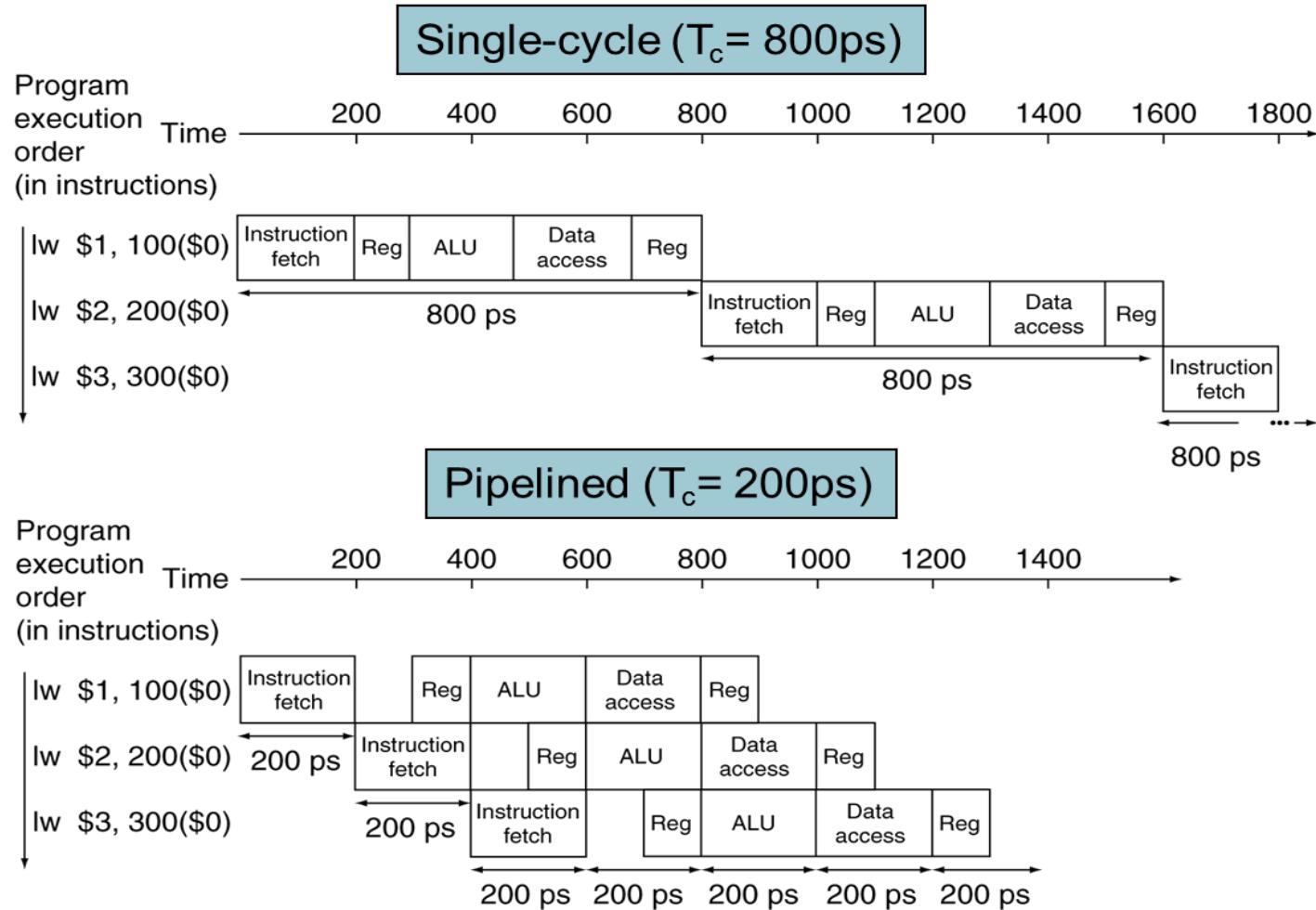


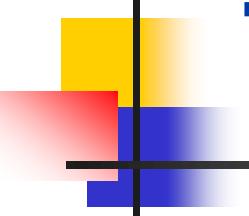
Hiệu suất ống

- Giả sử thời gian thực hiện cho các công đoạn
 - 100ps để đọc hoặc ghi thanh ghi
 - 200ps cho các công đoạn khác
- So sánh lộ trình xử lý ống và chu kỳ đơn như bảng dưới đây:

| Lệnh | Nạp lệnh | Đọc th/ghi | ALU op | Truy cập bộ nhớ | Ghi th/ghi | Tổng thời gian |
|----------|----------|------------|--------|-----------------|------------|----------------|
| lw | 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 suất ống (tt.)

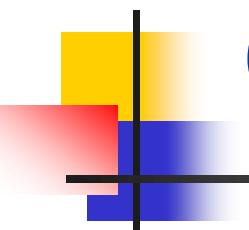




Tăng tốc của ống

- Nếu công việc các công đoạn như nhau
 - Ví dụ: có cùng thời gian thực hiện
 - Time between instructions_{pipelined} =
$$\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
- Nếu các công đoạn không đều nhau → Độ tăng tốc sẽ ít hơn
- Độ tăng tốc thể hiện hiệu suất (throughput) tăng
 - Vì thời gian thực thi cho mỗi lệnh không thay đổi (giảm)

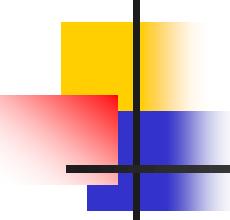




Cơ chế ống với MIPS ISA

- MIPS ISA được thiết kế với cơ chế ống
 - Tất cả các lệnh 32-bits
 - Dễ dàng nạp & giải mã trong 1 chu kỳ
 - Khác với x86: 1- đến 17-bytes/lệnh
 - Lệnh ít dạng và có quy tắc
 - Giải mã & đọc th/ghi trong 1 chu kỳ
 - Địa chỉ trong lệnh Load/store
 - Có thể tính trong công đoạn 3, truy cập bộ nhớ trong công đoạn 4
 - Các toán hạng bộ nhớ truy cập trong 1 cùng 1 chu kỳ



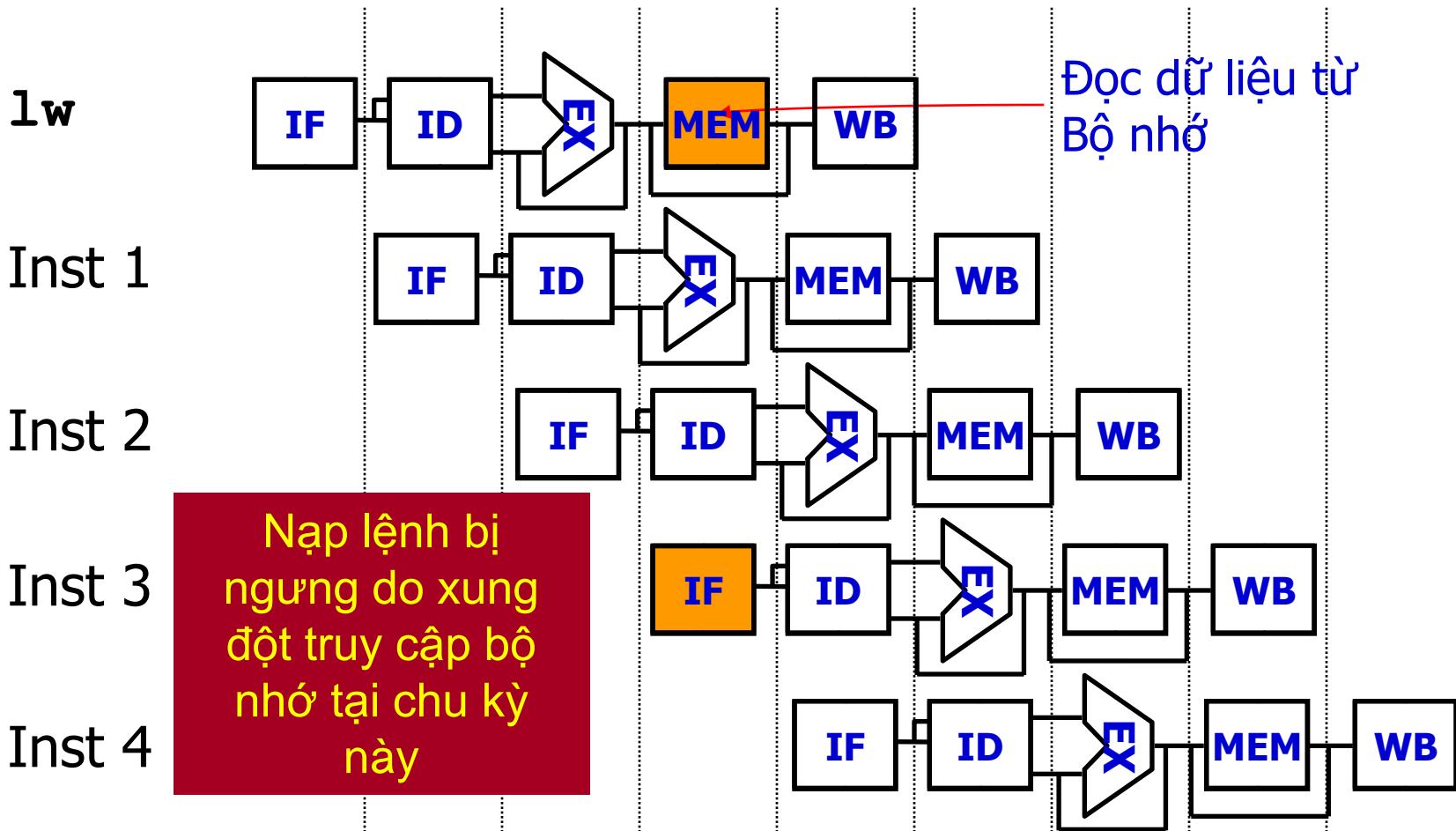


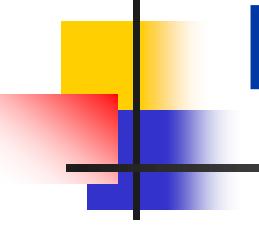
Rủi ro (Hazards) trong cơ chế ống

- Có trường hợp: Lệnh kế tiếp không thể thực hiện trong chu kỳ kế → Rủi ro.
Tồn tại 3 loại rủi ro:
- Rủi ro về cấu trúc (Structure Hazard)
 - Một tài nguyên được yêu cầu, nhưng bận
- Rủi ro về dữ liệu
 - Đợi lệnh trước hoàn tất tác vụ đọc/ghi dữ
- Rủi ro về điều khiển
 - Quyết định bước tiếp theo phụ thuộc vào lệnh trước đó



Rủi ro về cấu trúc





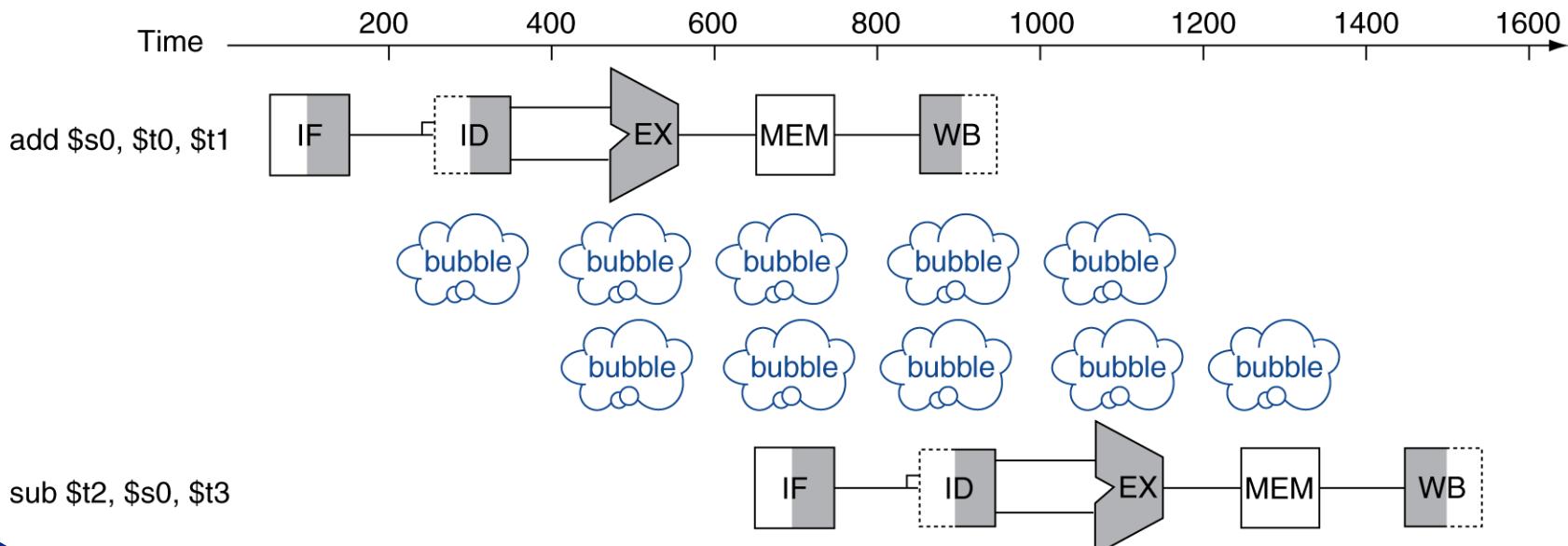
Rủi ro về cấu trúc

- Tranh chấp sử dụng tài nguyên
- Trong MIPS, cơ chế ống với 1 loại bộ nhớ
 - Load/store yêu cầu đọc/ghi dữ liệu
 - Nạp lệnh sẽ bị “kẹt” trong chu kỳ đó
 - Trường hợp đó gọi là sự “khụng lại” hay “bong bóng” (bubble)
- Vì vậy, trong cơ chế ống lộ trình xử lý lệnh cần tách 2 bộ nhớ riêng biệt (lệnh, data)
 - ít ra thì 2 vùng cache riêng



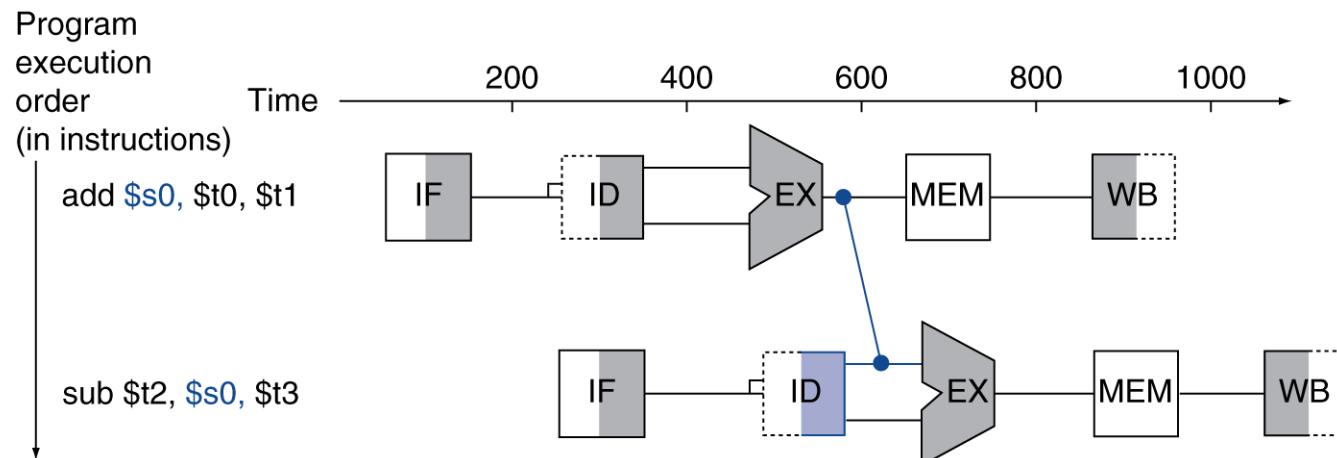
Rủi ro về dữ liệu

- Kết quả truy xuất dữ liệu thuộc lệnh trước ảnh đến lệnh sau
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



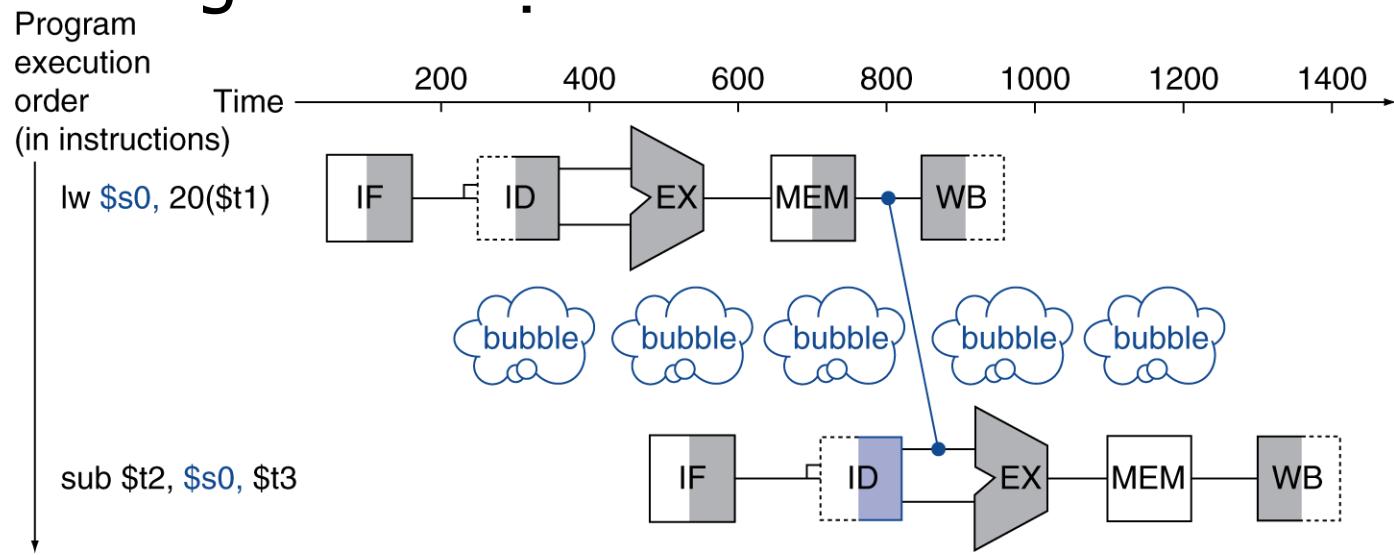
Xúc tiến sớm (Forwarding)

- Sử dụng ngay kết quả vừa tính toán xong của lệnh trước
 - Không cần đợi kết quả cất lại thanh ghi
 - Cần có thêm kết nối trong lộ trình



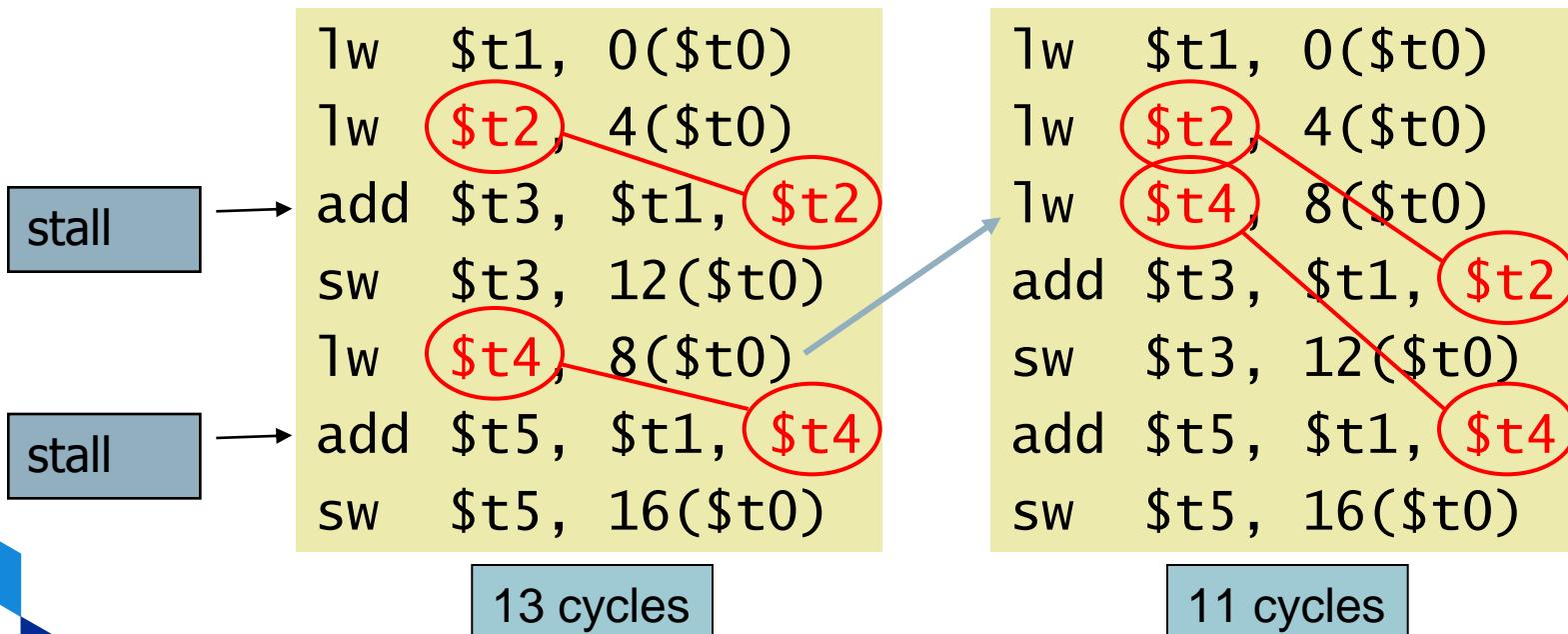
Rủi ro dữ liệu khi dùng Load

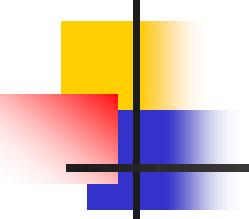
- Forwarding không phải lúc nào cũng giải quyết sự “khựng lại” trong ống
 - Nếu cần kết quả là lệnh truy xuất bộ nhớ cho lệnh kế
 - Không thể lùi lại!**



Khắc phục

- Sắp xếp lại code để tránh sử dụng kết quả của lệnh load trong lệnh kế
- C code: $A = B + E; C = B + F;$





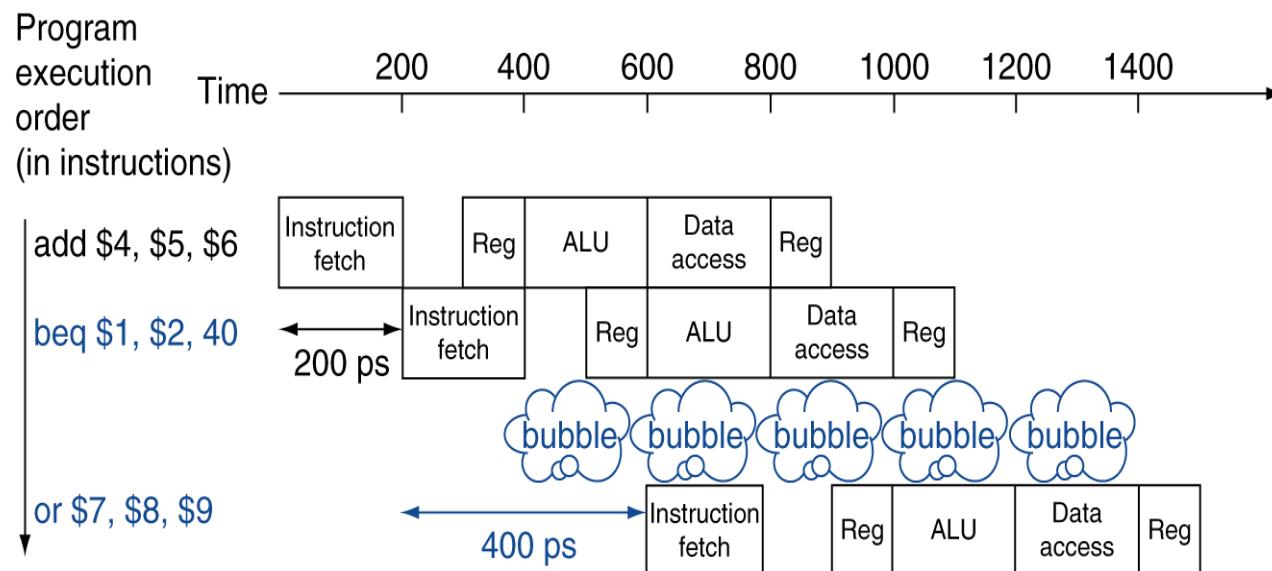
Rủi ro về điều khiển

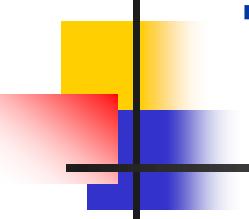
- Rẽ nhánh thay đổi lộ trình thực hiện
 - Nạp lệnh kế phụ thuộc vào kết quả của điều kiện rẽ nhánh
 - Với cơ chế ống: khó xác định đúng
 - Thực hiện trong công đoạn giải mã lệnh
- Trong cơ chế ống của MIPS
 - Giá trị các thanh ghi được so sánh & tính ra địa chỉ đích
 - Sử dụng thêm phần cứng để thực hiện trong bước giải mã lệnh



Sự “khụng lại” trong rẽ nhánh

- Đợi cho đến khi xác định được khi nào sẽ nạp lệnh kế.



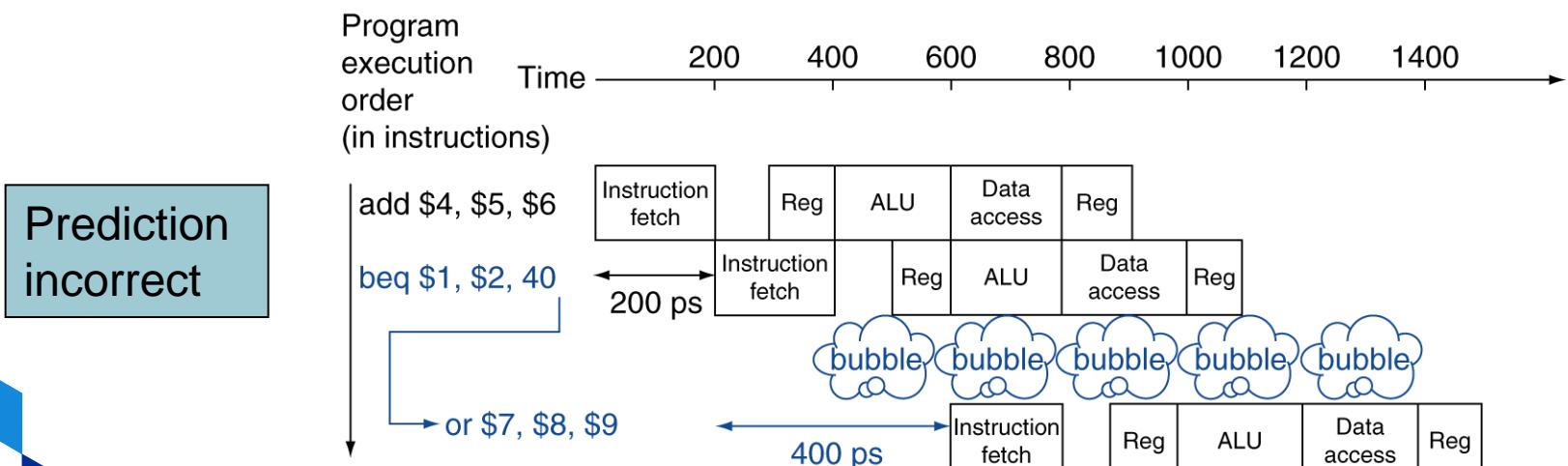
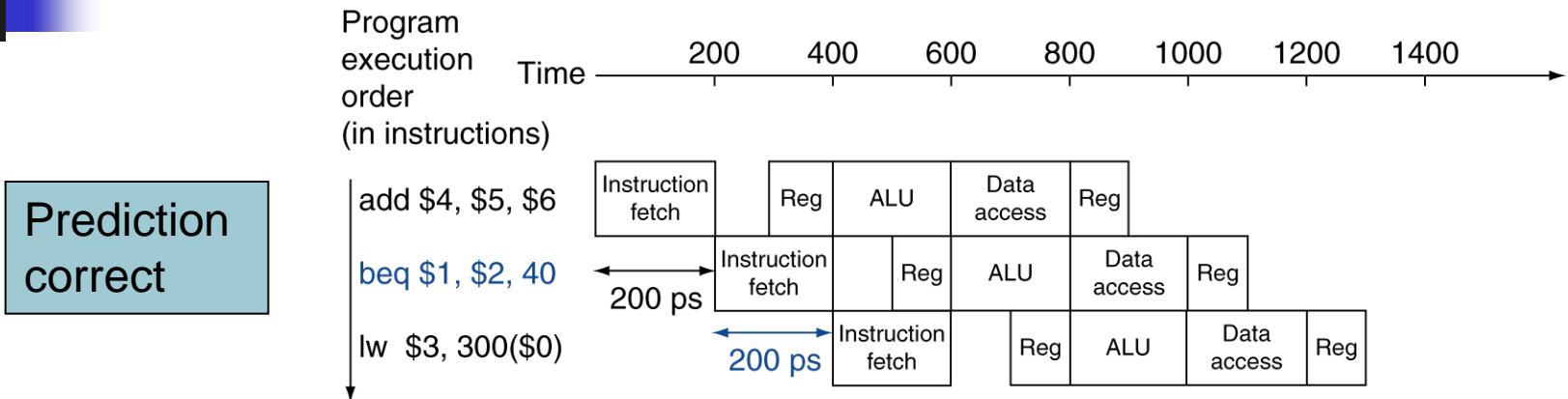


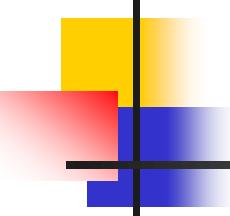
Tiên đoán khi có rẽ nhánh

- Đối với ống dài: có thể xác định sớm
 - Sự “khụng lại” → giảm hiệu xuất
- Tiên đoán trước
 - 50:50 → “Khụng lại”
- Trong cơ chế ống MIPS
 - Có thể tiên đoán
 - Tự động lấy lệnh kế



Ví dụ: Tiên đoán 50:50

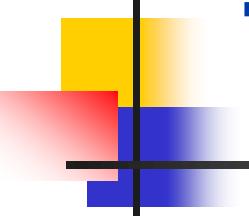




Giải pháp tiên đoán thực tế

- Tiên đoán tĩnh (Static branch prediction)
 - Dựa trên hành vi rẽ nhánh thường xảy ra
 - Ví dụ: Vòng lặp với phát biểu if
 - Tiên đoán sẽ là rẽ nhánh quay lại (backward branches)
 - Tiên đoán rẽ nhánh xuôi (forward) không xuất hiện
- Tiên đoán động (Dynamic branch prediction)
 - Bộ phận phần cứng sẽ đo đạc hành vi xảy ra
 - Ví dụ: lưu lại lịch sử mỗi rẽ nhánh
 - Giả thiết tương lai từ việc đo đạc
 - Nếu không đúng, cập nhật lại lịch sử, chấp nhận sự “khụng lại”



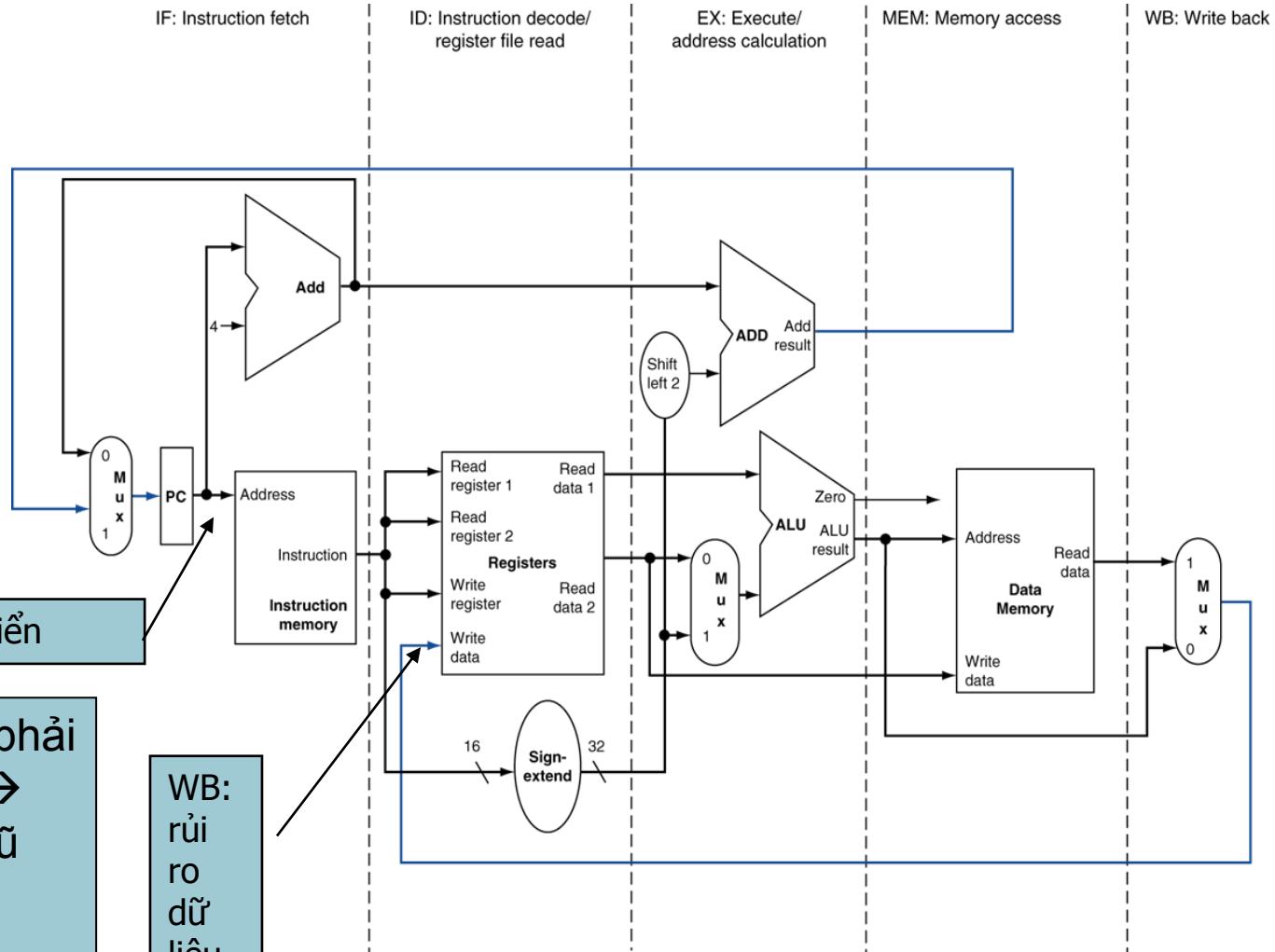


Tổng kết về Cơ chế ống

- Cơ chế ống cải thiện hiệu suất thực hiện lệnh (throughput)
 - Thực hiện nhiều lệnh cùng lúc
 - Mỗi lệnh có thời gian thực thi không đổi
- Vấn đề nảy sinh: rủi ro
 - Cấu trúc, dữ liệu , điều khiển
- Thiết kế tập lệnh (theo nguyên tắc thiết kế) có thể làm phức tạp quá trình thực thi cơ chế ống

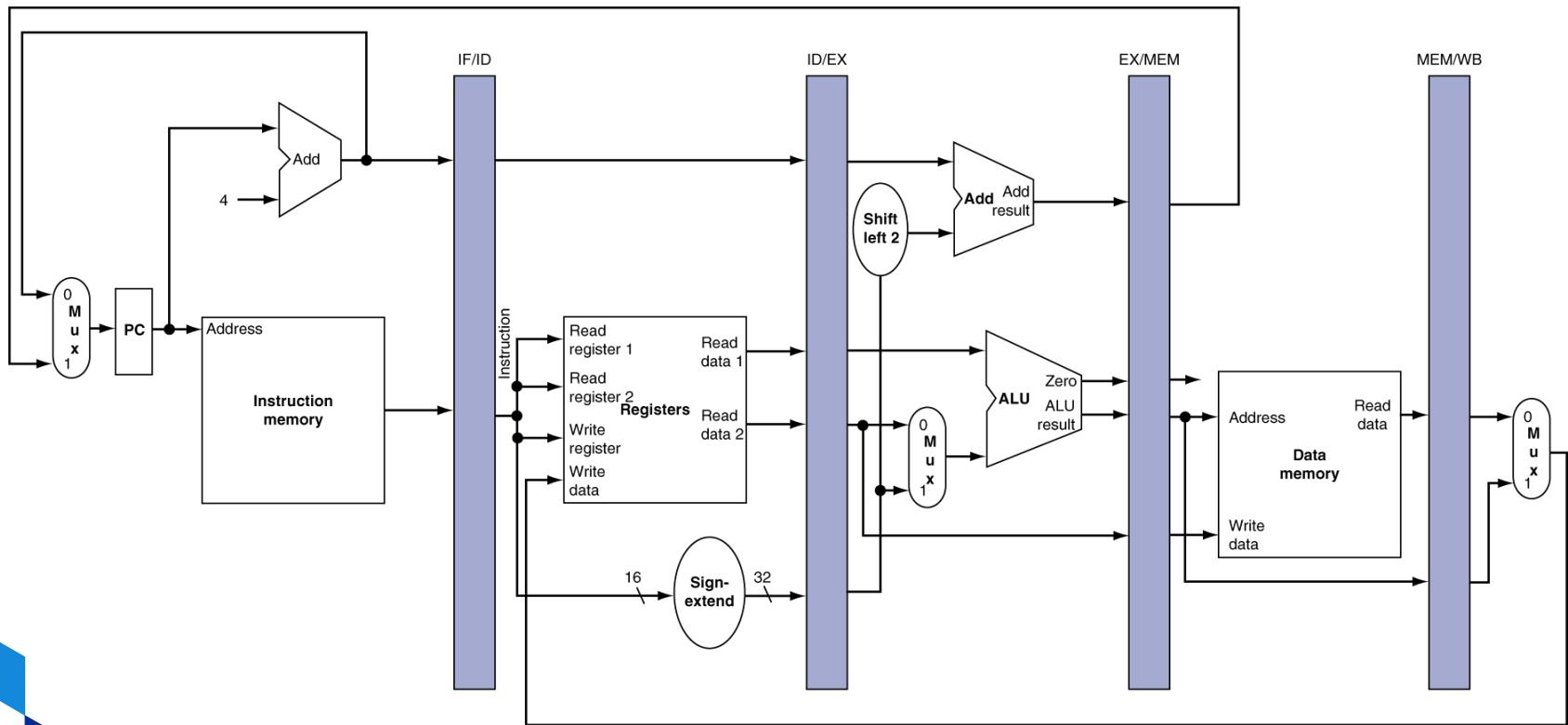


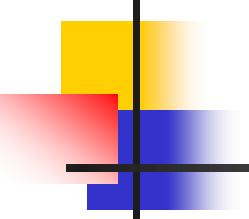
Lộ trình MIP theo bước (óng)



Thanh ghi đệm giữa các bước

- Cần có các thanh ghi đệm giữa các công đoạn (bước): lưu t/tin bước trước đó



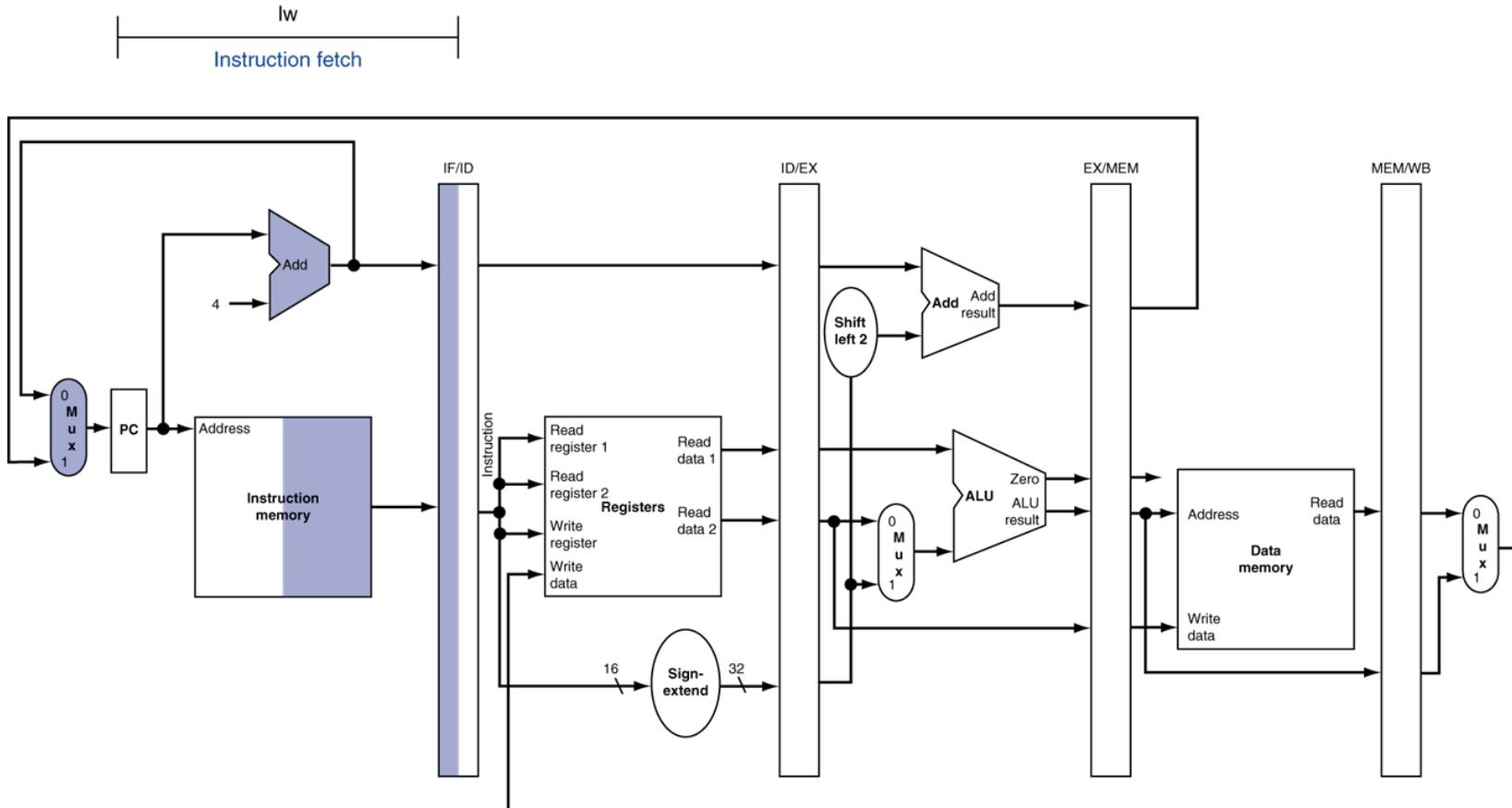


Hoạt động trong ống

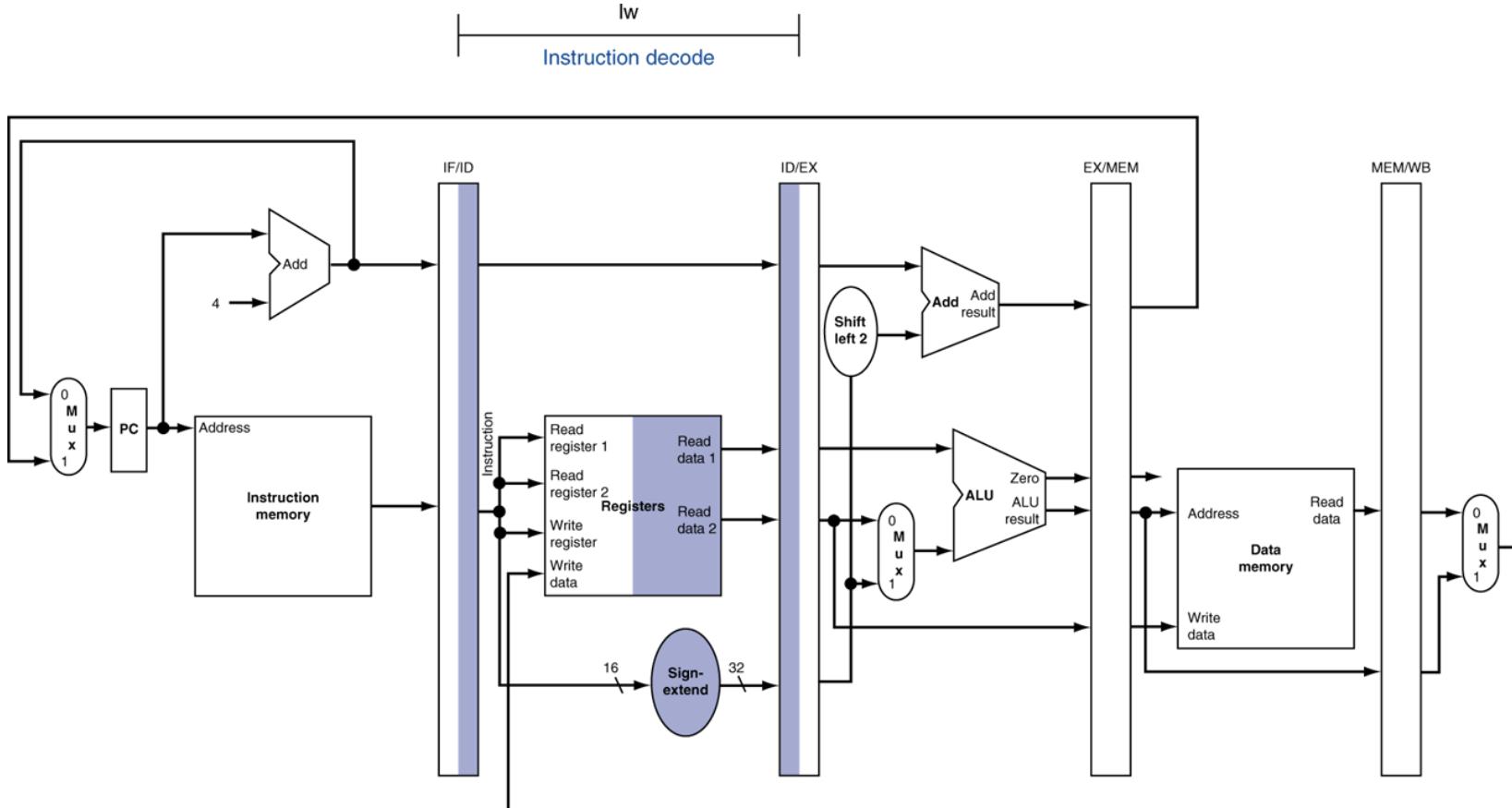
- Các lệnh sẽ được thực hiện theo luồng trong lộ trình dữ liệu ống (theo từng chu kỳ)
 - Biểu diễn theo chu kỳ đơn
 - Thể hiện lệnh/chu kỳ đồng hồ
 - Tô đậm các tài nguyên sử dụng
 - Ngược với biểu diễn theo đa chu kỳ
 - Biểu đồ tác vụ theo thời gian
- Chúng ta sẽ quan sát quá trình thực hiện từng bước với lệnh load & store



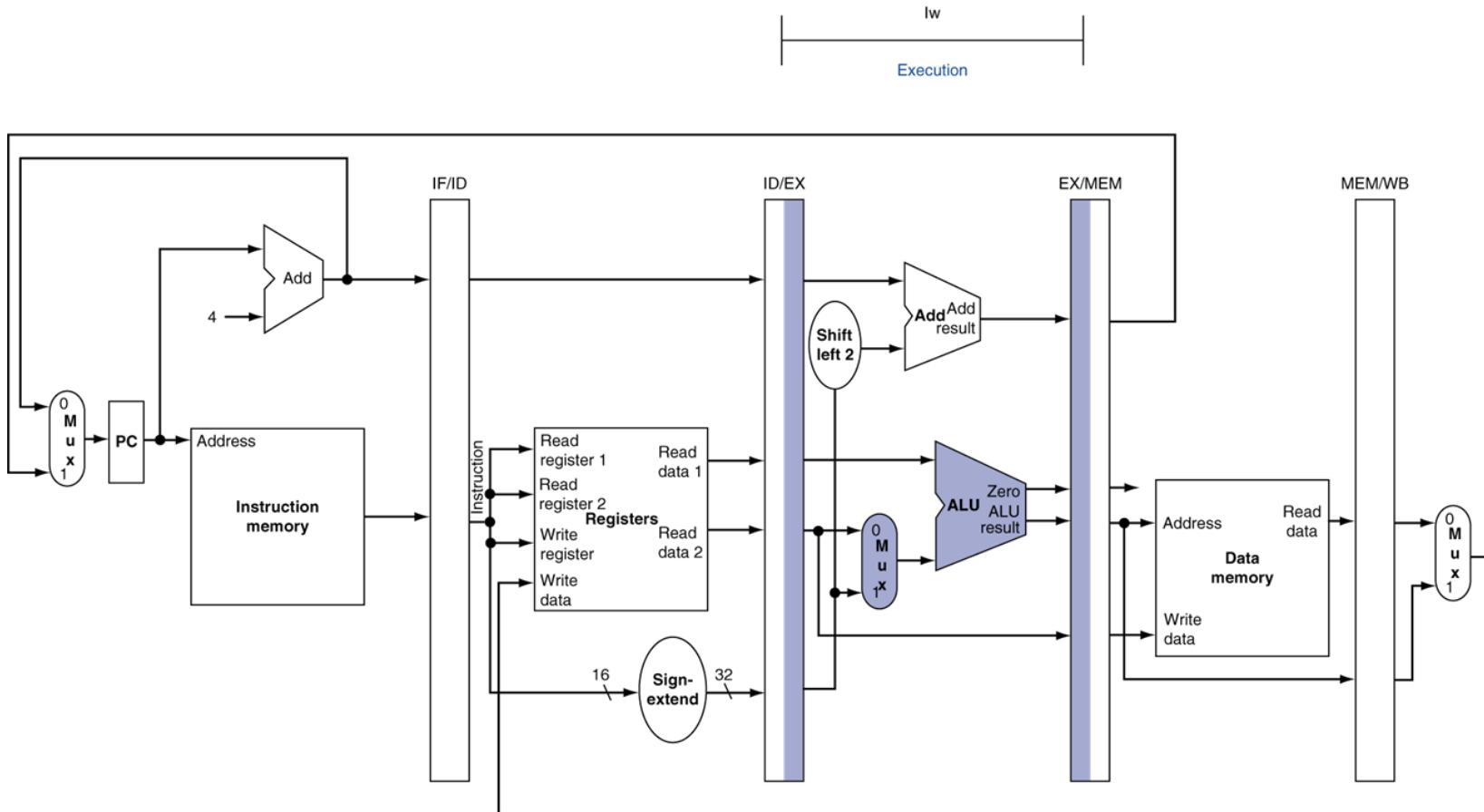
Bước Nạp lệnh (Load, Store, ...)



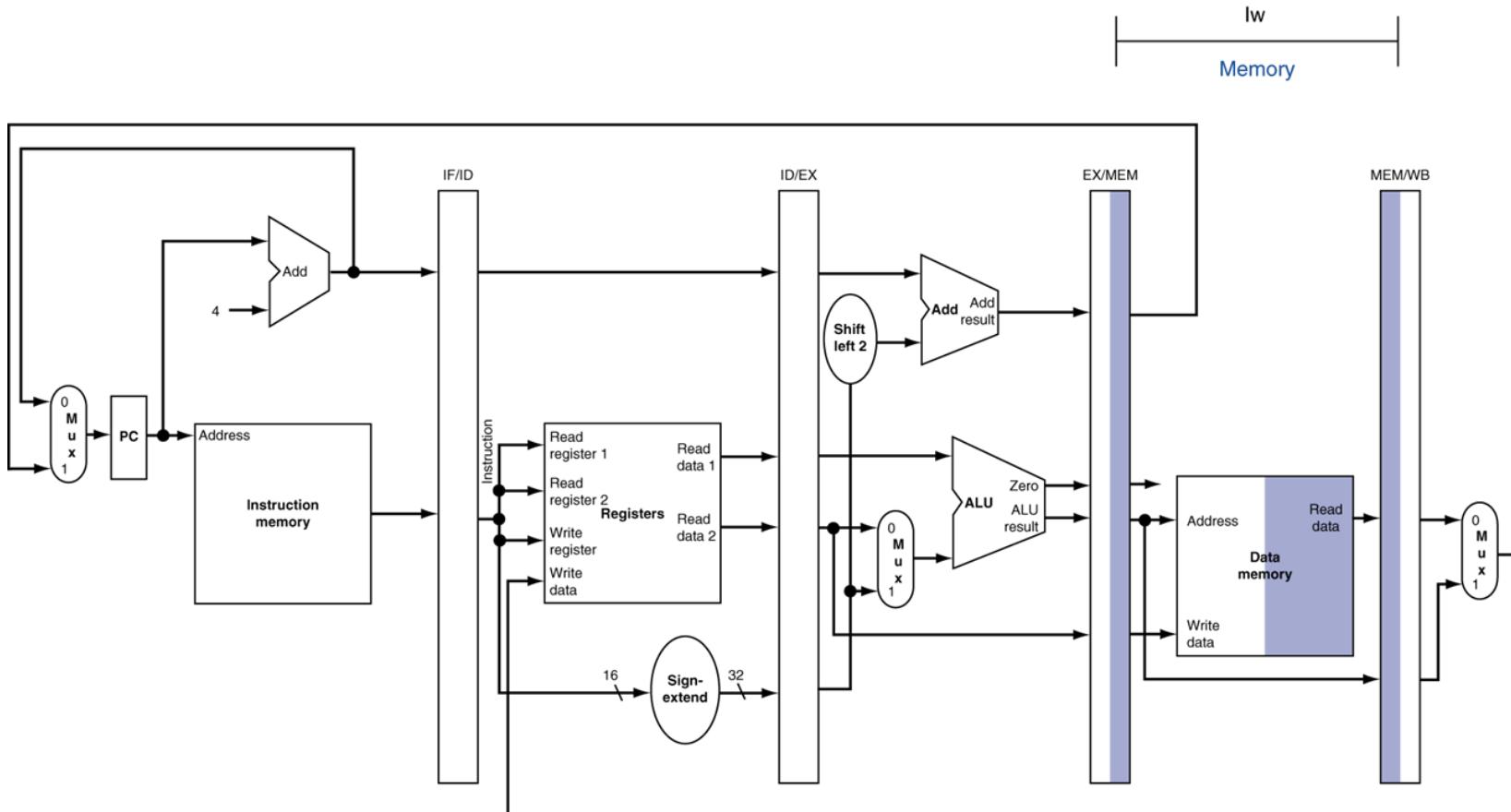
Bước Giải mã lệnh (Load, Store, ..)



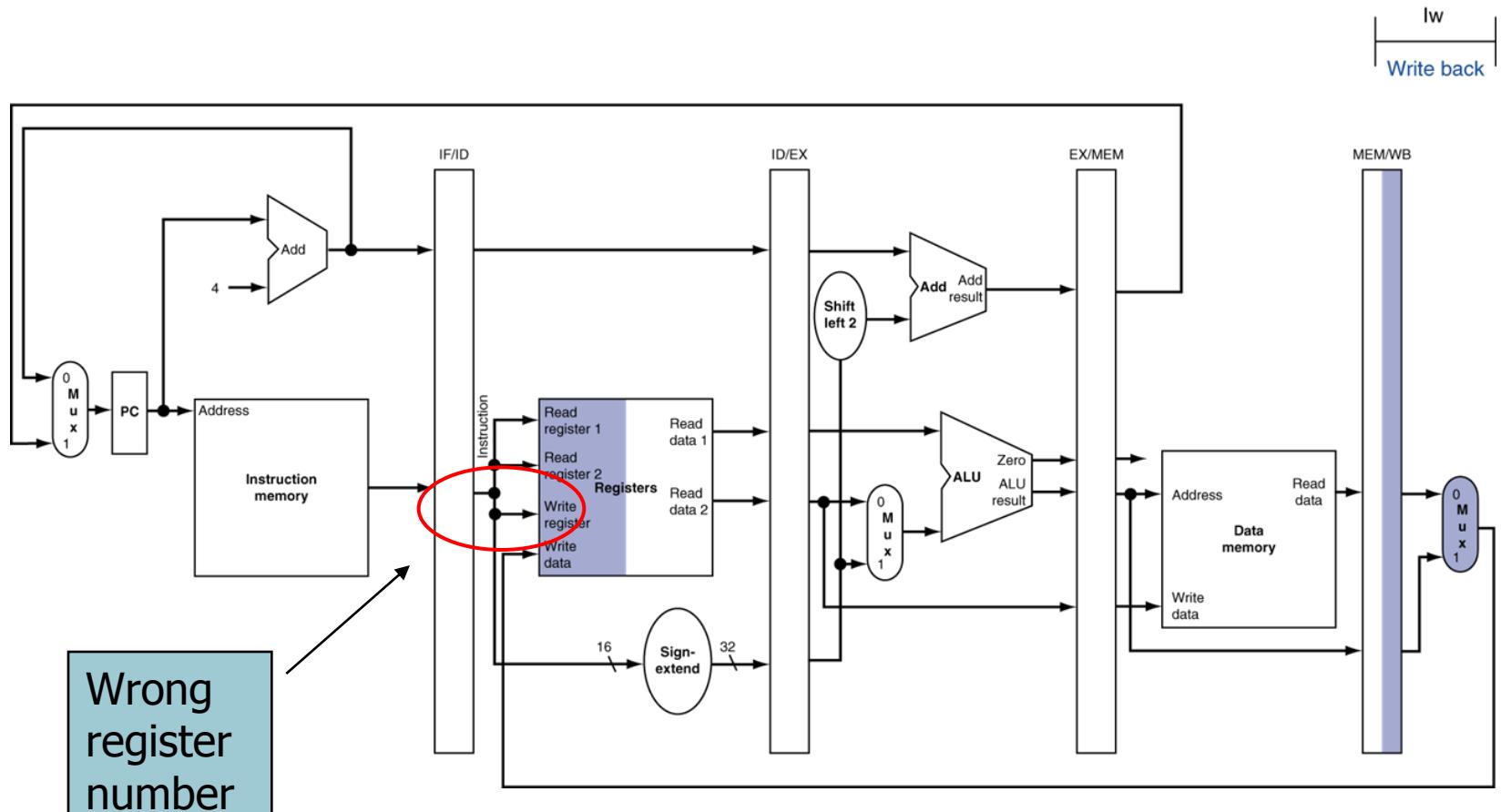
Bước thực hiện lệnh (Load)



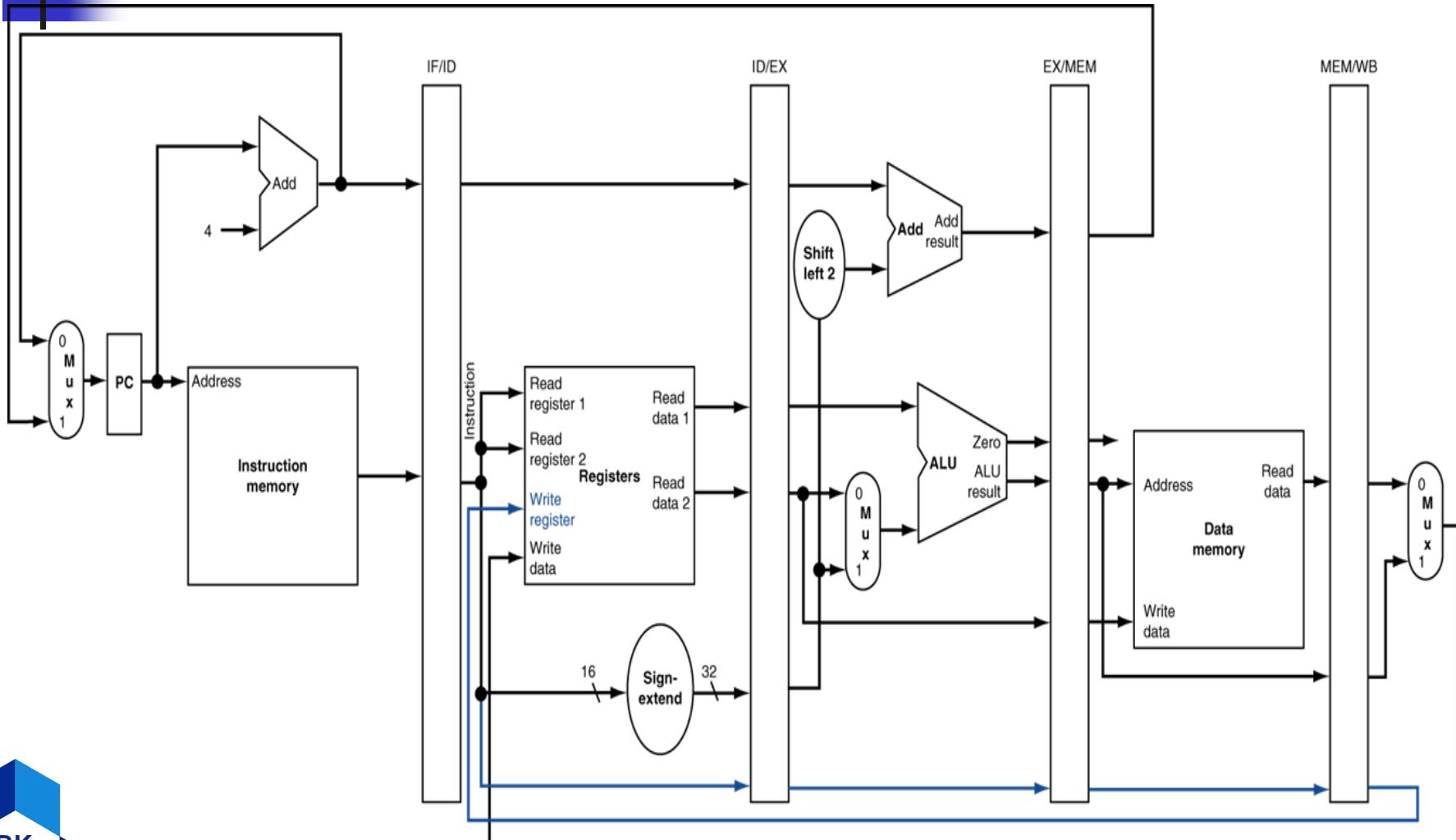
Bước truy cập bộ nhớ (Load)



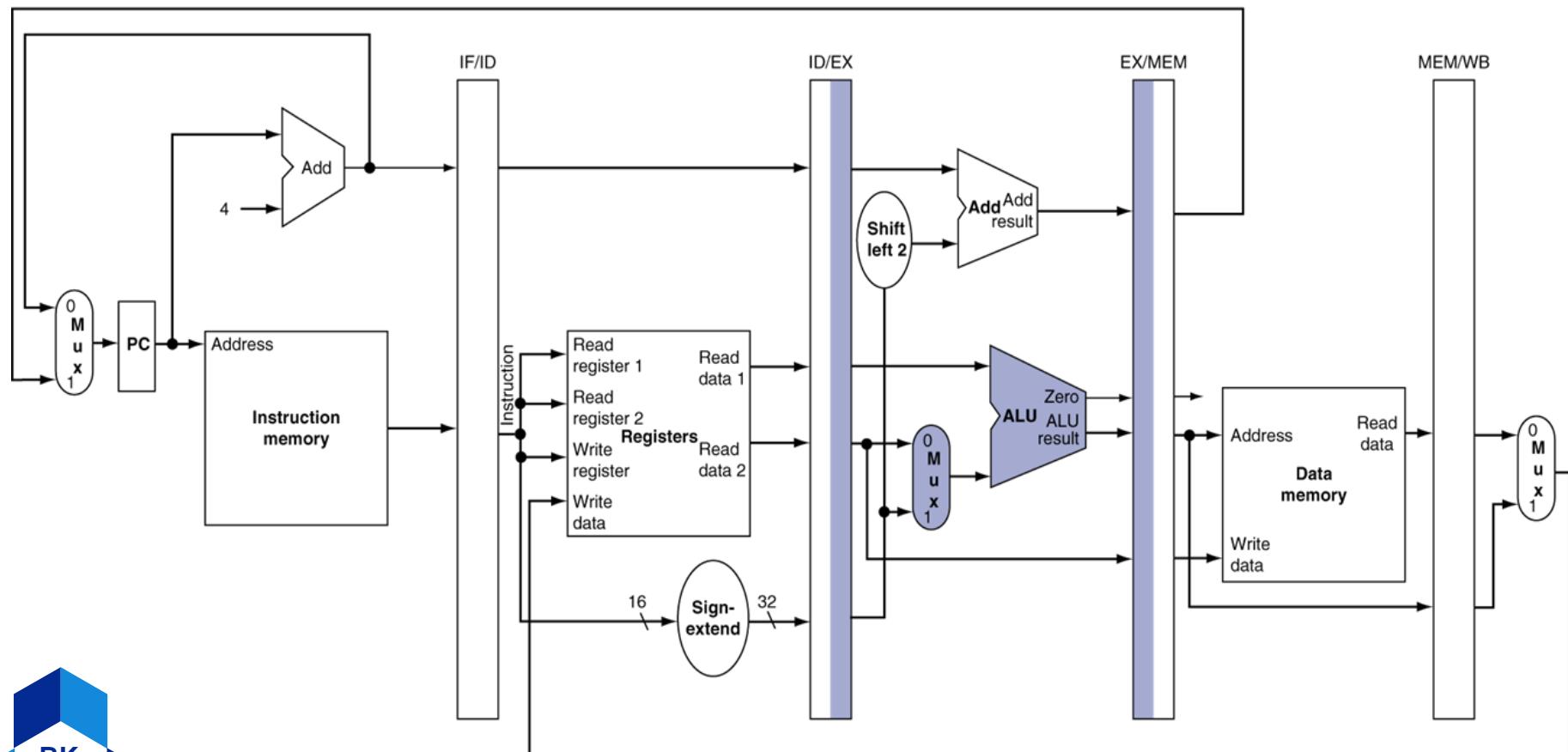
Bước ghi thanh ghi (Load)



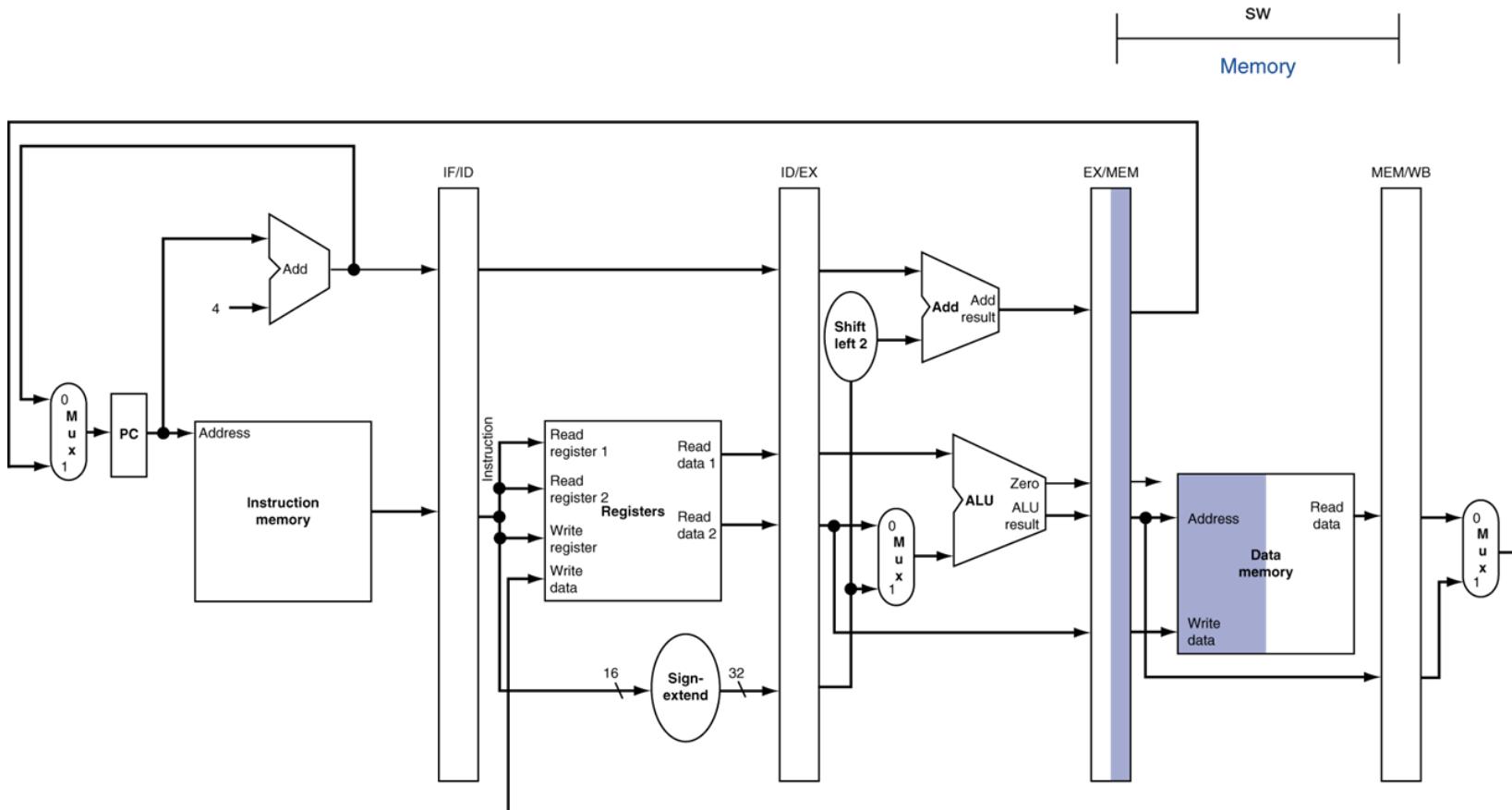
Lộ trình đúng (Load)



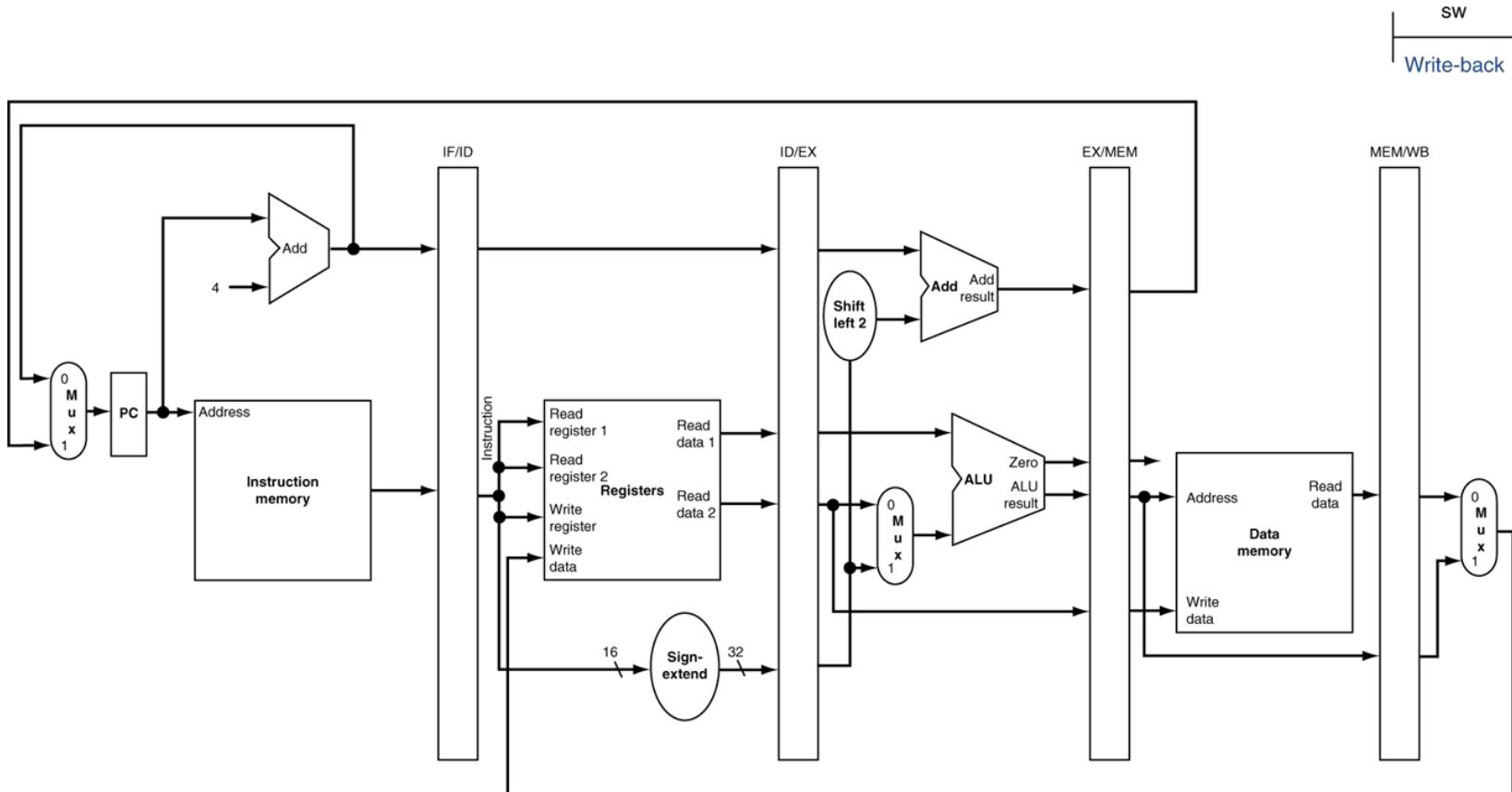
Bước thực hiện (Store)



Bước ghi MEM (Store)

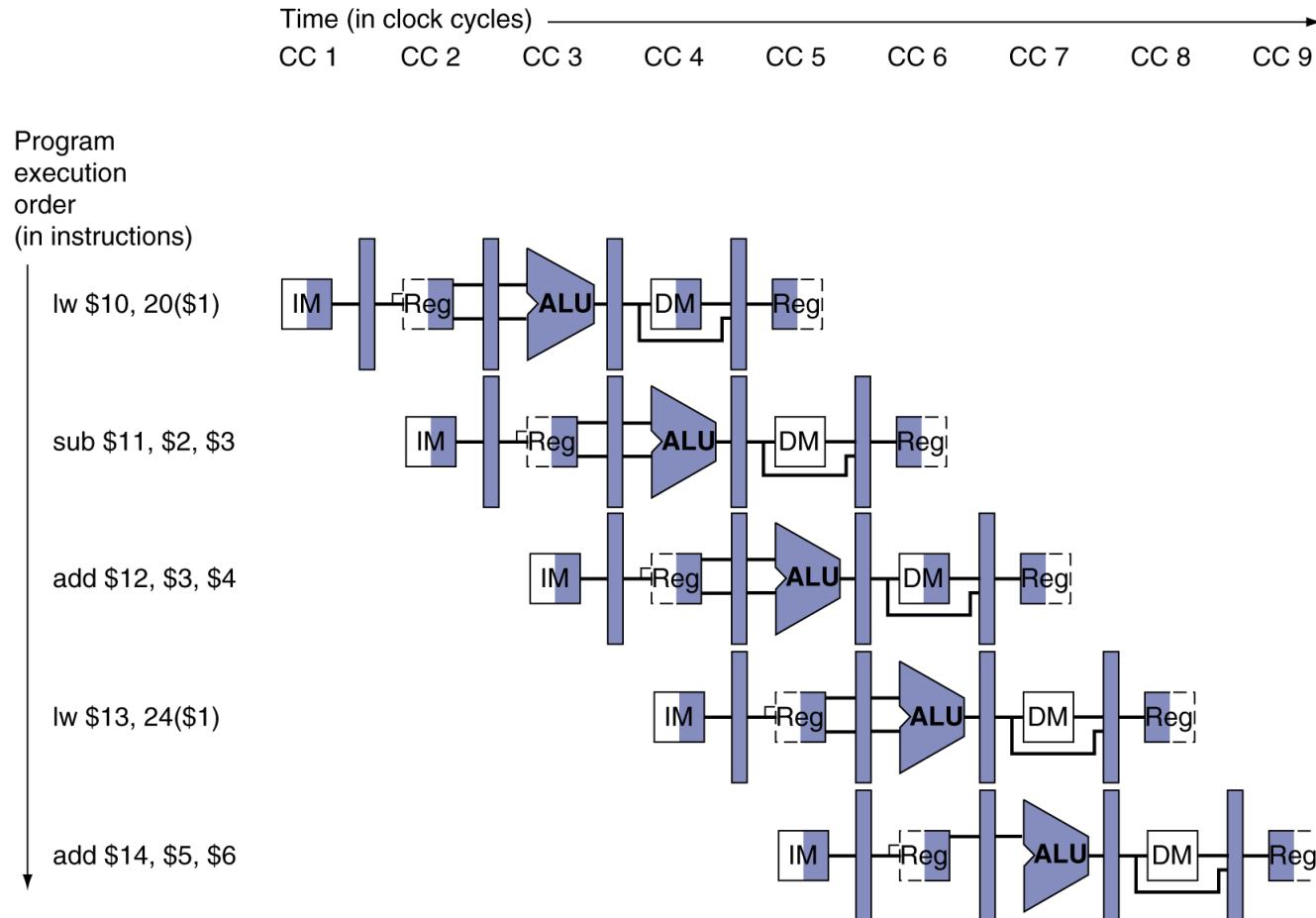


Bước ghi lên bộ nhớ (Store)



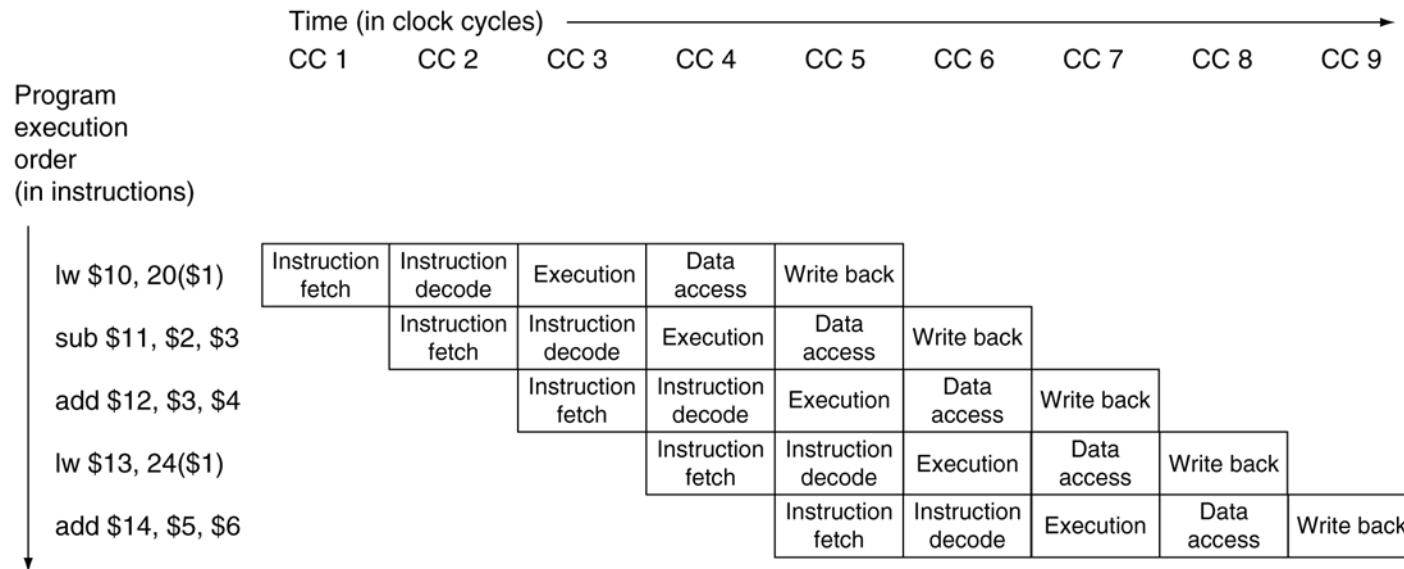
Biểu đồ ống đa bước (chu kỳ)

- “Multiple-Clock-Cycle”



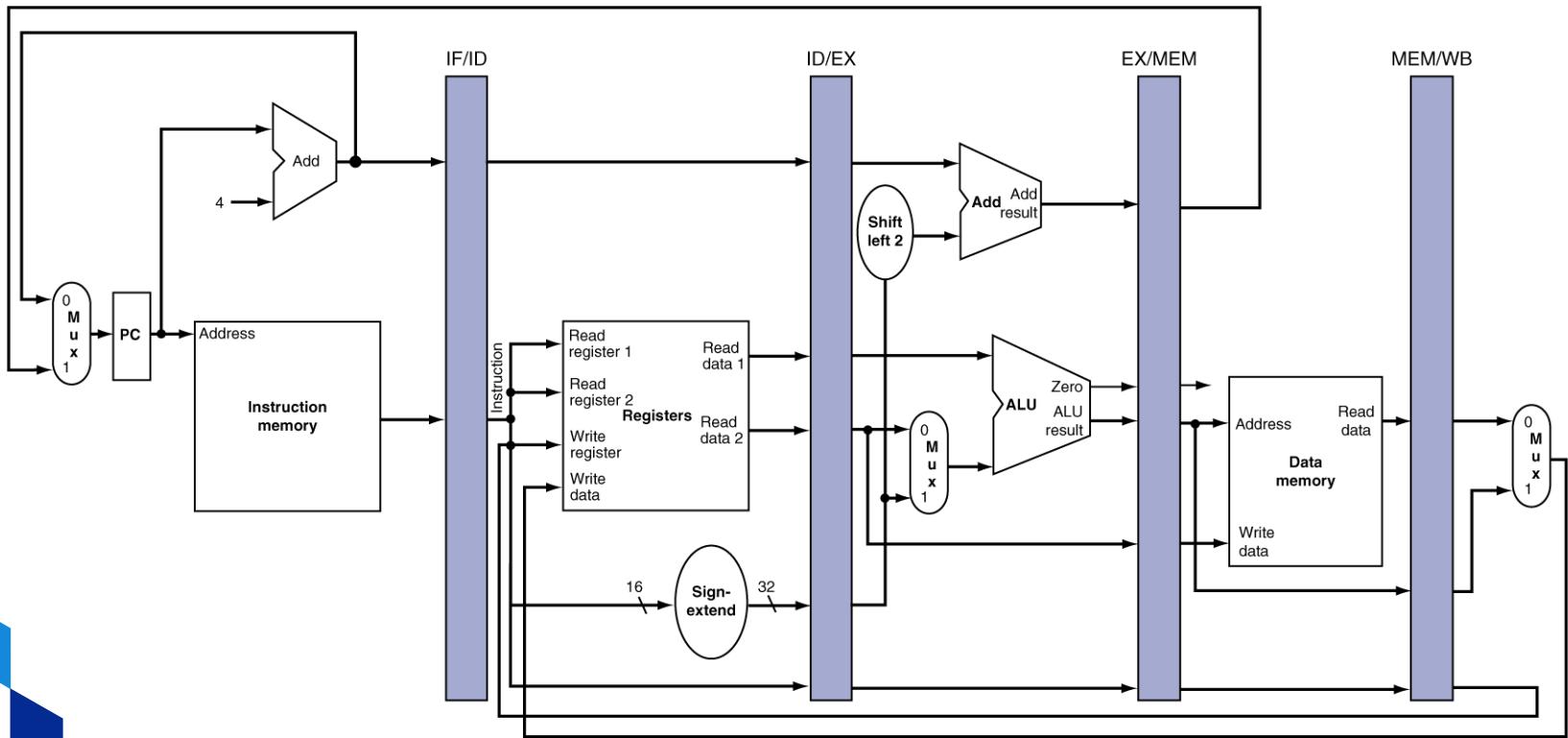
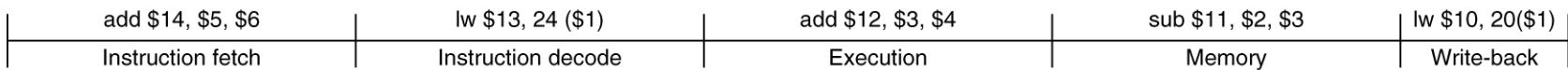
Biểu đồ ống đa bước (tt.)

■ Cách biểu diễn truyền thống

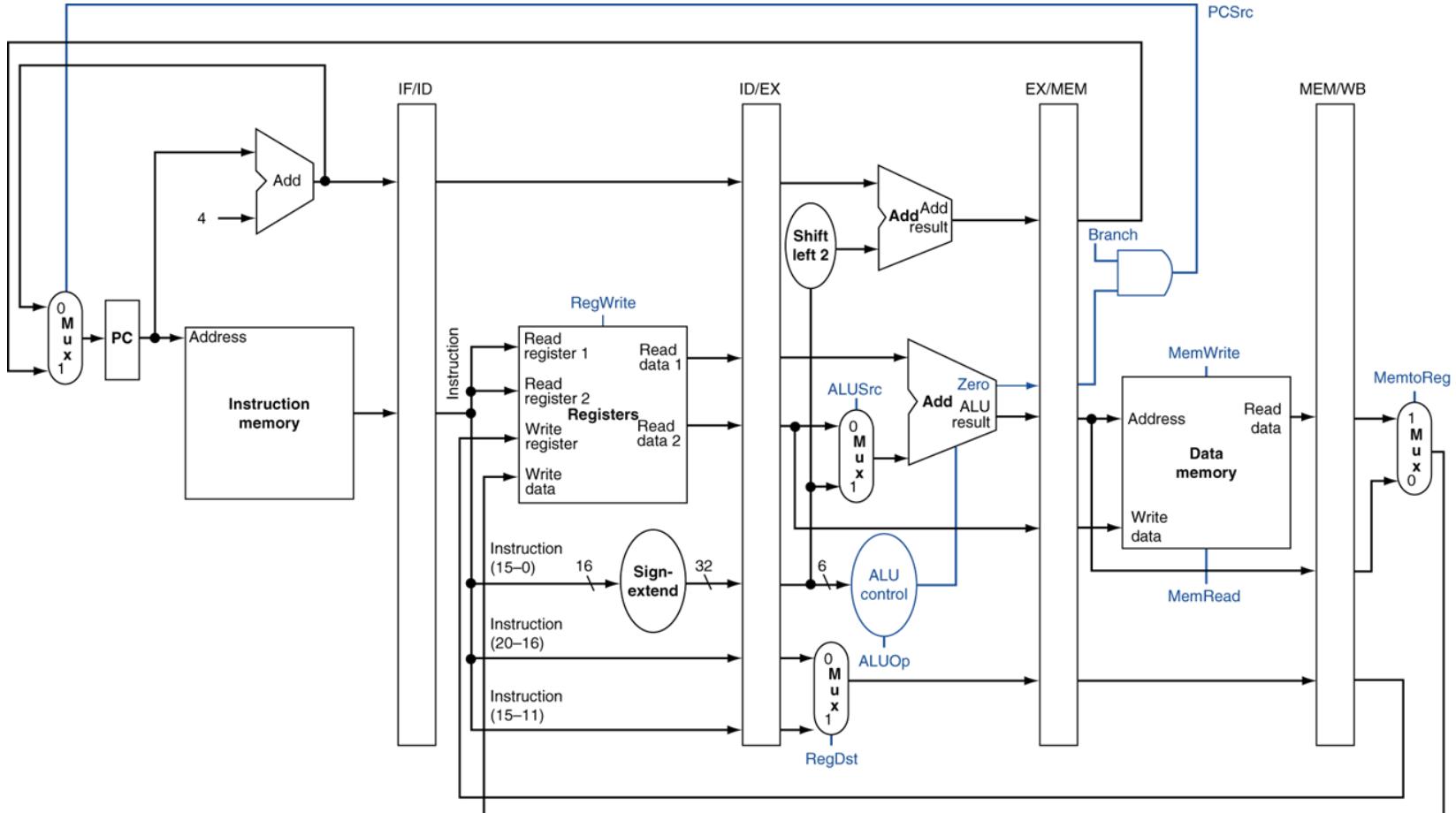


Biểu đồ ống đơn bước

- “Single-Clock-Cycle”
- Trạng thái của ống trong 1 chu kỳ

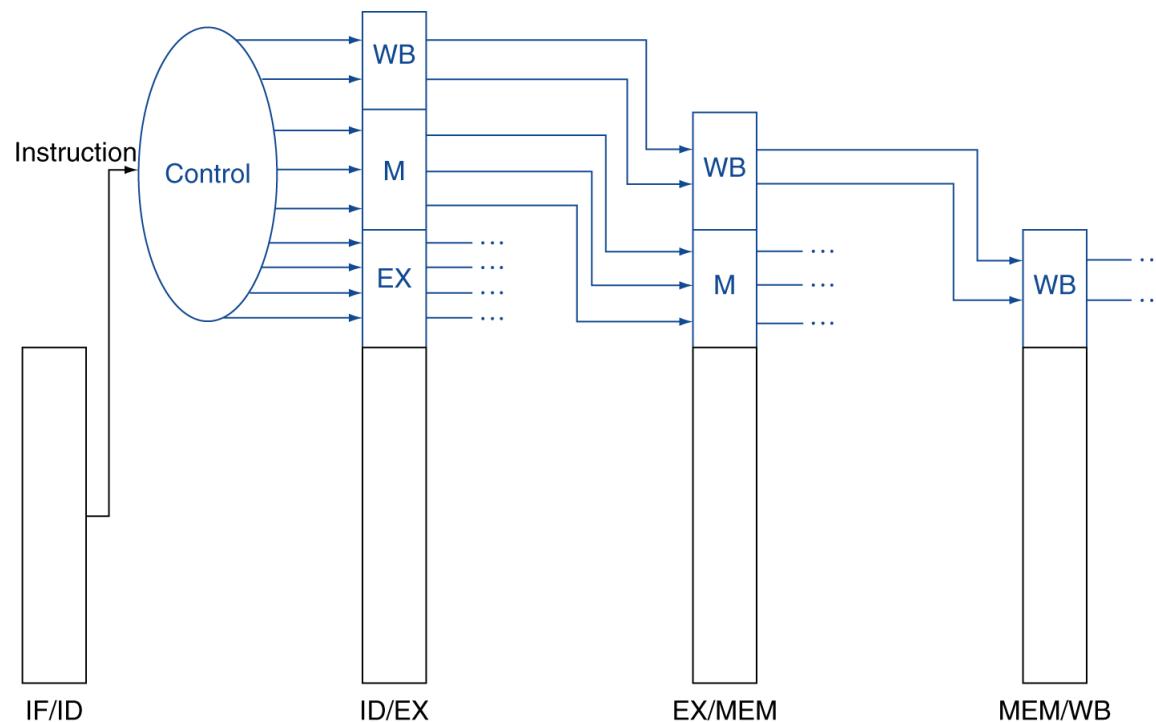


Điều khiển cơ chế ống (đã đơn giản)

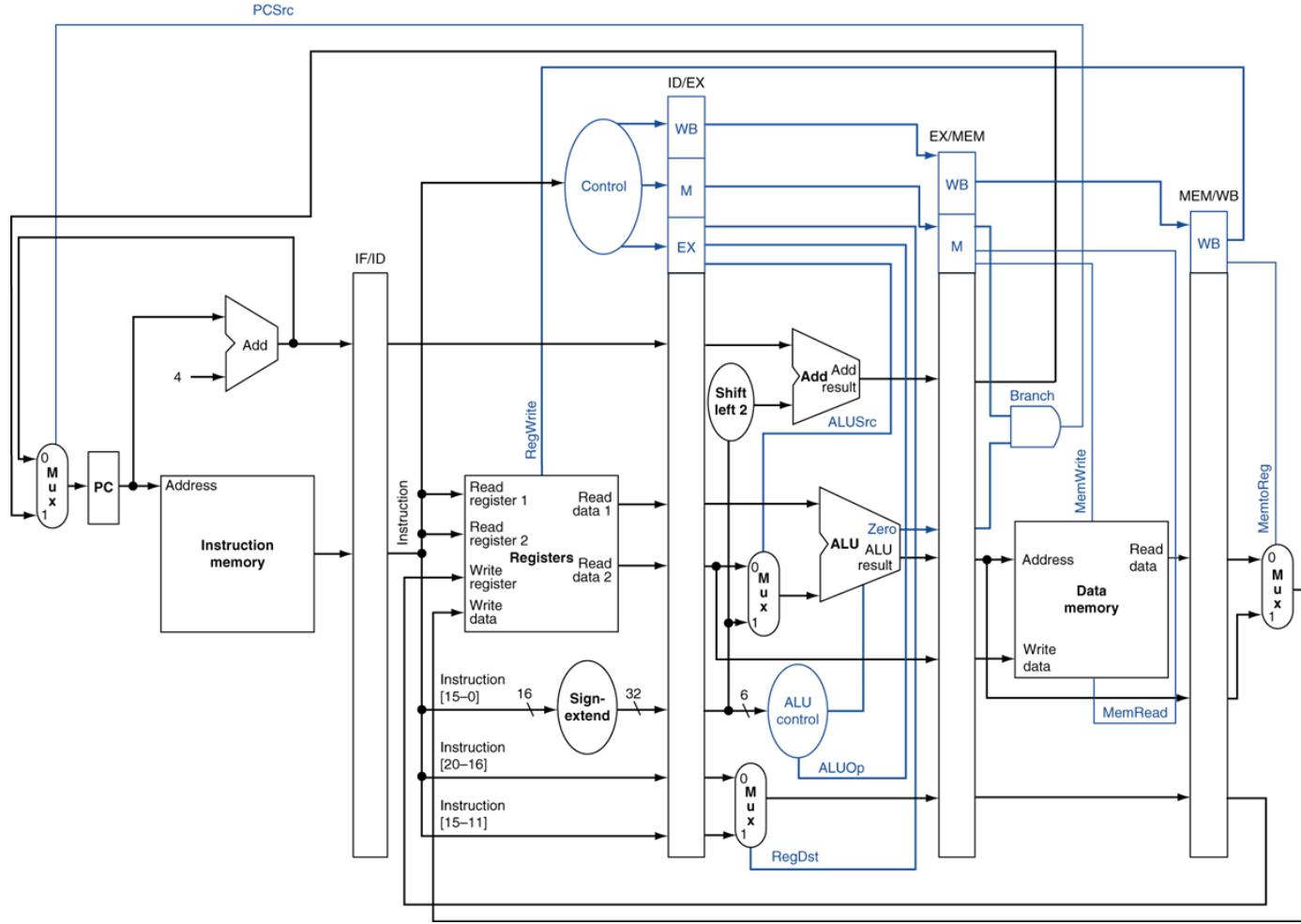


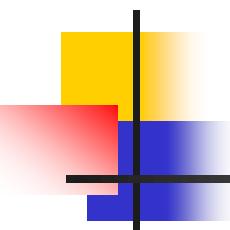
Điều khiển cơ chế ống (tt.)

- Tín hiệu điều khiển xác lập từ lệnh:
 - thực hiện đơn bước



Điều khiển cơ chế ống





Rủi ro dữ liệu khi thực hiện lệnh ALU

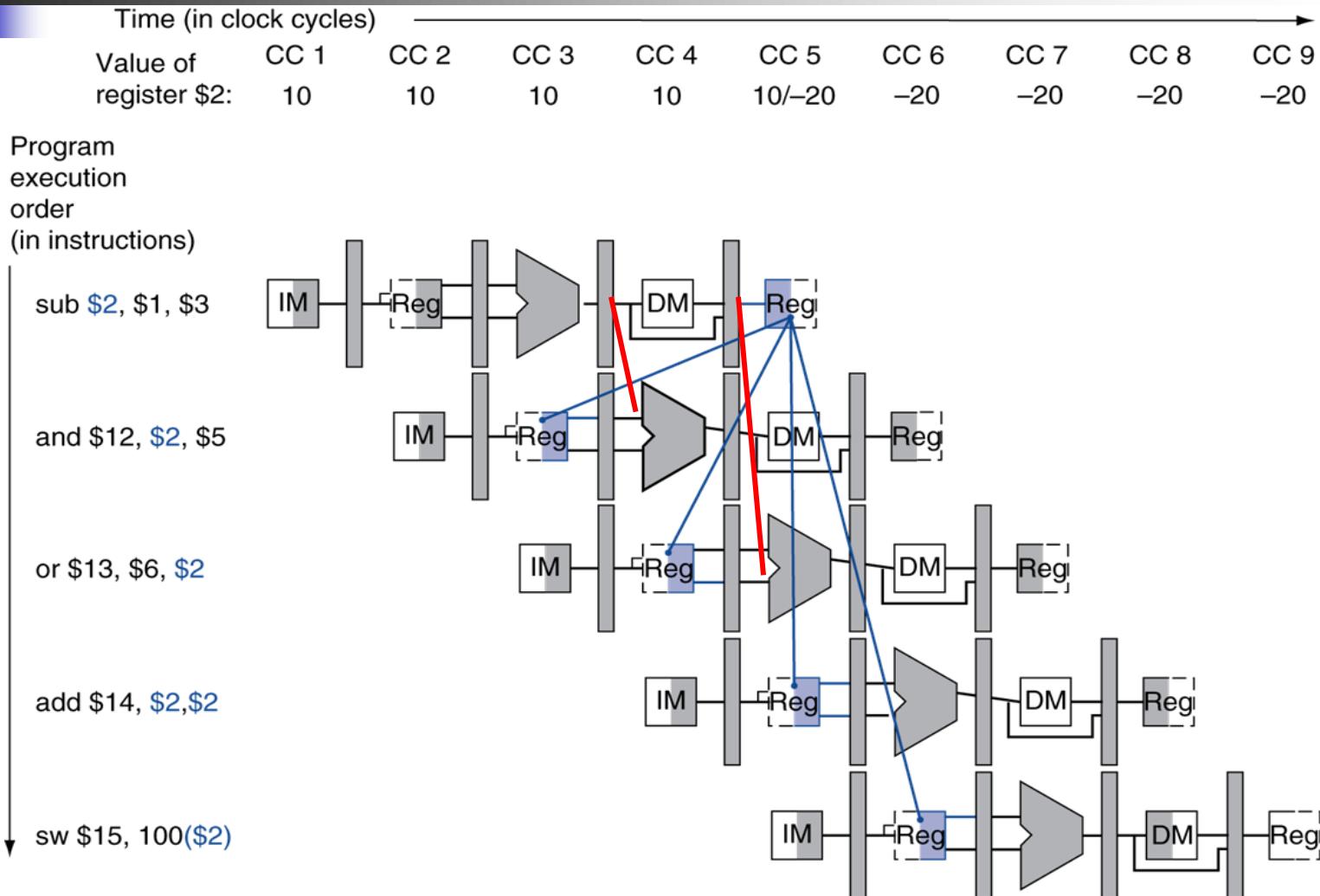
- Quan sát đoạn code sau:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- Ta có thể áp dụng phương pháp forwarding để giải quyết rủi ro
 - Làm thế nào để xác định khi nào forwarding?



Sự ràng buộc & Forwarding

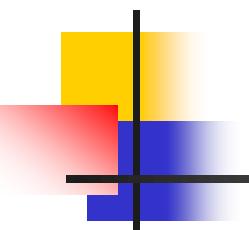


Phát hiện yêu cầu Forward

- Chuyển Chỉ số thanh ghi theo đường ống
 - Ví dụ: ID/EX.RegisterRs = Chỉ số của Rs trong thanh ghi ống giai đoạn ID/EX
- Chỉ số thanh ghi toán hạng (ALU) trong công đoạn thực hiện (EX) lệnh sẽ là
 - ID/EX.RegisterRs, ID/EX.RegisterRt
- Rủi ro dữ liệu xuất hiện khi:
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Xúc tiến sớm
từ th/ghi
EX/MEM

Xúc tiến sớm
từ th/ghi
MEM/WB

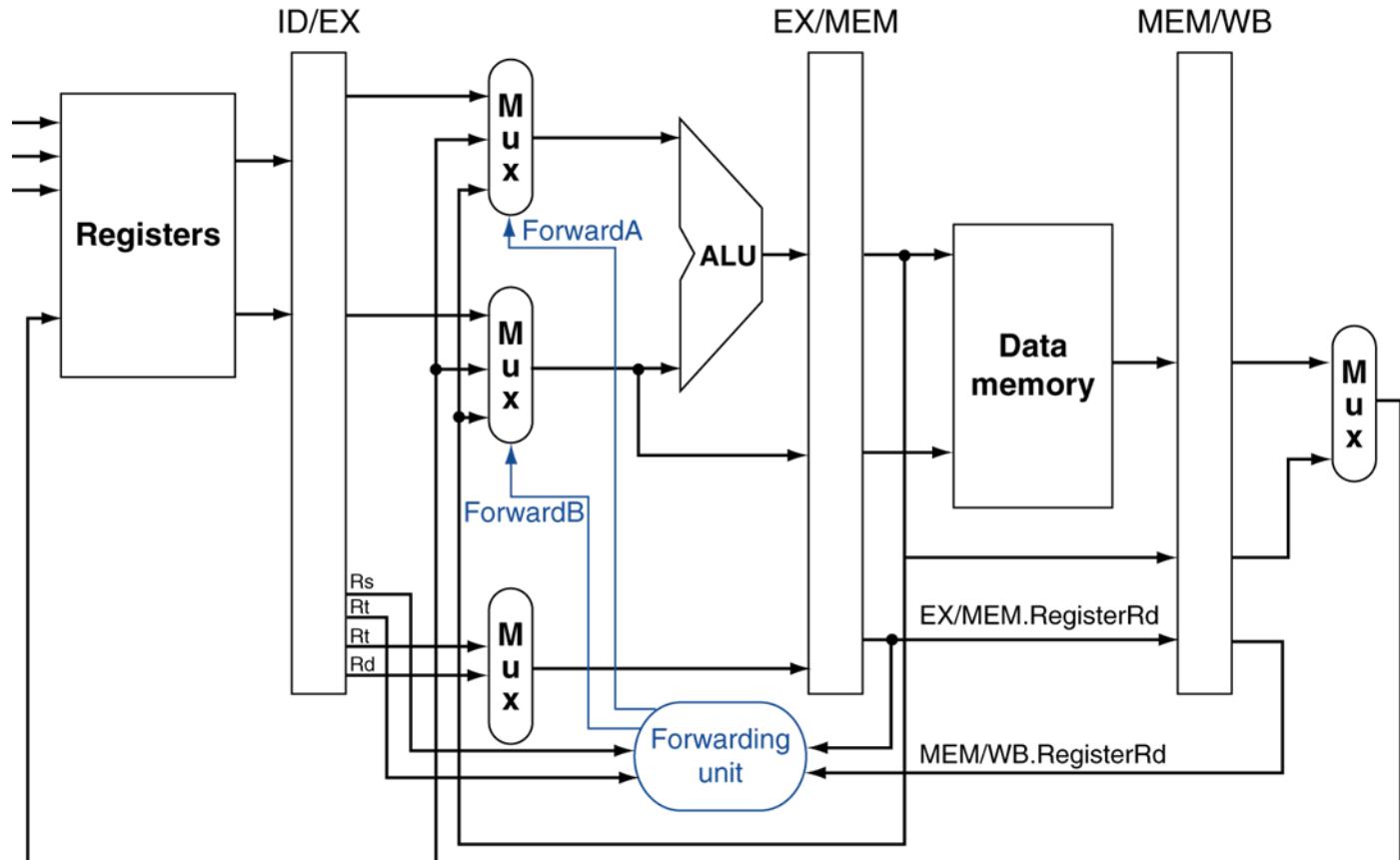


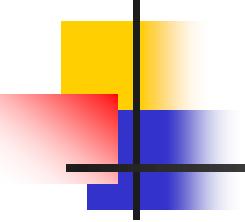
Phát hiện yêu cầu Forward (tt.)

- Nhưng chỉ với trường hợp lệnh cần xúc tiến sớm có ghi ra thanh ghi, đó là
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- Và thanh ghi Rd không phải là th/ghi \$zero
 - EX/MEM.RegisterRd $\neq 0$,
MEM/WB.RegisterRd $\neq 0$



Lộ trình xúc tiến sớm

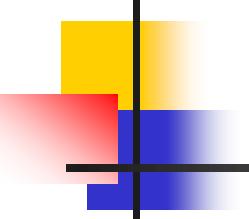




Các điều kiện xúc tiến sớm

- EX hazard
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

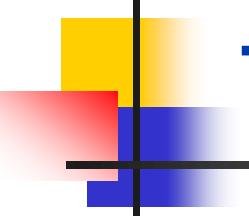




Rủi ro dữ liệu đúp

- Quan sát 3 lệnh dưới đây (Cộng dồn các phần tử của 1 dãy – Vector):
add \$1,\$1,\$2
add \$1,\$1,\$3
add \$1,\$1,\$4
- Xảy ra 2 loại Hazards: Ex và MEM
 - Dùng kết quả mới nhất của \$1
- Xét lại điều kiện để xúc tiến sớm



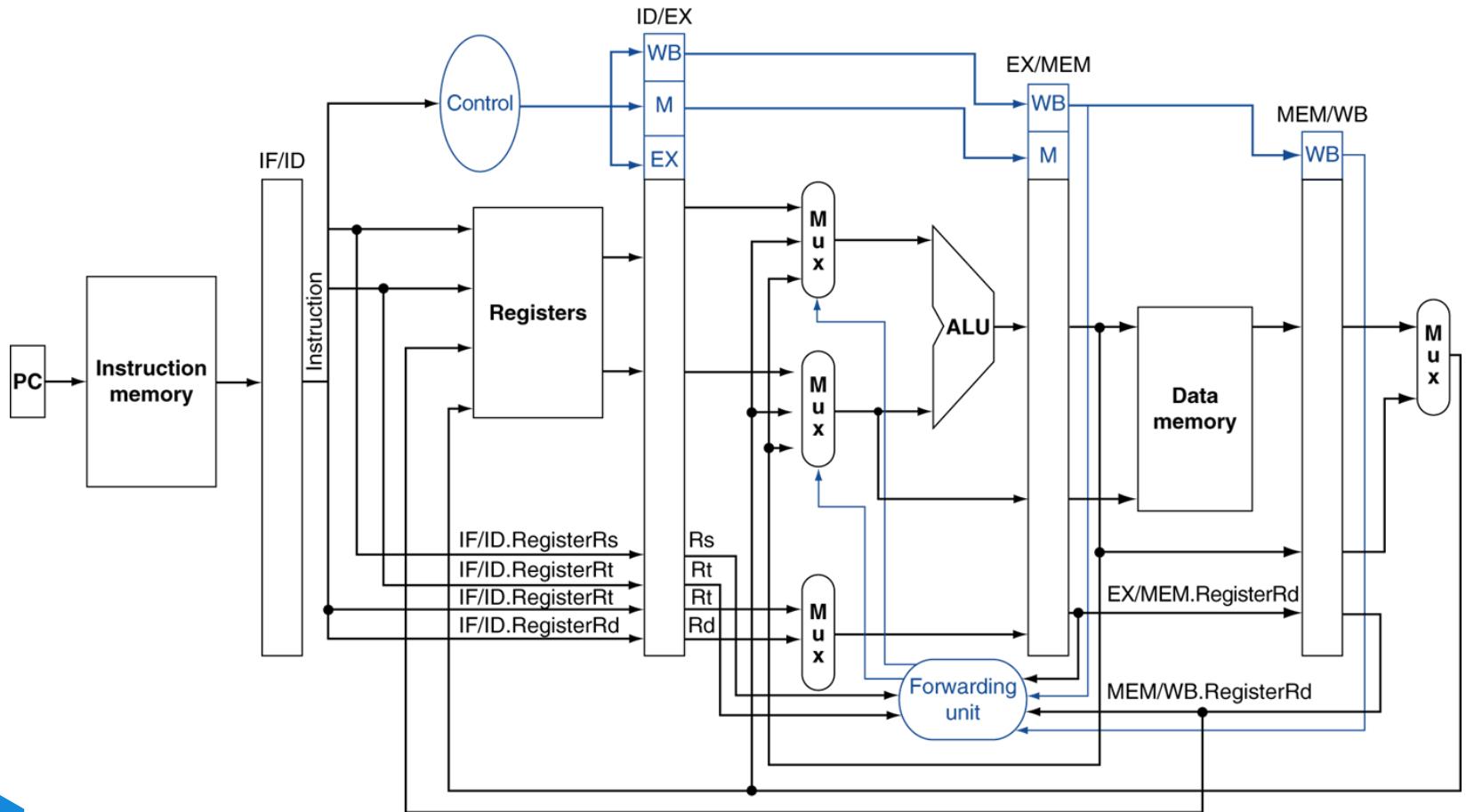


Điều kiện xúc tiến sớm xét lại

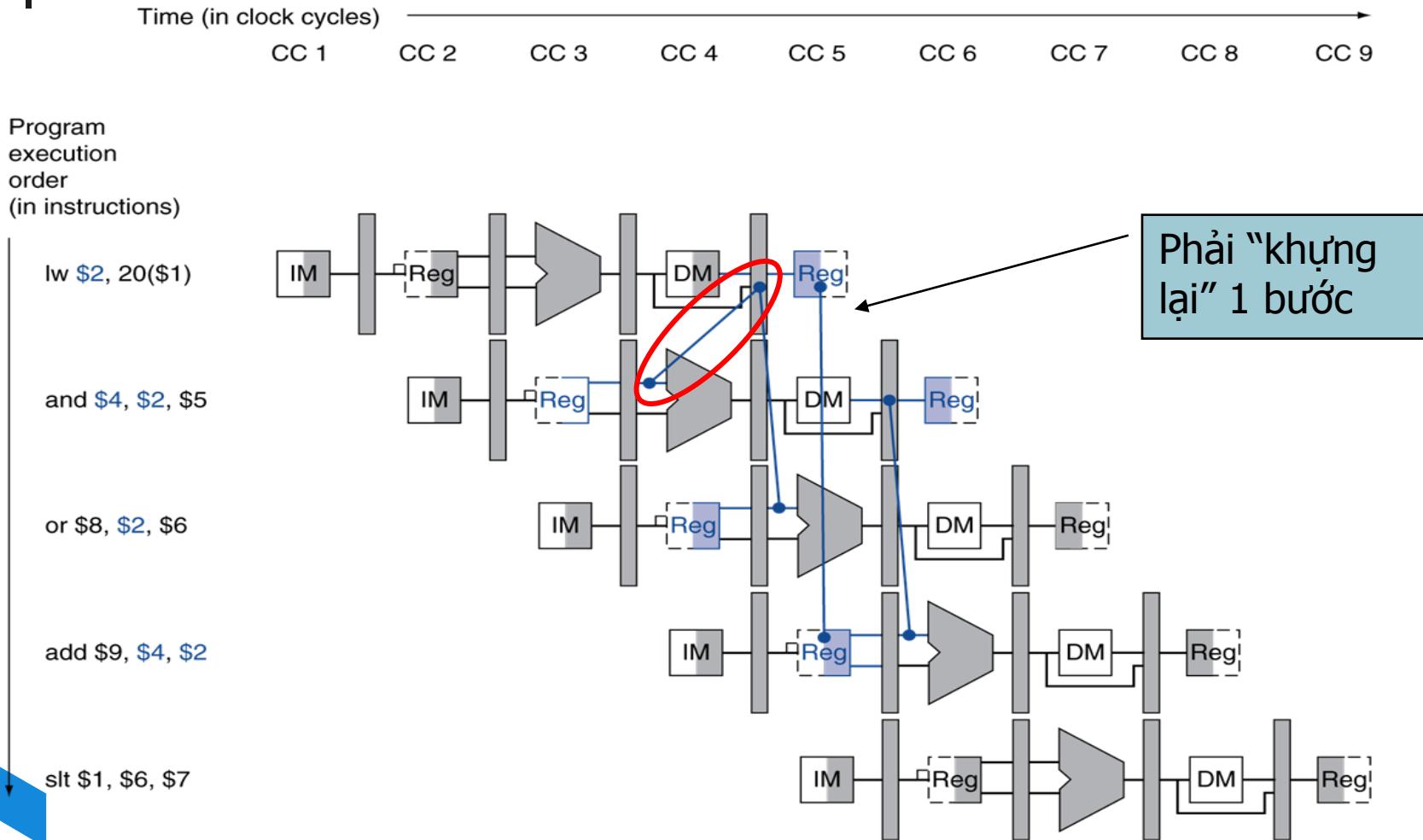
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

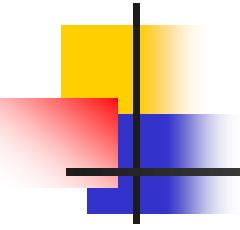


Lộ trình với Forwarding



Rủi ro dữ liệu với lệnh Load

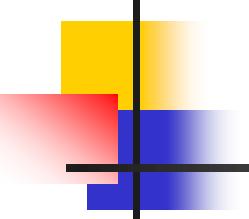




Phát hiện rủi ro do lệnh Load

- Kiểm tra lệnh trong giai đoạn giải mã (ID)
- Thanh ghi toán hạng của lệnh (inputs of ALU):
 - IF/ID.RegisterRs, IF/ID.RegisterRt
- Rủi ro khi thực hiện Load nếu
 - ID/EX.MemRead and
 $((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$
- Nếu phát hiện, thì khung lại và “nop”



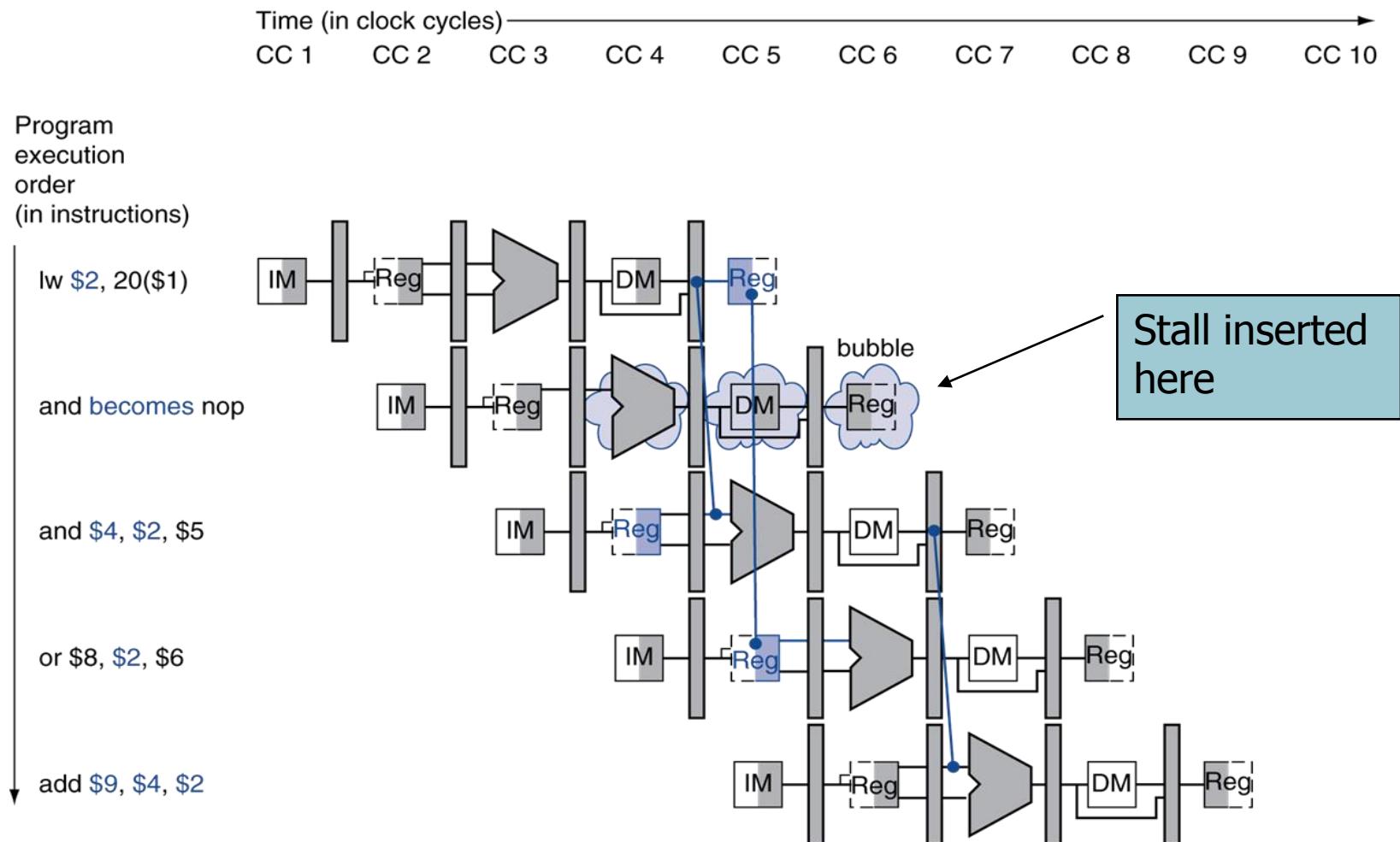


Làm “Khụng lại” ?

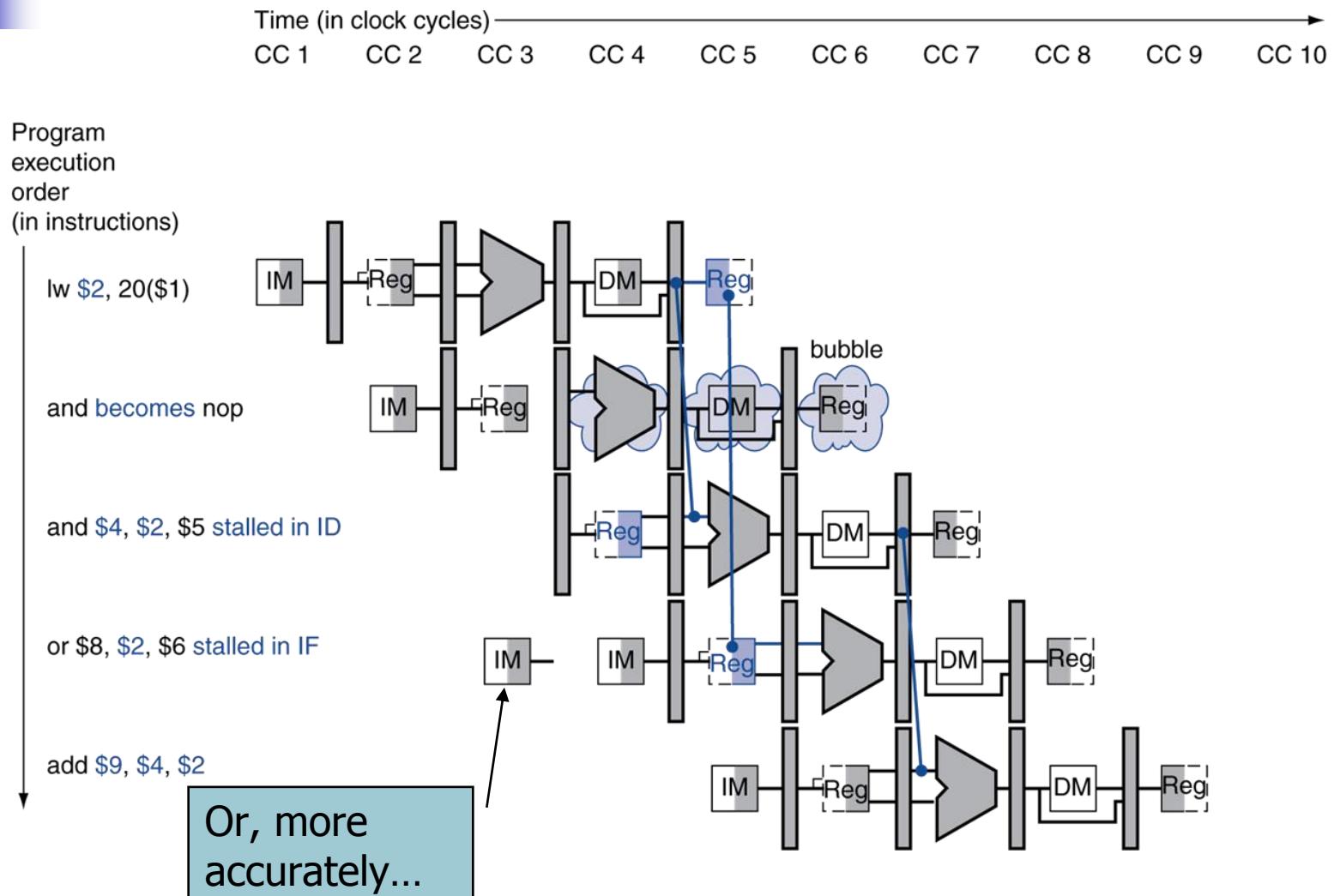
- Giữ các giá trị điều khiển thanh ghi trong bước ID/EX bằng 0
 - EX, MEM & WB thực hiện nop (no-op)
- Không cập nhật PC & IF/ID register
 - Sử dụng lại bước giải mã lệnh
 - Nạp lệnh tiếp theo lần nữa
 - 1-cyc đủ để đọc dữ liệu từ MEM đối với 1w
 - Can subsequently forward to EX stage



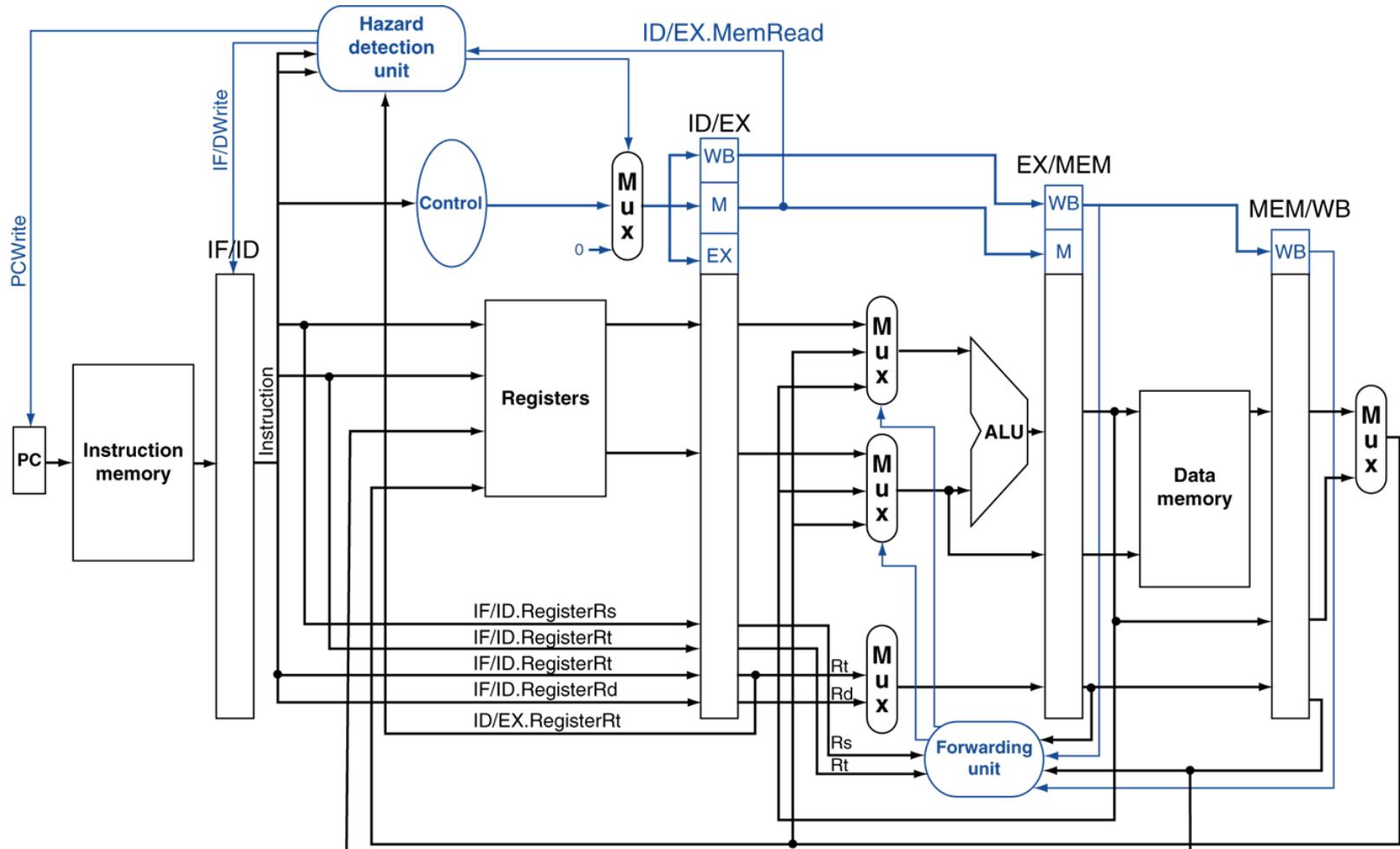
Stall/Bubble in the Pipeline

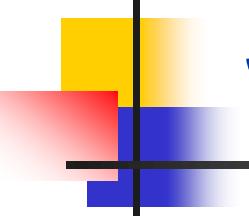


Stall/Bubble in the Pipeline



Lộ trình dữ liệu với bộ phát hiện rủi ro





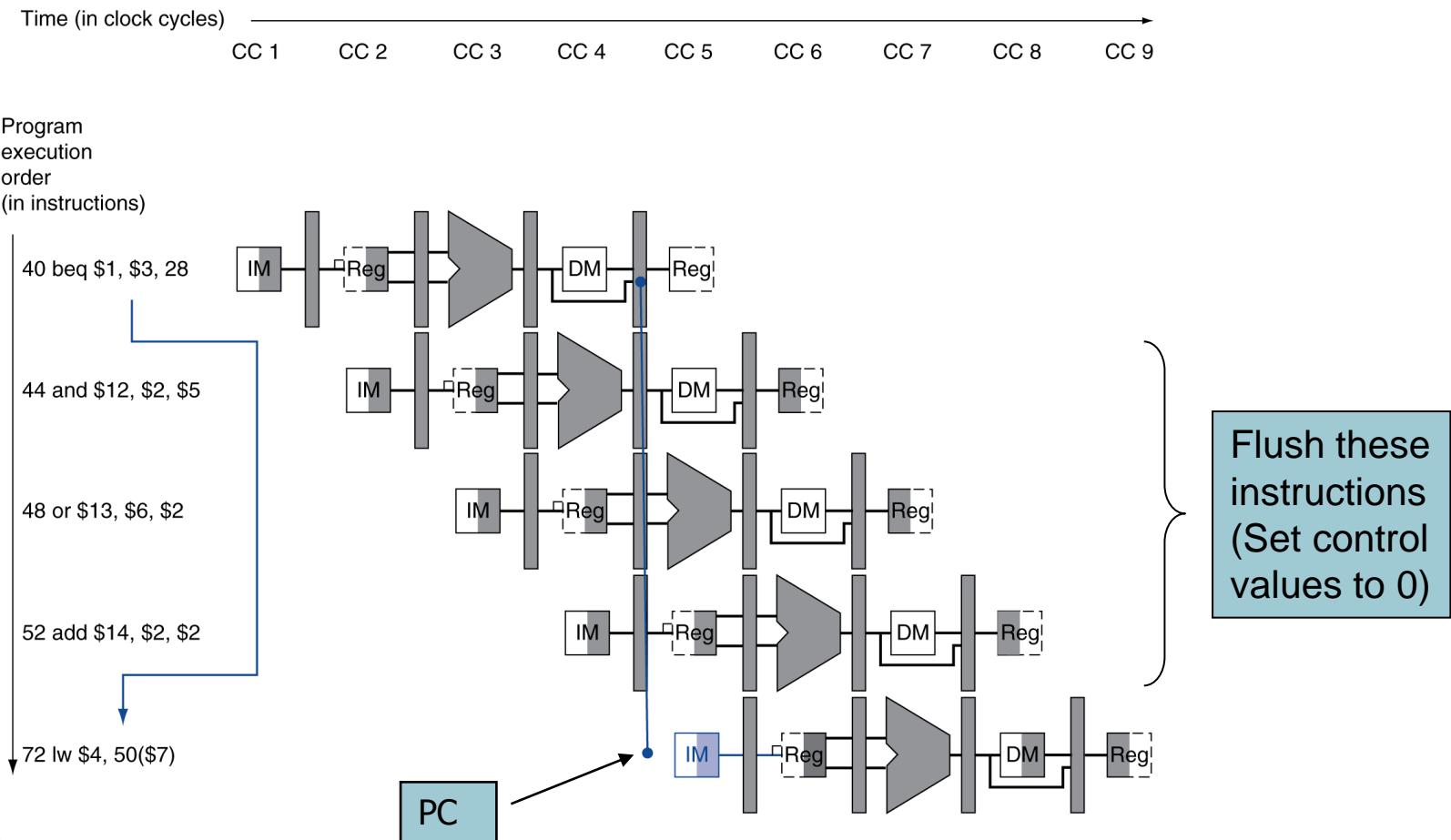
Sự “khụng lại” & Hiệu suất

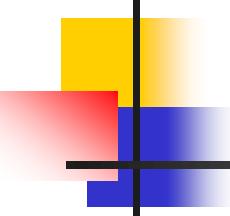
- Sự “Khụng lại” làm giảm hiệu suất
 - Nhưng cần thiết để cho kết quả đúng
- Biên dịch có thể sắp xếp lại trật tự các lệnh sao cho rủi ro và sự “ khụng lại” không xảy ra
 - Yêu cầu thông tin về cấu trúc thực hiện trong ống



Rủi ro điều khiển (rẽ nhánh)

- Nếu rẽ nhánh được xác định trong bước MEM





Giảm độ trễ do thực hiện rẽ nhánh

- Xác định sớm bằng phần cứng ở giai đoạn ID
 - Bộ cộng địa chỉ đích (Target address adder)
 - Bộ so sánh thanh ghi (Register comparator)
- Ví dụ: Giả thiết rẽ nhánh (branch taken)

36: sub \$10, \$4, \$8

40: beq \$1, \$3, 7

44: and \$12, \$2, \$5

48: or \$13, \$2, \$6

52: add \$14, \$4, \$2

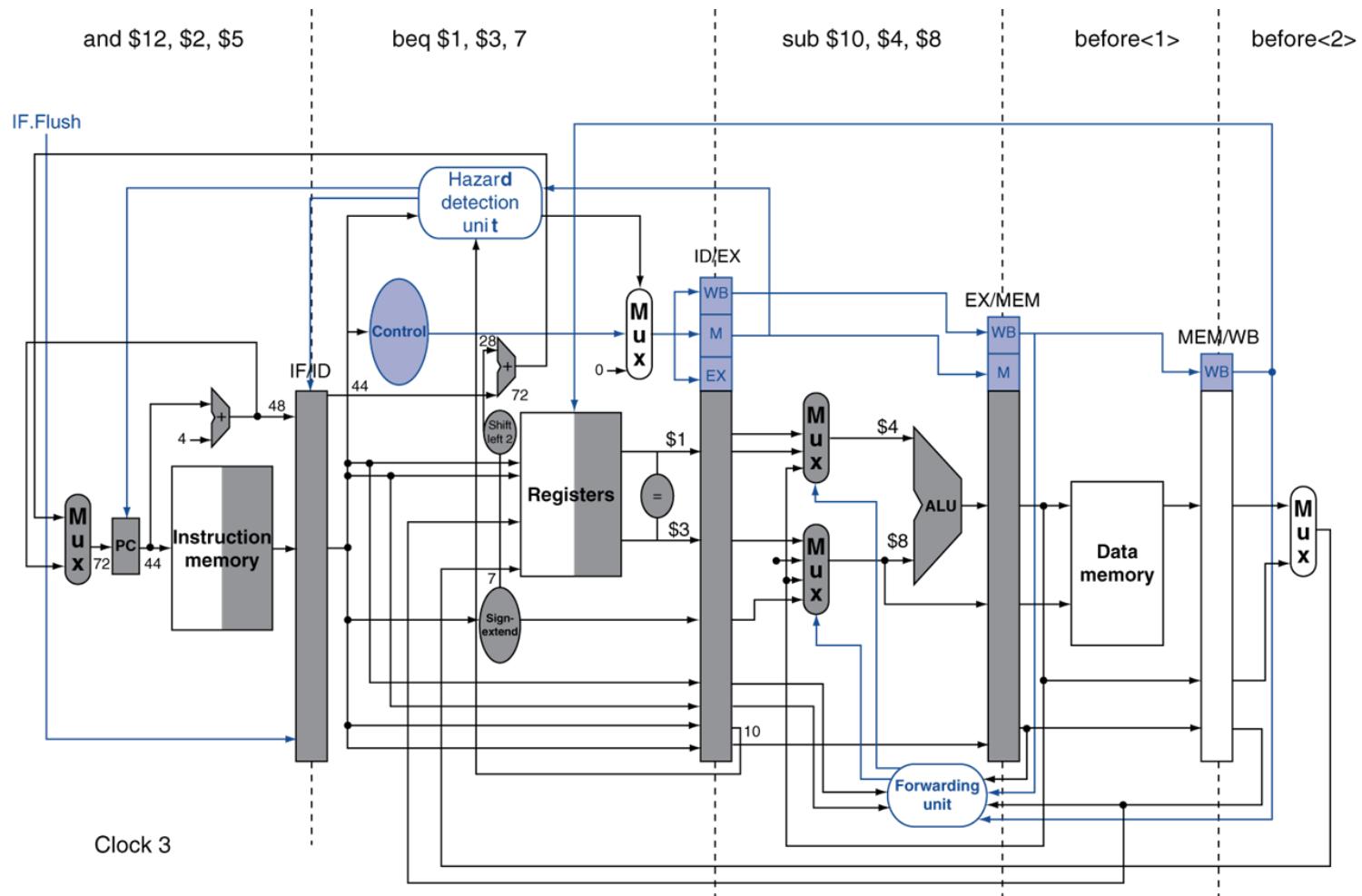
56: slt \$15, \$6, \$7

...

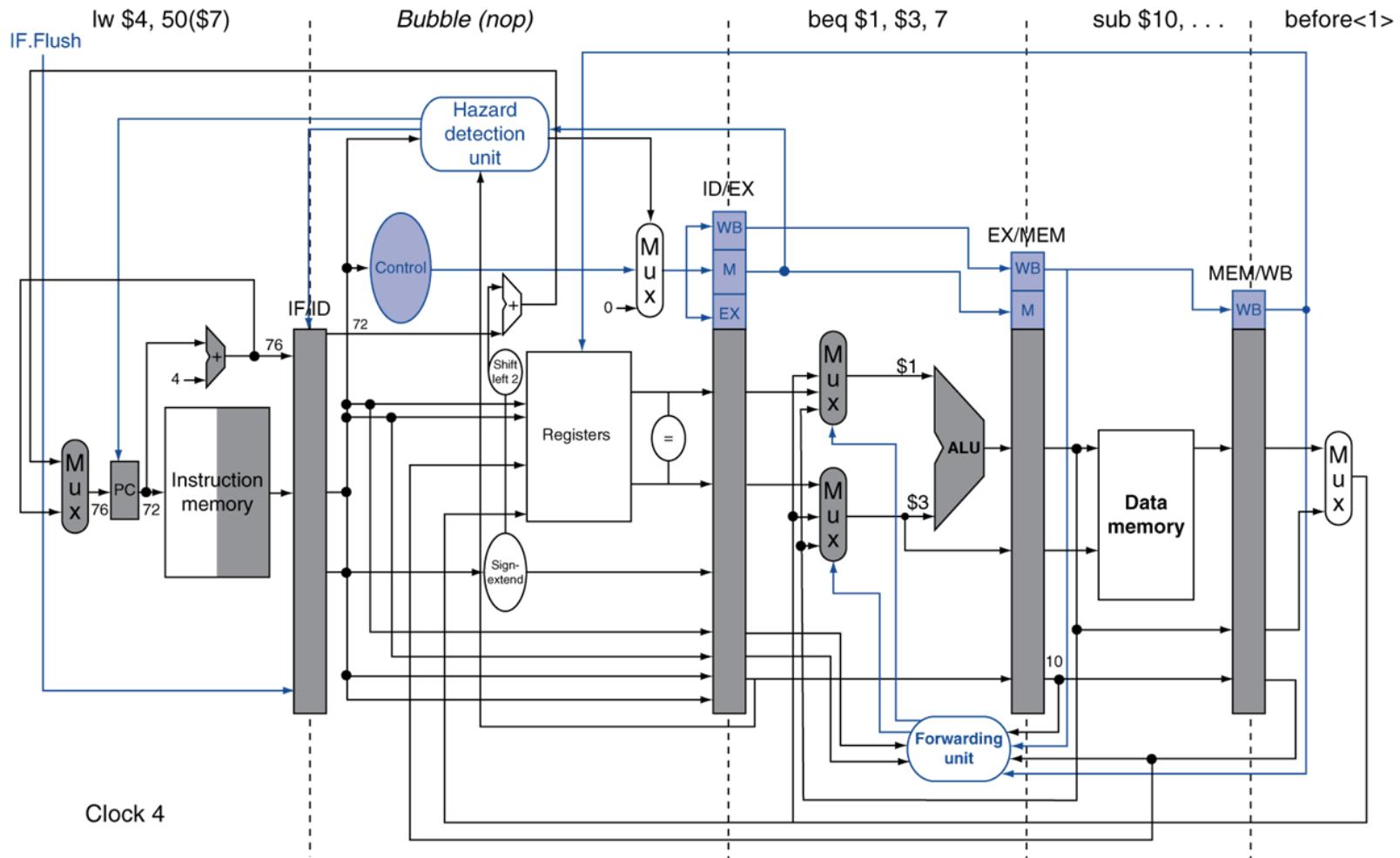
72: lw \$4, 50(\$7)



Ví dụ: Rẽ nhánh xảy ra

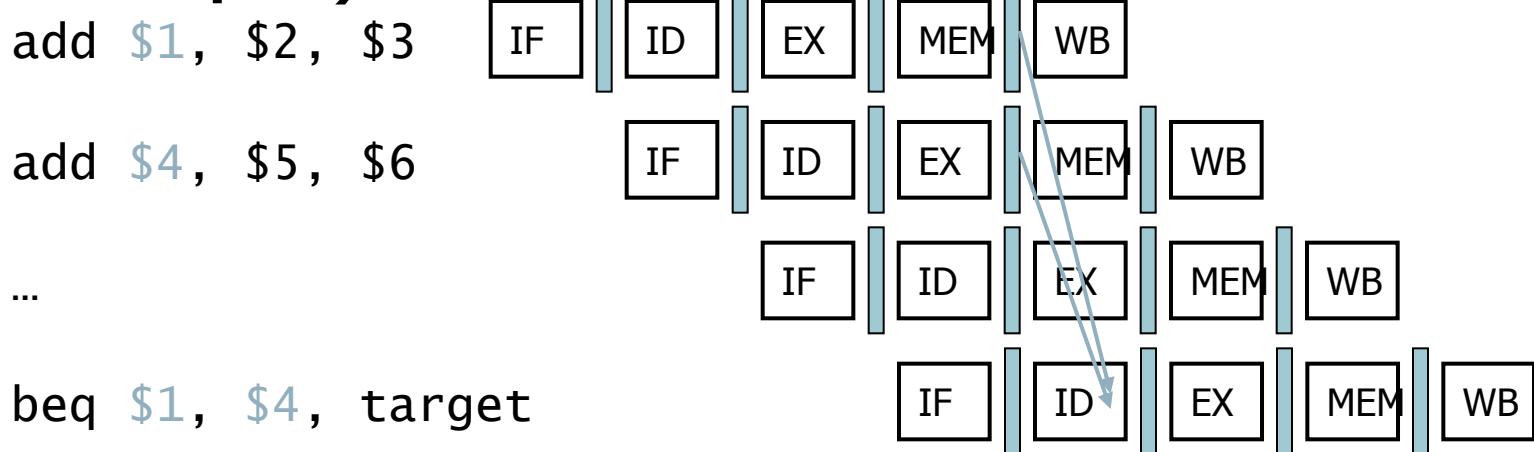


Ví dụ: Rẽ nhánh xảy ra (tt.)



Rủi ro dữ liệu với rẽ nhánh

- Nếu 1 th/ghi của lệnh so sánh là kết quả của 1 lệnh ALU trước đó (2 hay 3 lệnh)

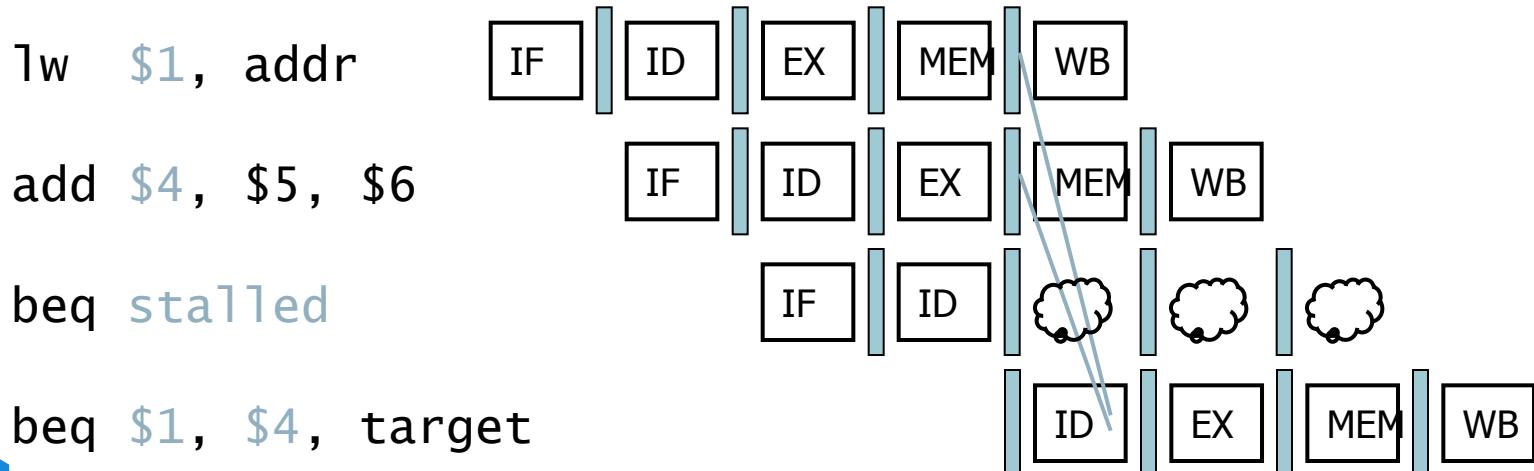


- Giải quyết bằng xúc tiếp sớm



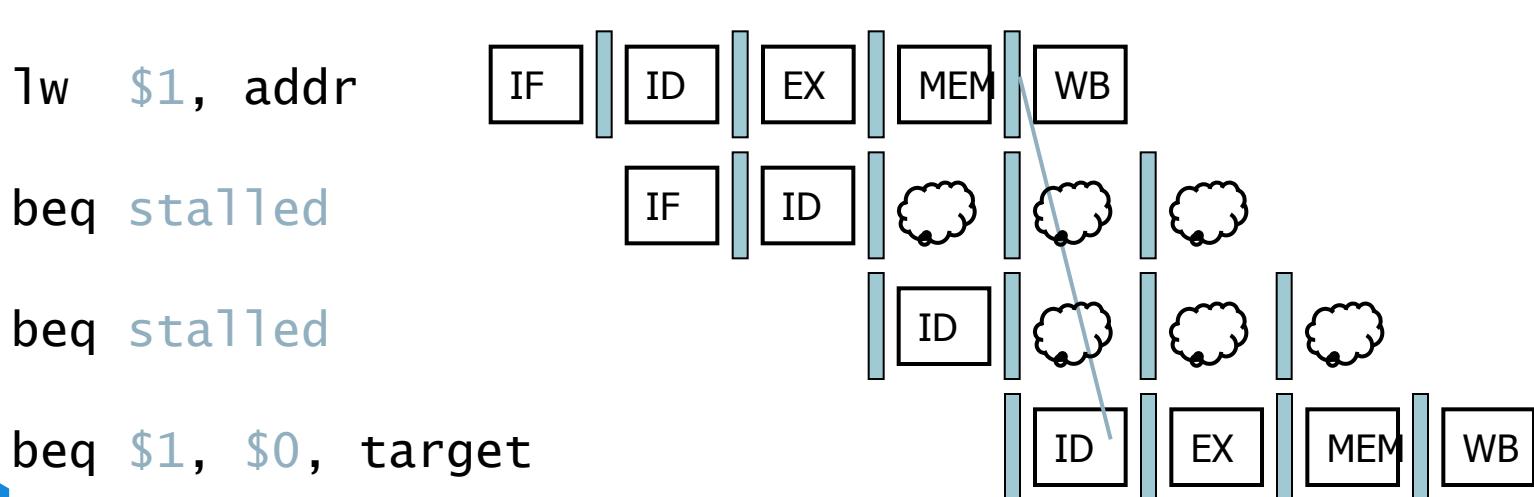
Rủi ro dữ liệu với rẽ nhánh (tt.)

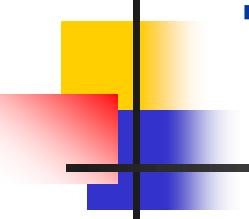
- Nếu 1 th/ghi của lệnh so sánh là kết quả của lệnh ALU ngay trước đó hoặc lệnh Load trước đó 2 lệnh
 - Cần 1 bước “khụng lại”



Rủi ro dữ liệu với rẽ nhánh (tt.)

- Nếu 1 th/ghi của lệnh so sánh là kết quả của lệnh Load ngay trước đó
 - Cần 2 bước “Khụng lại”





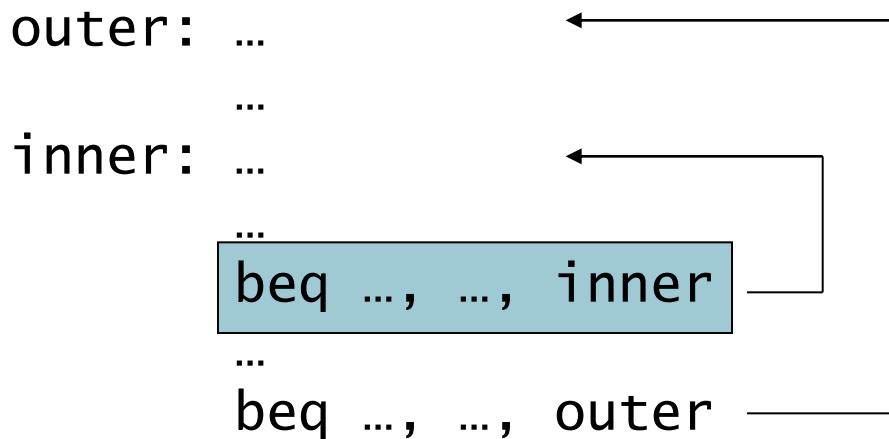
Tiên đoán động rẽ nhánh

- Ở những ống có nhiều bước, rủi ro điều khiển sẽ làm giảm hiệu xuất đáng kể
- Sử dụng phương pháp tiên đoán động
 - Bộ đệm tiên đoán (Bảng lưu lịch sử quá khứ rẽ nhánh)
 - Đánh dấu chỉ số các địa chỉ rẽ nhánh
 - Cắt kết quả rẽ nhánh (rẽ/không rẽ=tiếp tục)
 - Thực hiện rẽ nhánh bằng cách
 - Kiểm tra bảng lưu: → cùng mong đợi
 - Bắt đầu quy trình nạp (from fall-through or target)
 - Nếu sai, Xóa lưu ông, cập nhật tiên đoán



1-Bit Predictor: Shortcoming

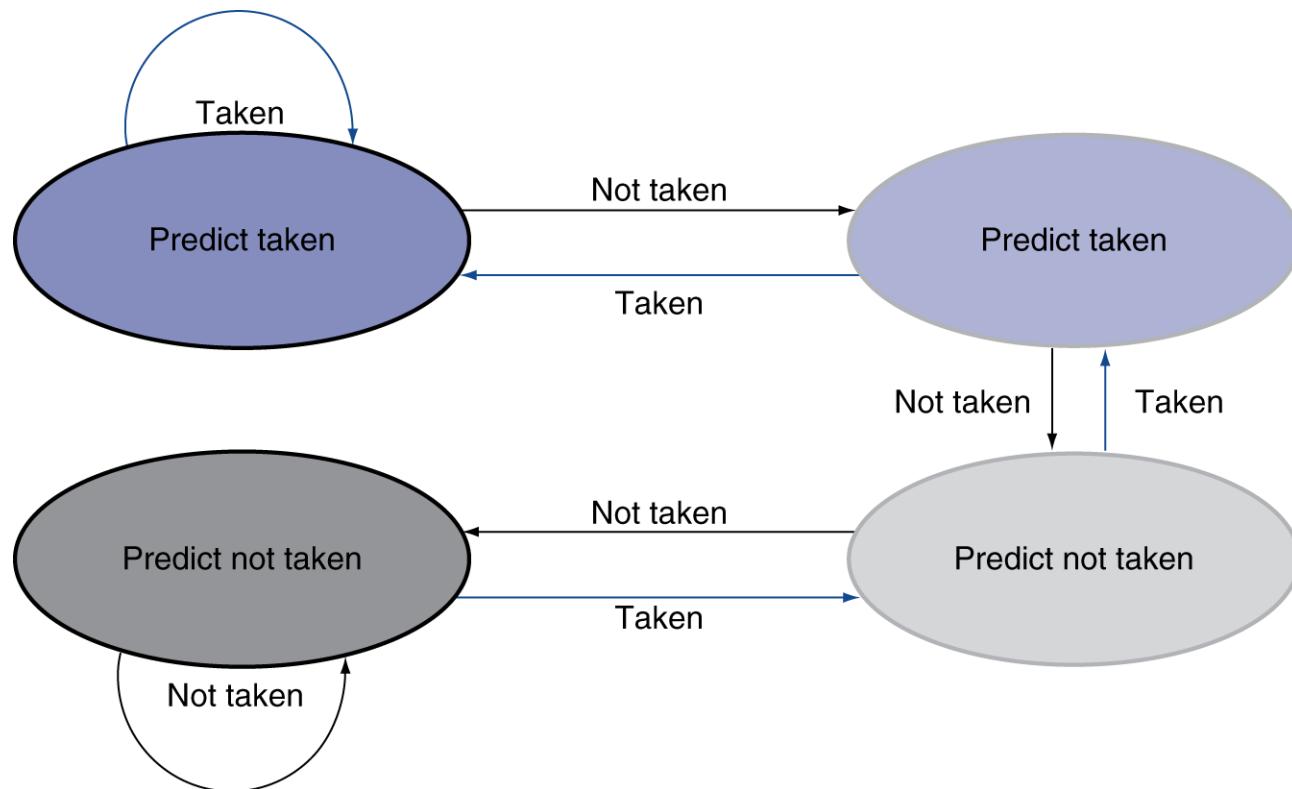
- Inner loop branches mispredicted twice!

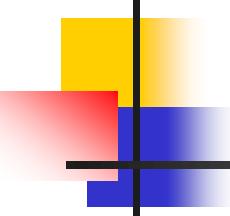


- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions

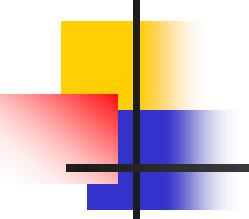




Calculating the Branch Target

- Even with predictor, still need to calculate the target address
 - 1-cycle penalty for a taken branch
- Branch target buffer
 - Cache of target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately

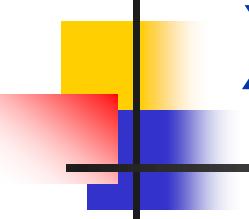




Ngoại lệ & Ngắt quãng

- Một sự kiện không mong đợi xảy ra làm cho thay đổi lộ trình thực hiện chương trình
 - ISA khác nhau sử dụng theo cách khác nhau
- Ngoại lệ
 - Xuất hiện khi CPU thực hiện
 - Ví dụ: mã lệnh sai, tràn, lệnh gọi ...
- Ngắt quãng
 - Bởi thiết bị ngoại vi
- Giải quyết mà không làm ảnh hưởng đến hiệu năng → vẫn đề khó

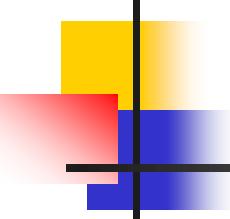




Xử lý ngoại lệ

- Trong MIP, ngoại lệ được quản lý bởi Bộ xử lý (kết hợp) điều khiển khiển hệ thống (CP0)
- Cắt PC của lệnh gây ra ngoại lệ (hoặc ngắt)
 - MIPS: Exception Program Counter (EPC)
- Cắt dấu hiệu vẫn để sinh ra ngoại lệ
 - MIPS: Thanh ghi nguyên nhân
 - Giả sử 1-bit
 - 0: opcode không tồn tại, 1: tràn
- Nhảy đến chương trình xử lý ngoại lệ: tại địa chỉ 8000 00180

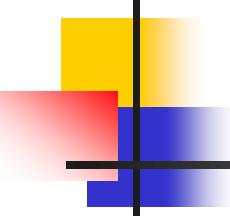




Phương thức xử lý ngoại lệ khác

- Bảng (Vectored Interrupts)
 - Địa chỉ mỗi phần tử bảng xác định lý do ngoại lệ
- Ví dụ:
 - Lệnh không tồn tại: C000 0000
 - Tràn: C000 0020
 - ...: C000 0040
- Lệnh xử lý ngoại lệ sẽ là
 - Giải quyết trực tiếp với ngắt
 - Hoặc nhảy đến c/trình xử lý

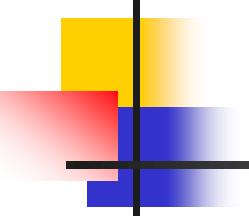




Công việc xử lý ngoại lệ

- Xác định nguyên nhân và chuyển đến c/trình xử lý tương ứng
- Xác định các việc phải giải quyết
- Nếu phải tiếp tục sau khi xử lý
 - Giải quyết vấn đề
 - Sử dụng EPC để trở về c/trình cũ
- Nếu không
 - Kết thúc c/trình
 - Báo lỗi, sử dụng EPC, nguyên nhân, etc....



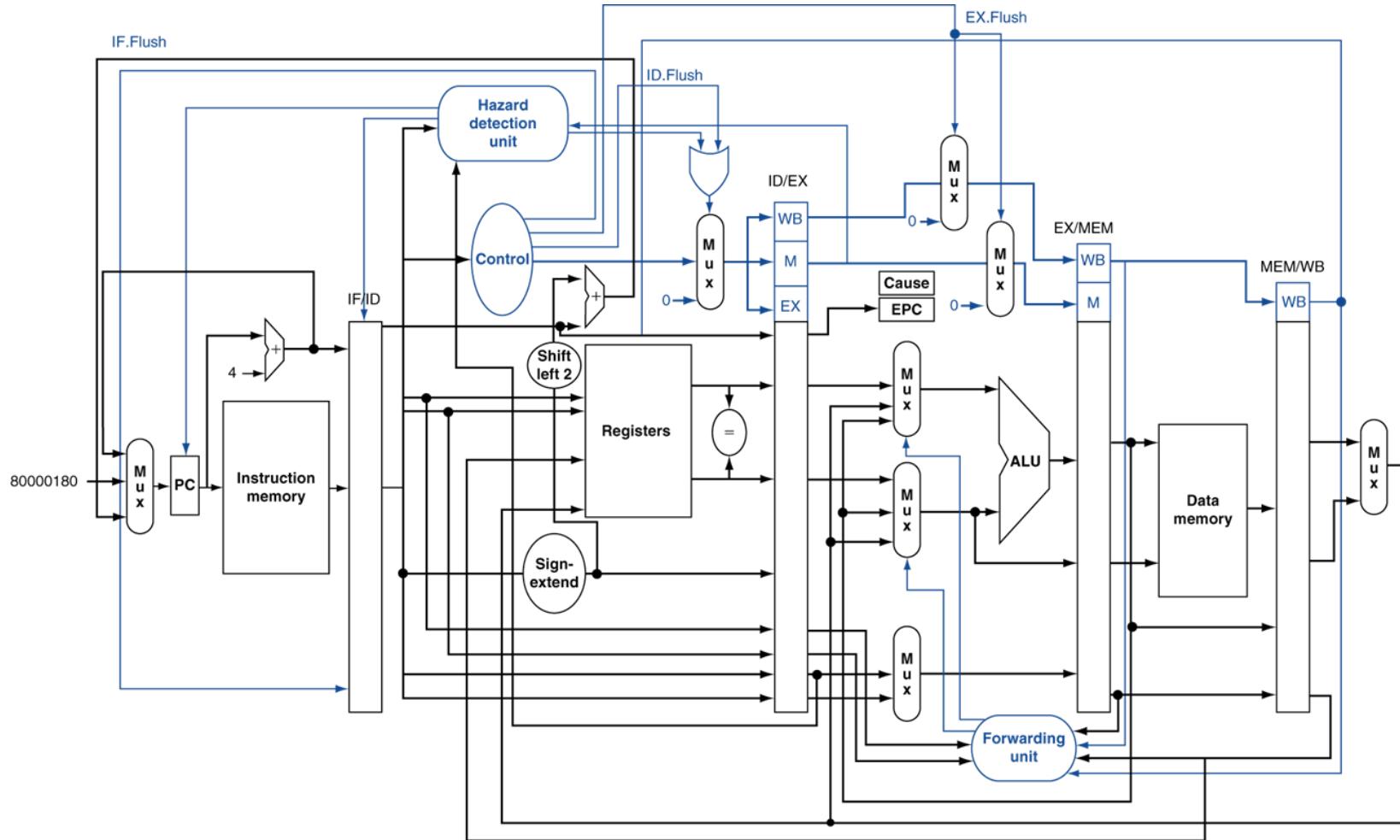


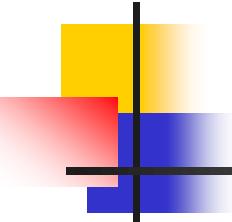
Ngoại lệ trong cơ chế ống

- Một dạng khác thuộc rủi ro điều khiển
- Giả sử tràn lệnh add trong bước EX
 - Tránh thay đổi giá trị \$1
 - Hoàn chỉnh lệnh trước đó
 - Xóa bỏ lệnh add và các lệnh sau
 - Gán nguyên nhân và giá trị t/ghi EPC
 - Chuyển điều khiển ch/trình xử lý tràn
- Tương tự cho việc rẽ nhánh với địa chỉ tiên đoán: sử dụng lại phần cứng



Cơ chế ống với ngoại lệ

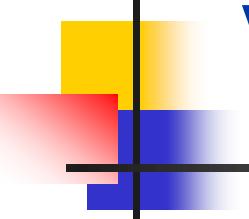




Exception Properties

- Restartable exceptions
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Refetched and executed from scratch
- PC saved in EPC register
 - Identifies causing instruction
 - Actually PC + 4 is saved
 - Handler must adjust





Ví dụ: ngoại lệ

- Ngoại lệ xảy ra tại lệnh **add** trong đoạn code:

```
40      sub    $11, $2, $4  
44      and    $12, $2, $5  
48      or     $13, $2, $6  
4C      add    $1, $2, $1  
50      slt    $15, $6, $7  
54      lw     $16, 50($7)
```

...

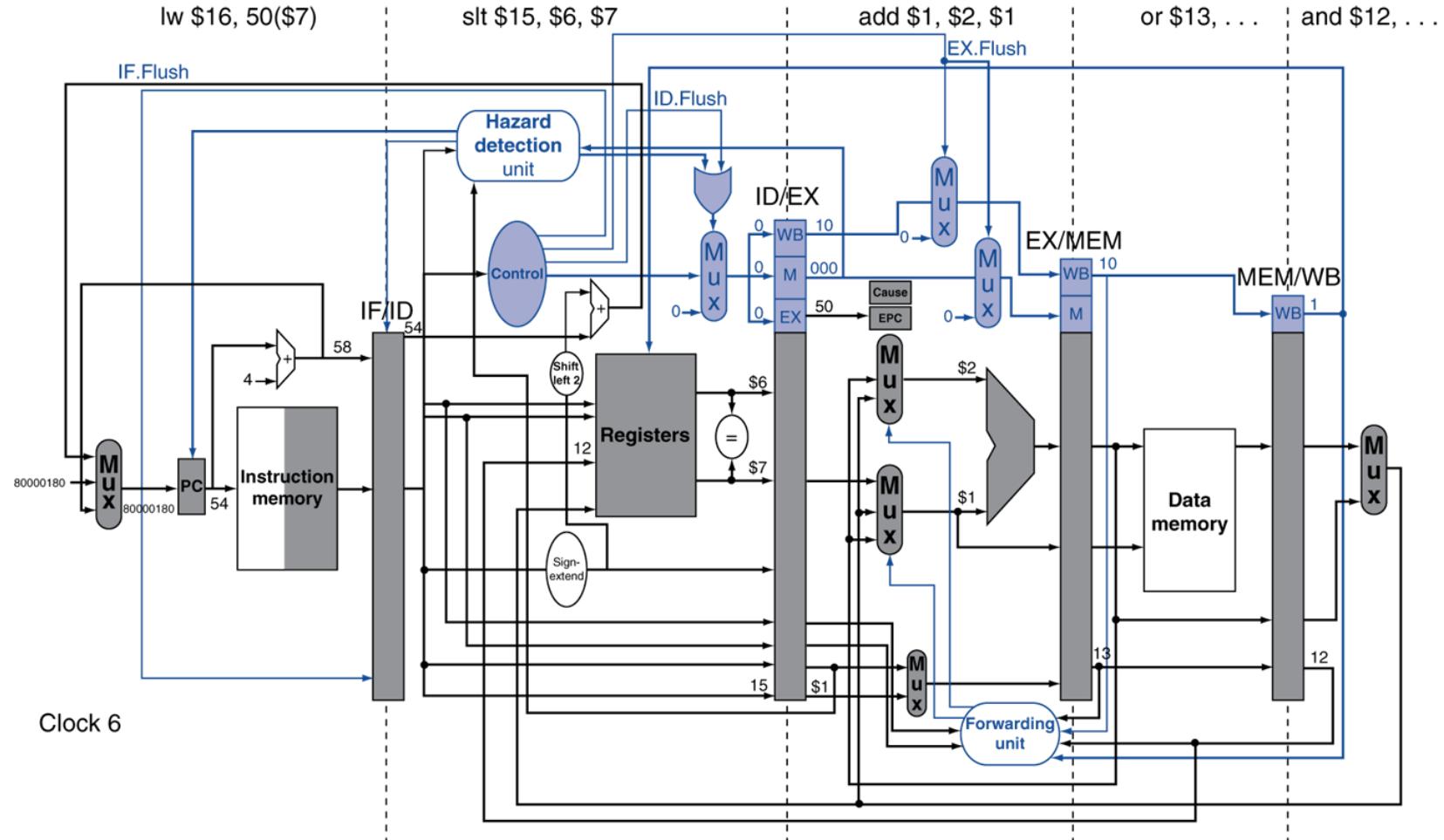
- Xử lý ngoại lệ

```
80000180    sw     $25, 1000($0)  
80000184    sw     $26, 1004($0)
```

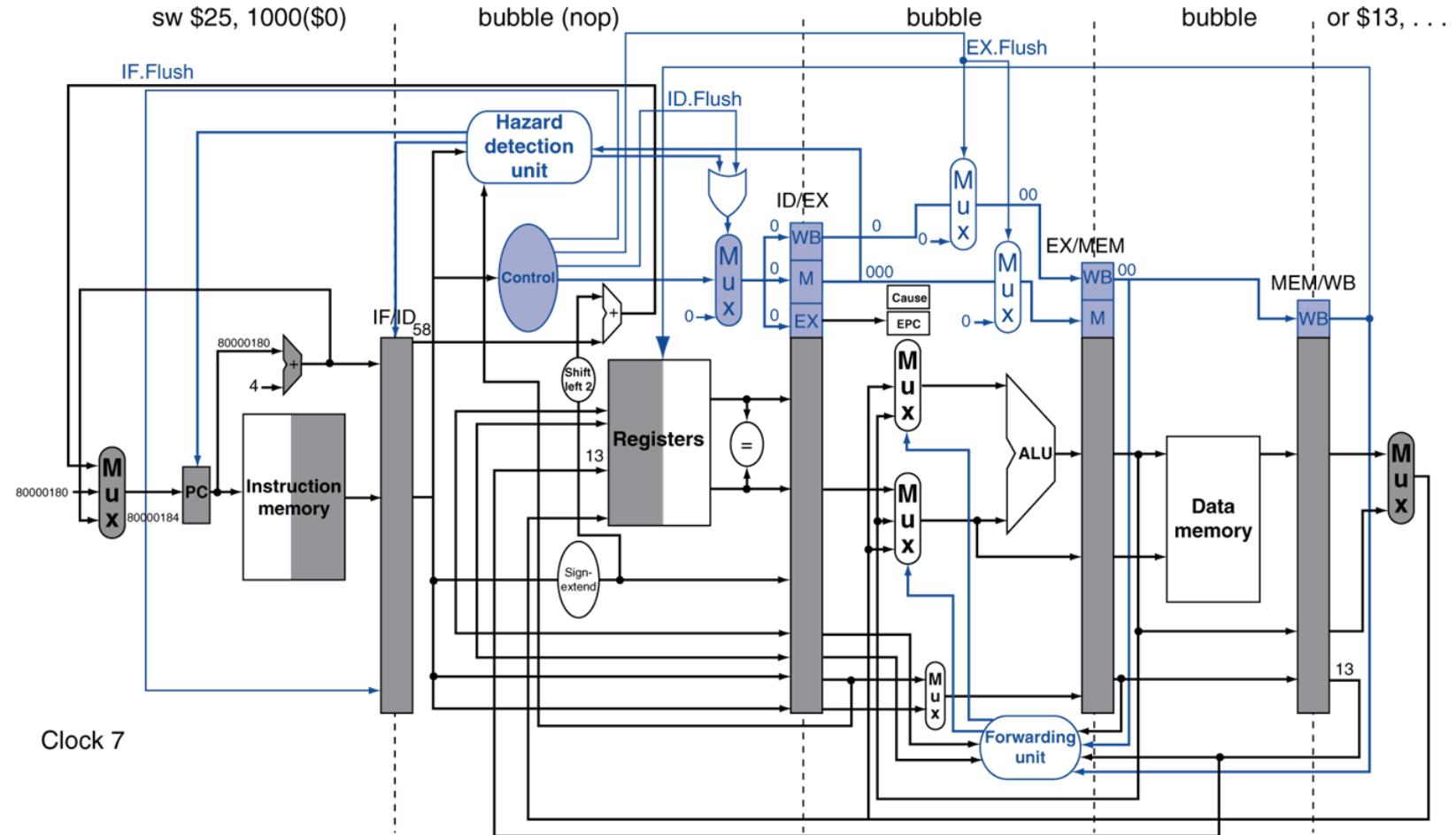
...



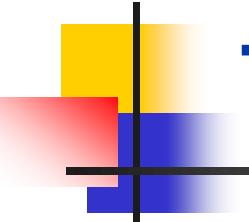
Ví dụ: Ngoại lệ



Ví dụ: Ngoại lệ (tt.)



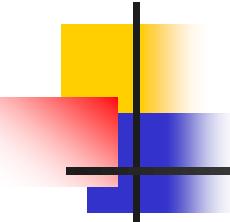
Clock 7



Đa ngoại lệ

- Nhiều lệnh thực thi phủ lấp nhau trong ống
 - Dẫn đến xuất hiện ngoại lệ cùng lúc
- Phương án đơn giản: Giải quyết ngoại lệ xảy ra đầu tiên
 - Xóa các lệnh kế tiếp
 - “Precise” exceptions
- Ống phức tạp
 - Nhiều lệnh trong cùng 1 chu kỳ
 - Không còn khả năng hoàn tất
 - Giải quyết ngoại lệ một cách chính xác: khó

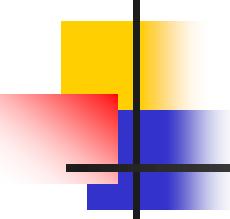




Imprecise Exceptions

- Just stop pipeline and save state
 - Including exception cause(s)
- Let the handler work out
 - Which instruction(s) had exceptions
 - Which to complete or flush
 - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

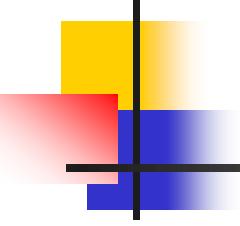




Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - Multiple issue
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - CPI < 1, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak CPI = 0.25, peak IPC = 4
 - But dependencies reduce this in practice

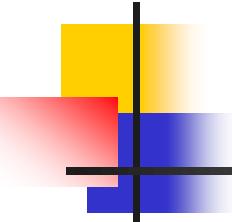




Multiple Issue

- Static multiple issue
 - Compiler groups instructions to be issued together
 - Packages them into “issue slots”
 - Compiler detects and avoids hazards
- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime

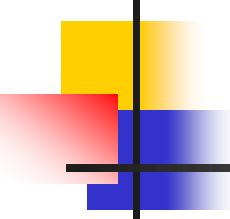




Speculation

- “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load
 - Roll back if location is updated

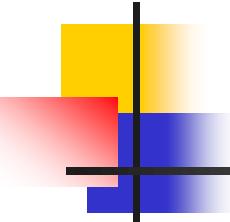




Compiler/Hardware Speculation

- Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation

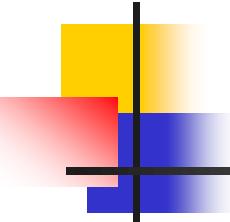




Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
 - e.g., speculative load before null-pointer check
- Static speculation
 - Can add ISA support for deferring exceptions
- Dynamic speculation
 - Can buffer exceptions until instruction completion (which may not occur)

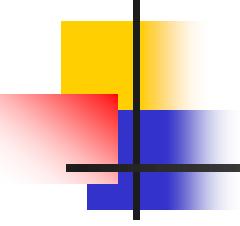




Static Multiple Issue

- Compiler groups instructions into “issue packets”
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
 - ⇒ Very Long Instruction Word (VLIW)





Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies with a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary

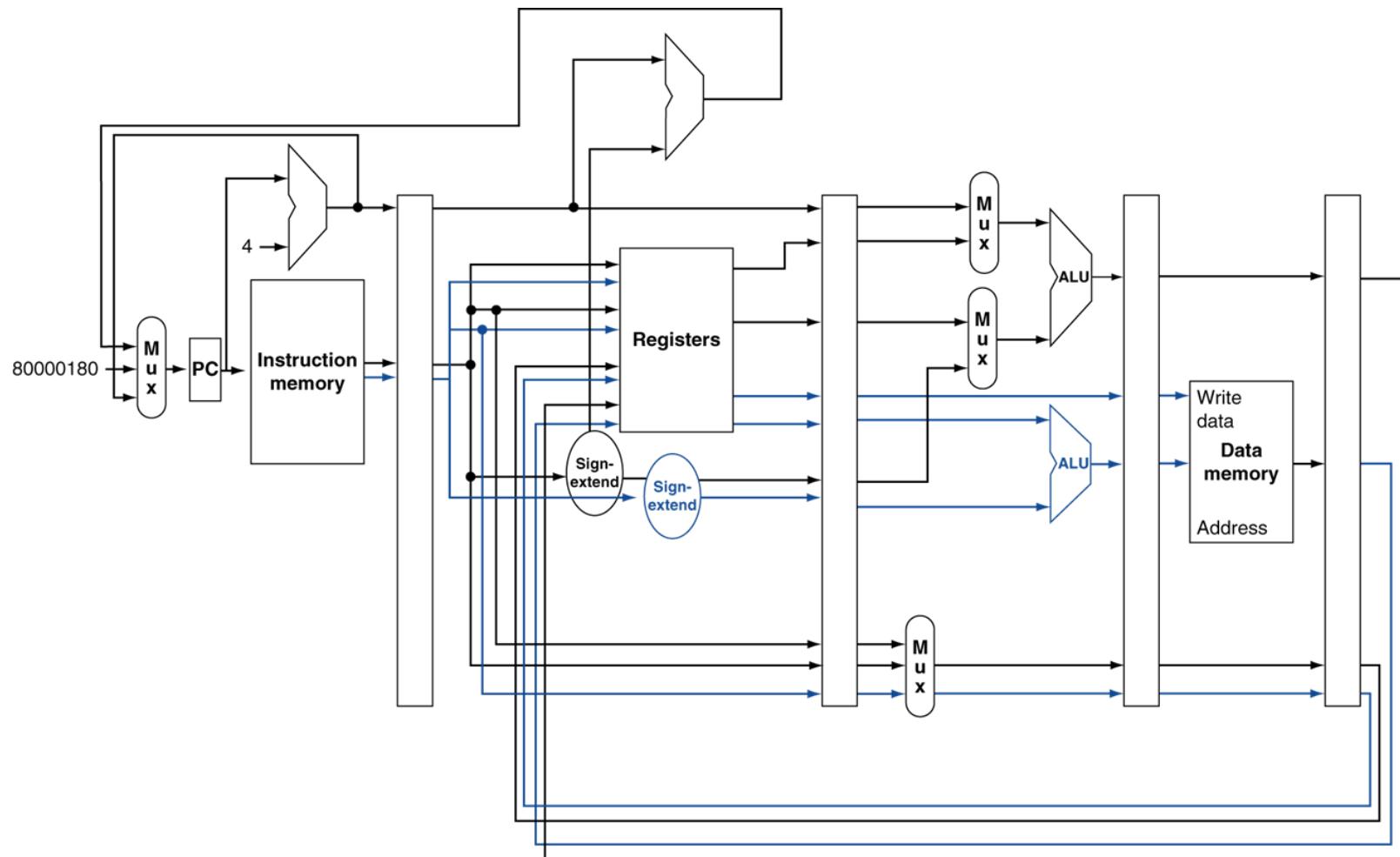


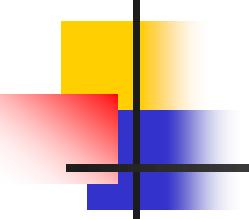
MIPS with Static Dual Issue

- Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

| Address | Instruction type | Pipeline Stages | | | | | | |
|---------|------------------|-----------------|----|----|-----|-----|-----|-----|
| | | IF | ID | EX | MEM | WB | | |
| n | ALU/branch | | | | | | | |
| n + 4 | Load/store | IF | ID | EX | MEM | WB | | |
| n + 8 | ALU/branch | | IF | ID | EX | MEM | WB | |
| n + 12 | Load/store | | | IF | ID | EX | MEM | WB |
| n + 16 | ALU/branch | | | | IF | ID | EX | MEM |
| n + 20 | Load/store | | | | IF | ID | EX | MEM |

MIPS with Static Dual Issue





Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add \$t0, \$s0, \$s1
 - Load \$s2, 0(\$t0)
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required



Scheduling Example

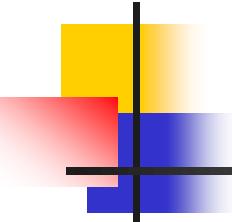
- Schedule this for dual-issue MIPS

```
Loop: lw    $t0, 0($s1)      # $t0=array element
       addu $t0, $t0, $s2      # add scalar in $s2
       sw    $t0, 0($s1)      # store result
       addi $s1, $s1,-4        # decrement pointer
       bne  $s1, $zero, Loop   # branch $s1!=0
```

| | ALU/branch | Load/store | cycle |
|-------|-------------------------|---------------------|-------|
| Loop: | nop | lw \$t0, 0(\$s1) | 1 |
| | addi \$s1, \$s1,-4 | nop | 2 |
| | addu \$t0, \$t0, \$s2 | nop | 3 |
| | bne \$s1, \$zero, Loop | sw \$t0, 4(\$s1) | 4 |

- $IPC = 5/4 = 1.25$ (c.f. peak IPC = 2)





Loop Unrolling

- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - Reuse of a register name

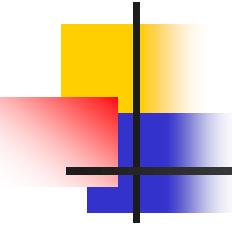


Loop Unrolling Example

| | ALU/branch | Load/store | cycle |
|-------|------------------------|-------------------|-------|
| Loop: | addi \$s1, \$s1,-16 | lw \$t0, 0(\$s1) | 1 |
| | nop | lw \$t1, 12(\$s1) | 2 |
| | addu \$t0, \$t0, \$s2 | lw \$t2, 8(\$s1) | 3 |
| | addu \$t1, \$t1, \$s2 | lw \$t3, 4(\$s1) | 4 |
| | addu \$t2, \$t2, \$s2 | sw \$t0, 16(\$s1) | 5 |
| | addu \$t3, \$t4, \$s2 | sw \$t1, 12(\$s1) | 6 |
| | nop | sw \$t2, 8(\$s1) | 7 |
| | bne \$s1, \$zero, Loop | sw \$t3, 4(\$s1) | 8 |

- $\text{IPC} = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

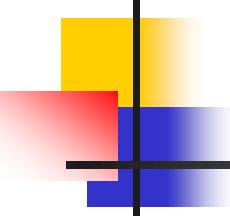




Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU





Dynamic Pipeline Scheduling

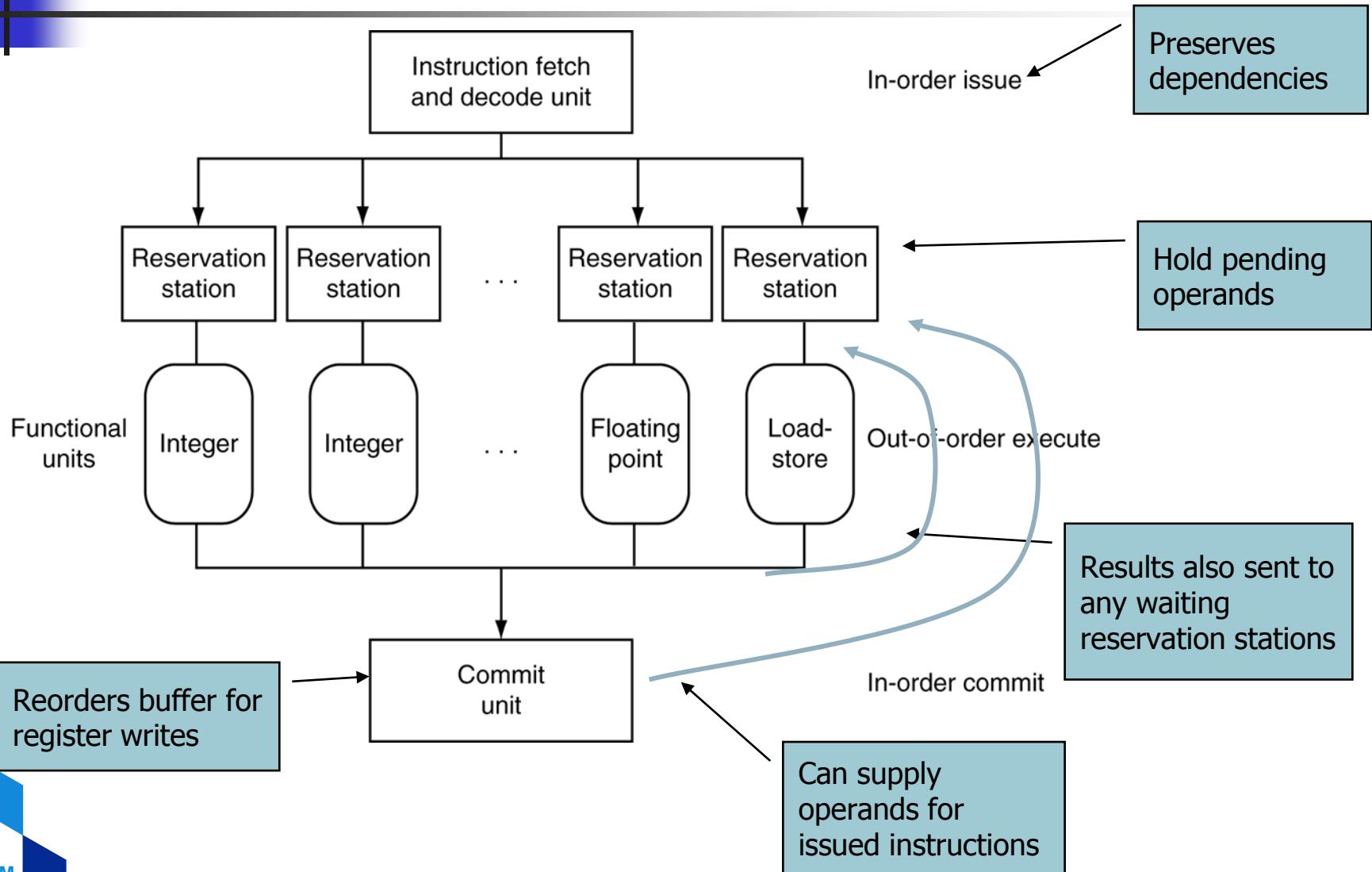
- Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order
- Example

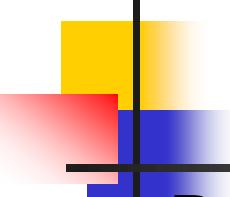
| | | | |
|------|-------|----------|------|
| lw | \$t0, | 20(\$s2) | |
| addu | \$t1, | \$t0, | \$t2 |
| sub | \$s4, | \$s4, | \$t3 |
| slti | \$t5, | \$s4, | 20 |

- Can start sub while addu is waiting for lw



Dynamically Scheduled CPU

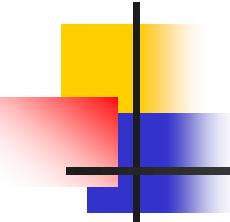




Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
 - If operand is available in register file or reorder buffer
 - Copied to reservation station
 - No longer required in the register; can be overwritten
 - If operand is not yet available
 - It will be provided to the reservation station by a function unit
 - Register update may not be required

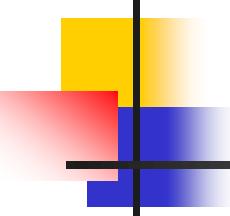




Speculation

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - Avoid load and cache miss delay
 - Predict the effective address
 - Predict loaded value
 - Load before completing outstanding stores
 - Bypass stored values to load unit
 - Don't commit load until speculation cleared

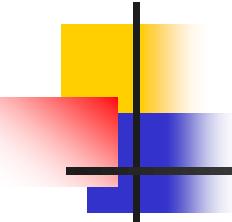




Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
 - e.g., cache misses
- Can't always schedule around branches
 - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards





Does Multiple Issue Work?

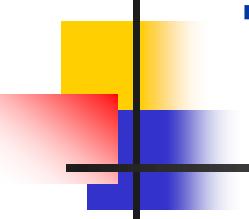
- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
 - e.g., pointer aliasing
- Some parallelism is hard to expose
 - Limited window size during instruction issue
- Memory delays and limited bandwidth
 - Hard to keep pipelines full
- Speculation can help if done well



Tiết kiệm năng lượng

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

| Microprocessor | Year | Clock Rate | Pipeline Stages | Issue width | Out-of-order/Speculation | Cores | Power |
|----------------|------|------------|-----------------|-------------|--------------------------|-------|-------|
| i486 | 1989 | 25MHz | 5 | 1 | No | 1 | 5W |
| Pentium | 1993 | 66MHz | 5 | 2 | No | 1 | 10W |
| Pentium Pro | 1997 | 200MHz | 10 | 3 | Yes | 1 | 29W |
| P4 Willamette | 2001 | 2000MHz | 22 | 3 | Yes | 1 | 75W |
| P4 Prescott | 2004 | 3600MHz | 31 | 3 | Yes | 1 | 103W |
| Core | 2006 | 2930MHz | 14 | 4 | Yes | 2 | 75W |
| UltraSparc III | 2003 | 1950MHz | 14 | 4 | No | 1 | 90W |
| UltraSparc T1 | 2005 | 1200MHz | 6 | 1 | No | 8 | 70W |



Tổng kết

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
 - More instructions completed per second
 - Latency for each instruction not reduced
- Rủi ro: cấu trúc, dữ liệu, điều khiển
- Multiple issue and dynamic scheduling (ILP)
 - Dependencies limit achievable parallelism
 - Complexity leads to the power wall

