COMPUTER ARCHITECTURE
CSE 2015

Faculty of Computer Science and Engineering

Department of Computer Engineering

Vo Tan Phuong

http://www.cse.hcmut.edu.vn/~vtphuong

Chapter 4.2

Thiết kế bộ xử lý đường ống (Pipelined Processor Design)

Nội dung

- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

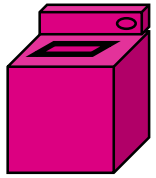
Ví dụ cơ chế đường ống

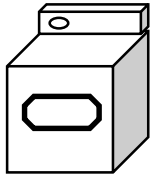
- Dịch vụ giặt đồ: 3 bước

1. Giặt

2. Sấy

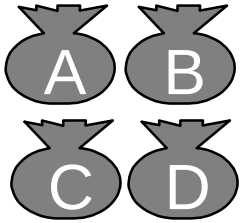
3. Gấp



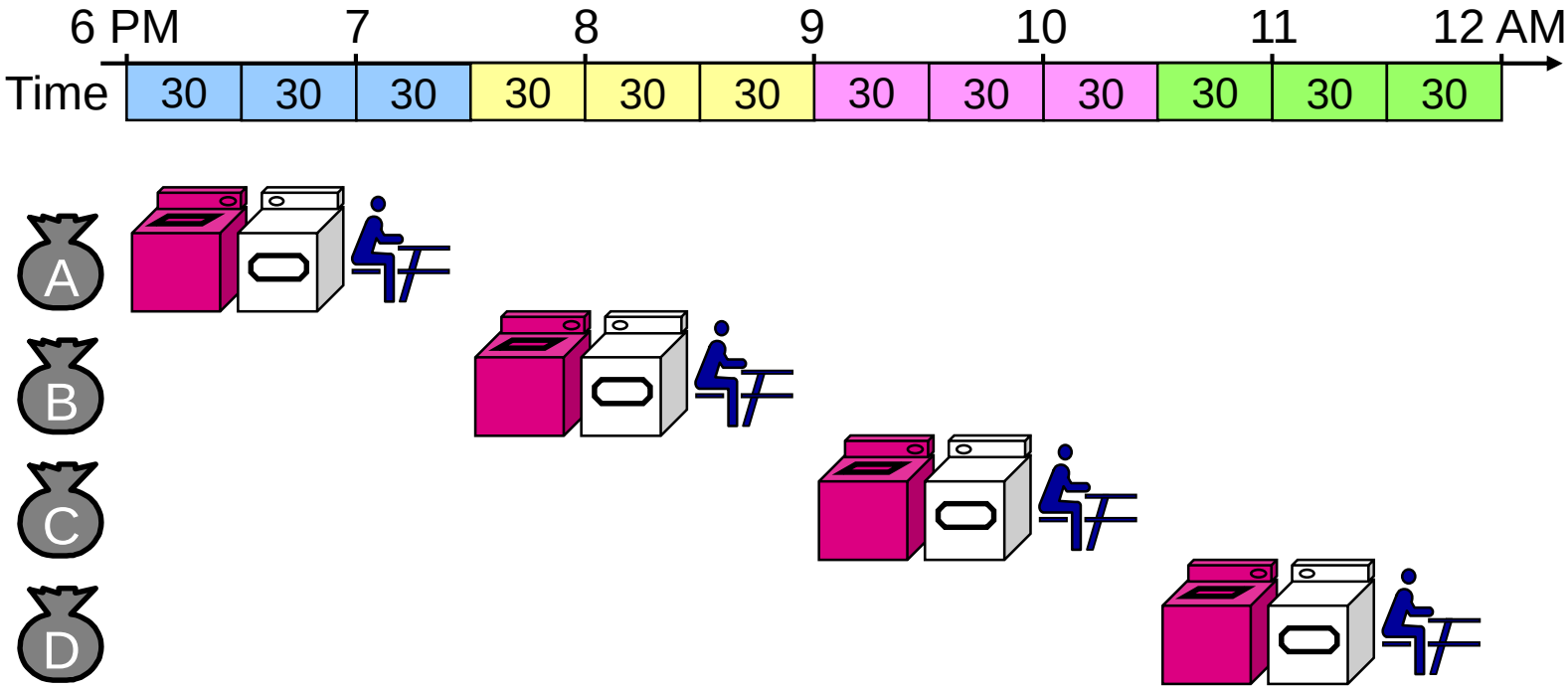


- Mỗi bước thực hiện trong 30 phút

- Có 4 mẻ

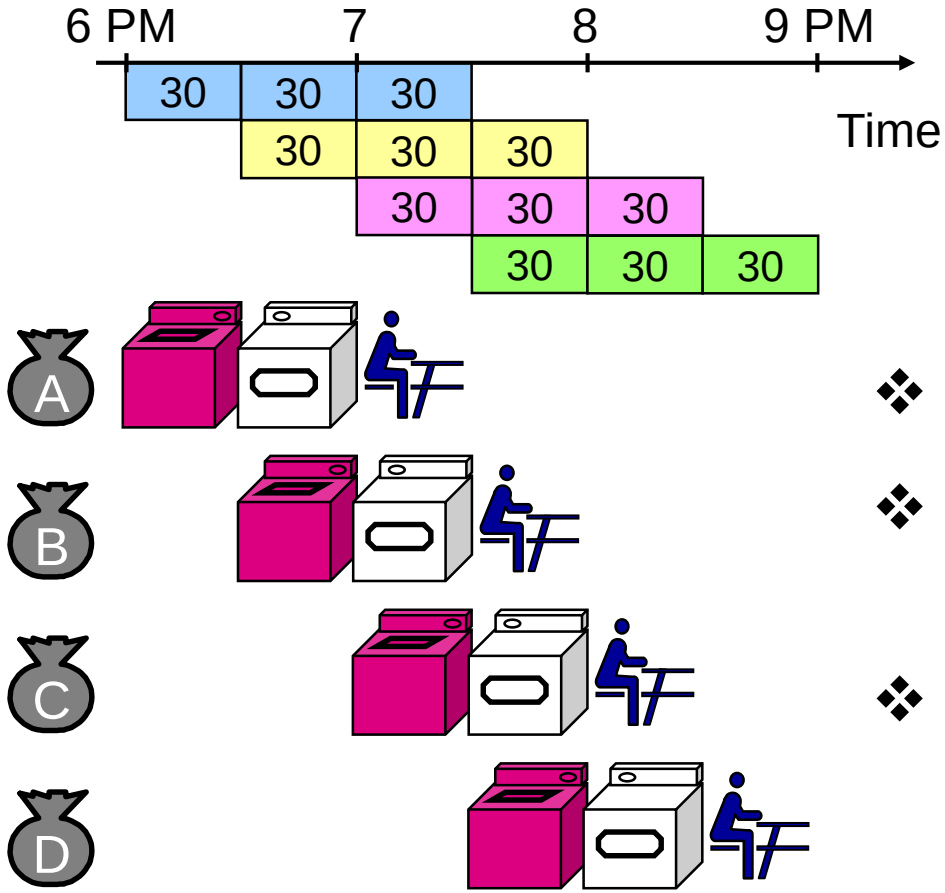


Phương pháp tuần tự



- Cần 6 tiếng để hoàn thành 4 mẻ
- Dễ thấy cách làm này còn có thể được cải thiện

Áp dụng cơ chế đường ống



- Cần 3 tiếng cho 4 mẻ
- Hiệu quả hơn 2 lần cho 4 mẻ
- Thời gian xử lý một mẻ không đổi (90 phút)

Hiệu suất của cơ chế đường ống

- Mỗi công việc cần k công đoạn
- Với τ_i = thời gian của mỗi công đoạn S_i
- Chu kỳ xung nhịp $\tau = \max(\tau_i)$ là thời gian của công đoạn dài nhất
- Tần số xung nhịp $f = 1/\tau = 1/\max(\tau_i)$
- Thời gian xử lý n công việc = $(k + n - 1)*\tau$
  - k chu kỳ để hoàn thành công việc đầu tiên
  - $n - 1$ chu kỳ còn lại hoàn thành $n - 1$ công việc
- Speed up trong trường hợp lý tưởng

$$S_k = \frac{\text{Số chu kỳ cho cách tuần tự}}{\text{Số chu lỳ cho cách pipeline}} = \frac{nk}{k + n - 1} \qquad S_k \rightarrow k \text{ khi } n \text{ lớn}$$

Bộ xử lý MIPS theo cơ chế Pipeline

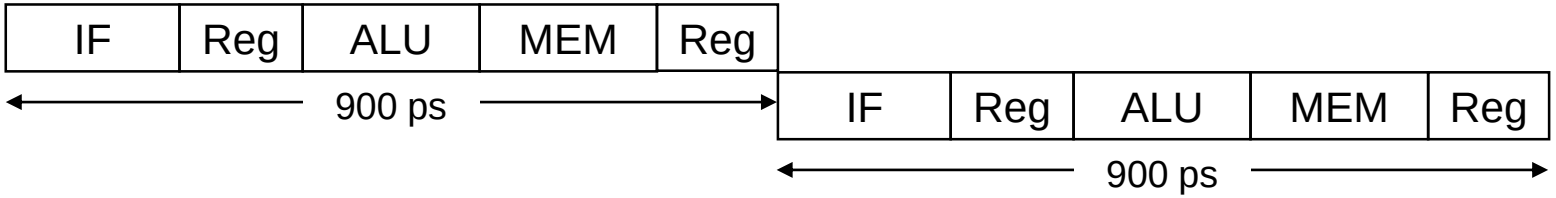
- Gồm 5 công đoạn, mỗi công đoạn là một chu kỳ

1. IF: Instruction Fetch (nạp lệnh)
2. ID: Instruction Decode (giải mã lệnh)
3. EX: Execute (thực thi phép toán)
4. MEM: Memory access (truy xuất bộ nhớ dữ liệu)
5. WB: Write Back (ghi kết quả vào thanh ghi đích)

So sánh Single-Cycle với Pipelined

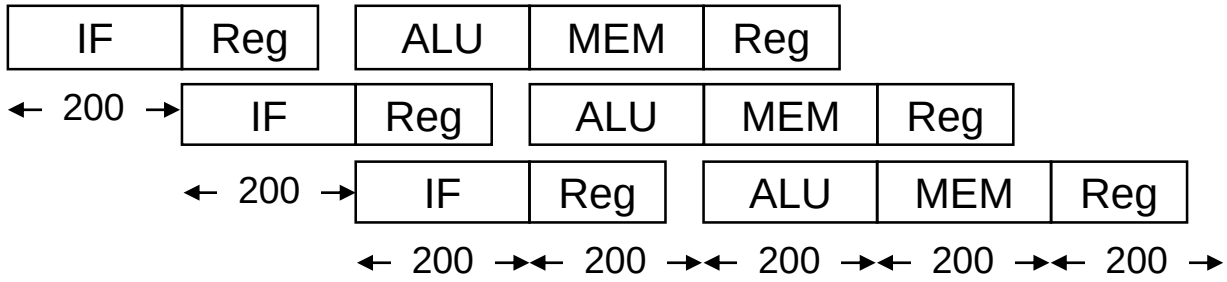
- Giả sử 5 công đoạn trong quá trình thực thi lệnh có thời gian như sau:
  - Nạp lệnh (IF) = ALU thực thi (ALU) = truy xuất bộ nhớ dữ liệu (MEM) = 200 ps
  - Đọc thanh ghi (RegR) = ghi thanh ghi (RegW) = 150 ps
- Tính chu kỳ của bộ xử lý đơn chu kỳ (Ts)?
- Tính chu kỳ của bộ xử lý đơn đường ống (Tp)?
- Tính speedup?
- Lời giải:

Ts = 200+150+200+200+150 = 900 ps



So sánh (tiếp theo)

- Tp = max(200, 150) = 200 ps



- CPI cho bộ xử lý pipeline = 1
  - Xét trường hợp số lượng lệnh lớn
- Speedup của bộ xử lý pipeline = 900 ps / 200 ps = 4.5
  - IC và CPI bằng nhau cho cả hai trường hợp
- Speedup nhỏ hơn 5 (số công đoạn)
  - Do thời gian các công đoạn không cân bằng

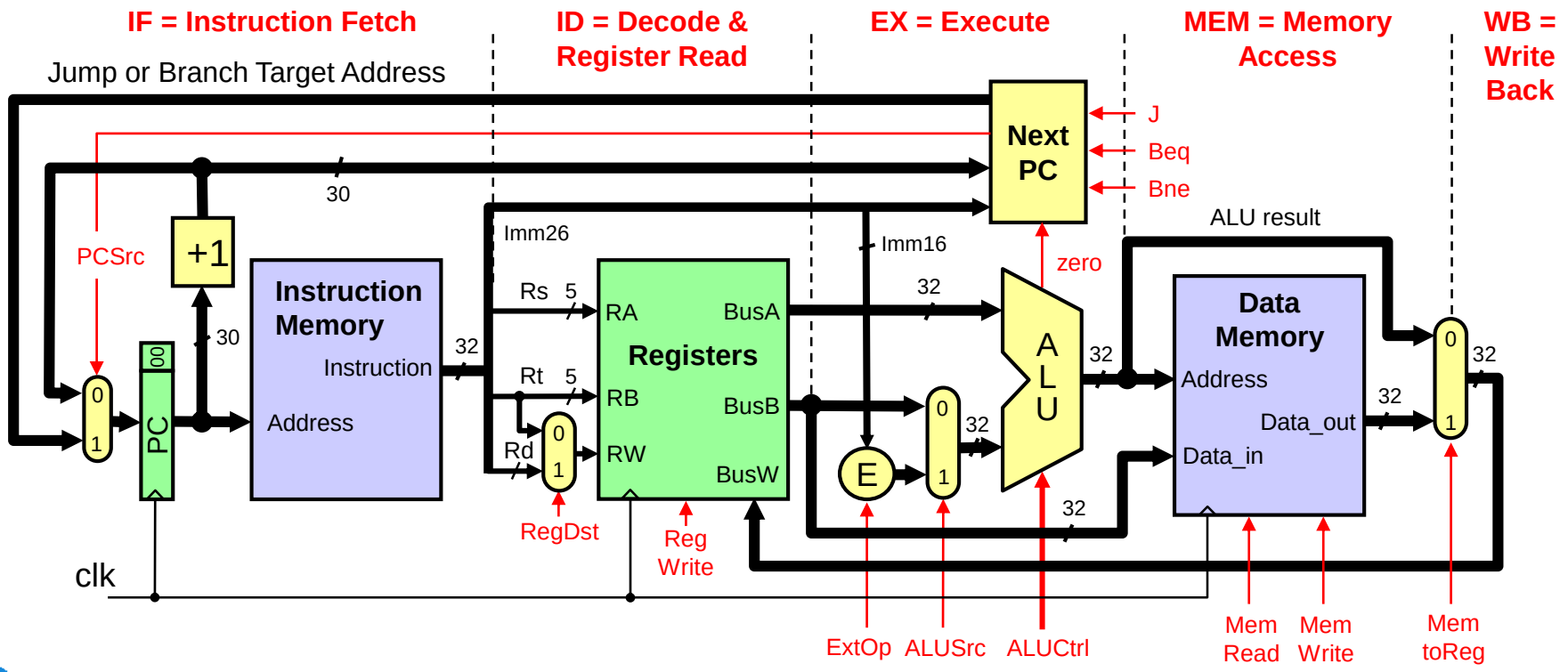
Tiếp theo…

- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

Datapath đơn chu kỳ

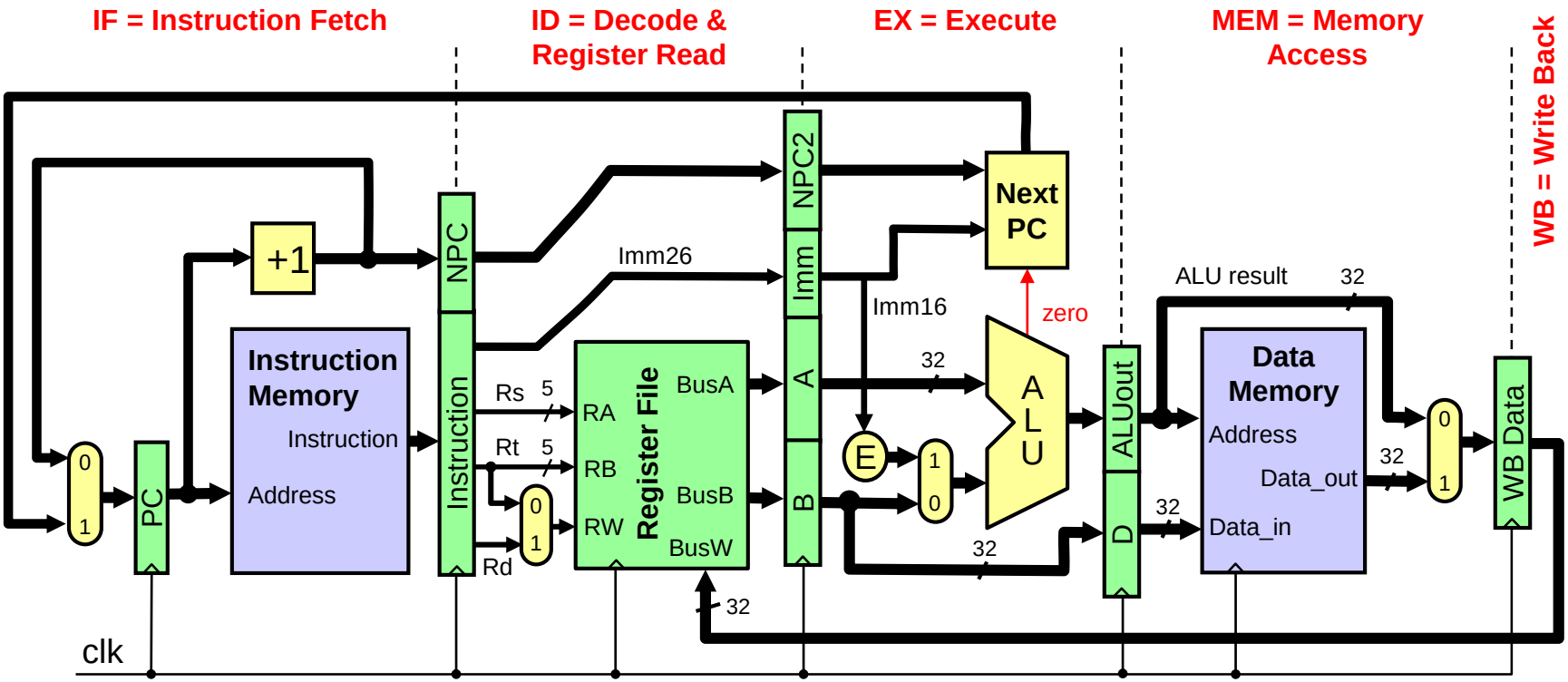
- Hình bên dưới là datapath của bộ xử lý đơn chu kỳ
- Thay đổi datapath như thế nào để trở thành pipelined?

  Trả lời: Thêm thanh ghi vào cuối những công đoạn



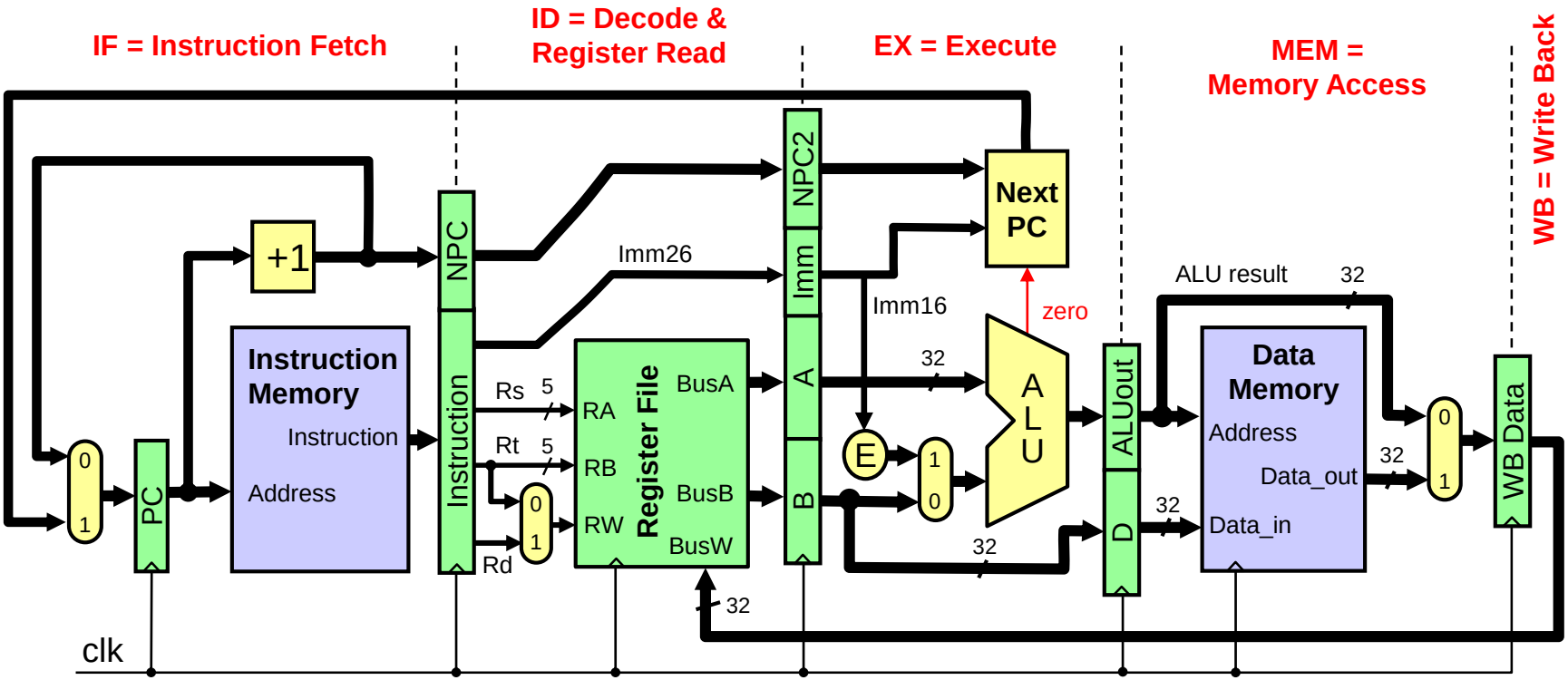
Datapath đường ống

- Các thanh ghi thêm vào có màu xanh, bao gồm PC
- Cùng chung tín hiệu xung nhịp với bộ thanh ghi và bộ nhớ dữ liệu (cho lệnh store)



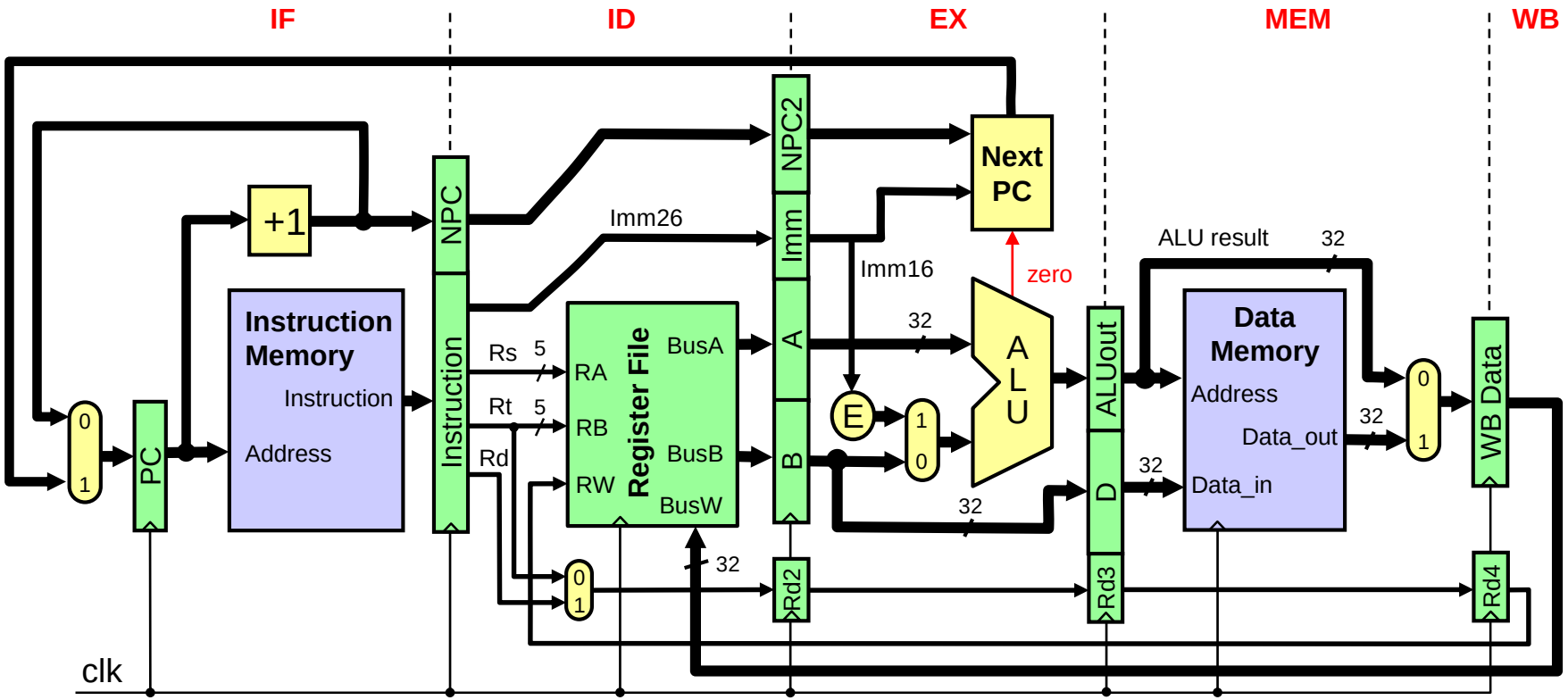
Vấn đề với thanh ghi đích (RW)

- Có vấn đề với địa chỉ thanh ghi đích?
  - Các giá trị RA, RB, RW trong công đoạn ID khác với công đoạn WB
  - Địa chỉ thanh ghi đích RW khi WB không phải là giá trị đúng cần ghi vào



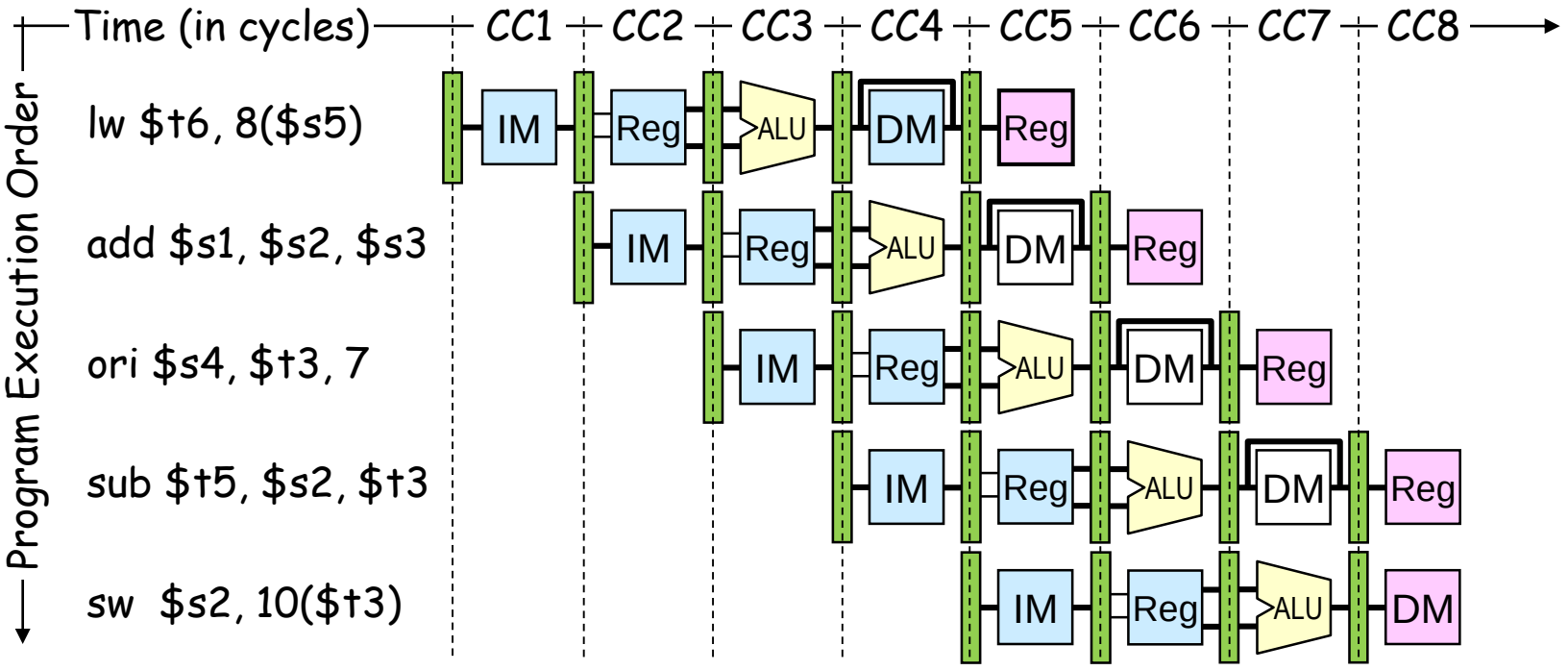
Thêm đường ống cho địa chỉ thanh ghi đích

- Địa chỉ thanh ghi đích cần thêm vào đường ống
  - Địa chỉ thanh ghi đích truyền từ công đoạn ID sang WB
  - Lúc này công đoạn WB có giá trị đúng của địa chỉ thanh ghi đích



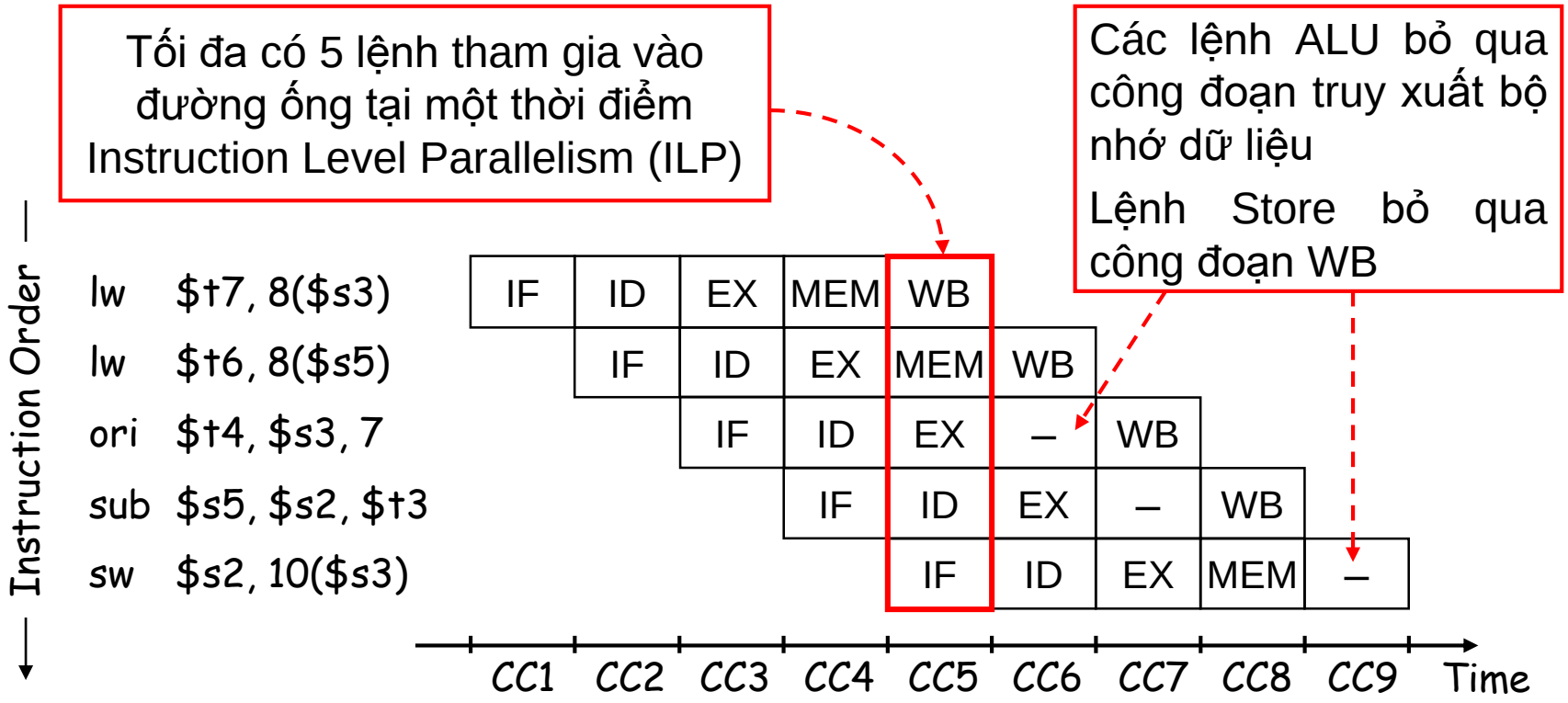
Giảng đồ thực thi lệnh theo đường ống

- Diễn tả quá trình thực thi lệnh theo chu kỳ xung nhịp
  - Lệnh được liệt kê theo thứ tự từ trên xuống
  - Xung nhịp theo thứ tự từ trái qua phải
  - Mô tả chi tiết công đoạn của từng lệnh, tài nguyên sử dụng theo chu kỳ xung nhịp

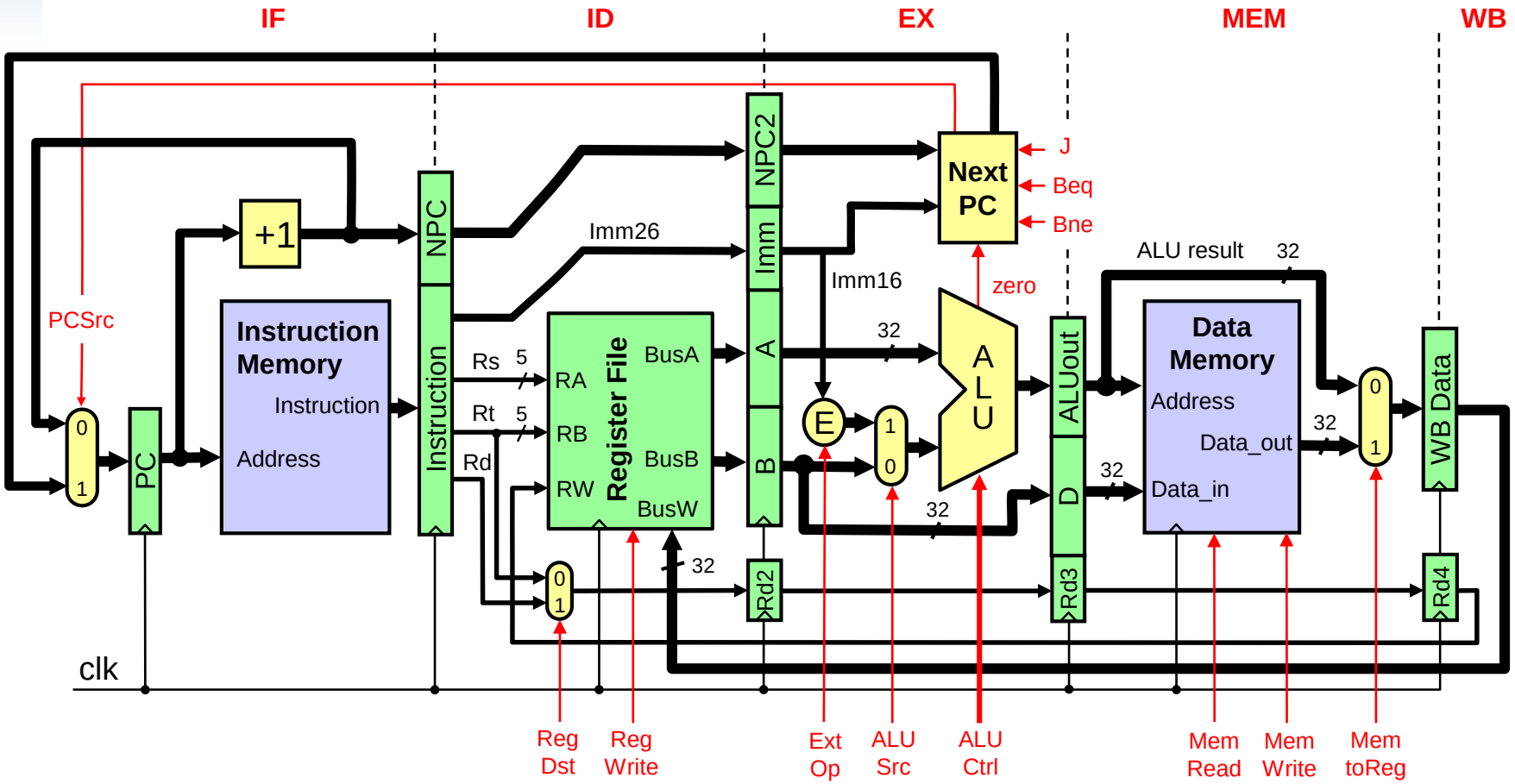


Giảng đồ tóm lược

- Giảng đồ tóm lược mô tả:
  - Công đoạn thực thi của từng lệnh theo chu kỳ xung nhịp
- Các lệnh đi qua đường ống gồm 5 trạng thái

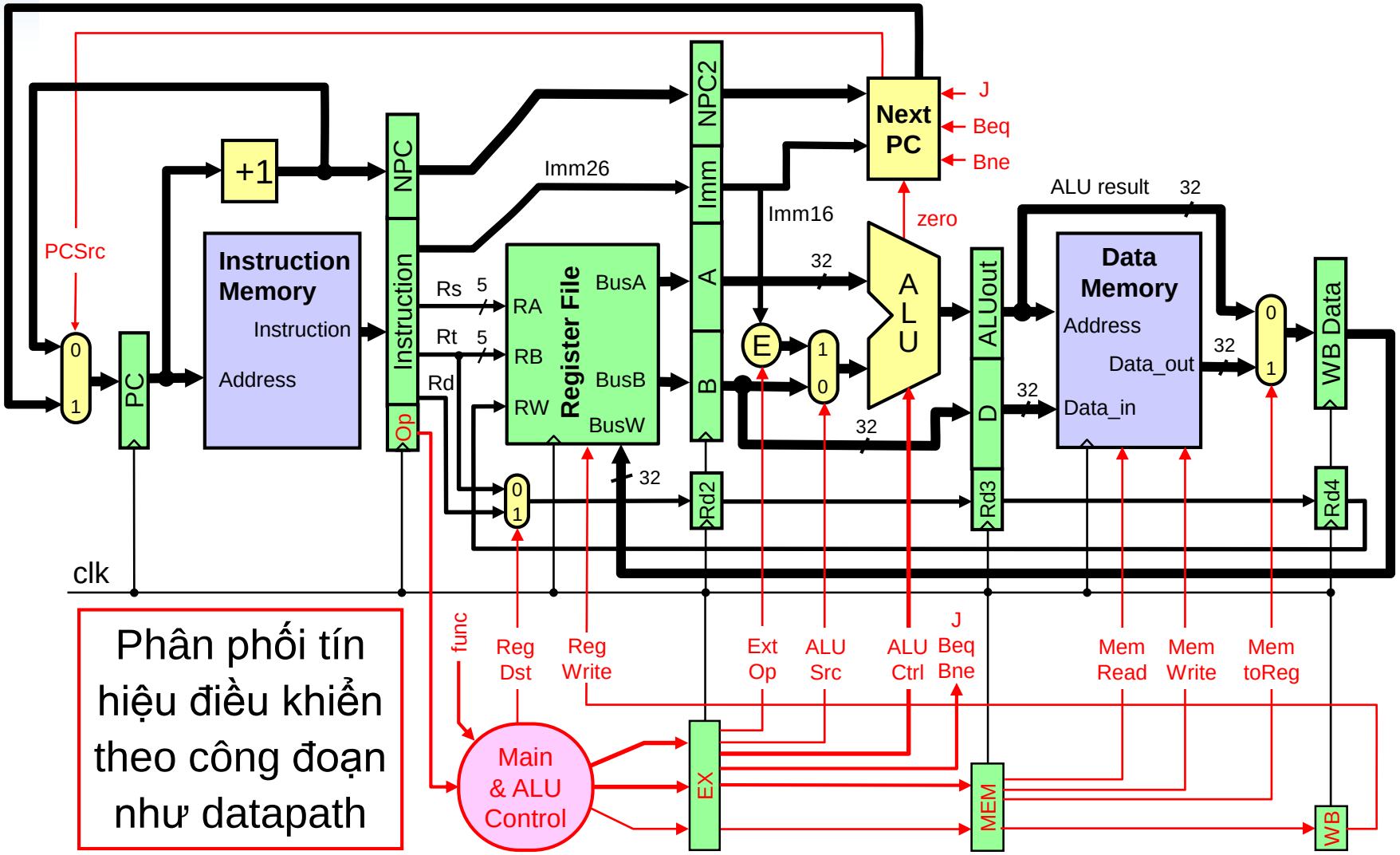


Tín hiệu điều khiển



Các tín hiệu điều khiển giống như bộ xử lý đơn chu kỳ

Điều khiển theo kiểu đường ống



Tiếp theo…

- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

Các rủi ro (hazards) với thiết kế pipeline

- Hazards: Tình huống làm quá trình thực thi không đúng theo thứ tự pipeline bình thường

1. Rủi ro về mặt cấu trúc
   - Tài nguyên (khối chức năng) được dùng bởi hai lệnh cùng một thời điểm
2. Rủi ro về mặt dữ liệu
   - Một lệnh tính ra kết quả được sử dụng bởi các lệnh kế
   - Phần cứng có thể phát hiện sự phụ thuộc này giữa các lệnh
3. Rủi ro về mặt điều khiển
   - Sinh ra bởi lệnh thay đổi dòng thực thi (nhảy/rẽ nhánh)

- Rủi ro làm thiết kế pipeline thêm phức tạp và giảm hiệu suất.

Rủi ro về mặt điều khiển

Vấn đề

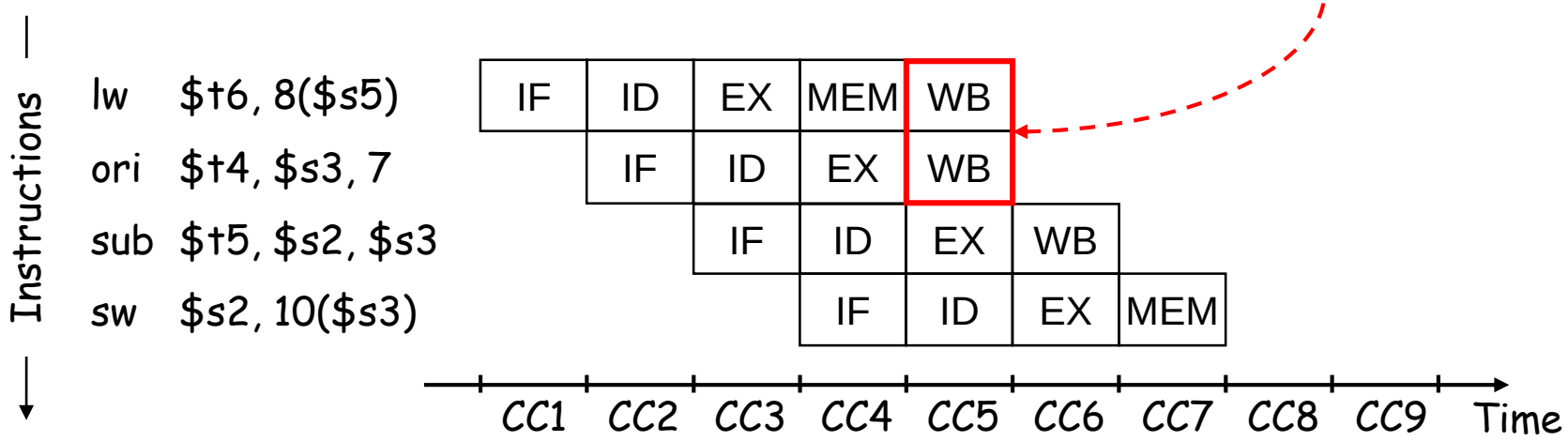
- Tranh chấp sử dụng một khối chức năng bởi hai lệnh khác nhau vào cùng một chu kỳ xung nhịp

Ví dụ

- Ghi lại kết quả ALU trong công đoạn 4
- Đụng độ với ghi kết quả ô nhớ trong công đoạn 5

Structural Hazard

Hai lệnh cùng ghi kết quả vào thanh bộ thanh ghi



Tiếp theo…

- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

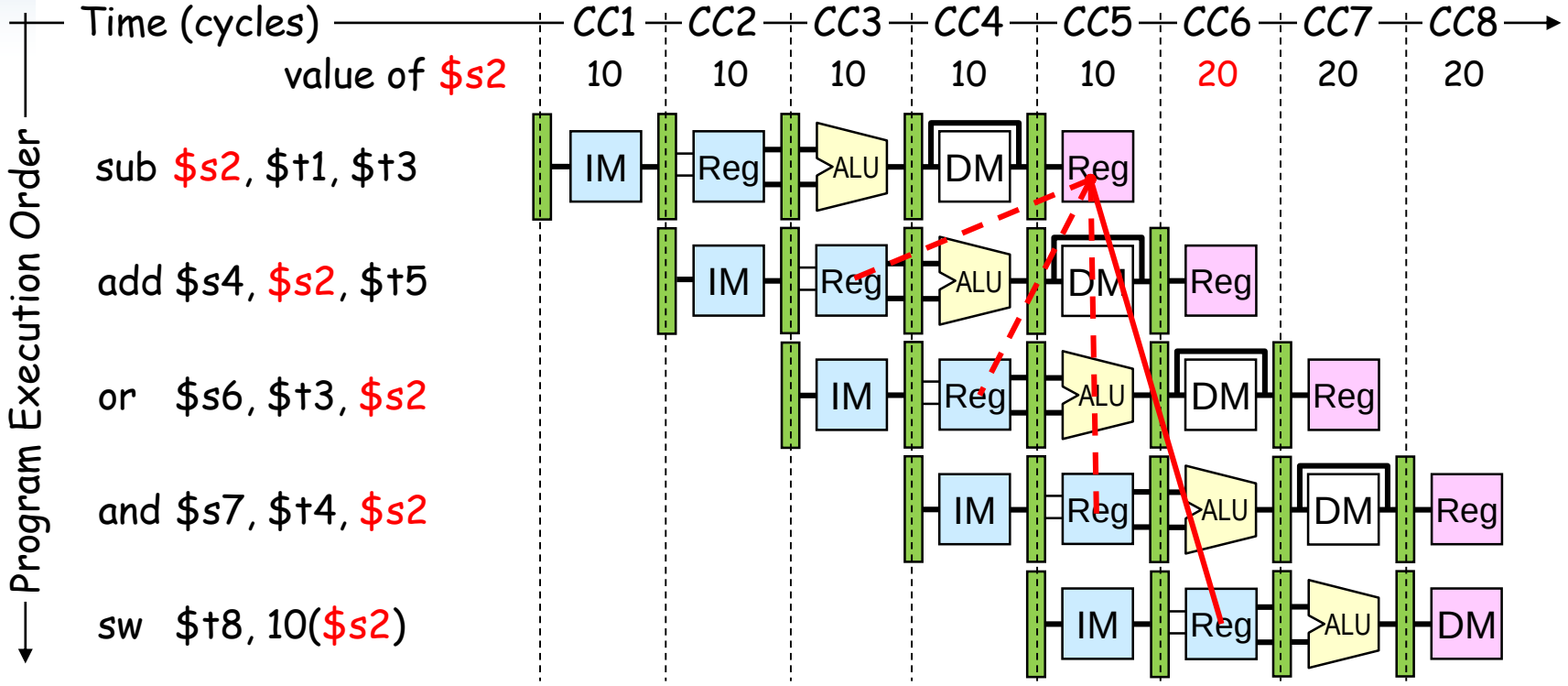
Rủi ro về mặt dữ liệu

- Phụ thuộc dữ liệu giữa các lệnh tạo ra rủi ro loại này
- Các lệnh phụ thuộc dữ liệu nằm kề với nhau
- Read After Write – Rủi ro RAW
  - Cho 2 lệnh *I* và *J*, với *I* được thực thi trước lệnh *J*
  - Lệnh *J* cần (đọc) toán hạn được ghi bởi lệnh *I*
  - Gọi là một phụ thuộc dữ liệu

```
I: add $s1, $s2, $s3     # $s1 is written
J: sub $s4, $s1, $s3     # $s1 is read
```

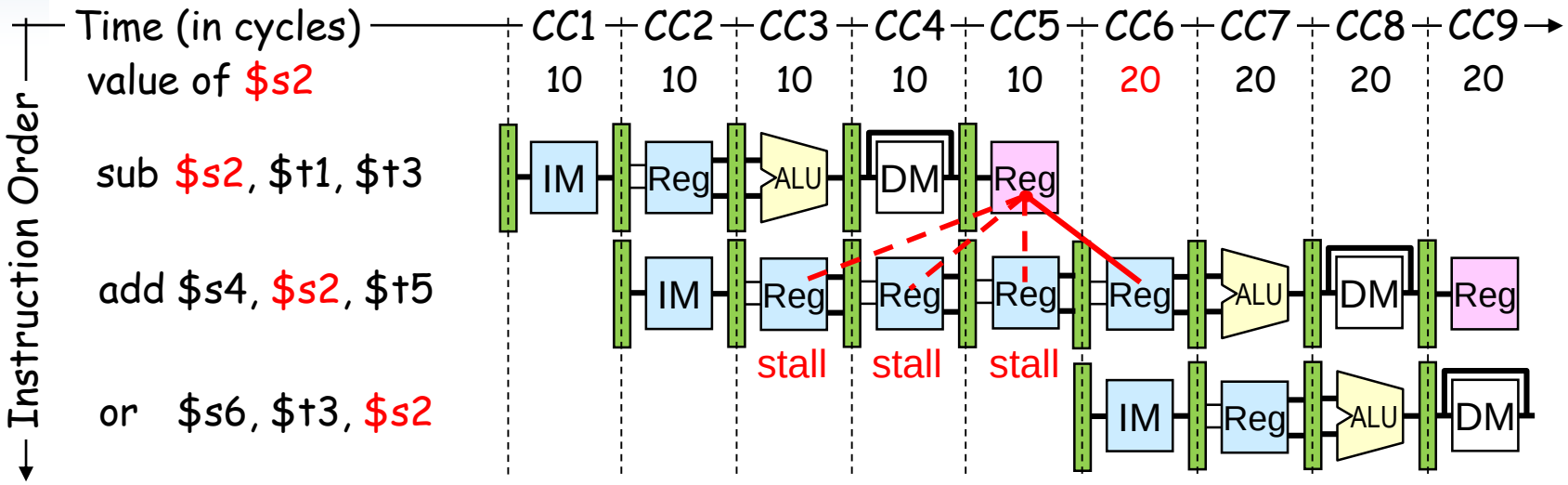
  - Rủi ro xảy ra khi lệnh *J* đọc toán hạn $s1 trước khi lệnh *I* cập nhật lại giá trị mới nhất

Ví dụ về rủi ro loại RAW



- Kết quả của lệnh sub cần bởi lệnh add, or, and, & sw
- Lệnh add & or sẽ đọc giá trị cũ của $s2 từ bộ thanh ghi
- Trong CC5, $s2 được ghi giá trị mới vào cuối chu kỳ

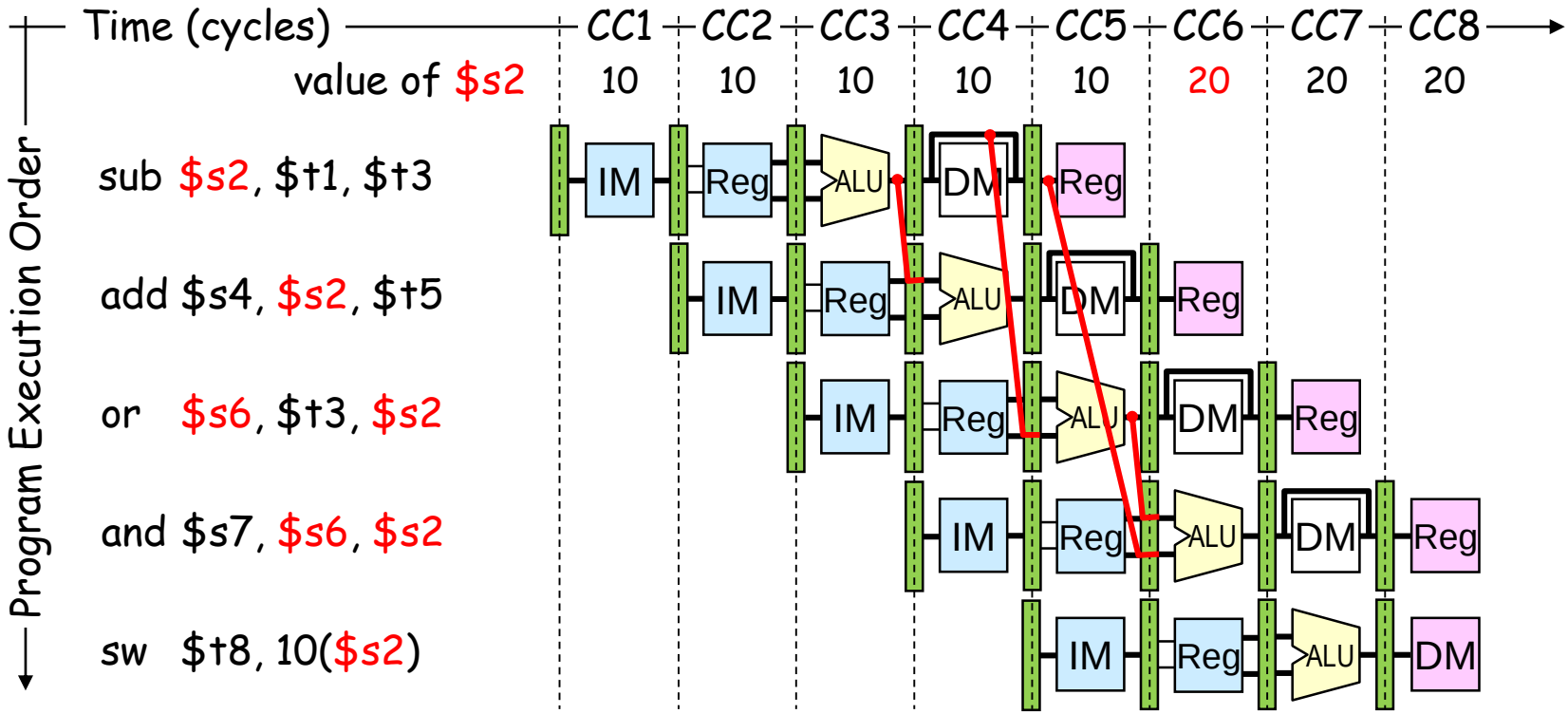
Giải pháp 1: Khựng quá trình pipeline



- Khựng (stall) 3 chu kỳ từ CC3 đến CC5
  - Các chu kỳ khựng làm trễ quá trình thực thi của lênh add & và quá trịnh nạp lệnh của lênh or
- Lệnh add không thể đọc $s2 cho đến đầu chu kỳ CC6
  - Lệnh add giữ nguyên mã lệnh cho đến chu kỳ CC6
  - Thanh ghi PC không thay đổi cho đến đầu chu kỳ CC6

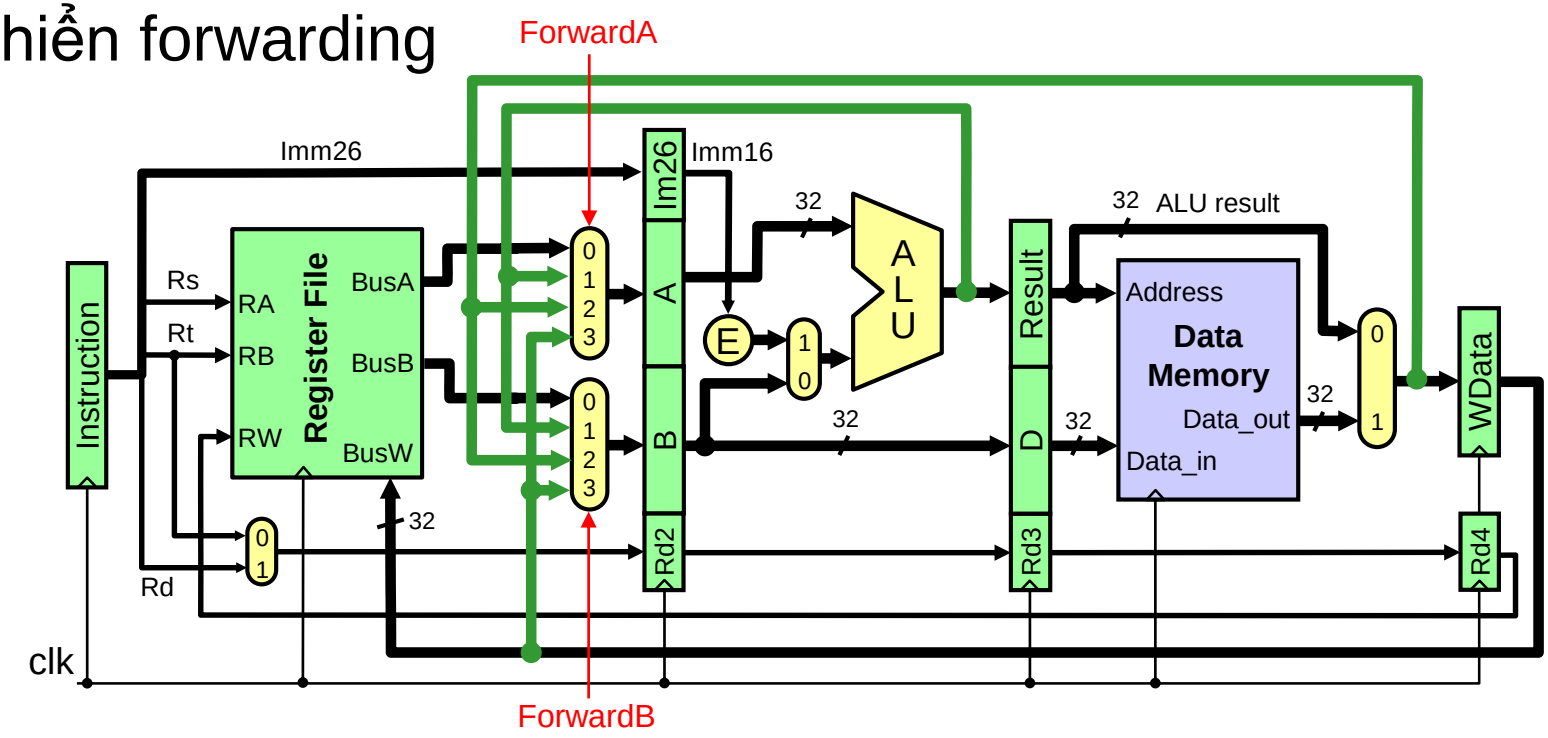
Giải pháp 2: Sử dụng sớm kết quả ALU

- Kết quả ALU được đưa sang ngõ vào ALU
  - Không cần khựng lại quá trình pipeline => không lãng phí chu kỳ nào
- Kết quả ALU được lấy từ công đoạn ALU, MEM, và WB



Hiện thực xúc tiến sớm (forwarding)

- Thêm vào 2 bộ hợp kênh (mux) 4 ngõ vào cho thanh ghi A & B
  - Dữ liệu từ công đoạn ALU, MEM, và WB được nối ngược lại
- Các tín hiệu điều khiển: ForwardA và ForwardB để điều khiển forwarding



Tín hiệu điều khiển forwarding

Signal	Explanation
ForwardA = 0	Toán hạn thứ nhất của ALU = giá trị thanh ghi Rs
ForwardA = 1	Lấy kết quả ALU của lệnh trước (đang ở công đoạn ALU)
ForwardA = 2	Lấy kết quả ALU của lệnh trước 2 chu kỳ (đang ở MEM)
ForwardA = 3	Lấy kết quả ALU của lệnh trước 3 chu kỳ (đang ở WB)
ForwardB = 0	Toán hạn thứ hai của ALU = giá trị thanh ghi Rt
ForwardB = 1	Lấy kết quả ALU của lệnh trước (đang ở công đoạn ALU)
ForwardB = 2	Lấy kết quả ALU của lệnh trước 2 chu kỳ (đang ở MEM)
ForwardB = 3	Lấy kết quả ALU của lệnh trước 3 chu kỳ (đang ở WB)

Forwarding Example

Instruction sequence:

```
lw   $t4, 4($t0)
ori  $t7, $t1, 2
sub  $t3, $t4, $t7
```

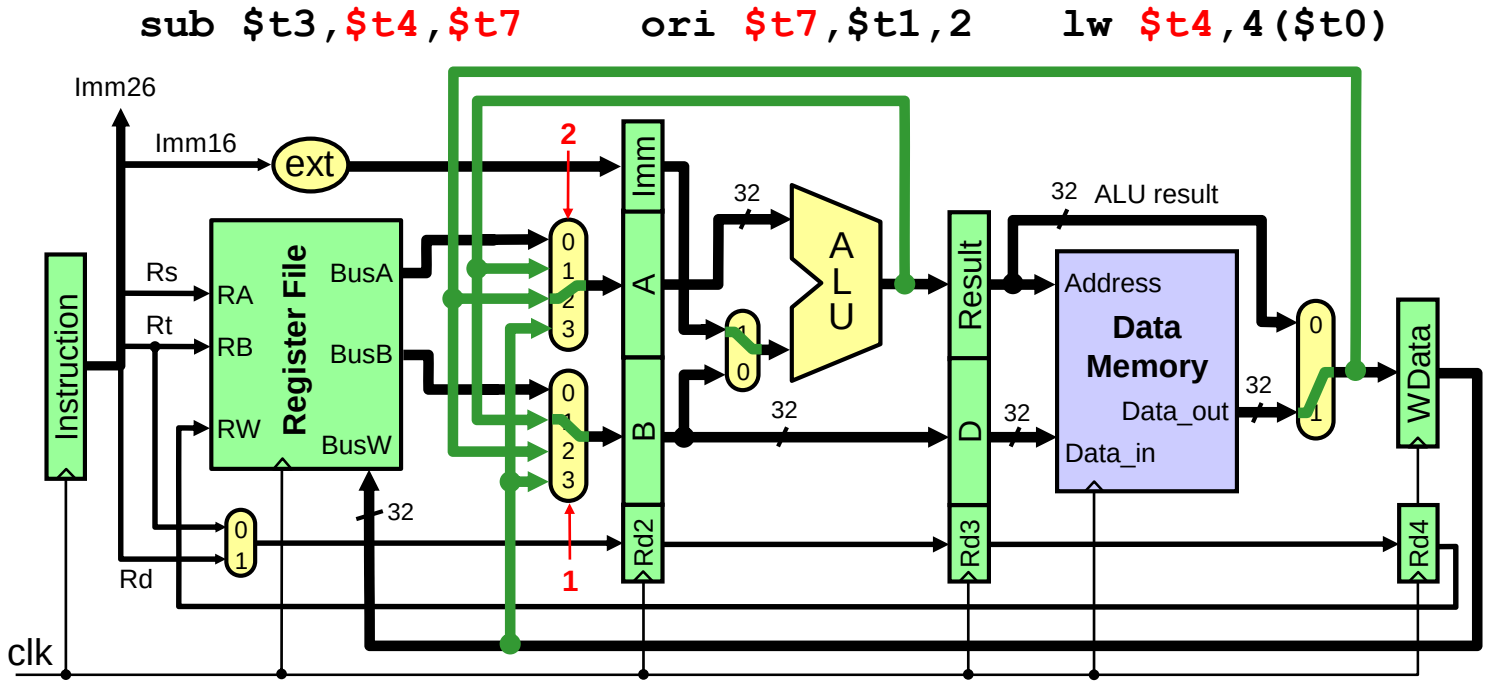
When **sub** instruction is fetched

ori will be in the ALU stage

lw will be in the MEM stage

ForwardA = 2 from MEM stage

ForwardB = 1 from ALU stage



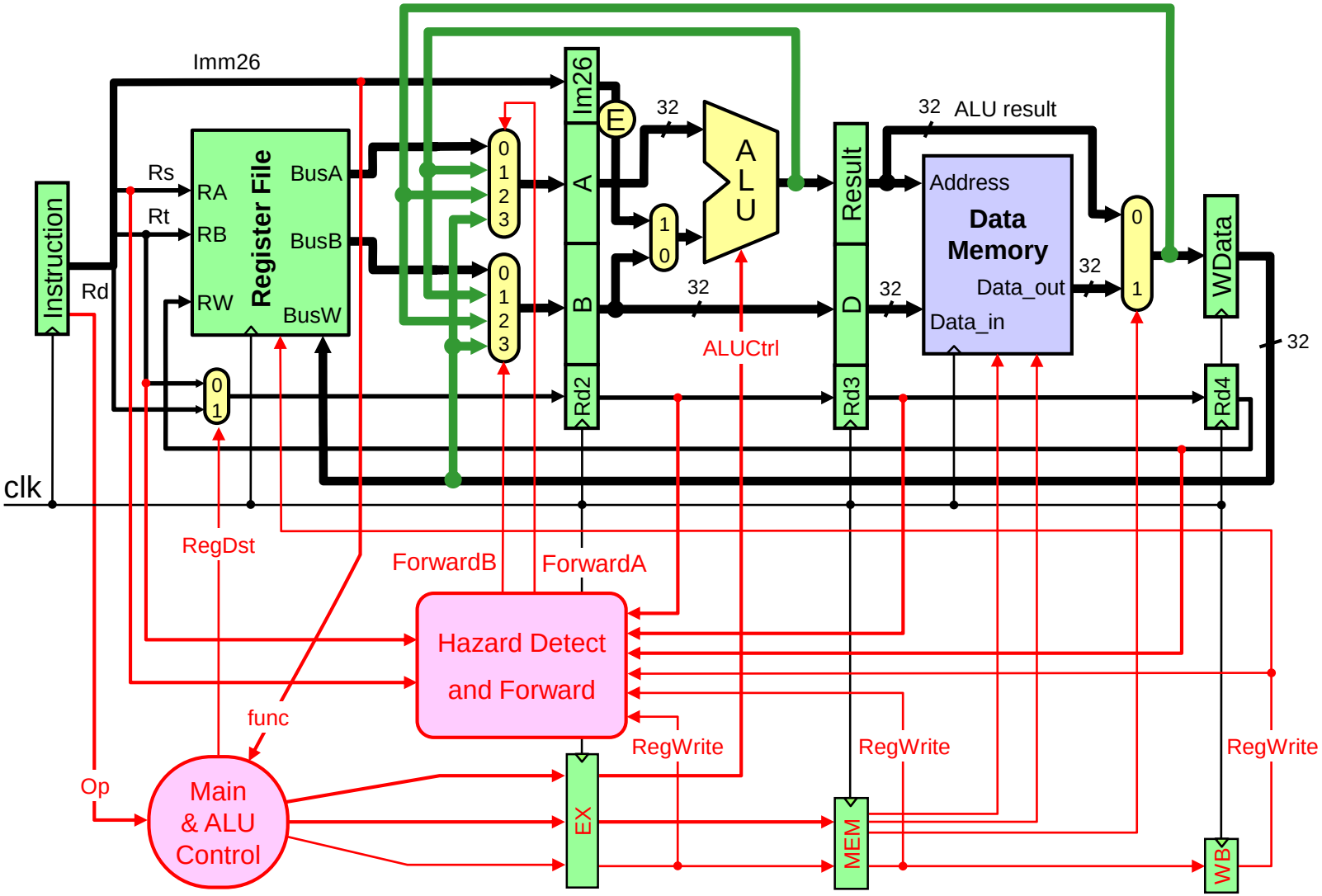
RAW Hazard Detection

- Current instruction being decoded is in Decode stage
  - Previous instruction is in the Execute stage
  - Second previous instruction is in the Memory stage
  - Third previous instruction in the Write Back stage

```
If  ((Rs != 0) and (Rs == Rd2) and (EX.RegWrite))          ForwardA ← 1
Else if          ((Rs != 0) and (Rs == Rd3) and (MEM.RegWrite))  ForwardA ← 2
Else if          ((Rs != 0) and (Rs == Rd4) and (WB.RegWrite))    ForwardA ← 3
Else                                                      ForwardA ← 0

If  ((Rt != 0) and (Rt == Rd2) and (EX.RegWrite))          ForwardB ← 1
Else if          ((Rt != 0) and (Rt == Rd3) and (MEM.RegWrite))  ForwardB ← 2
Else if          ((Rt != 0) and (Rt == Rd4) and (WB.RegWrite))    ForwardB ← 3
Else                                                      ForwardB ← 0
```

Hazard Detect and Forward Logic

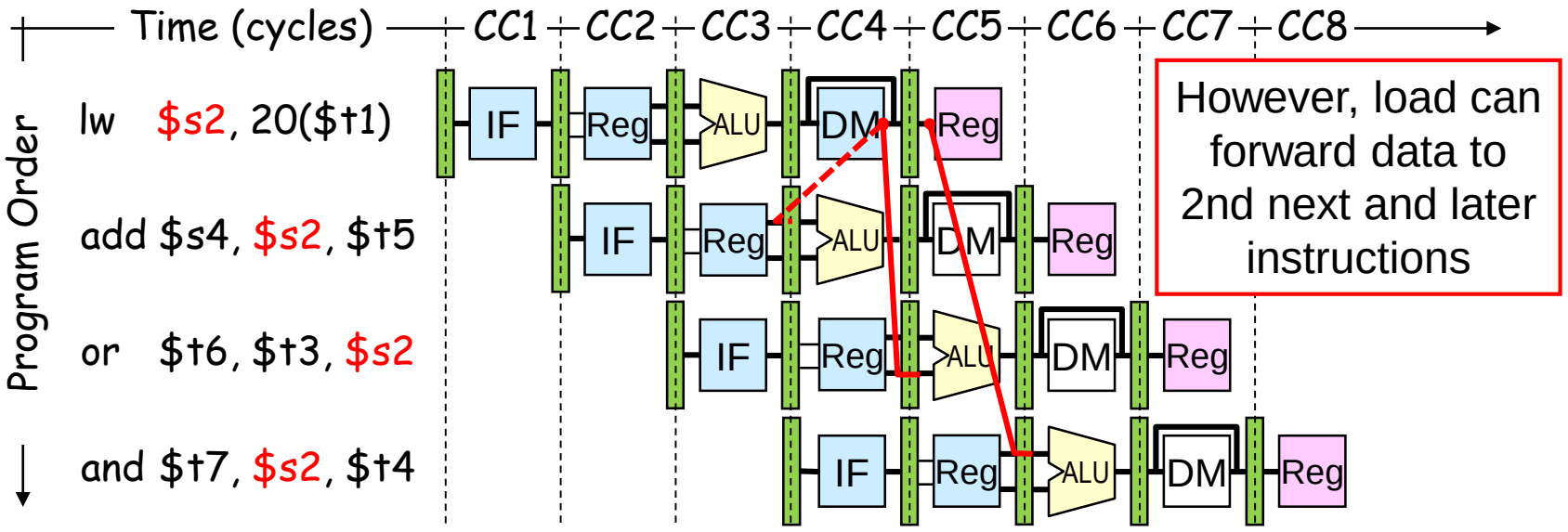


Tiếp theo…

- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

Load Delay

- Unfortunately, not all data hazards can be forwarded
  - Load has a delay that cannot be eliminated by forwarding
- In the example shown below …
  - The LW instruction does not read data until end of CC4
  - Cannot forward data to ADD at end of CC3 - NOT possible



Detecting RAW Hazard after Load

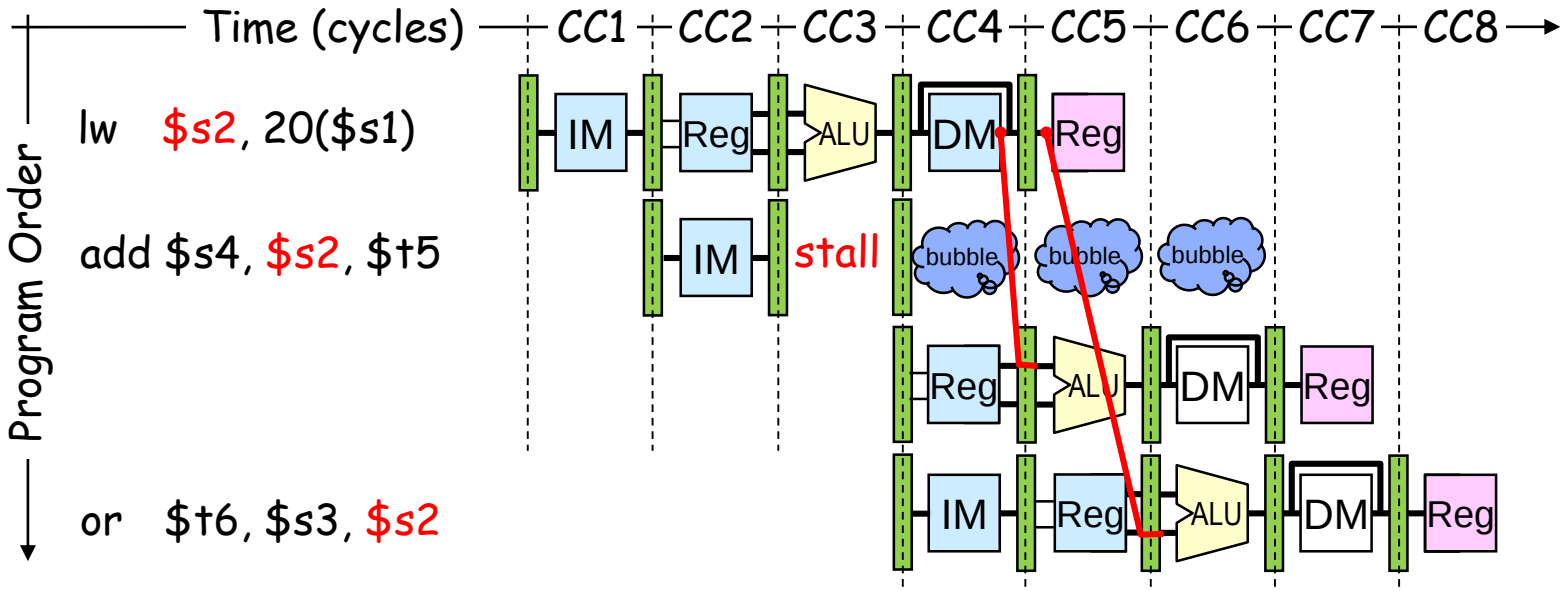
- Detecting a RAW hazard after a Load instruction:
  - The load instruction will be in the EX stage
  - Instruction that depends on the load data is in the decode stage
- Condition for stalling the pipeline

```
if  ((EX.MemRead == 1)  // Detect Load in EX stage
and (ForwardA==1 or ForwardB==1)) Stall  // RAW Hazard
```

- Insert a bubble into the EX stage after a load instruction
  - Bubble is a no-op that wastes one clock cycle
  - Delays the dependent instruction after load by once cycle
    - Because of RAW hazard

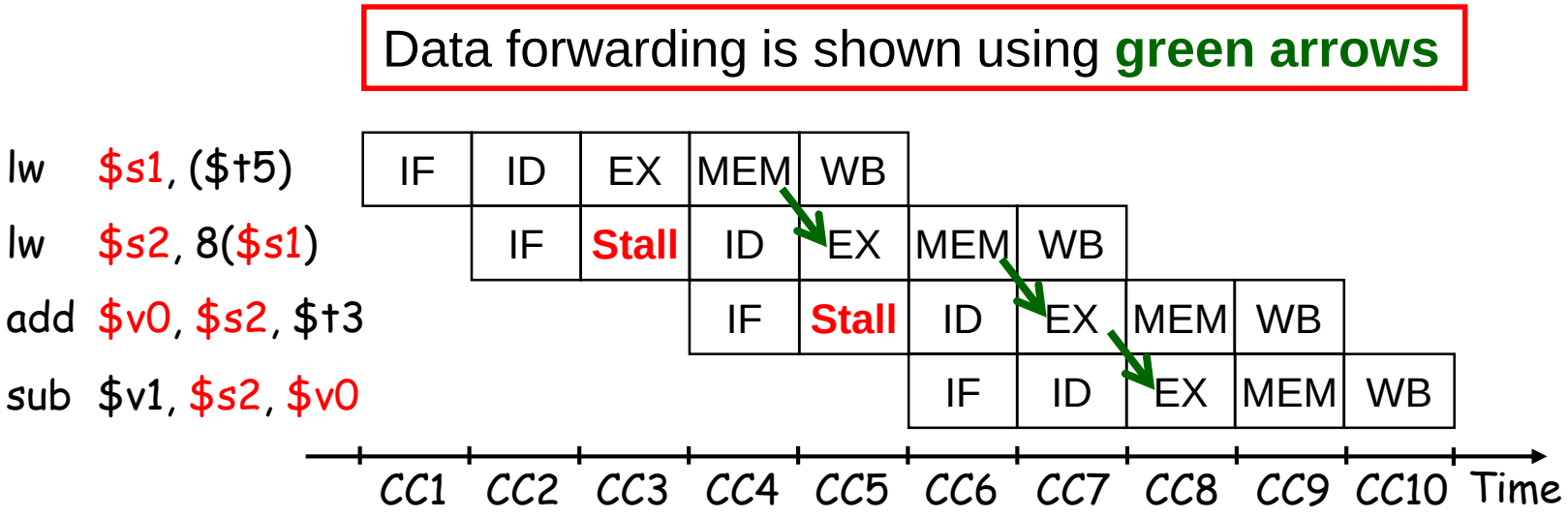
Stall the Pipeline for one Cycle

- ADD instruction depends on LW ➔ stall at CC3
  - Allow Load instruction in ALU stage to proceed
  - Freeze PC and Instruction registers (NO instruction is fetched)
  - Introduce a bubble into the ALU stage (bubble is a NO-OP)
- Load can forward data to next instruction after delaying it

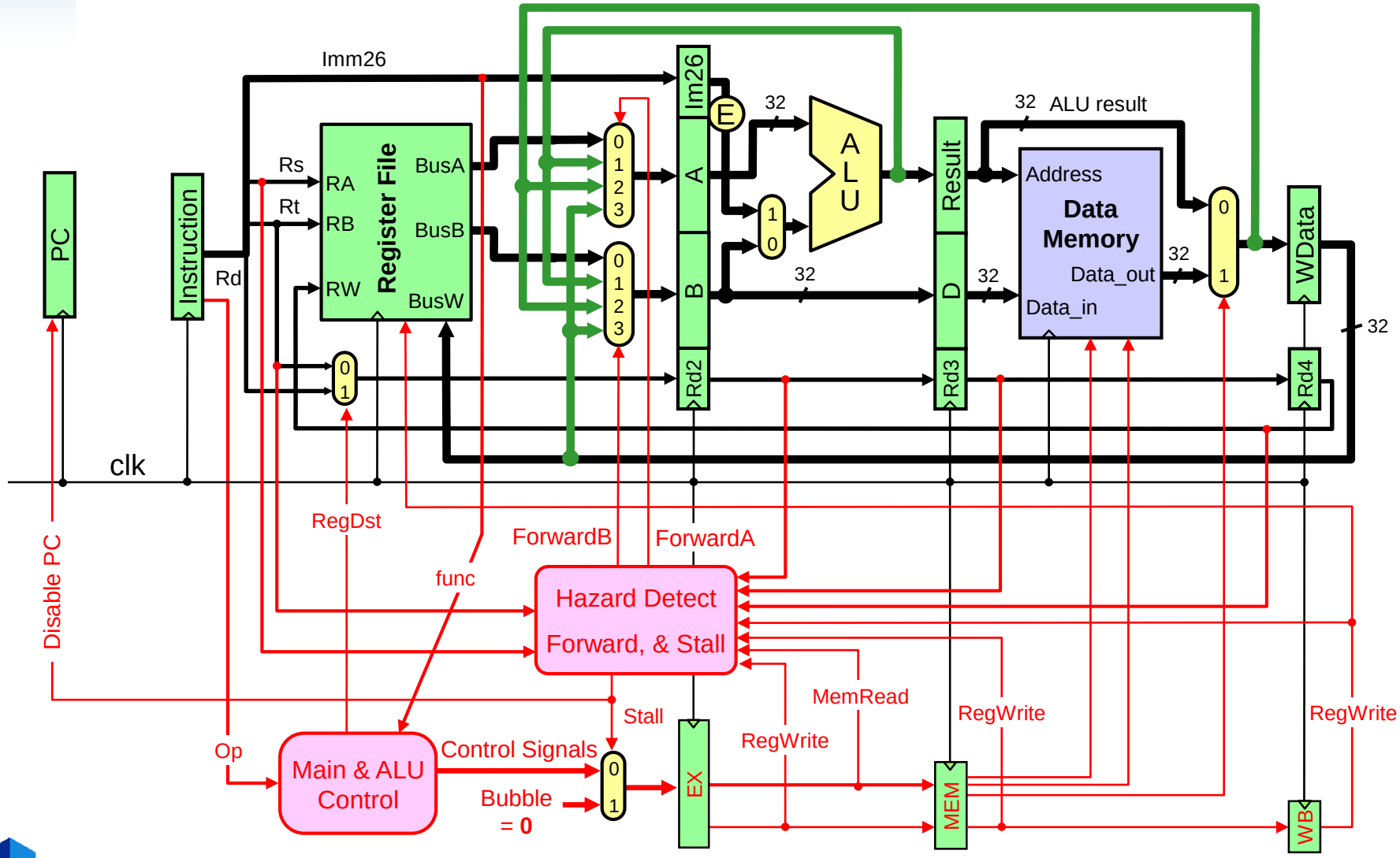


Showing Stall Cycles

- Stall cycles can be shown on instruction-time diagram
- Hazard is detected in the Decode stage
- Stall indicates that instruction is delayed
- Instruction fetching is also delayed after a stall
- Example:



Hazard Detect, Forward, and Stall



Code Scheduling to Avoid Stalls

- Compilers reorder code in a way to avoid load stalls
- Consider the translation of the following statements:

```
A = B + C; D = E - F; // A thru F are in Memory
```

- Slow code:

```
lw   $t0,4($s0)    # &B = 4($s0)
lw   $t1,8($s0)    # &C = 8($s0)
add  $t2,$t0,$t1   # stall cycle
sw   $t2,0($s0)    # &A = 0($s0)
lw   $t3,16($s0)   # &E = 16($s0)
lw   $t4,20($s0)   # &F = 20($s0)
sub  $t5,$t3,$t4   # stall cycle
sw   $t5,12($0)    # &D = 12($0)
```

- Fast code: No Stalls

```
lw   $t0, 4($s0)
lw   $t1, 8($s0)
lw   $t3, 16($s0)
lw   $t4, 20($s0)
add  $t2, $t0, $t1
sw   $t2, 0($s0)
sub  $t5, $t3, $t4
sw   $t5, 12($s0)
```

Name Dependence: Write After Read

- Instruction **J** should write its result after it is read by **I**
- Called anti-dependence by compiler writers

```
I: sub $t4, $t1, $t3 # $t1 is read
J: add $t1, $t2, $t3 # $t1 is written
```

- Results from reuse of the name **$t1**
- NOT a data hazard in the 5-stage pipeline because:
  - Reads are always in stage 2
  - Writes are always in stage 5, and
  - Instructions are processed in order
- Anti-dependence can be eliminated by **renaming**
  - Use a different destination register for **add** (eg, **$t5**)

Name Dependence: Write After Write

- Same destination register is written by two instructions
- Called output-dependence in compiler terminology

```
I: sub $t1, $t4, $t3      # $t1 is written
J: add $t1, $t2, $t3      # $t1 is written again
```

- Not a data hazard in the 5-stage pipeline because:
  - All writes are ordered and always take place in stage 5
- However, can be a hazard in more complex pipelines
  - If instructions are allowed to complete out of order, and
  - Instruction `J` completes and writes `$t1` before instruction `I`
- Output dependence can be eliminated by **renaming `$t1`**
- Read After Read is NOT a name dependence

Tiếp theo…

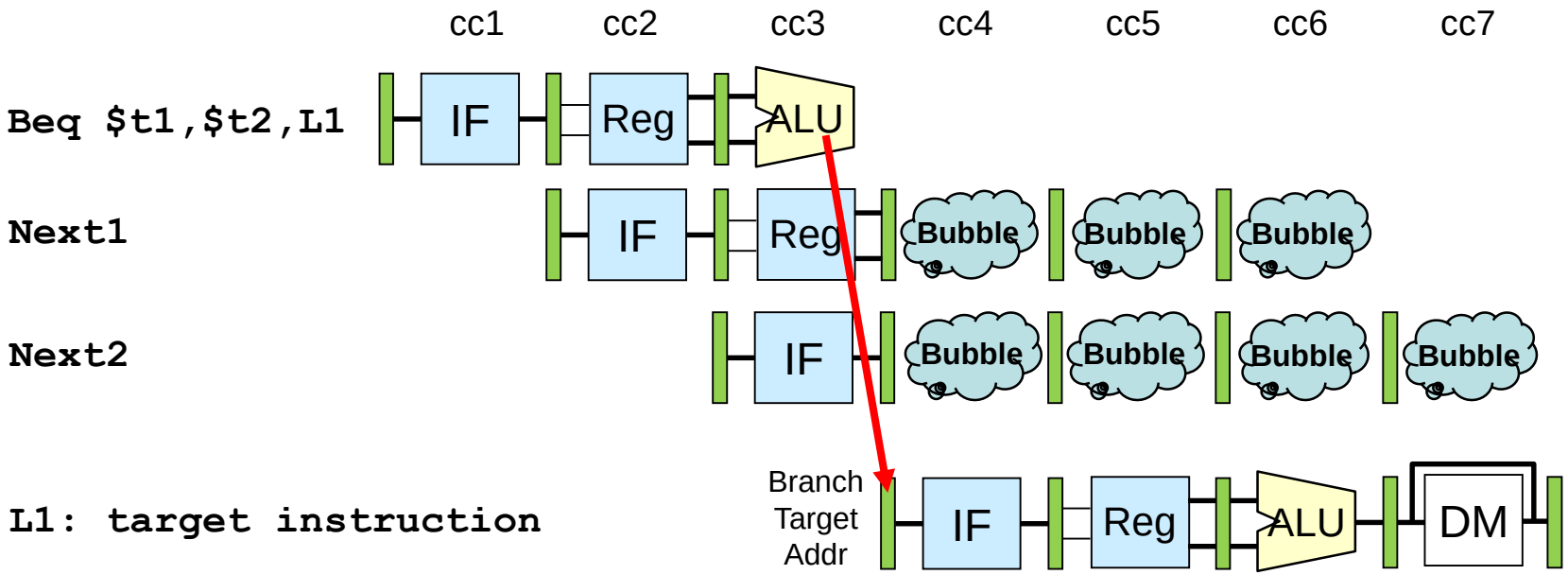
- Thực thi theo kiểu đường ống so với tuần tự
- Datapath & Control theo kiểu đường ống
- Rủi ro (Hazard) trong hiện thực đường ống
- Rủi ro về dữ liệu và phương pháp xúc tiến sớm
- Chờ trong lệnh “Load”, phát hiện rủi ro và khựng
- Rủi ro về điều khiển

Control Hazards

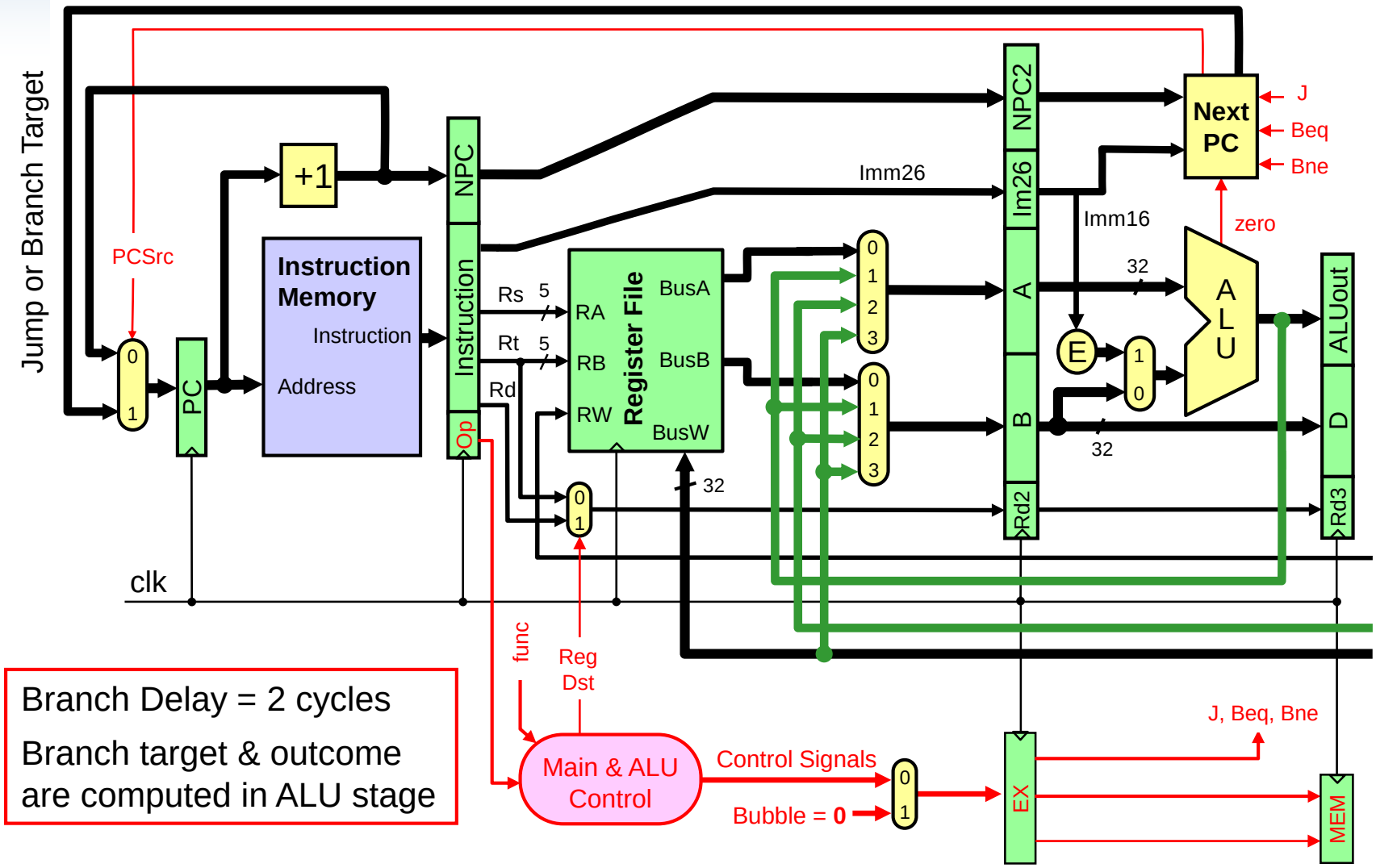
- Jump and Branch can cause great performance loss
- Jump instruction needs only the jump target address
- Branch instruction needs two things:
  - Branch Result — Taken or Not Taken
  - Branch Target Address
    - PC + 4 — If Branch is NOT taken
    - PC + 4 + 4 × immediate — If Branch is Taken
- Jump and Branch targets are computed in the ID stage
  - At which point a new instruction is already being fetched
  - Jump Instruction: 1-cycle delay
  - Branch: 2-cycle delay for branch result (taken or not taken)

2-Cycle Branch Delay

- Control logic detects a Branch instruction in the 2nd Stage
- ALU computes the Branch outcome in the 3rd Stage
- Next1 and Next2 instructions will be fetched anyway
- Convert Next1 and Next2 into bubbles if branch is taken



Implementing Jump and Branch

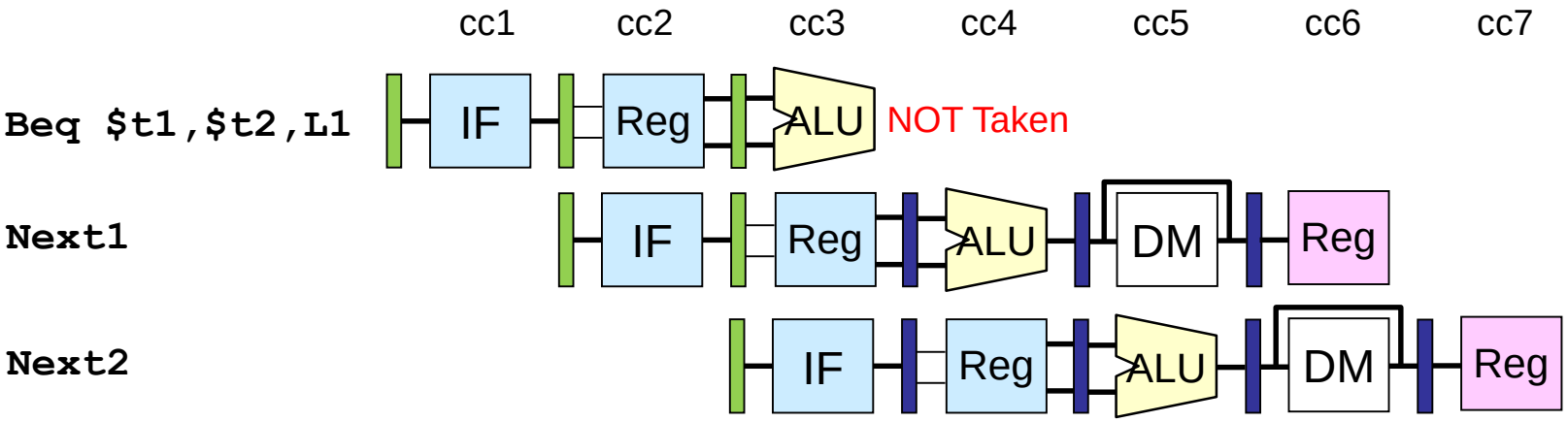


Branch Delay = 2 cycles

Branch target & outcome are computed in ALU stage

Predict Branch NOT Taken

- Branches can be predicted to be NOT taken
- If branch outcome is NOT taken then
  - Next1 and Next2 instructions can be executed
  - Do not convert Next1 & Next2 into bubbles
  - No wasted cycles



Reducing the Delay of Branches

- Branch delay can be reduced from 2 cycles to just 1 cycle
- Branches can be determined earlier in the Decode stage
  - A comparator is used in the decode stage to determine branch decision, whether the branch is taken or not
  - Because of forwarding the delay in the second stage will be increased and this will also increase the clock cycle
- Only one instruction that follows the branch is fetched
- If the branch is taken then only one instruction is flushed
- We should insert a bubble after jump or taken branch
  - This will convert the next instruction into a NOP

Reducing Branch Delay to 1 Cycle