Chương 3: Tập lệnh máy tính

Chương 3: Nội dung chính

- Giới thiệu về tập lệnh
- Khuôn dạng và các thành phần của lệnh
- Các dạng toán hạng lệnh
- Các chế độ địa chỉ
- Một số dạng lệnh thông dụng
- Cơ chế ống lệnh

Giới thiệu chung

- Lệnh máy tính là một từ nhị phân (binary word) mà thực hiện một nhiệm vụ cụ thể:
  - Lệnh được lưu trong bộ nhớ
  - Lệnh được đọc từ bộ nhớ vào CPU để giải mã và thực hiện
  - Mỗi lệnh có chức năng riêng của nó
- Tập lệnh gồm nhiều lệnh, có thể được chia thành các nhóm theo chức năng:
  - Chuyển dữ liệu (data movement)
  - Tính toán (computational)
  - Điều kiện và rẽ nhánh (conditioning & branching)
  - Các lệnh khác …

Giới thiệu chung

- Quá trình thực hiện/ chạy lệnh được chia thành các pha hay giai đoạn (stage). Mỗi lệnh có thể được thực hiện theo 4 giai đoạn:
  - Đọc lệnh IF(Instruction Fetch): lệnh được đọc từ bộ nhớ vào CPU
  - Giải mã lệnh ID(Instruction Decode): CPU giải mã lệnh
  - Chay lệnh IE(Instruction Execution): CPU thực hiện lệnh
  - Ghi WB(Write Back): kết quả lệnh (nếu có) được ghi vào thanh ghi hoặc bộ nhớ

Chu kỳ thực hiện lệnh

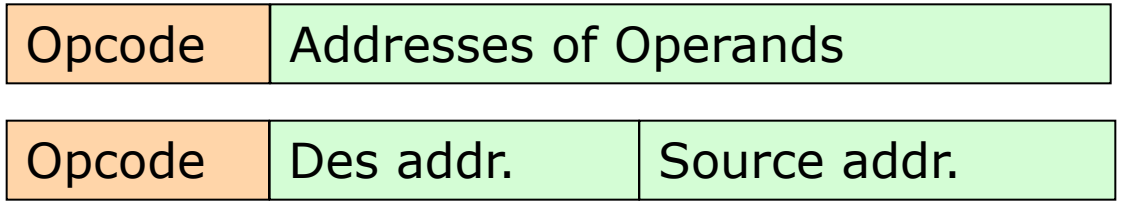
- Chu kỳ thực hiện lệnh (instruction execution cycle) là khoảng thời gian mà CPU thực hiện xong một lệnh
  - Một chu kỳ thực hiện lệnh gồm một số giai đoạn thực hiện lệnh
  - Một giai đoạn thực hiện lệnh có thể gồm một số chu kỳ máy
  - Một chu kỳ máy có thể gồm một số chu kỳ đồng hồ

Chu kỳ thực hiện lệnh

- Một chu kỳ thực hiện lệnh có thể gồm các thành phần sau:
  - Chu kỳ đọc lệnh
  - Chu kỳ đọc bộ nhớ (memory read)
  - Chu kỳ ghi bộ nhớ (memory write)
  - Chu kỳ đọc thiết bị ngoại vi (I/O read)
  - Chu kỳ ghi thiết bị ngoại vi (I/O write)
  - Chu kỳ bus rỗi (bus idle)

Khuôn dạng lệnh

- Khuôn dạng lệnh thông thường bao gồm 2 phần:
  - Mã lệnh (opcode): mỗi lệnh đều có riêng một mã
  - Địa chỉ các toán hạng (addresses of operands): số lượng toán hạng phụ thuộc vào lệnh. Có thể có các dạng địa chỉ toán hạng sau:
    - 3 địa chỉ
    - 2 địa chỉ
    - 1 địa chỉ
    - 1.5 địa chỉ
    - 0 địa chỉ



Toán hạng 3 địa chỉ

- Khuôn dạng:
  - opcode addr1, addr2, addr3
  - Mỗi địa chỉ addr1, addr2, addr3: tham chiếu tới một ô nhớ hoặc 1 thanh ghi
- Ví dụ
  1. ADD R_1, R_2, R_3; $R_2 + R_3 \rightarrow R_1$

     R_2 cộng R_3 sau đó kết quả đưa vào R_1

     R_i là các thanh ghi CPU
  2. ADD A, B, C; M[B]+M[C] → M[A]

     A, B, C là các vị trí trong bộ nhớ

Toán hạng 2 địa chỉ

- Khuôn dạng:
  - opcode addr1, addr2
  - Mỗi địa chỉ addr1, addr2: tham chiếu tới 1 thanh ghi hoặc 1 vị trí trong bộ nhớ
- Ví dụ
  1. ADD R_1, R_2; $R_1 + R_2 \rightarrow R_1$

     R_1 cộng R_2 sau đó kết quả đưa vào R_1

     R_i là các thanh ghi CPU
  2. ADD A, B; M[A]+M[B] → M[A]

     A, B là các vị trí trong bộ nhớ

Toán hạng 1 địa chỉ

- Khuôn dạng:
  - opcode addr
  - addr: tham chiếu tới 1 thanh ghi hoặc 1 vị trí trong bộ nhớ
  - Khuôn dạng này sử dụng R_{acc} (thanh ghi tích lũy) mặc định cho địa chỉ thứ 2
- Ví dụ
  1. ADD R_1; $R_1 + R_{acc} \rightarrow R_{acc}$
     R_1 cộng R_{acc} sau đó kết quả đưa vào R_{acc}
     R_i là các thanh ghi CPU
  2. ADD A; $M[A]+R_{acc} \rightarrow R_{acc}$
     A là vị trí trong bộ nhớ

Toán hạng 1.5 địa chỉ

- Khuôn dạng:
  - opcode addr1, addr2
  - Một địa chỉ tham chiếu tới 1 ô nhớ và địa chỉ còn lại tham chiếu tới 1 thanh ghi
  - Là dạng hỗn hợp giữa các toán hạng thanh ghi và vị trí bộ nhớ
- Ví dụ
  1. ADD R_1, B; M[B] + R_1 → R_1

Toán hạng 0 địa chỉ

- Được thực hiện trong các lệnh mà thực hiện các thao tác ngăn xếp: push & pop

Các chế độ địa chỉ

- Chế độ địa chỉ là cách thức CPU tổ chức các toán hạng
  - Chế độ địa chỉ cho phép CPU kiểm tra dạng và tìm các toán hạng của lệnh
- Một số chế độ địa chỉ tiêu biểu:
  - Chế độ địa chỉ tức thì (Immediate)
  - Chế độ địa chỉ trực tiếp (Direct)
  - Chế độ địa chỉ gián tiếp qua thanh ghi (Register Indirect)
  - Chế độ địa chỉ gián tiếp qua bộ nhớ (Memory Indirect)
  - Chế độ địa chỉ chỉ số (Indexed)
  - Chế độ địa chỉ tương đối (Relative)

Chế độ địa chỉ tức thì

- Giá trị của toán hạng nguồn có sẵn trong lệnh (hằng số)
- Toán hạng đích có thể là thanh ghi hoặc một vị trí bộ nhớ
- Ví dụ:

LOAD R_1, #1000; 1000 → R_1

giá trị 1000 được tải vào thanh ghi R1

LOAD B, #500; 500 → M[B]

Giá trị 500 được tải vào vị trí B trong bộ nhớ

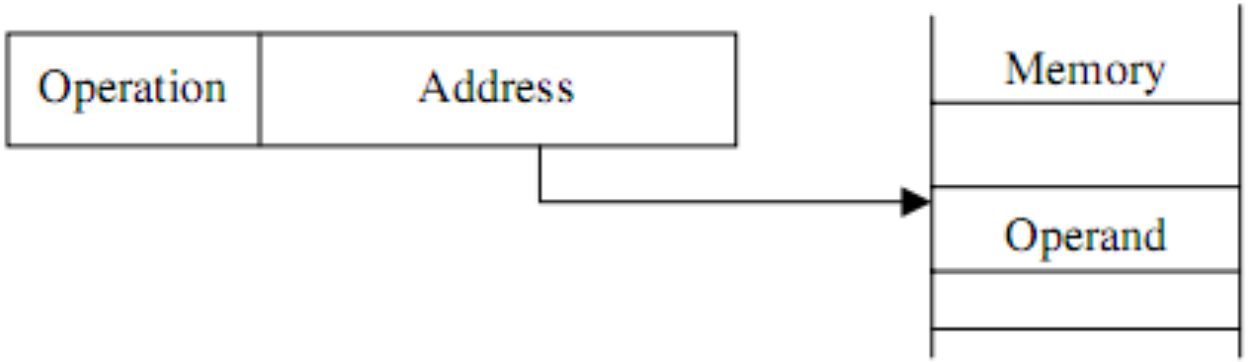
Chế độ địa chỉ trực tiếp/ tuyệt đối

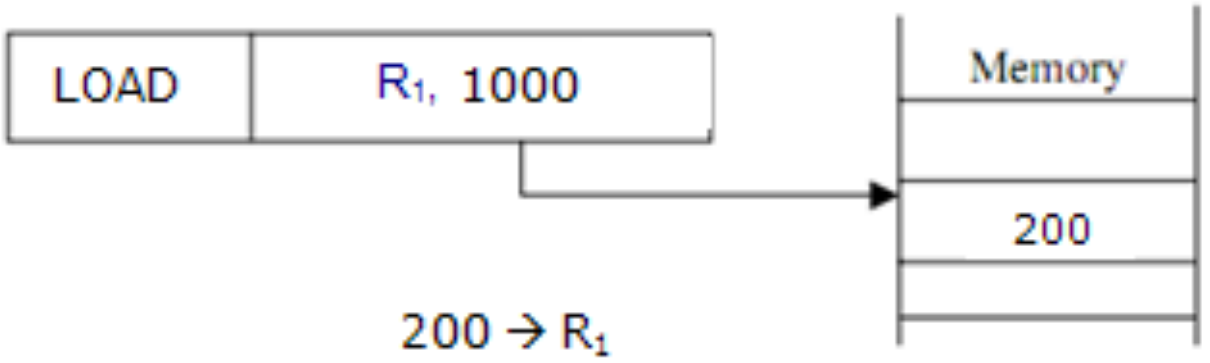
- Một toán hạng là địa chỉ của một vị trí trong bộ nhớ chứa dữ liệu
- Toán hạng kia là thanh ghi hoặc 1 địa chỉ ô nhớ
- Ví dụ:

LOAD R_1, 1000; M[1000] → R_1

giá trị lưu trong vị trí 1000 ở bộ nhớ được tải vào thanh ghiR1

Chế độ địa chỉ trực tiếp/ tuyệt đối





Chế độ địa chỉ gián tiếp

- Một thanh ghi hoặc một vị trí trong bộ nhớ được sử dụng để lưu địa chỉ của toán hạng
  - Gián tiếp thanh ghi:

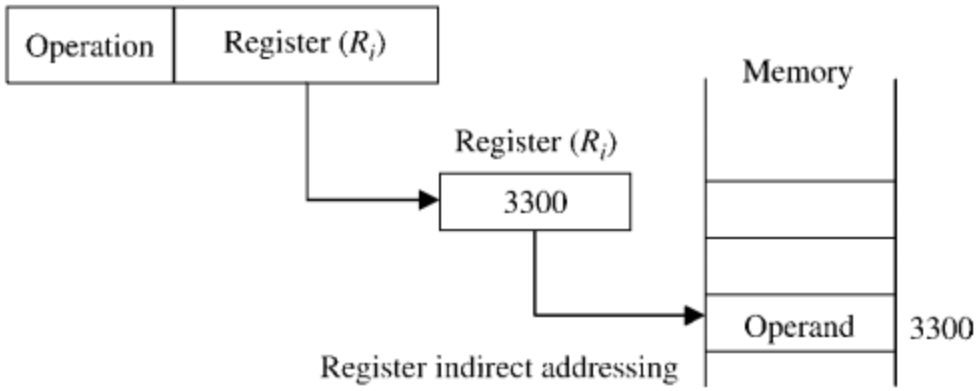
    LOAD R_j ,(R_i); M[R_i] → R_j

    Tải giá trị tại vị trí bộ nhớ có địa chỉ được lưu trong Ri vào thanh ghi Rj
  - Gián tiếp bộ nhớ:

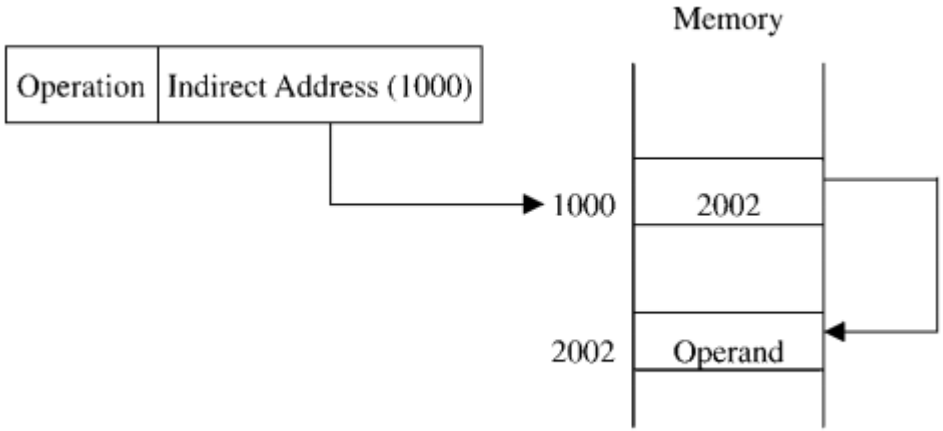
    LOAD R_i , (1000); M[M[1000]] → R_i

    Giá trị của vị trí bộ nhớ có địa chỉ được lưu tại vị trí 1000 vào Ri

Chế độ địa chỉ gián tiếp



Register indirect addressing

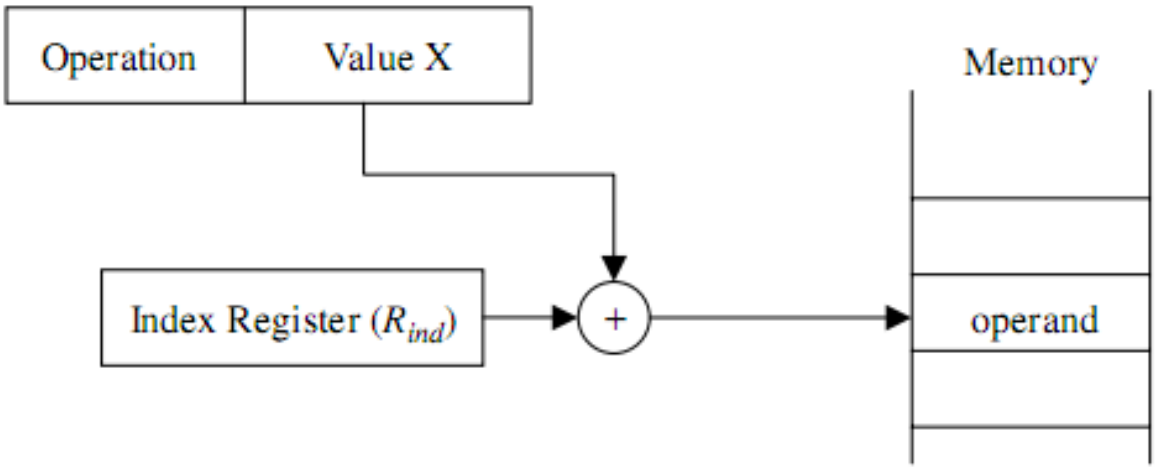


Memory indirect addressing

Chế độ địa chỉ chỉ số

- Địa chỉ của toán hạng có được bằng cách cộng thêm hằng số vào nội dung của một thanh ghi, là thanh ghi chỉ số
- Ví dụ

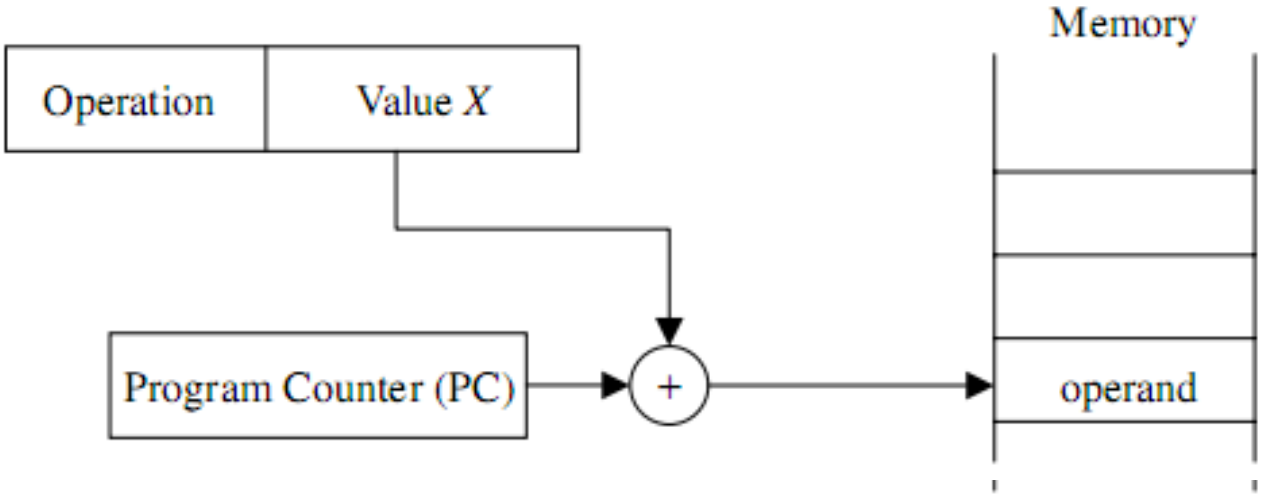
LOAD R_i, $X(R_{ind})$; $M[X+R_{ind}] \rightarrow R_i$



Chế độ địa chỉ tương đối

- Địa chỉ của toán hạng có được bằng cách cộng thêm hằng số vào nội dung của một thanh ghi, là thanh ghi con đếm chương trình PC
- Ví dụ

  LOAD R_i, X(PC); M[X+PC] → R_i



Tổng kết các chế độ địa chỉ

Chế độ địa chỉ	Ý nghĩa	Ví dụ	Thực hiện
Tức thì	Giá trị của toán hạng được chứa trong lệnh	LOAD Ri, #1000	Ri ← 1000
Trực tiếp	Địa chỉ của toán hạng được chứa trong lệnh	LOAD Ri, 1000	Ri ← M[1000]
Gián tiếp thanh ghi	Giá trị của thanh ghi trong lệnh là địa chỉ bô nhớ chứa toán hạng	LOAD Ri, (Rj)	Ri ← M[Rj]
Gián tiếp bộ nhớ	Địa chỉ bộ nhớ trong lệnh chứa địa chỉ bộ nhớ của toán hạng	LOAD Ri, (1000)	Ri ← M[M[1000]]
Chỉ số	Địa chỉ của toán hạng là tổng của hằng số (trong lệnh) và giá trị của một thanh ghi chỉ số	LOAD Ri, X(Rind)	Ri ← M[X+ Rind]
Tương đối	Địa chỉ của toán hạng là tổng của hằng số và giá trị của thanh ghi con đếm chương trình	LOAD Ri, X(PC)	Ri ← M[X+ PC]

Một số dạng lệnh thông dụng

- Các lệnh vận chuyển dữ liệu
- Các lệnh số học và logic
- Các lệnh điều khiển chương trình
- Các lệnh vào/ ra

Lệnh vận chuyển dữ liệu

- Chuyển dữ liệu giữa các phần của máy tính
  - Giữa các thanh ghi trong CPU

    MOVE Ri, Rj ; Rj -> Ri
  - Giữa thanh ghi CPU và một vị trí trong bộ nhớ

    MOVE Rj,1000; M[1000] -> Rj
  - Giữa các vị trí trong bộ nhớ

    MOVE 1000, (Rj) ; M[Rj] -> M[1000]

Một số lệnh vận chuyển dữ liệu thông dụng

- MOVE: chuyển dữ liệu giữa thanh ghi – thanh ghi, ô nhớ - thanh ghi, ô nhớ - ô nhớ
- LOAD: nạp nội dung 1 ô nhớ vào 1 thanh ghi
- STORE: lưu nội dung 1 thanh ghi ra 1 ô nhớ
- PUSH: đẩy dữ liệu vào ngăn xếp
- POP: lấy dữ liệu ra khỏi ngăn xếp

Lệnh số học và logic

- Thực hiện các thao tác số học và logic giữa các thanh ghi và nội dung ô nhớ
- Ví dụ:

ADD R1, R2, R3; R2 + R3 -> R1

SUBSTRACT R1, R2, R3; R2 – R3 -> R1

Các lệnh tính toán số học thông dụng

- ADD: cộng 2 toán hạng
- SUBSTRACT: trừ 2 toán hạng
- MULTIPLY: nhân 2 toán hạng
- DIVIDE: chia số học
- INCREMENT: tăng 1
- DECREMENT: giảm 1

Các lệnh logic thông dụng

- NOT: phủ định
- AND: và
- OR: hoặc
- XOR: hoặc loại trừ
- COMPARE: so sánh
- SHIFT: dịch
- ROTATE: quay

Lệnh điều khiển/ tuần tự

- Được dùng để thay đổi trình tự các lệnh được thực hiện:
  - Các lệnh rẽ nhánh (nhẩy) có điều kiện (conditional branching/ jump)
  - Các lệnh rẽ nhánh (nhẩy) không điều kiện (unconditional branching/ jump)
  - CALL và RETURN: lệnh gọi thực hiện và trở về từ chương trình con
- Đặc tính chung của các lệnh này là quá trình thực hiện lệnh của chúng làm thay đổi giá trị PC
- Sử dụng các cờ ALU để xác định các điều kiện

Một số lệnh điều khiển thông dụng

- BRANCH – IF – CONDITION: chuyển đến thực hiện lệnh ở địa chỉ mới nếu điều kiện là đúng
- JUMP: chuyển đến thực hiện lệnh ở địa chỉ mới
- CALL: chuyển đến thực hiện chương trình con
- RETURN: trở về (từ chương trình con) thực hiện tiếp chương trình gọi

Một số lệnh điều khiển thông dụng

```
    LOAD  R1, #100
LAP:
    ADD R0, (R2)
    DECREMENT  R1
    BRANCH_IF >0 LAP
```

Các lệnh vào/ ra

- Được dùng để truyền dữ liệu giữa máy tính và các thiết bị ngoại vi
- Các thiết bị ngoại vi giao tiếp với máy tính thông qua các cổng. Mỗi cổng có một địa chỉ dành riêng
- Hai lệnh I/O cơ bản được sử dụng là các lệnh INPUT và OUTPUT
  - Lệnh INPUT được dùng để chuyển dữ liệu từ thiết bị ngoại vi vào tới bộ vi xử lý
  - Lệnh OUTPUT dùng để chuyển dữ liệu từ VXL ra thiết bị đầu ra

Các ví dụ

```
    CLEAR R0;
    MOVE R1, #100;
    CLEAR R2;
LAP:
    ADD R0, 1000(R2);
    INCREMENT R2;
    DECREMENT R1;
    BRANCH_IF>0  LAP;
    STORE 2000, R0;
```

Các ví dụ

CLEAR R0;	R0← 0
MOVE R1, #100;	R1← 100
CLEAR R2;	R2← 0
LAP:	
ADD R0, 1000(R2);	R0←R0+ M[R2+1000]
INCREMENT R2;	R2← R2+1
DECREMENT R1;	R1← R1-1
BRANCH_IF>0 LAP;	go to LAP if R1>0
STORE 2000, R0;	M[2000]← R0

BÀI TẬP

1. Cho đoạn lệnh sau:

   ADD R2, (R0);

   SUBSTRACT R2, (R1);

   MOVE 500(R0), R2;

   LOAD R2, #5000;

   STORE 100(R2), R0;

Biết R0=1500, R1=4500, R2=1000, M[1500]=3000, M[4500]=500

a. Chỉ rõ chế độ địa chỉ của từng lệnh

b. Hãy chỉ ra giá trị của thanh ghi và tại vị trí trong bộ nhớ qua mỗi lệnh thực hiện.

BÀI TẬP

2. Cho đoạn lệnh sau:

```
        MOVE R0, #100;
        CLEAR R1;
        CLEAR R2;
LAP:
        ADD R1, 2000(R2);
        ADD R2, #2;
        DECREMENT R0;
        BRANCH_IF>0 LAP;
        STORE 3000, R1;
```

a. Hãy giải thích ý nghĩa của từng lệnh
b. Chỉ ra chế độ địa chỉ của từng lệnh (đối với các lệnh có 2 toán hạng)
c. Đoạn lệnh trên thực hiện công việc gì?

BÀI TẬP

- Cho một mảng gồm 10 số, được lưu trữ liên tiếp nhau trong bộ nhớ, bắt đầu từ vị trí ô nhớ 1000. Viết đoạn chương trình tính tổng các số dương trong mảng đó và lưu kết quả vào ô nhớ 2000.

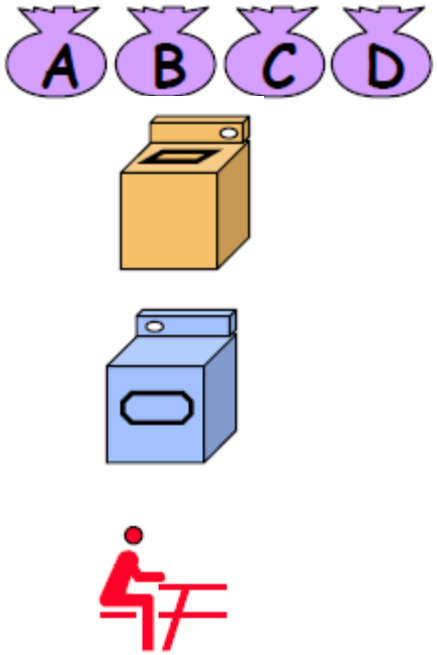
CPU pipeline

Nội dung chính

- Giới thiệu về CPU pipeline
- Các vấn đề của pipeline
- Xử lý xung đột dữ liệu và tài nguyên
- Xử lý rẽ nhánh (branch)
- Super pipeline

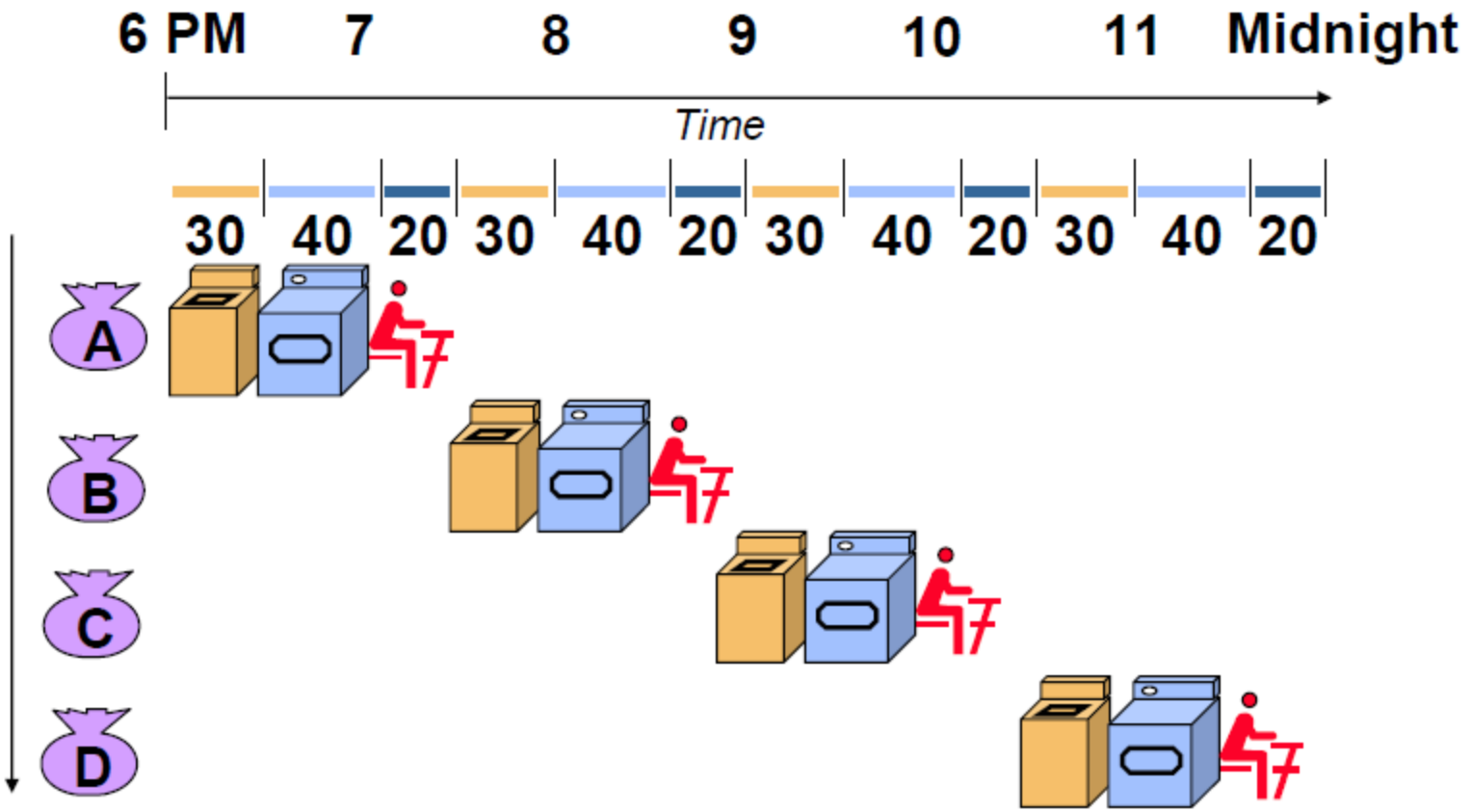
Pipeline – Ví dụ thực tế

- Bài toán giặt: A. B, C, D có 4 túi quần áo cần giặt, làm khô, gấp
- Giặt tốn 30 phút
- Sấy khô: 40 phút
- Gấp: 20 phút



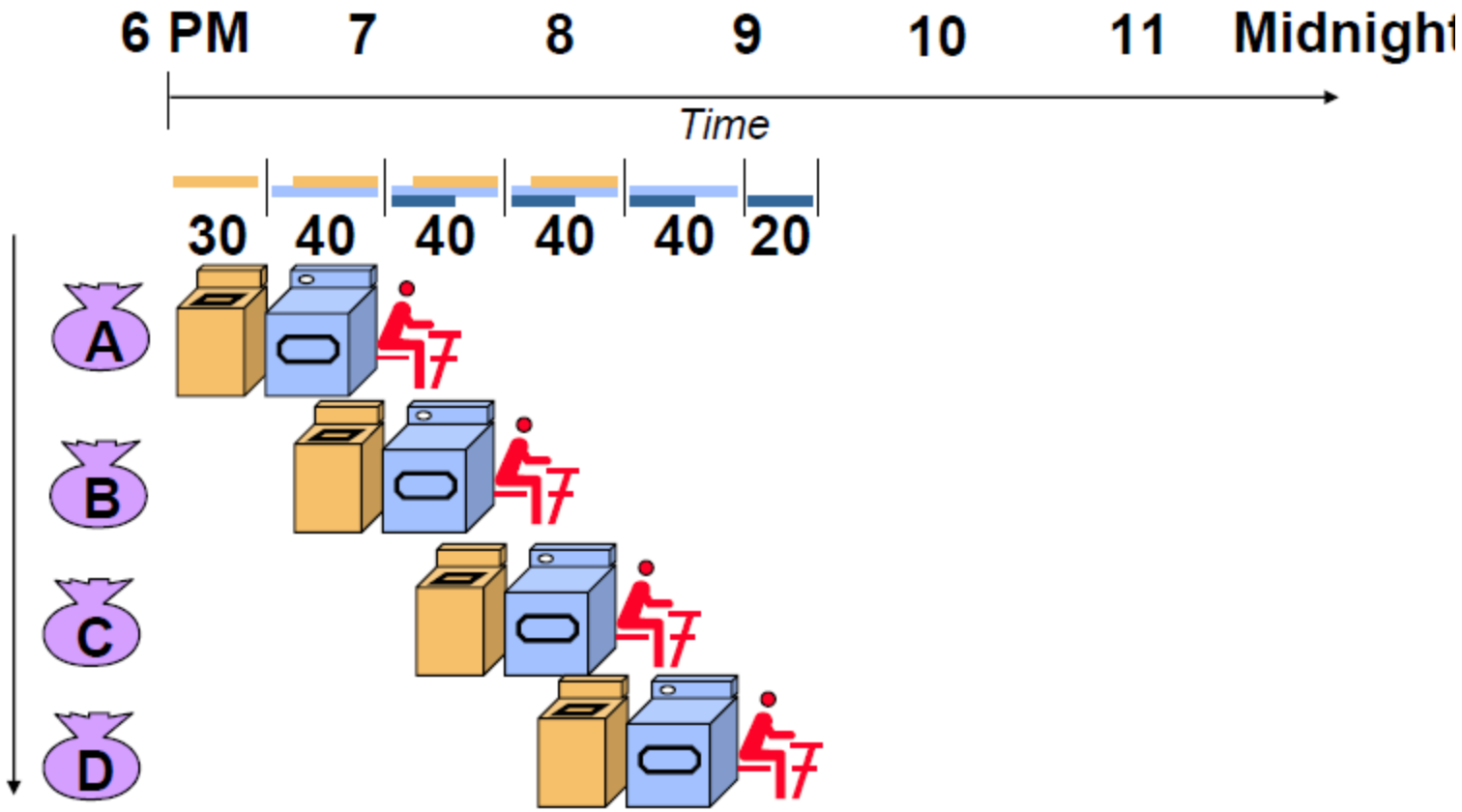
Pipeline – Ví dụ thực tế

Thực hiện tuần tự



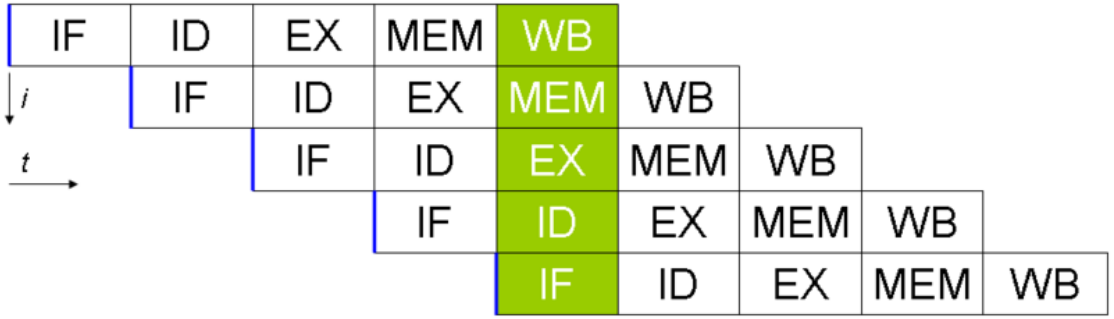
Pipeline – Ví dụ thực tế

Áp dụng pipeline

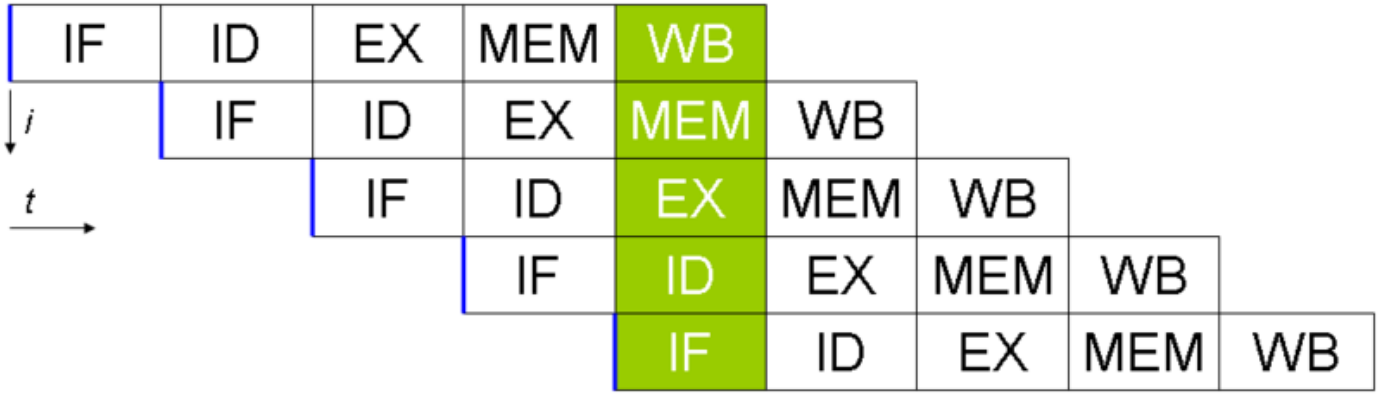


Giới thiệu về CPU Pipeline – Nguyên lý

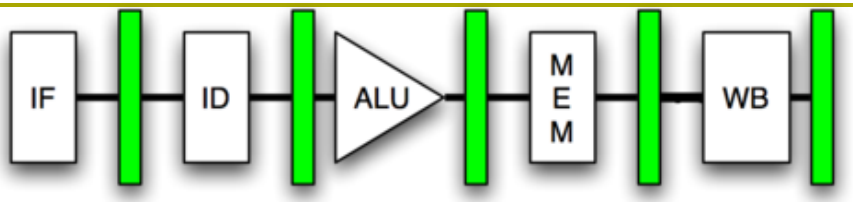
- Quá trình thực hiện lệnh được chia thành các giai đoạn
- 5 giai đoạn của hệ thống load – store:
  - Instruction fetch (IF): lấy lệnh từ bộ nhớ (hoặc cache)
  - Instruction Decode (ID): giải mã lệnh và lấy các toán hạng
  - Execute (EX): thực hiện lệnh: nếu là lệnh truy cập bộ nhớ thì tính toán địa chỉ bộ nhớ
  - Memory access (MEM): đọc/ ghi bộ nhớ ; nếu không truy cập bộ nhớ thì không có
  - Write back (WB): lưu kết quả vào thanh ghi
- Cải thiện hiệu năng bằng cách tăng số lượng lệnh vào xử lý

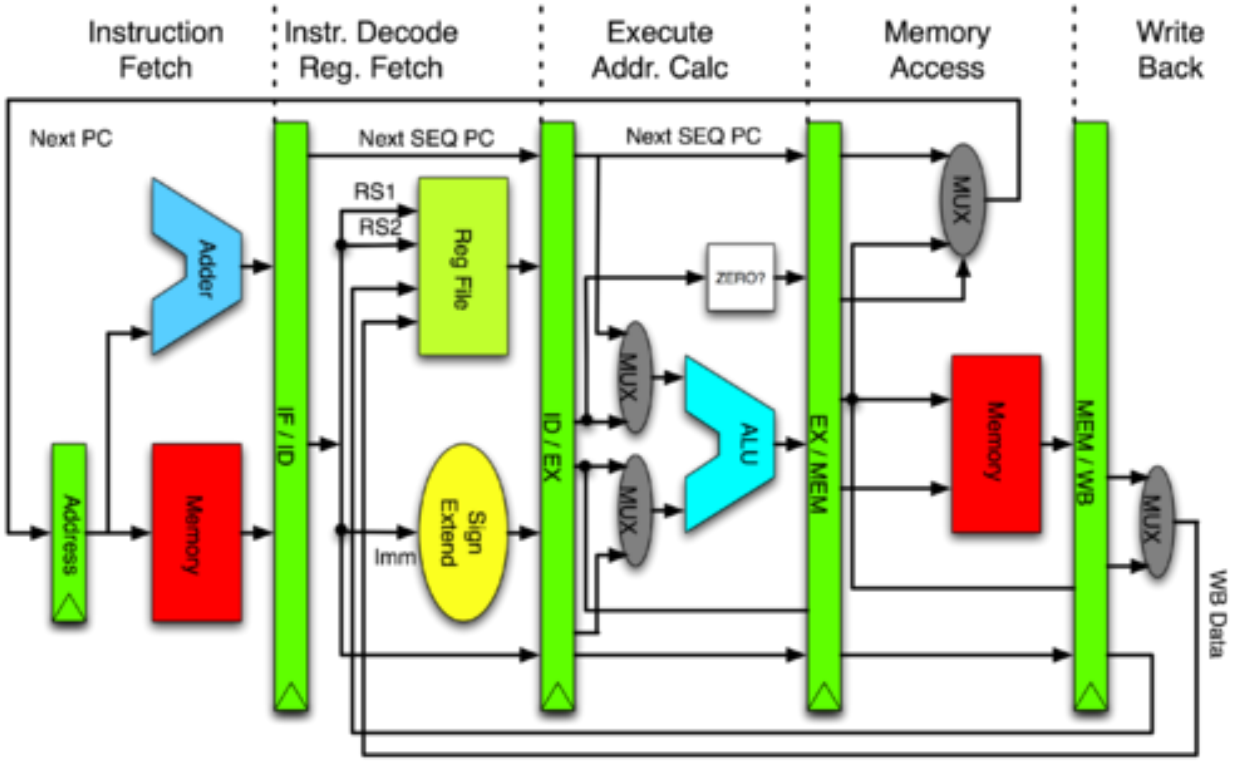


Giới thiệu về CPU Pipeline – Nguyên lý



Giới thiệu về CPU Pipeline – Nguyên lý





Giới thiệu về CPU Pipeline – Đặc điểm

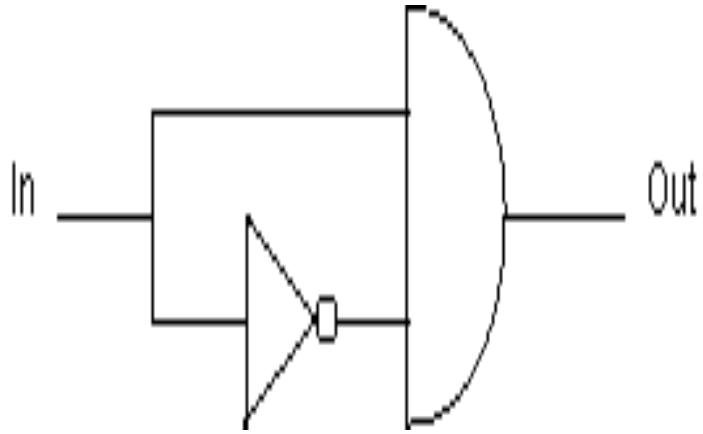
- Pipeline là kỹ thuật song song ở mức lệnh (ILP: Instruction Level Parallelism)
- Một pipeline là đầy đủ nếu nó luôn nhận một lệnh mới tại mỗi chu kỳ đồng hồ
- Một pipeline là không đầy đủ nếu có nhiều giai đoạn trễ trong quá trình xử lý
- Số lượng giai đoạn của pipeline phụ thuộc vào thiết kế CPU:
  - 2, 3, 5 giai đoạn: pipeline đơn giản
  - 14 giai đoạn: Pen II, Pen III
  - 20 – 31 giai đoạn: Pen IV
  - 12 -15 giai đoạn: Core

Giới thiệu về CPU Pipeline

Số lượng giai đoạn

- Thời gian thực hiện của các giai đoạn:
  - Mọi giai đoạn nên có thời gian thực hiện bằng nhau
  - Các giai đoạn chậm nên chia ra
- Lựa chọn số lượng giai đoạn:
  - Theo lý thuyết, số lượng giai đoạn càng nhiều thì hiệu năng càng cao
  - Nếu pipeline dài mà rỗng vì một số lý do, sẽ mất nhiều thời gian để làm đầy pipeline

Các vấn đề (rủi ro: hazard) của Pipeline



- Đầu ra mong muốn luôn là 0 (false)
- Nhưng trong một số trường hợp, đầu ra là 1 (true)

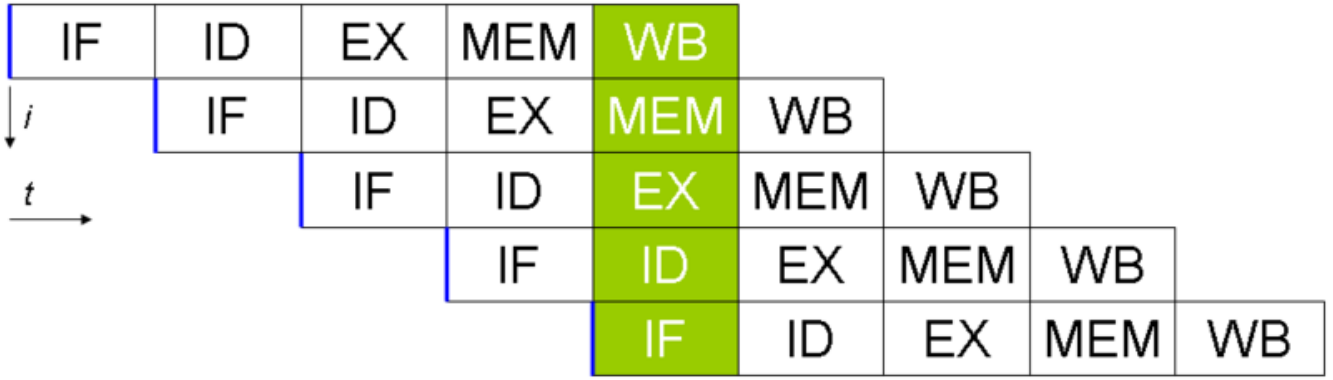
⇒ Hazard

Các vấn đề của Pipeline

- Vấn đề xung đột tài nguyên (resource conflict)
  - Xung đột truy cập bộ nhớ
  - Xung đột truy cập thanh ghi
- Xung đột/ tranh chấp dữ liệu (data hazard)
  - Hầu hết là RAW hay Read After Write Hazard
- Các lệnh rẽ nhánh (Branch Instruction)
  - Không điều kiện
  - Có điều kiện
  - Gọi thực hiện và trở về từ chương trình con

Xung đột tài nguyên

- Tài nguyên không đủ
- Ví dụ: nếu bộ nhớ chỉ hỗ trợ một thao tác đọc/ ghi tại một thời điểm, pipeline yêu cầu 2 truy cập bộ nhớ 1 lúc (đọc lệnh tại giai đoạn IF và đọc dữ liệu tại ID) -> nảy sinh xung đột



Xung đột tài nguyên

- Giải pháp:
  - Nâng cao khả năng tài nguyên
  - Memory/ cache: hỗ trợ nhiều thao tác đọc/ ghi cùng lúc
  - Chia cache thành cache lệnh và cache dữ liệu để cải thiện truy nhập

Xung đột dữ liệu

- Xét 2 lệnh sau:

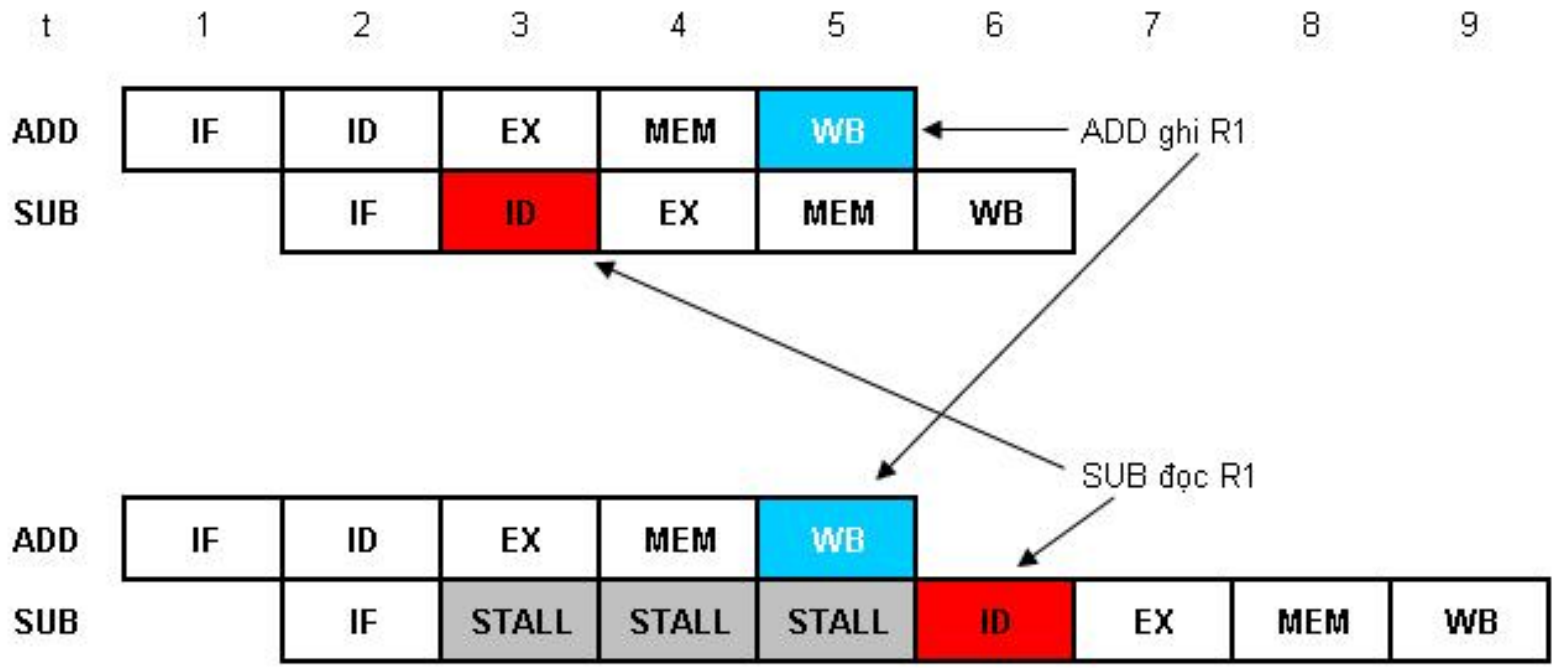
  ADD R1, R1, R3; R1 ← R1 + R3

  SUB R4, R1, R2; R4 ← R1 -R2

- SUB sử dụng kết quả lệnh ADD: có phụ thuộc dữ liệu giữa 2 lệnh này
- SUB đọc R1 tại giai đoạn 2 (ID); trong khi đó ADD lưu kết quả tại giai đoạn 5 (WB)
  - SUB đọc giá trị cũ của R1 trước khi ADD lưu trữ giá trị mới vào R1

⇒ *Dữ liệu chưa sẵn sàng cho các lệnh phụ thuộc tiếp theo*

Pipeline hazard – xung đột dữ liệu



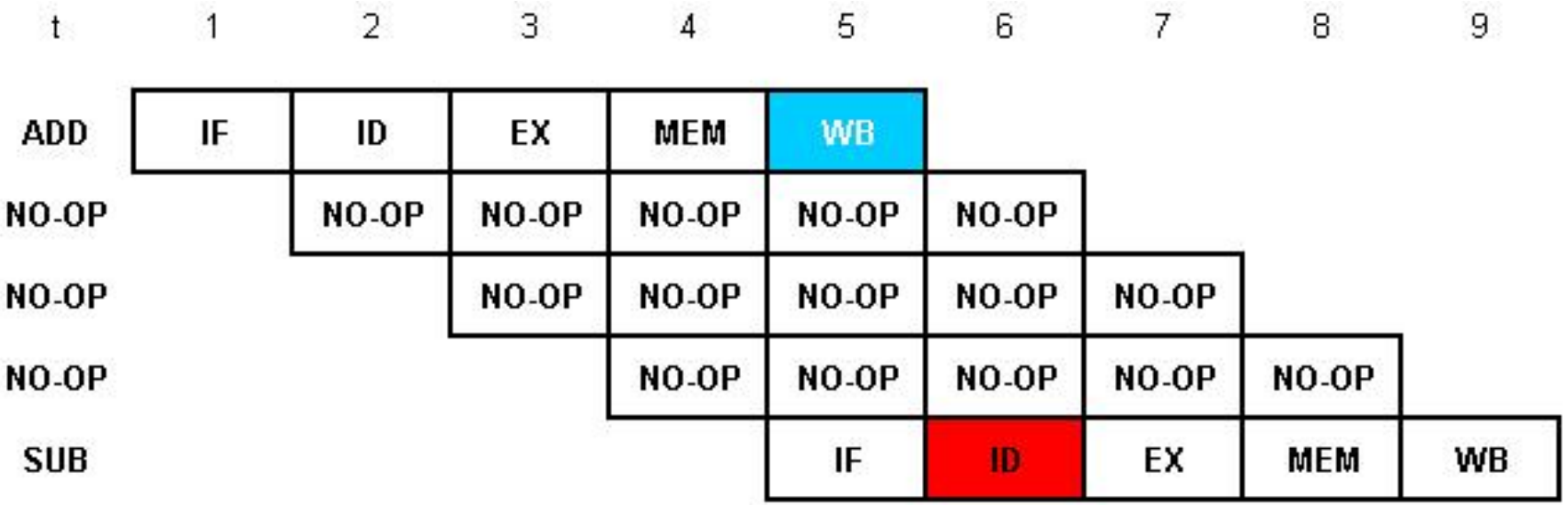
ADD R1, R1, R3; R1 ← R1 + R3

SUB R4, R1, R2; R4 ← R1 - R2

Hướng khắc phục xung đột dữ liệu

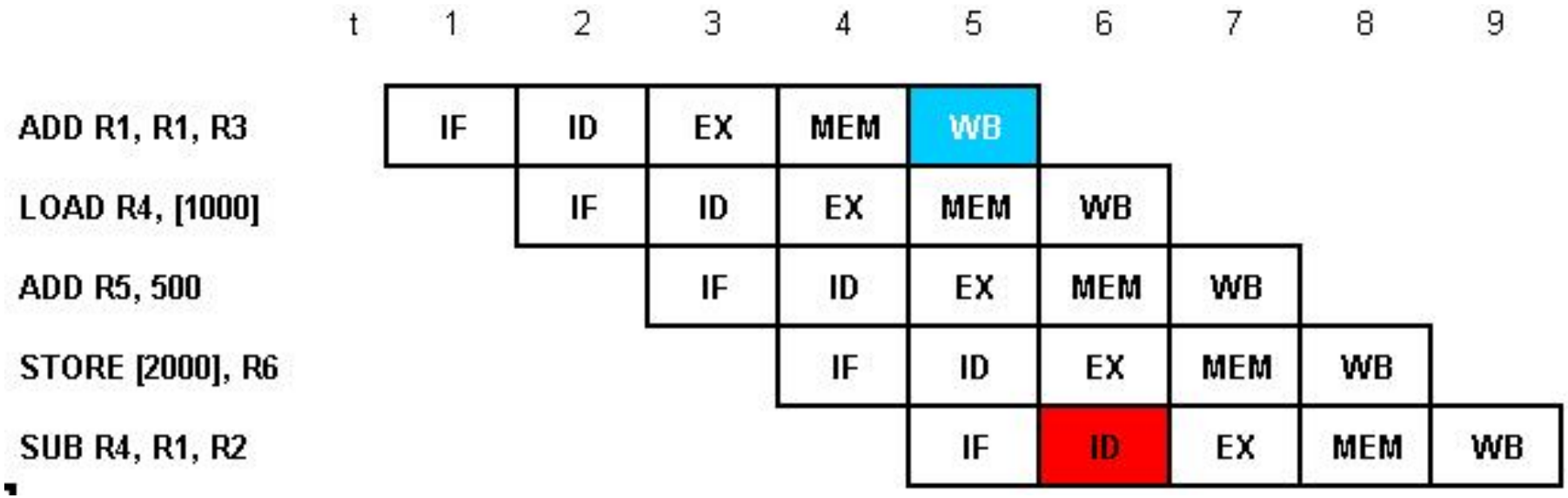
- Nhận biết nó xảy ra
- Ngưng pipeline (stall): phải làm trễ hoặc ngưng pipeline bằng cách sử dụng một vài phương pháp tới khi có dữ liệu chính xác
- Sử dụng complier để nhận biết RAW và:
  - Chèn các lệnh NO-OP vào giữa các lệnh có RAW
  - Thay đổi trình tự các lệnh trong chương trình và chèn các lệnh độc lập dữ liệu vào vị trí giữa 2 lệnh có RAW
- Sử dụng phần cứng để xác định RAW (có trong các CPUs hiện đại) và dự đoán trước giá trị dữ liệu phụ thuộc

Hướng khắc phục xung đột dữ liệu



- Làm trễ quá trình thực hiện lệnh SUB bằng cách chèn 3 NO-OP

Hướng khắc phục xung đột dữ liệu



- Chèn 3 lệnh độc lập dữ liệu vào giữa ADD và SUB

Ví dụ

- Viết chương trình tính:

a = b + c;

d = e + f;

Bài Tập

- Xác định lỗi và bố trí lại các câu lệnh tránh trì hoãn khi thiết kế pipeline cho các câu lệnh sau:

  load R_1, 0(R_0)

  load R_2, 4(R_0)

  Add R_3, R_1,R_2

  Store 12(R_0), R3

  load R_4, 8(R_0)

  Add R_5,R_1,R_4

  Store 16(R_0), R5

Bài Tập

- Xác định lỗi và bố trí lại các câu lệnh tránh trì hoãn khi thiết kế pipeline cho các câu lệnh sau:

  load R_1, $0(R_0)$

  load R_2, $4(R_0)$

  Add R_3, R_1,R_2

  Store R_3, $12(R_0)$

  load R_4, $8(R_0)$

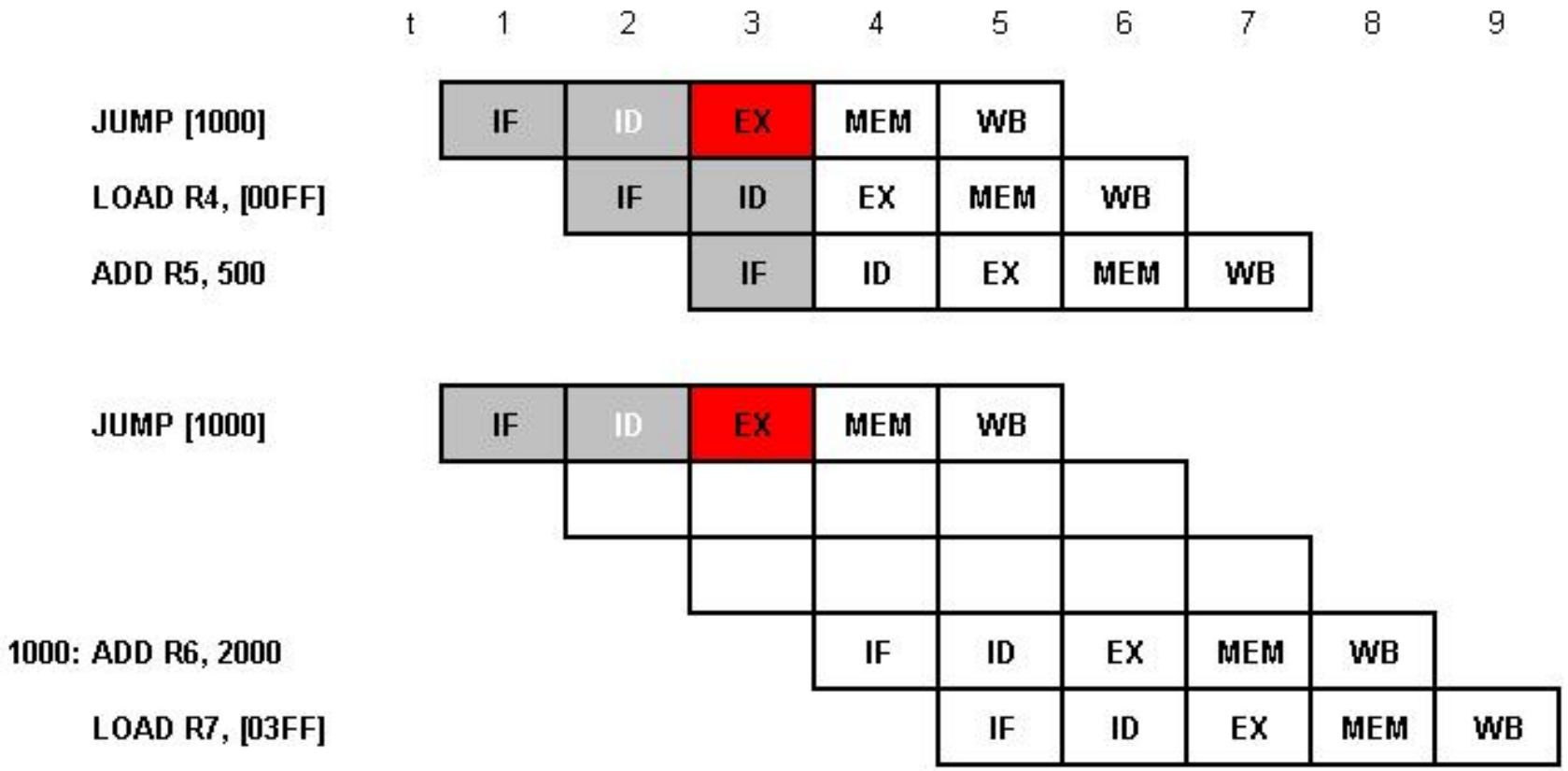
  Add R_5,R_1,R_4

  Store R_5,$16(R_0)$

Quản lý các lệnh rẽ nhánh trong pipeline

- Tỷ lệ các lệnh rẽ nhánh chiếm khoảng 10 - 30%. Các lệnh rẽ nhánh có thể gây ra:
  - Gián đoạn trong quá trình chạy bình thường của chương trình
  - Làm cho Pipeline rỗng nếu không có biện pháp ngăn chặn hiệu quả
- Với các CPU mà pipeline dài (P4 với 31 giai đoạn) và nhiều pipeline chạy song song, vấn đề rẽ nhánh càng trở nên phức tạp hơn vì:
  - Phải đẩy mọi lệnh đang thực hiện ra ngoài pipeline khi gặp lệnh rẽ nhánh
  - Tải mới các lệnh từ địa chỉ rẽ nhánh vào pipeline. Tiêu tốn nhiều thời gian để điền đầy pipeline

Quản lý các lệnh rẽ nhánh



- Khi 1 lệnh rẽ nhánh được thực hiện, các lệnh tiếp theo bị đẩy ra khỏi pipeline và các lệnh mới được tải

Giải pháp quản lý các lệnh rẽ nhánh

- Đích rẽ nhánh (branch target)
- Rẽ nhánh có điều kiện (conditional branches)
  - Làm chậm rẽ nhánh (delayed branching)
  - Dự báo rẽ nhánh (branch prediction)

Đích rẽ nhánh

- Khi một lệnh rẽ nhánh được thực hiện, lệnh tiếp theo được lấy là lệnh ở địa chỉ đích rẽ nhánh (target) chứ không phải lệnh tại vị trí tiếp theo lệnh nhảy

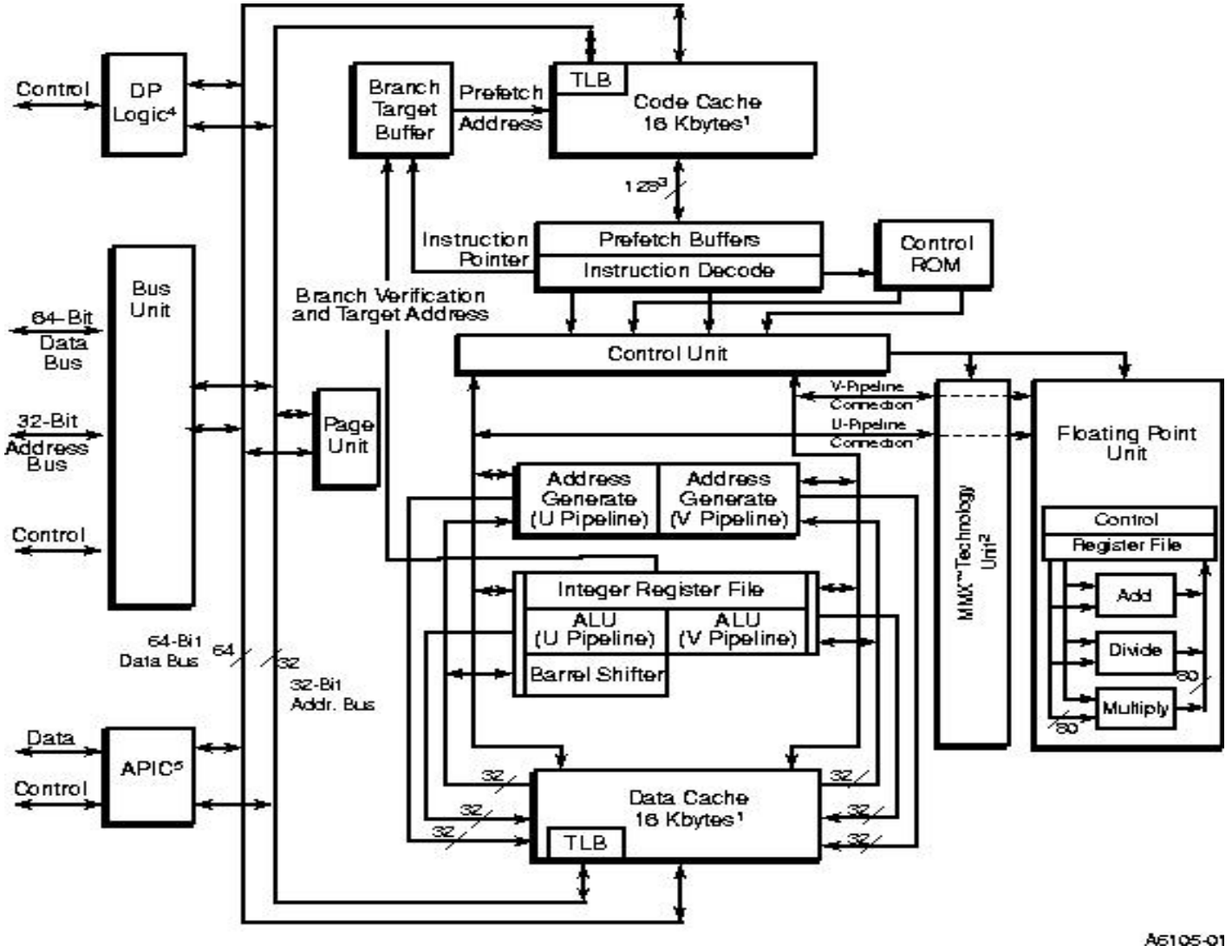
```
JUMP  <Address>
ADD R1, R2

Address: SUB R3, R4
```

Đích rẽ nhánh

- Các lệnh rẽ nhánh được xác định tại giai đoạn ID, vậy có thể biết trước chúng bằng cách giải mã trước
- Sử dụng đệm đích rẽ nhánh (BTB: branch target buffer) để lưu vết của các lệnh rẽ nhánh đã được thực thi:
  - Địa chỉ đích của các lệnh rẽ nhánh đã được thực hiện
  - Lệnh đích của các lệnh rẽ nhánh đã được thực hiện
- Nếu các lệnh rẽ nhánh được sử dụng lại (trong vòng lặp):
  - Các địa chỉ đích của chúng lưu trong BTB có thể được dùng mà không cần tính lại
  - Các lệnh đích có thể dùng trực tiếp không cần load lại từ bộ nhớ

  ⇒ Điều này có thể vì địa chỉ và lệnh đích thường không thay đổi

Đích rẽ nhánh của PIII



Lệnh rẽ nhánh có điều kiện

- Khó quản lý các lệnh rẽ nhánh có điều kiện hơn vì:
  - Có 2 lệnh đích để lựa chọn
  - Không thể xác định được lệnh đích tới khi lệnh rẽ nhánh được thực hiện xong
  - Sử dụng BTB riêng rẽ không hiệu quả vì phải đợi tới khi có thể xác định được lệnh đích.

Lệnh nhảy có điều kiện – các chiến lược

- Làm chậm rẽ nhánh
- Dự đoán rẽ nhánh

Làm chậm rẽ nhánh

- Dựa trên ý tưởng:
  - Lệnh rẽ nhánh không làm rẽ nhánh ngay lập tức
  - Mà nó sẽ bị làm chậm một vài chu kỳ đồng hồ phụ thuộc vào độ dài của pipeline
- Đặc điểm:
  - Hoạt động tốt trên các vi xử lý RISC trong đó các lệnh có thời gian xử lý bằng nhau
  - Pipeline ngắn (thông thường là 2 giai đoạn)
  - Lệnh sau lệnh nhảy luôn được thực hiện, không phụ thuộc vào kết quả lệnh rẽ nhánh

Làm chậm rẽ nhánh

- Cài đặt:
  - Sử dụng complier để chèn NO-OP vào vị trí ngay sau lệnh rẽ nhánh, hoặc
  - Chuyển một lệnh độc lập từ trước tới ngay sau lệnh rẽ nhánh

Làm chậm rẽ nhánh

- Xét các lệnh:

  ADD R2, R3, R4
  CMP R1,0
  JNE somewhere

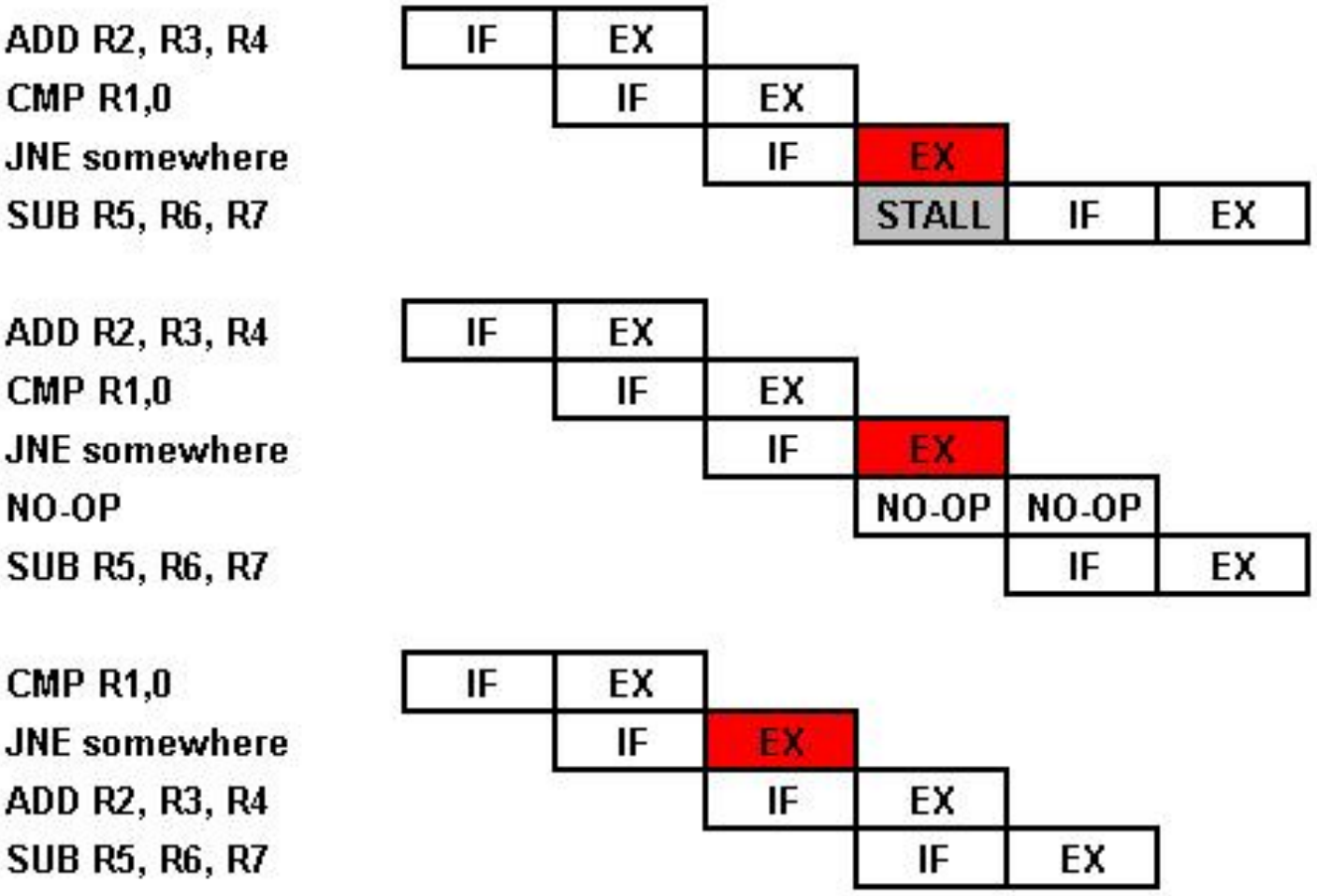
- Chèn NO-OP vào vị trí ngay sau lệnh rẽ nhánh

  ADD R2, R3, R4
  CMP R1,0
  JNE somewhere
  NO-OP

- Chuyển một lệnh độc lập từ trước tới ngay sau lệnh rẽ nhánh

  CMP R1,0
  JNE somewhere
  ADD R2, R3, R4

Làm chậm rẽ nhánh



Làm chậm rẽ nhánh – các nhận xét

- Dễ cài đặt nhờ tối ưu trình biên dịch (complier)
- Không cần phần cứng đặc biệt
- Nếu chỉ chèn NO-OP làm giảm hiệu năng khi pipeline dài
- Thay các lệnh NO-OP bằng các lệnh độc lập có thể làm giảm số lượng NO-OP cần thiết tới 70%

Làm chậm rẽ nhánh – các nhận xét

- Làm tăng độ phức tạp mã chương trình (code)
- Cần lập trình viên và người xây dựng trình biên dịch có mức độ hiểu biết sâu về pipeline vi xử lý => hạn chế lớn
- Giảm tính khả chuyển (portable) của mã chương trình vì các chương trình phải được viết hoặc biên dịch lại trên các nền VXL mới

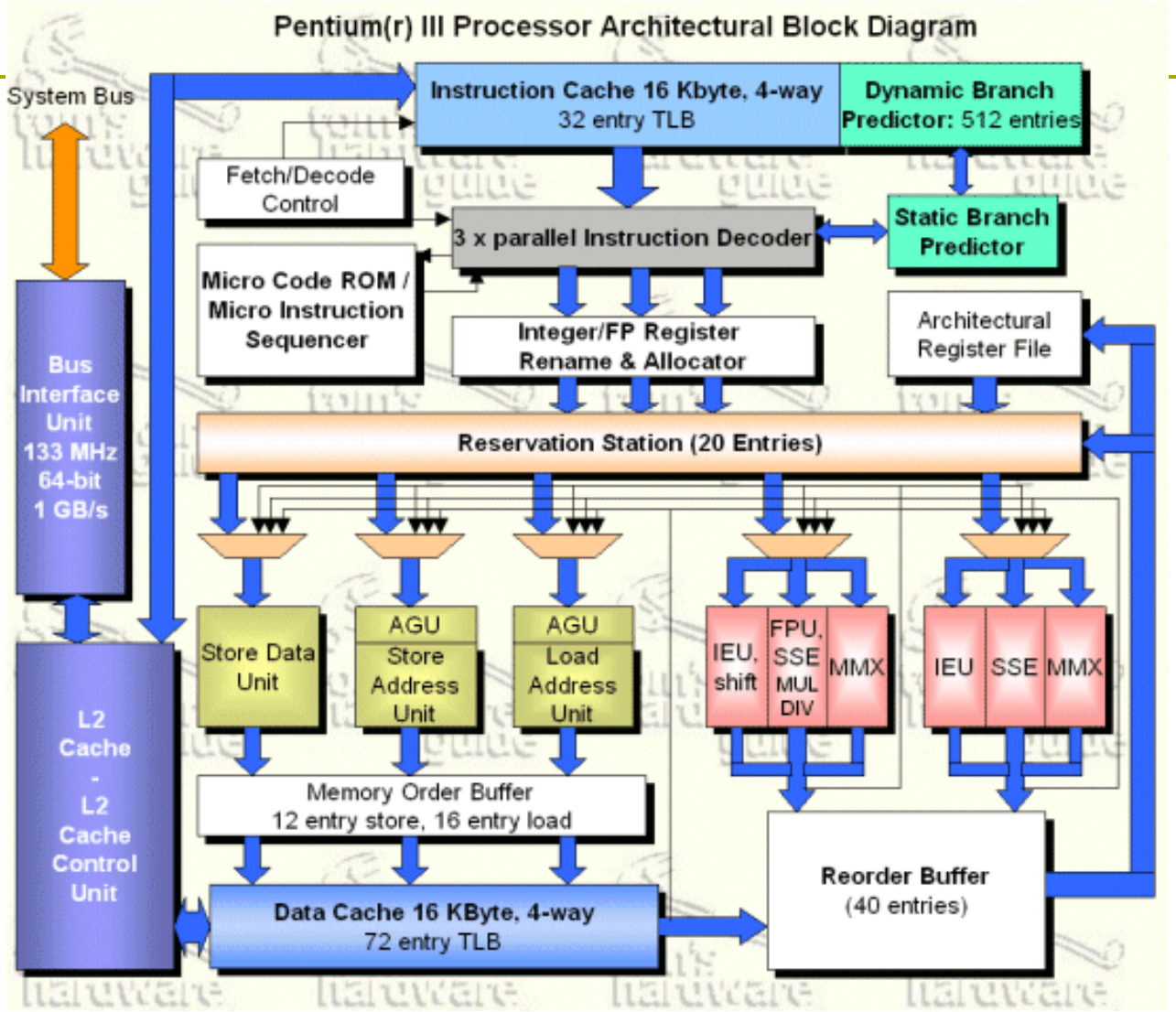
Dự đoán rẽ nhánh

- Có thể dự đoán lệnh đích của lệnh rẽ nhánh:
  - Dự đoán đúng: nâng cao hiệu năng
  - Dự đoán sai: đẩy các lệnh tiếp theo đã load và phải load lại các lệnh tại đích rẽ nhánh
  - Trường hợp xấu của dự đoán là 50% đúng và 50% sai

Dự đoán rẽ nhánh

- Các cơ sở để dự đoán:
  - Đối với các lệnh nhảy ngược (backward):
    - Thường là một phần của vòng lặp
    - Các vòng lặp thường được thực hiện nhiều lần
  - Đối với các lệnh nhảy xuôi (forward), khó dự đoán hơn:
    - Có thể là kết thúc lệnh loop
    - Có thể là nhảy có điều kiện

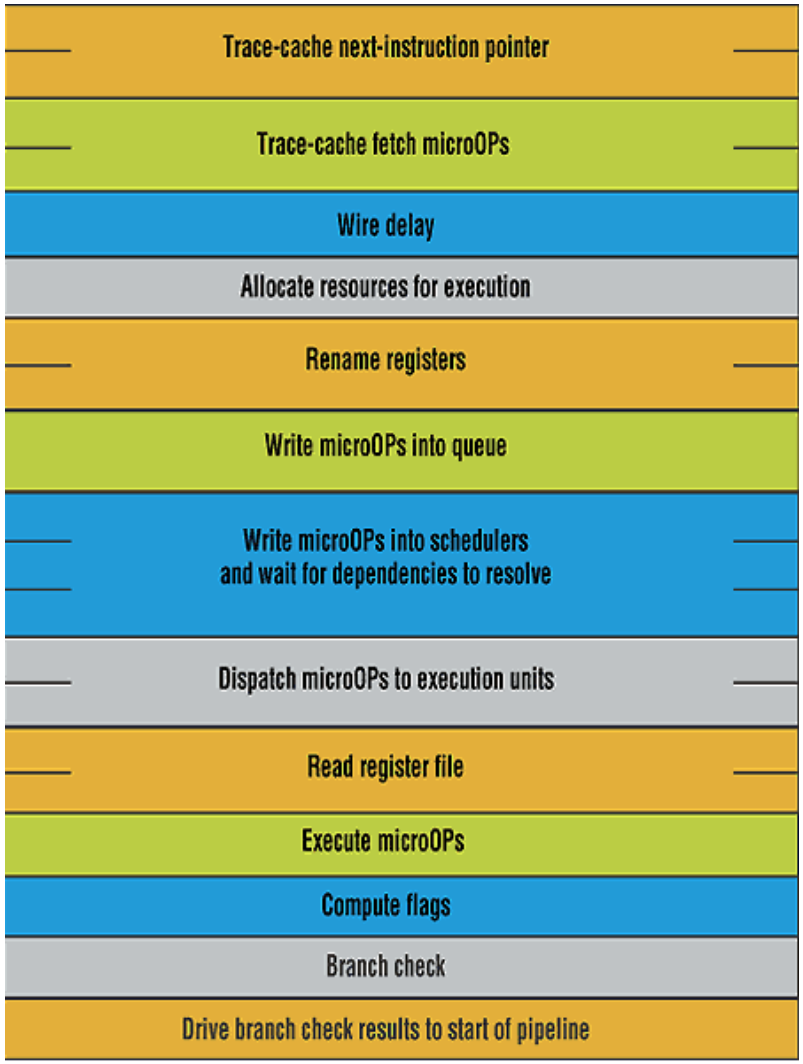
Branch Prediction – Intel PIII



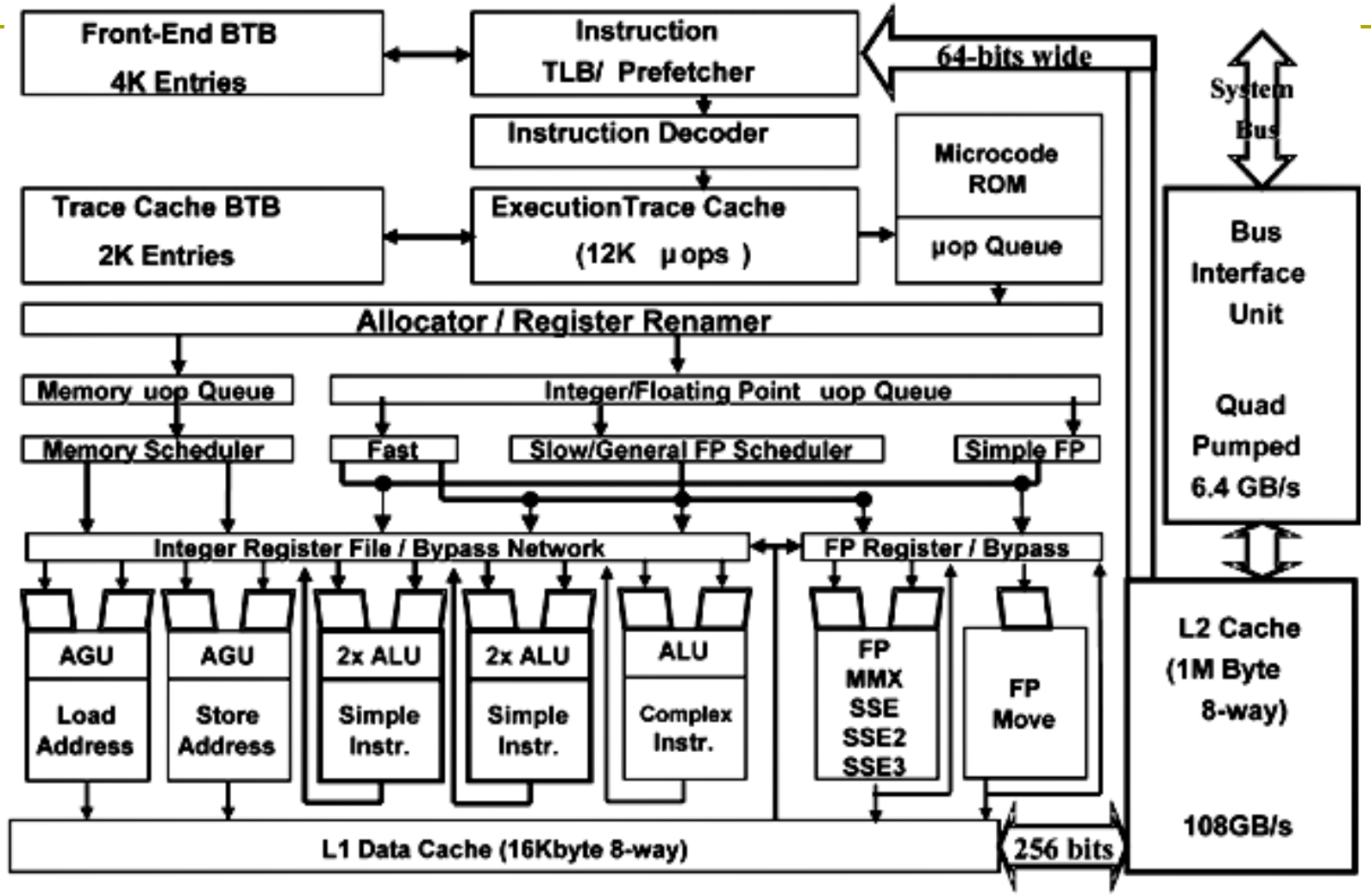
Siêu pipeline (superpipelining)

- Siêu pipeline là kỹ thuật cho phép:
  - Tăng độ sâu ống lệnh
  - Tăng tốc độ đồng hồ
  - Giảm thời gian trễ cho từng giai đoạn thực hiện lệnh
- Ví dụ: nếu giai đoạn thực hiện lệnh bởi ALU kéo dài -> chia thành một số giai đoạn nhỏ -> giảm thời gian chờ cho các giai đoạn ngắn
- Pentium 4 siêu ống với 20 giai đoạn

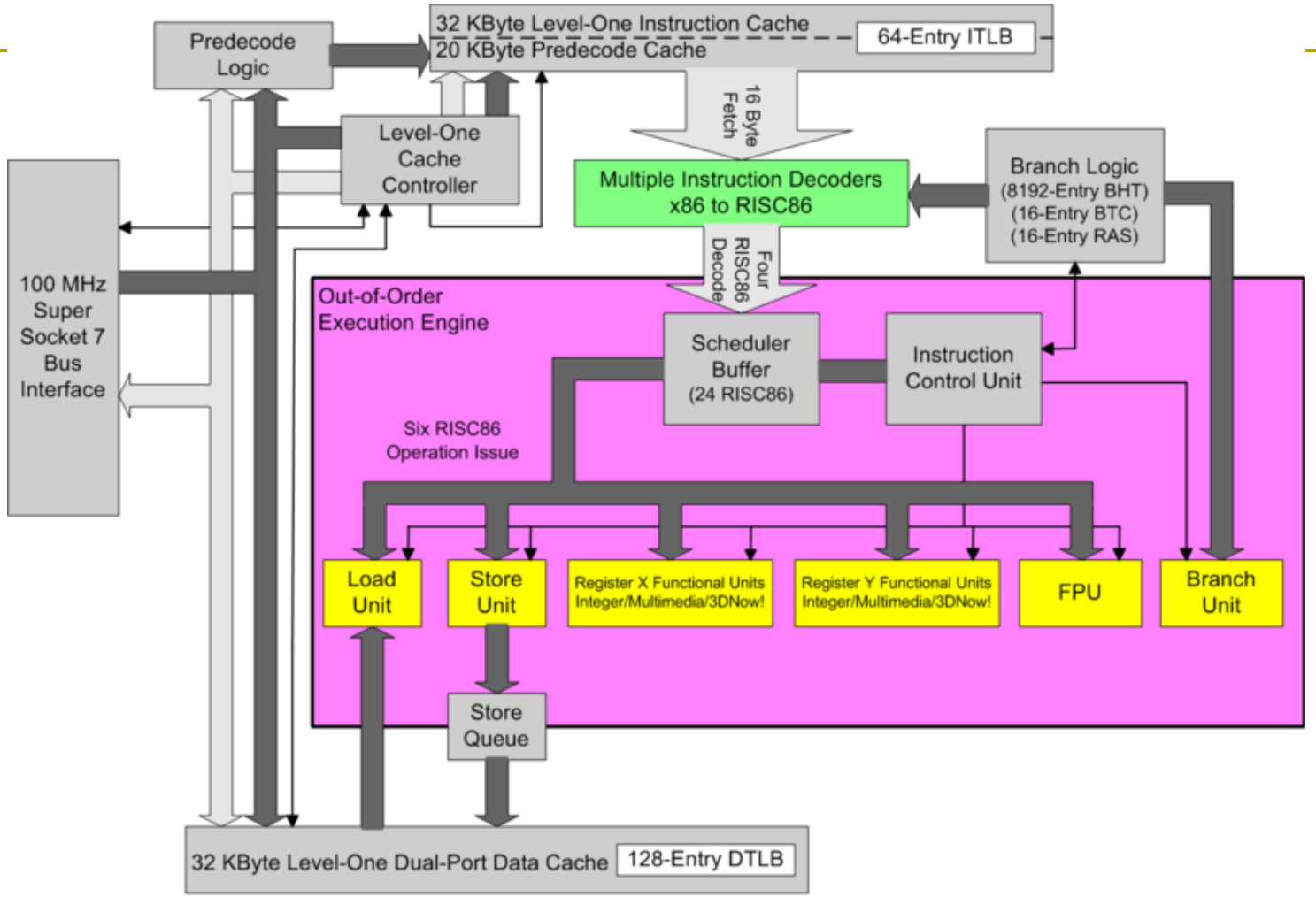
Pentium 4 siêu ống với 20 giai đoạn



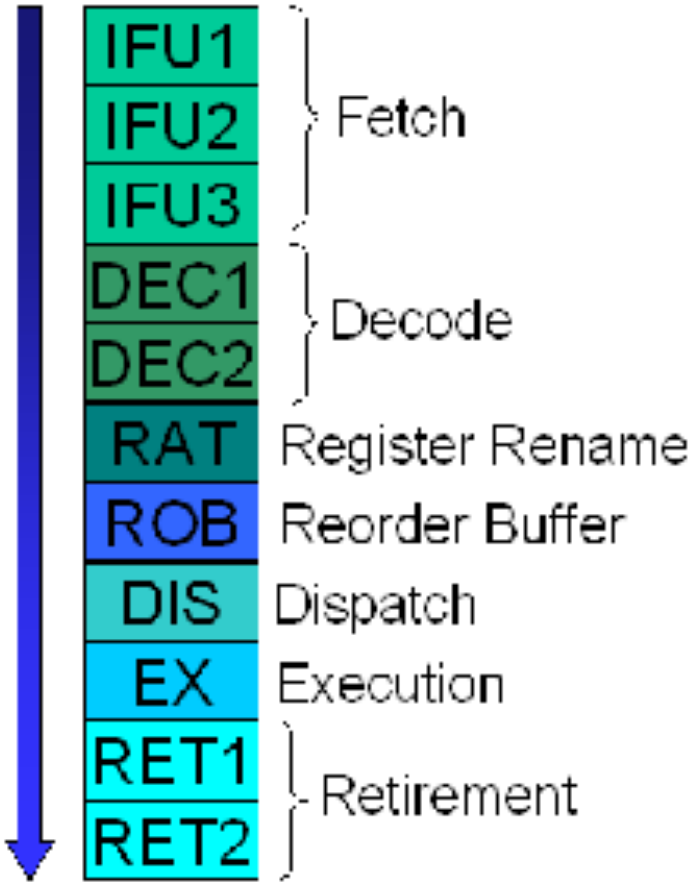
Branch Prediction – Intel P4



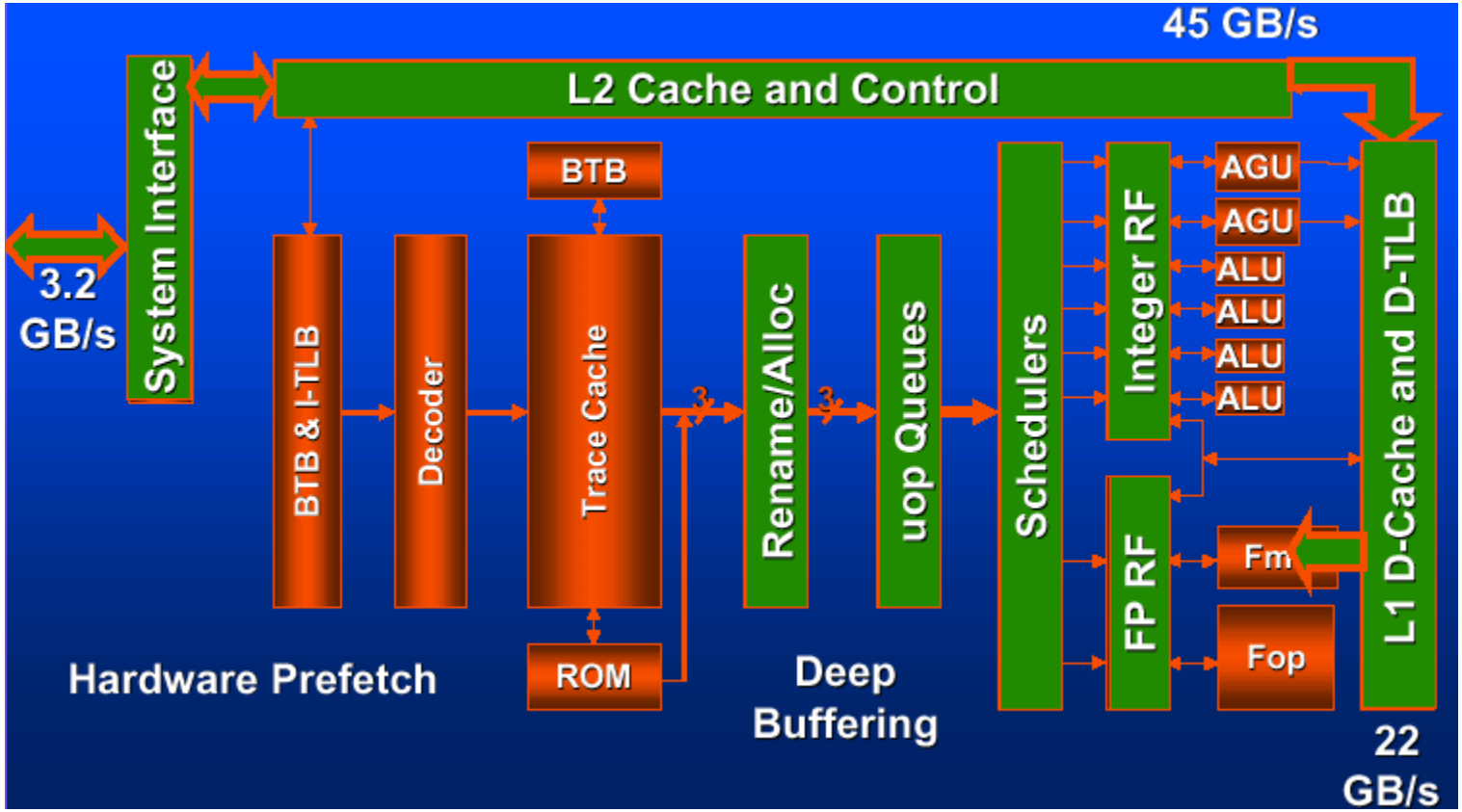
AMD K6-2 pipeline



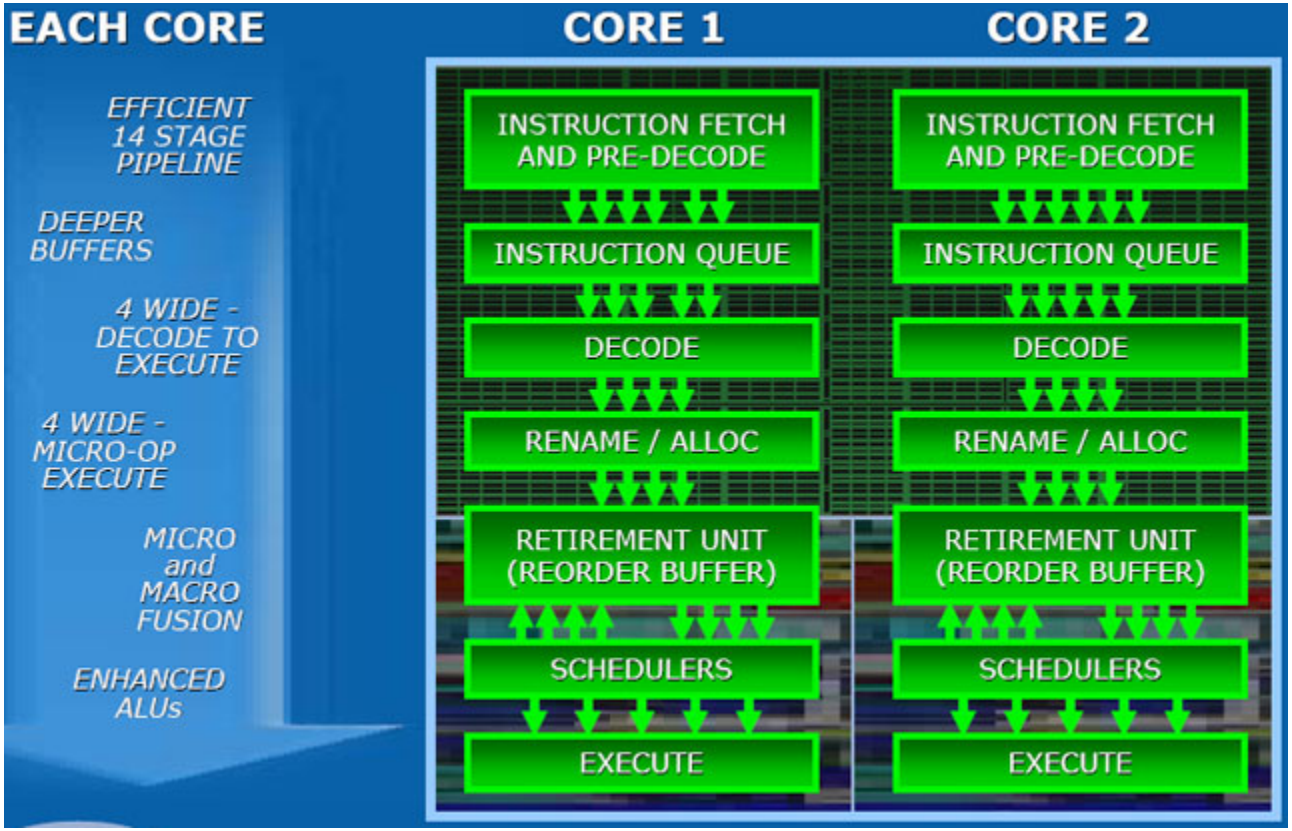
Pipeline –Pen III, M



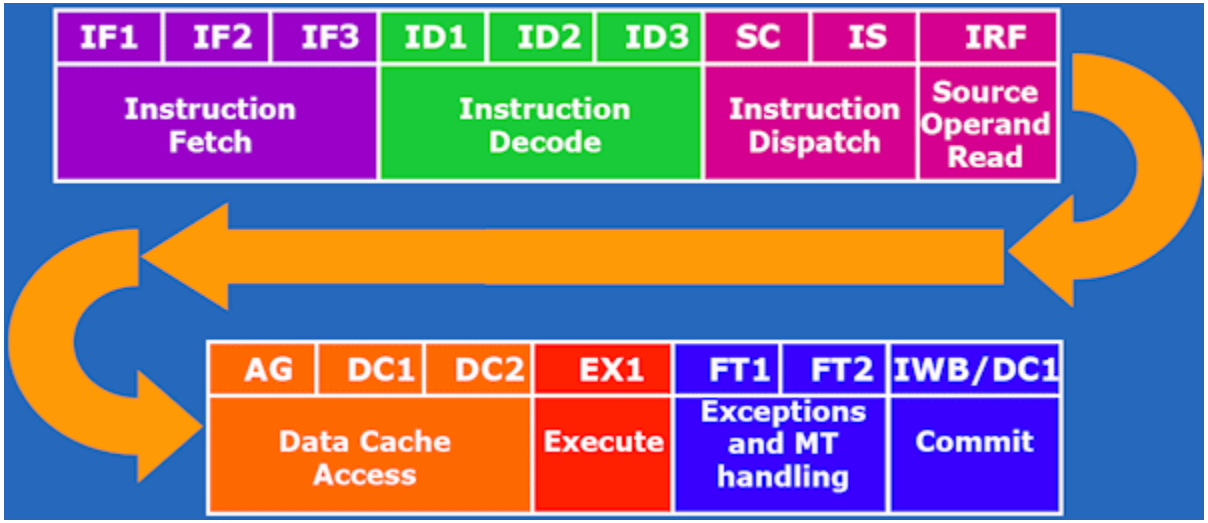
Intel Pen 4 Pipeline



Intel Core 2 Duo pipeline



Intel Atom 16-stage pipeline



Intel Core 2 Duo – Super Pipeline