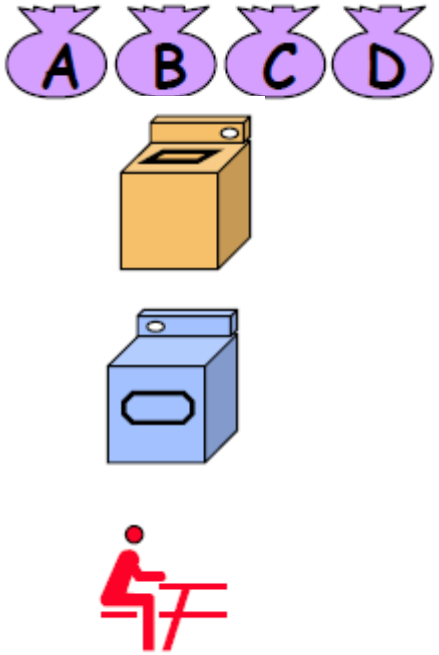
CPU pipeline

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

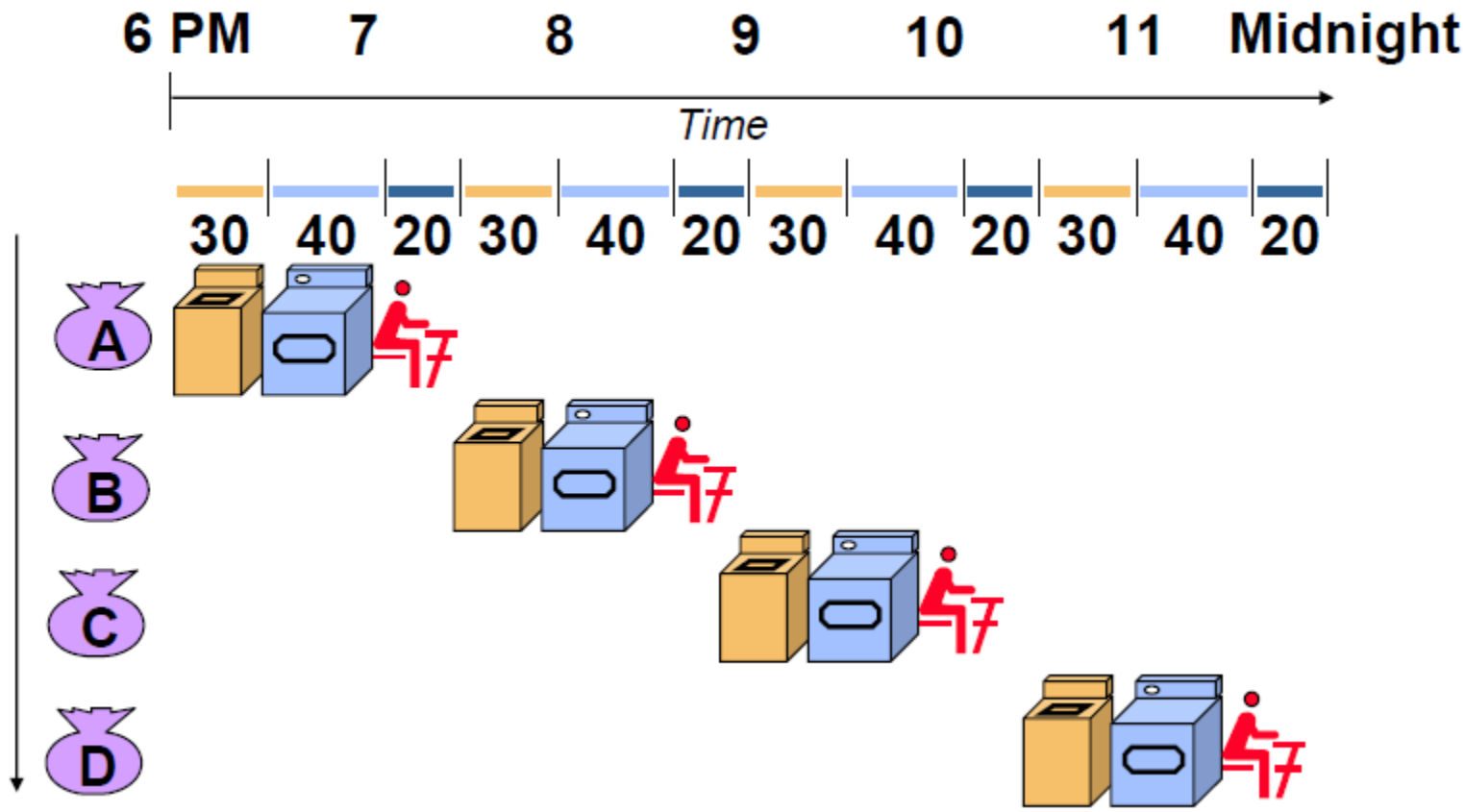
Pipeline – Ví dụ thực tế

- 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



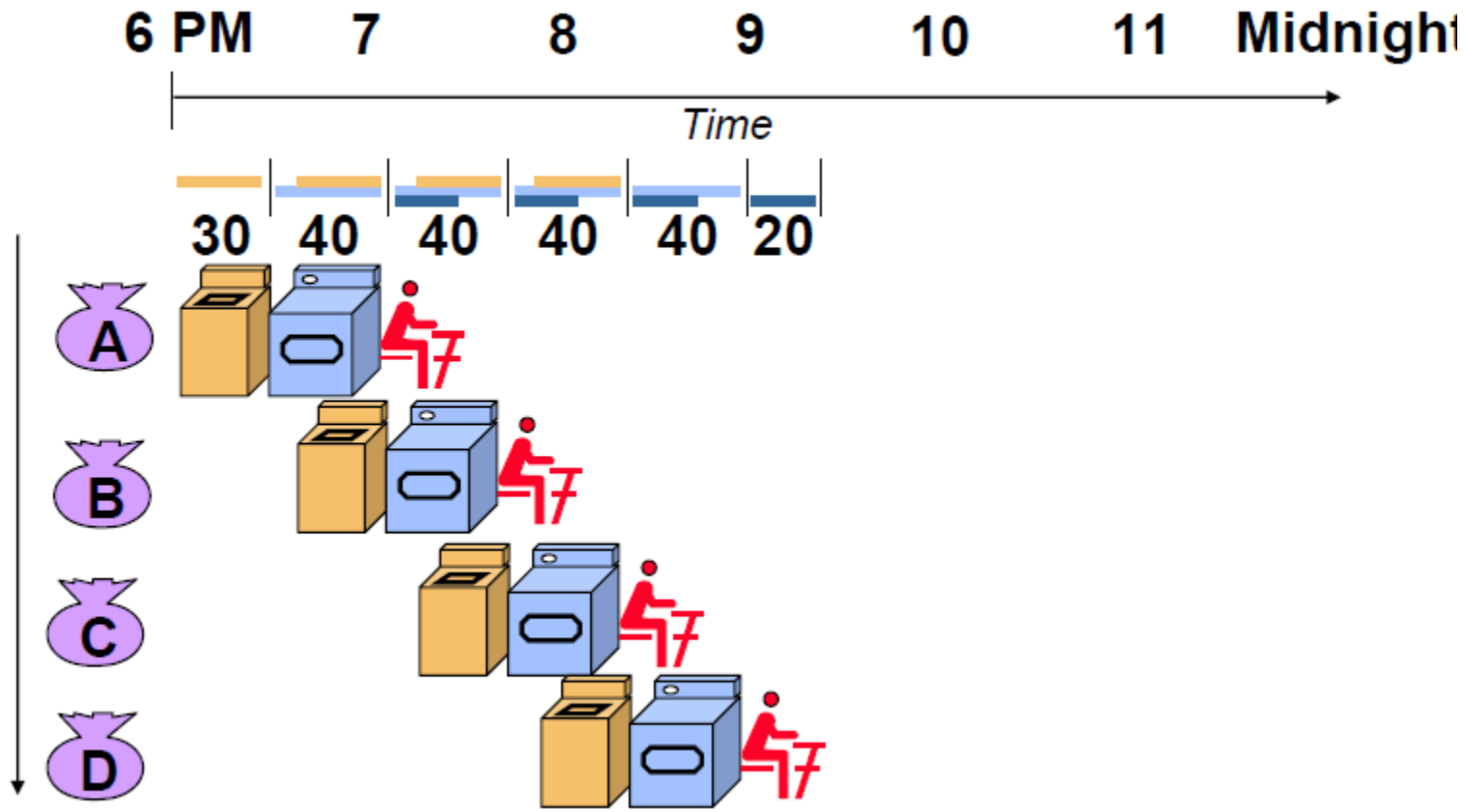
Pipeline – Ví dụ thực tế

Thực hiện tuần tự



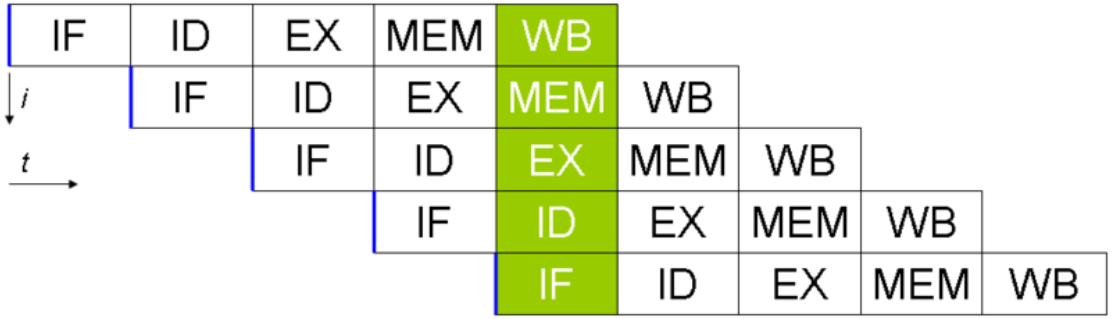
Pipeline – Ví dụ thực tế

Áp dụng pipeline

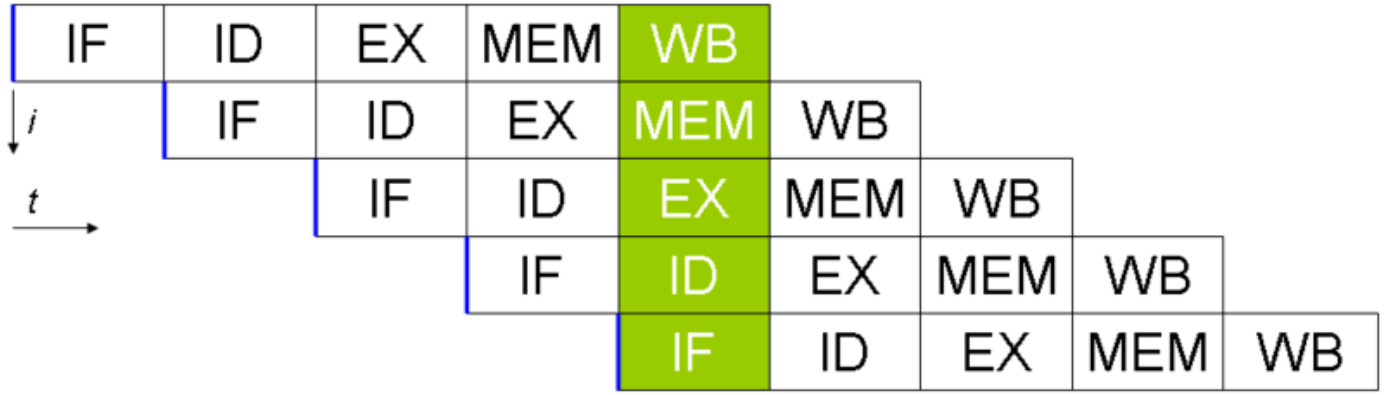


Giới thiệu về CPU Pipeline – Nguyên lý

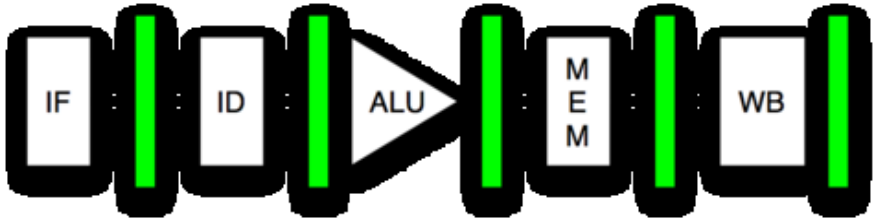
- Quá trình thực hiện lệnh được chia thành các giai đoạn
- 5 giai đoạn của hệ thống load – store:
  - Instruction fetch (IF): lấy lệnh từ bộ nhớ (hoặc cache)
  - Instruction Decode (ID): giải mã lệnh và lấy các toán hạng
  - Execute (EX): thực hiện lệnh: nếu là lệnh truy cập bộ nhớ thì tính toán địa 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

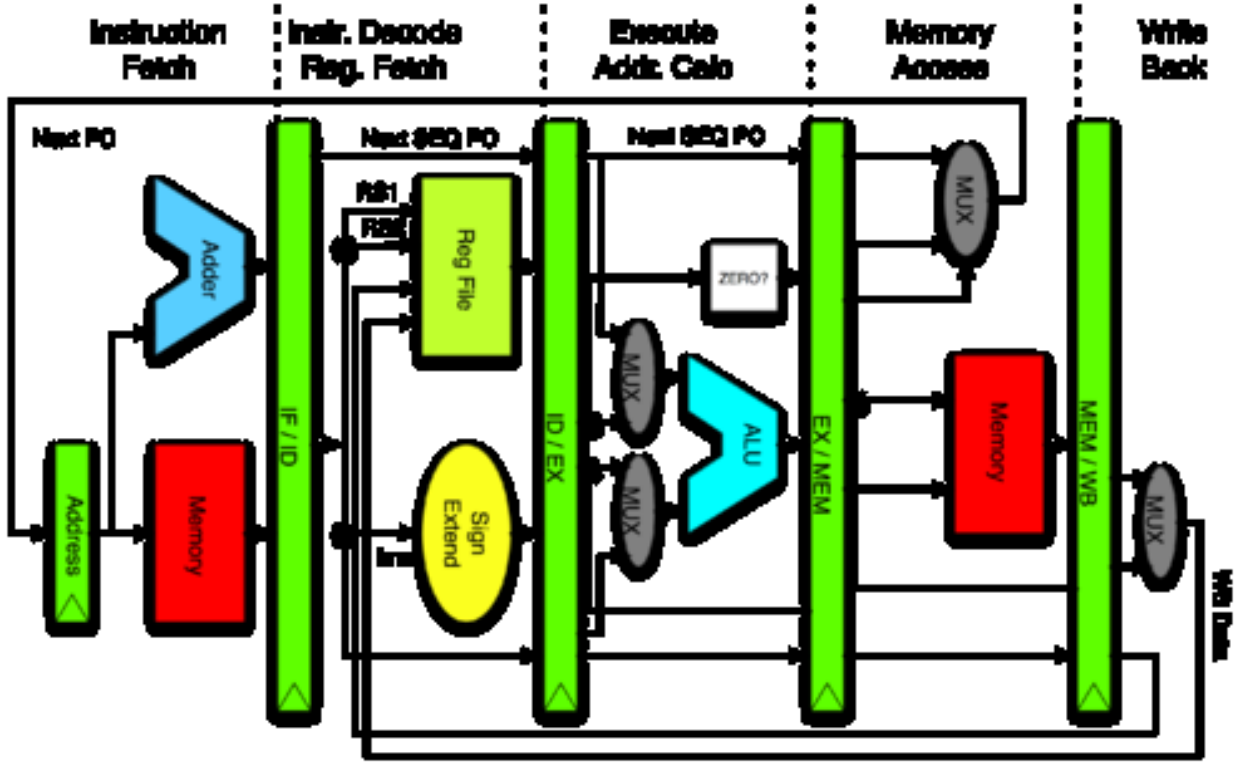


Giới thiệu về CPU Pipeline – Nguyên lý



Giới thiệu về CPU Pipeline – Nguyên lý





Giới thiệu về CPU Pipeline – Đặc điểm

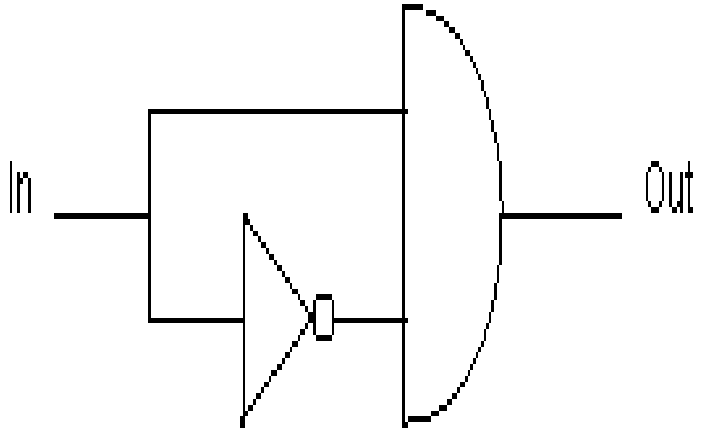
- Pipeline là kỹ thuật song song ở mức lệnh (ILP: Instruction Level Parallelism)
- Một pipeline là đầy đủ nếu nó luôn nhận một lệnh mới tại mỗi chu kỳ đồng hồ
- Một pipeline là không đầy đủ nếu có nhiều giai đoạn trễ trong quá trình xử lý
- Số lượng giai đoạn của pipeline phụ thuộc vào thiết kế CPU:
  - 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)

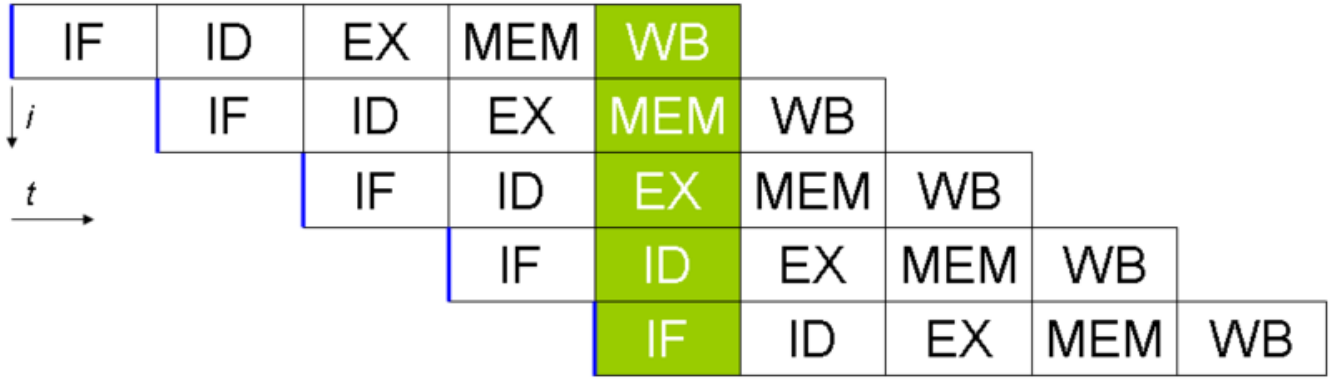
⇒ Hazard

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:

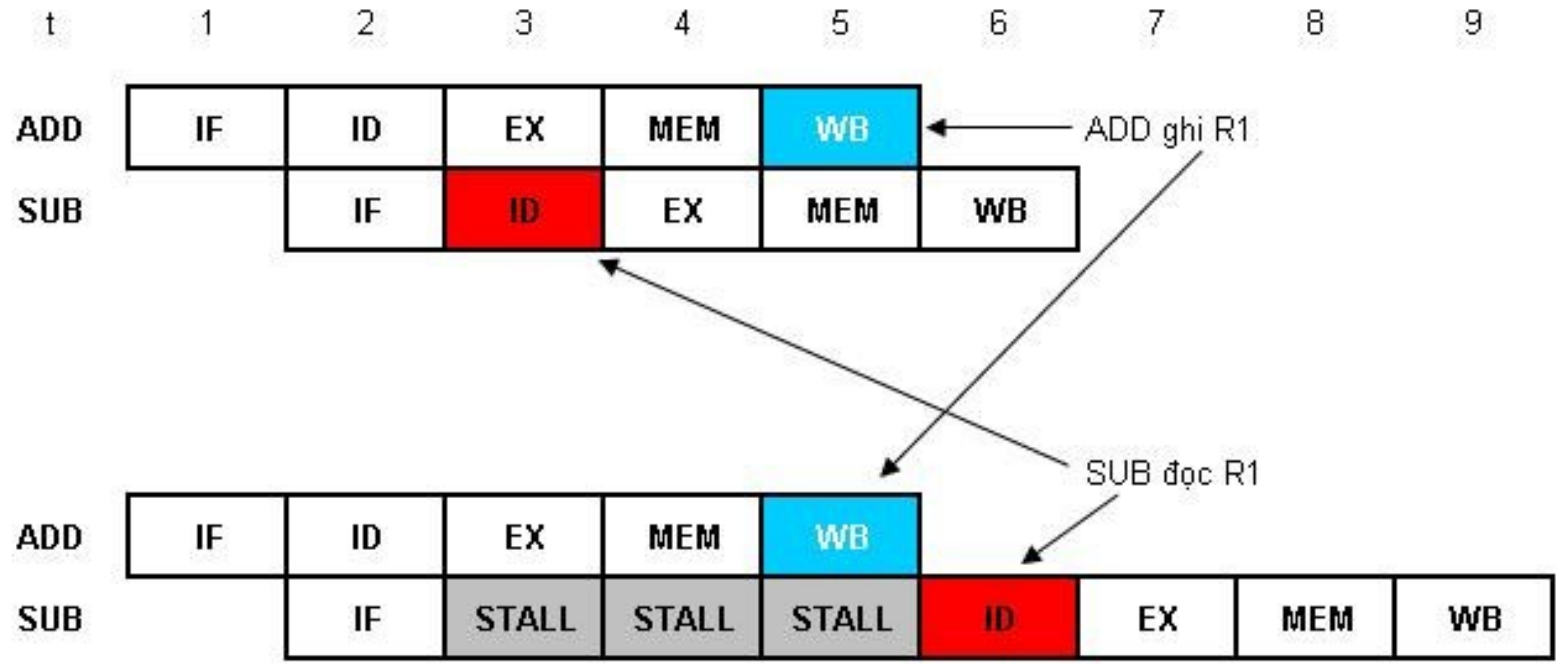
  ADD R1, R1, R3; R1 ← R1 + R3

  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



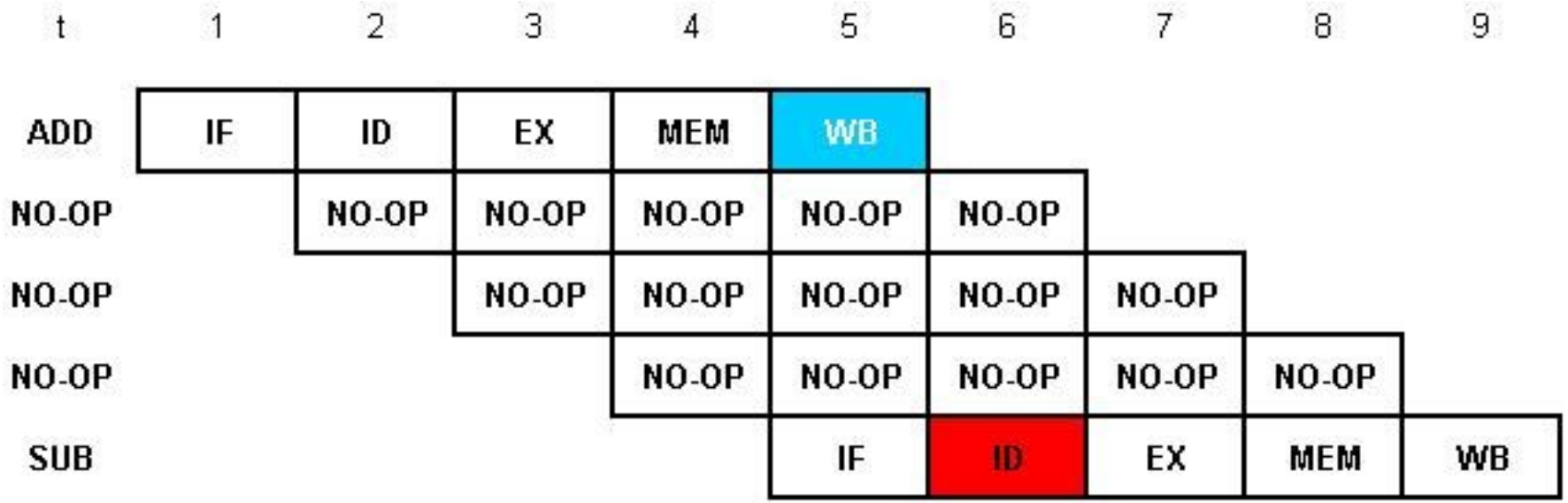
ADD R1, R1, R3; R1 ← R1 + R3

SUB R4, R1, R2; R4 ← R1 - R2

Hướng khắc phục xung đột dữ liệu

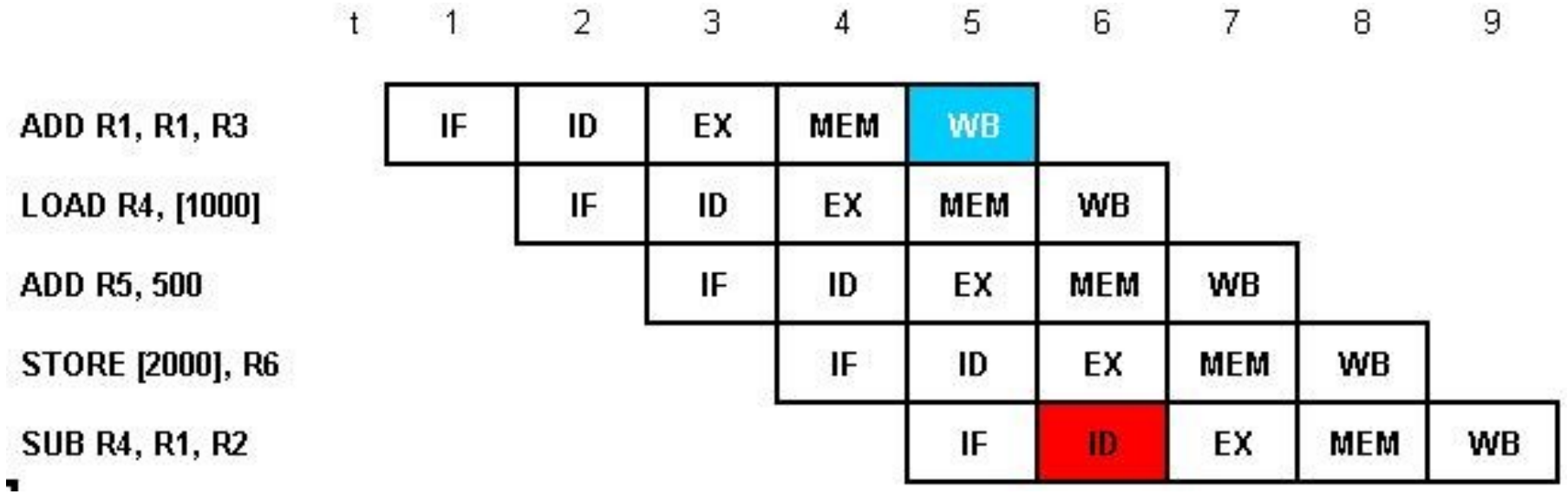
- Nhận biết nó xảy ra
- Ngưng pipeline (stall): phải làm trễ hoặc ngưng pipeline bằng cách sử dụng một vài phương pháp tới khi có dữ liệu chính xác
- Sử dụng complier để nhận biết RAW – (Read after Write) và:
  - Chèn các lệnh NO-OP vào giữa các lệnh có RAW
  - Thay đổi trình tự các lệnh trong chương 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 R_3, $12(R_0)$

  load R_4, $8(R_0)$

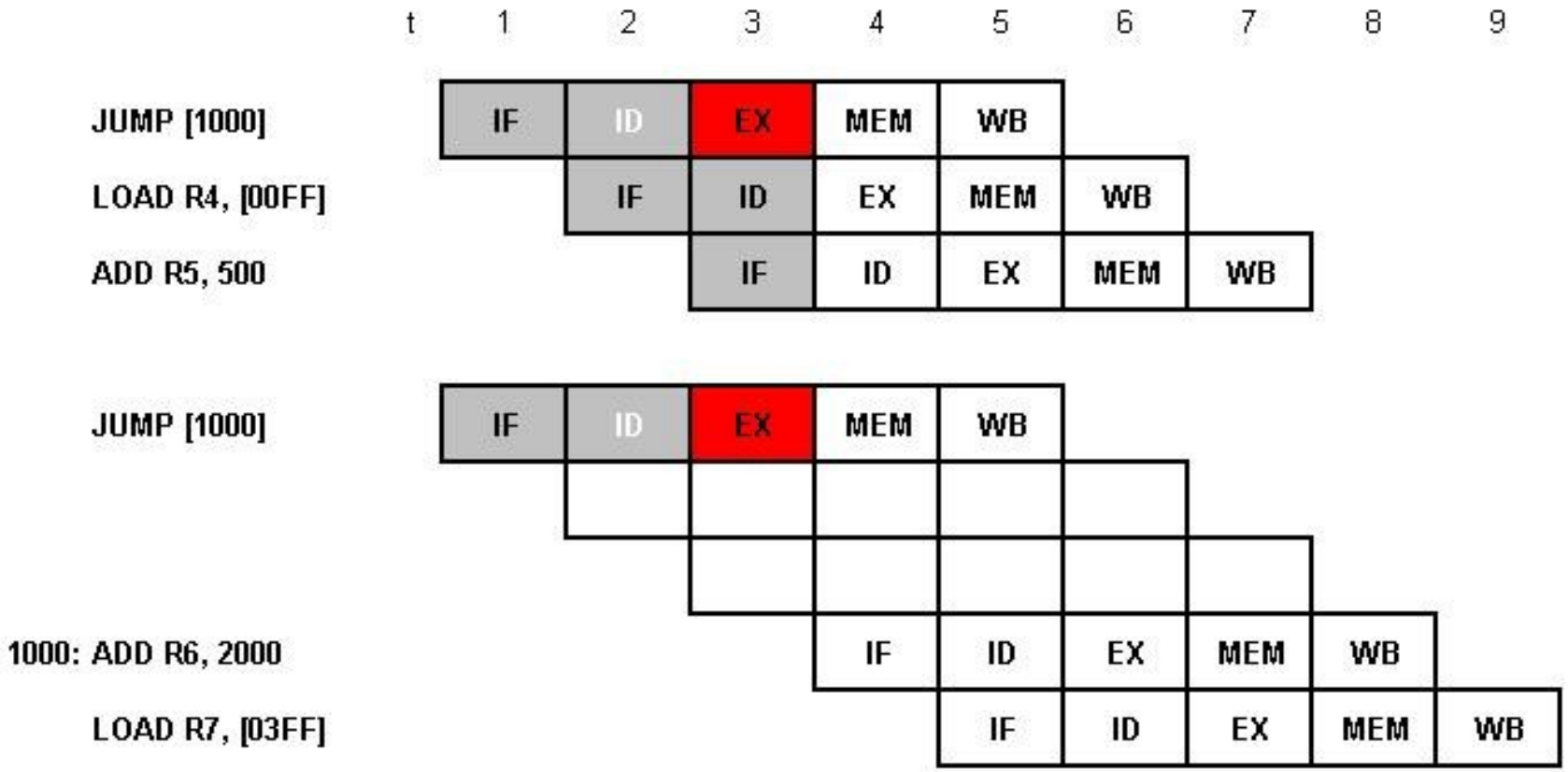
  Add R_5,R_1,R_4

  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

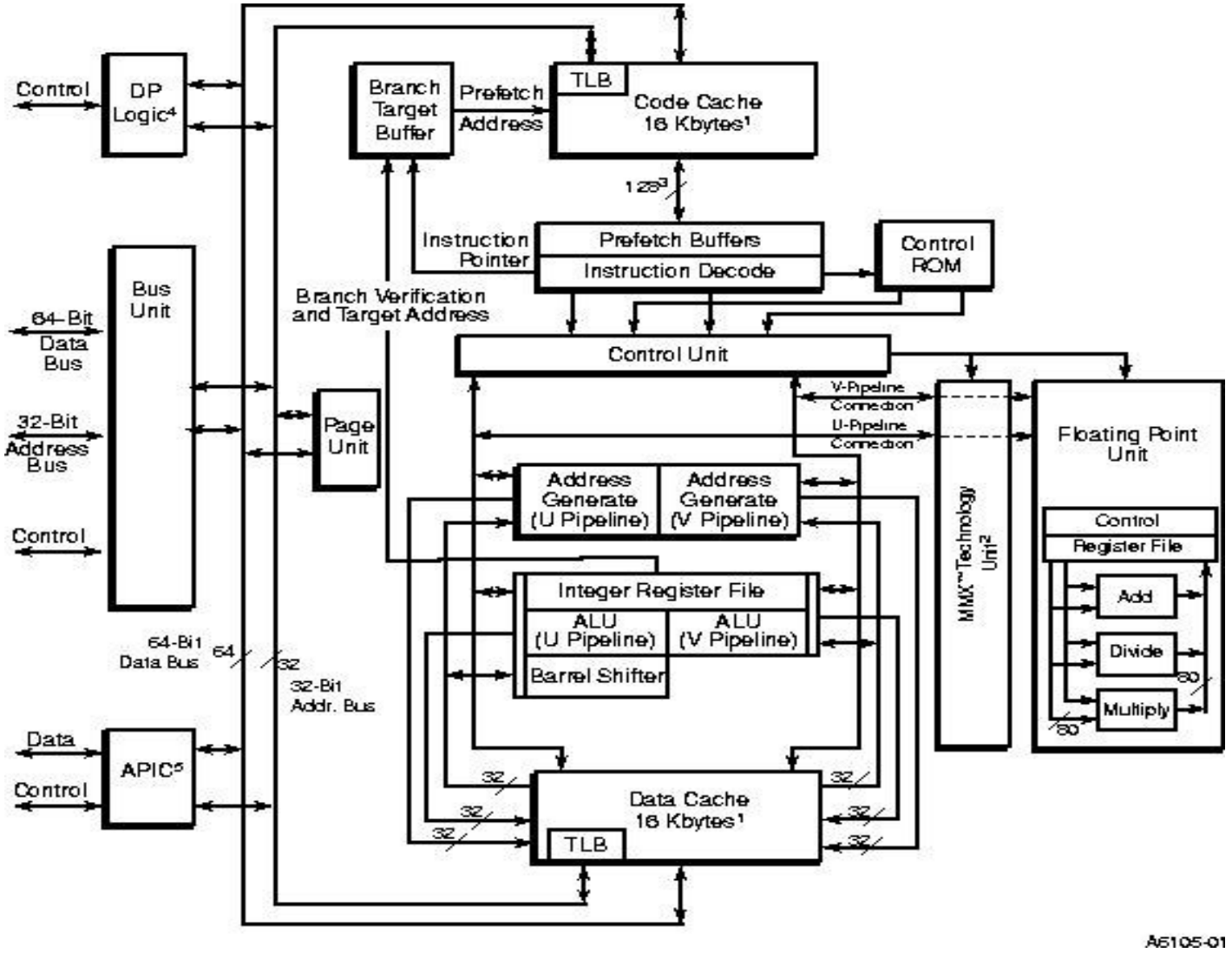
```
JUMP  <Address>
ADD R1, R2

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

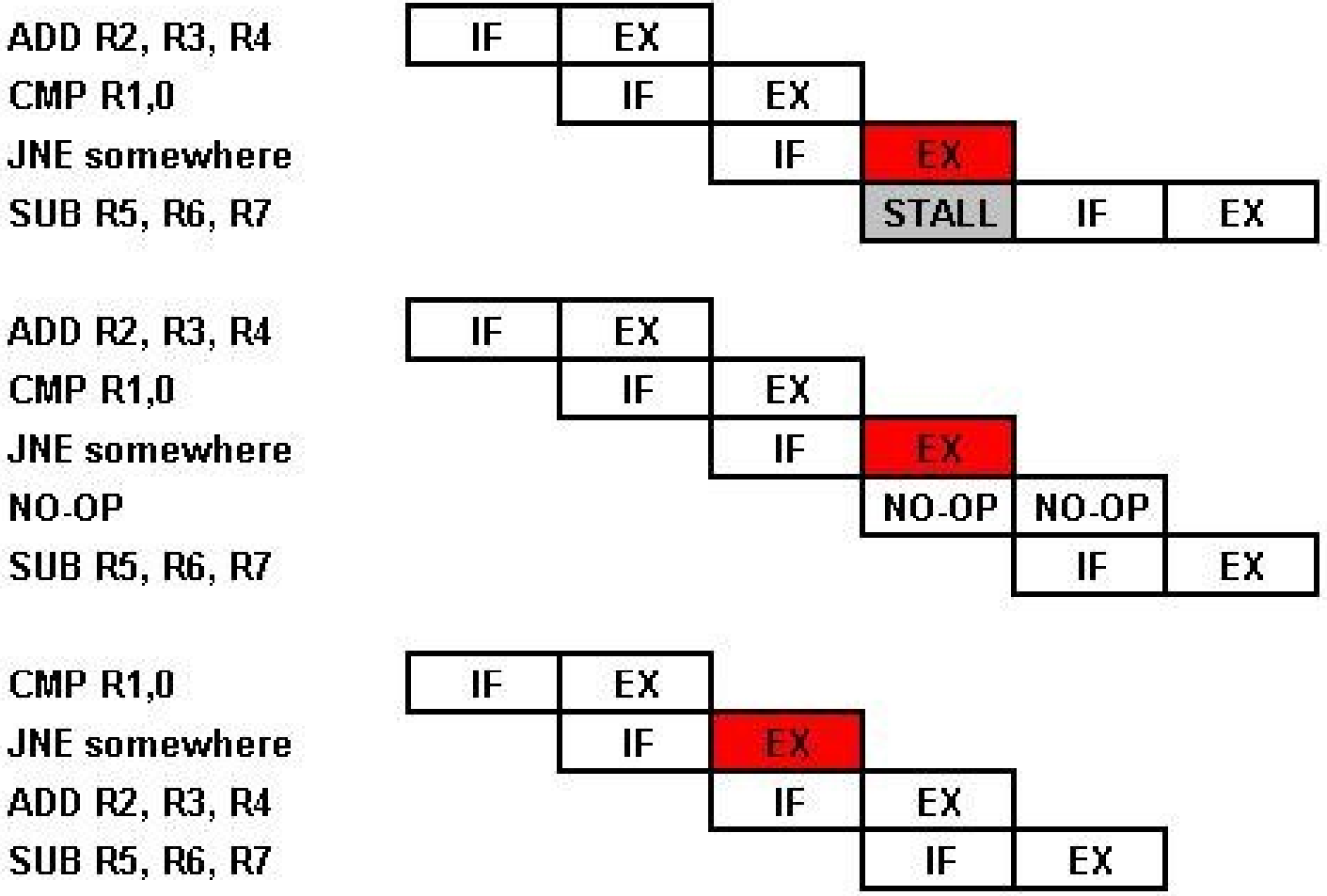
- Chèn NO-OP vào vị trí ngay sau lệnh rẽ nhánh

  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

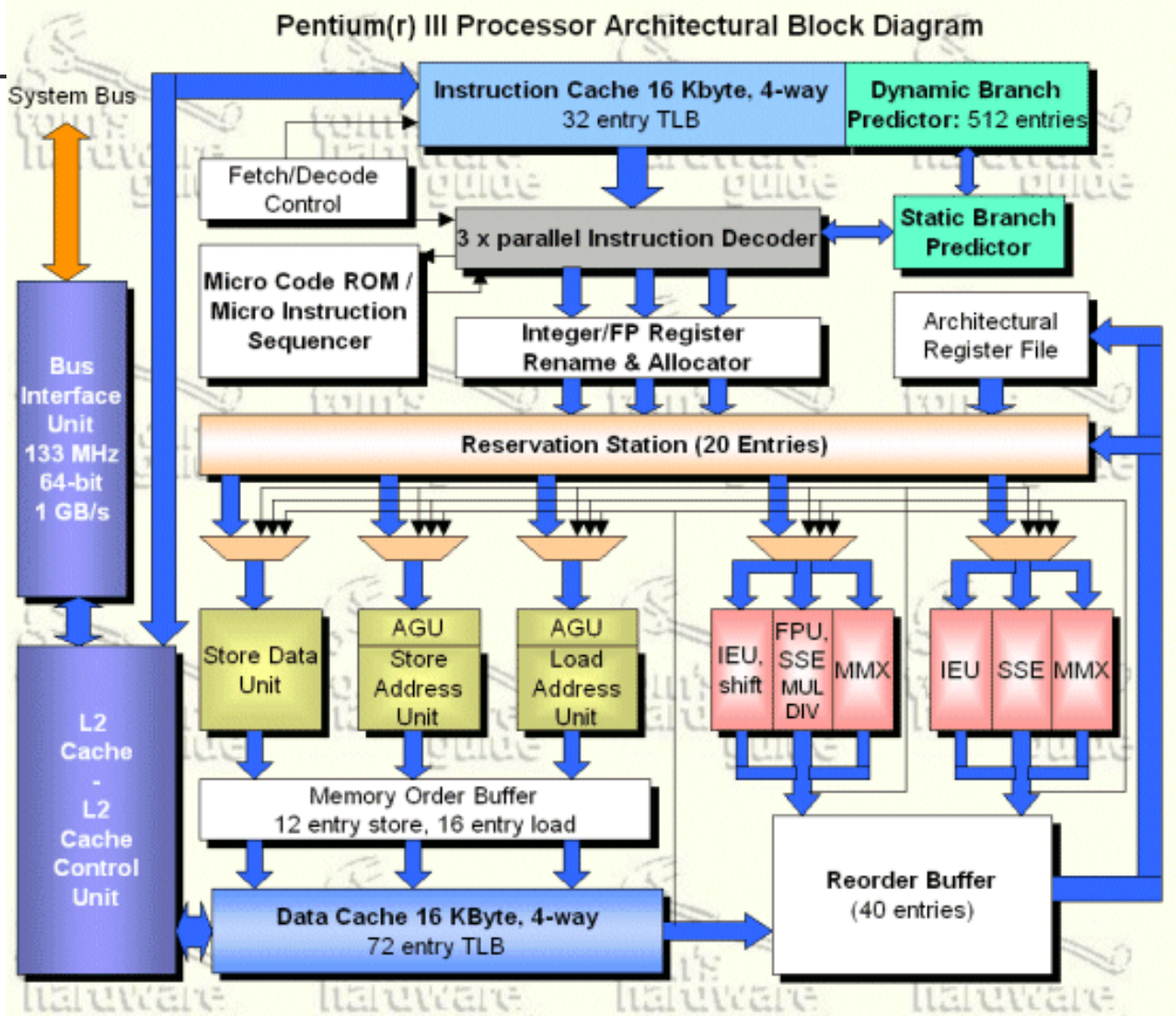
Dự đoán rẽ nhánh

- 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

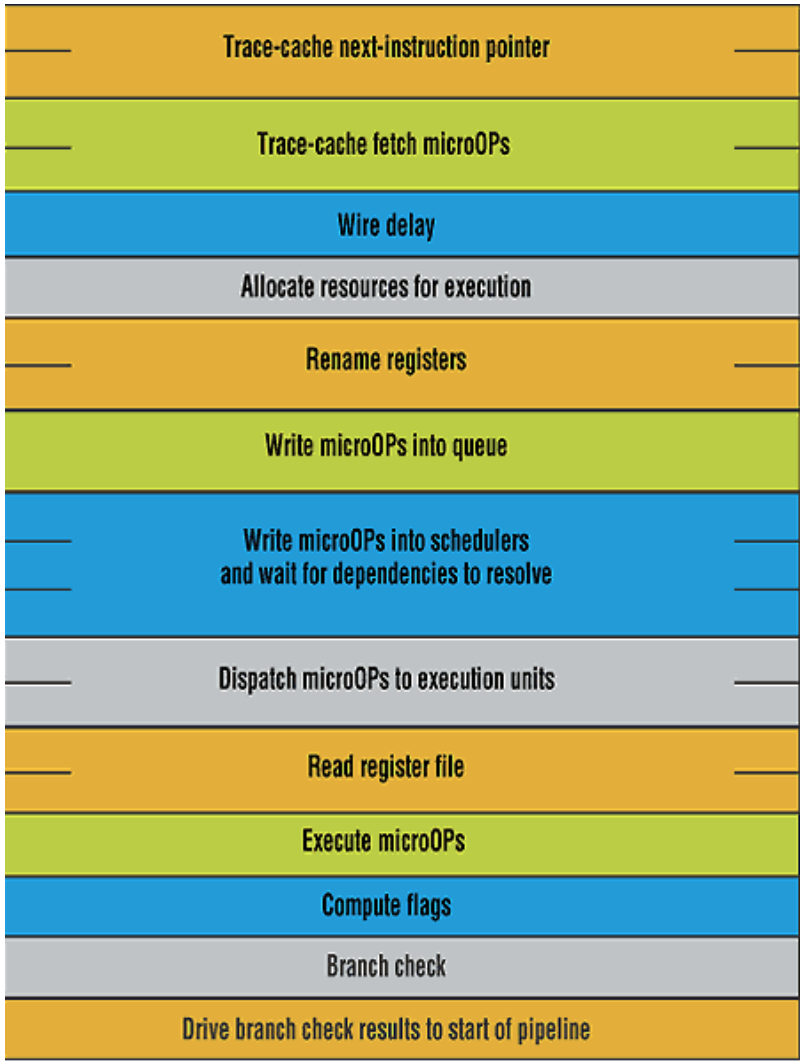
Branch Prediction – Intel PIII



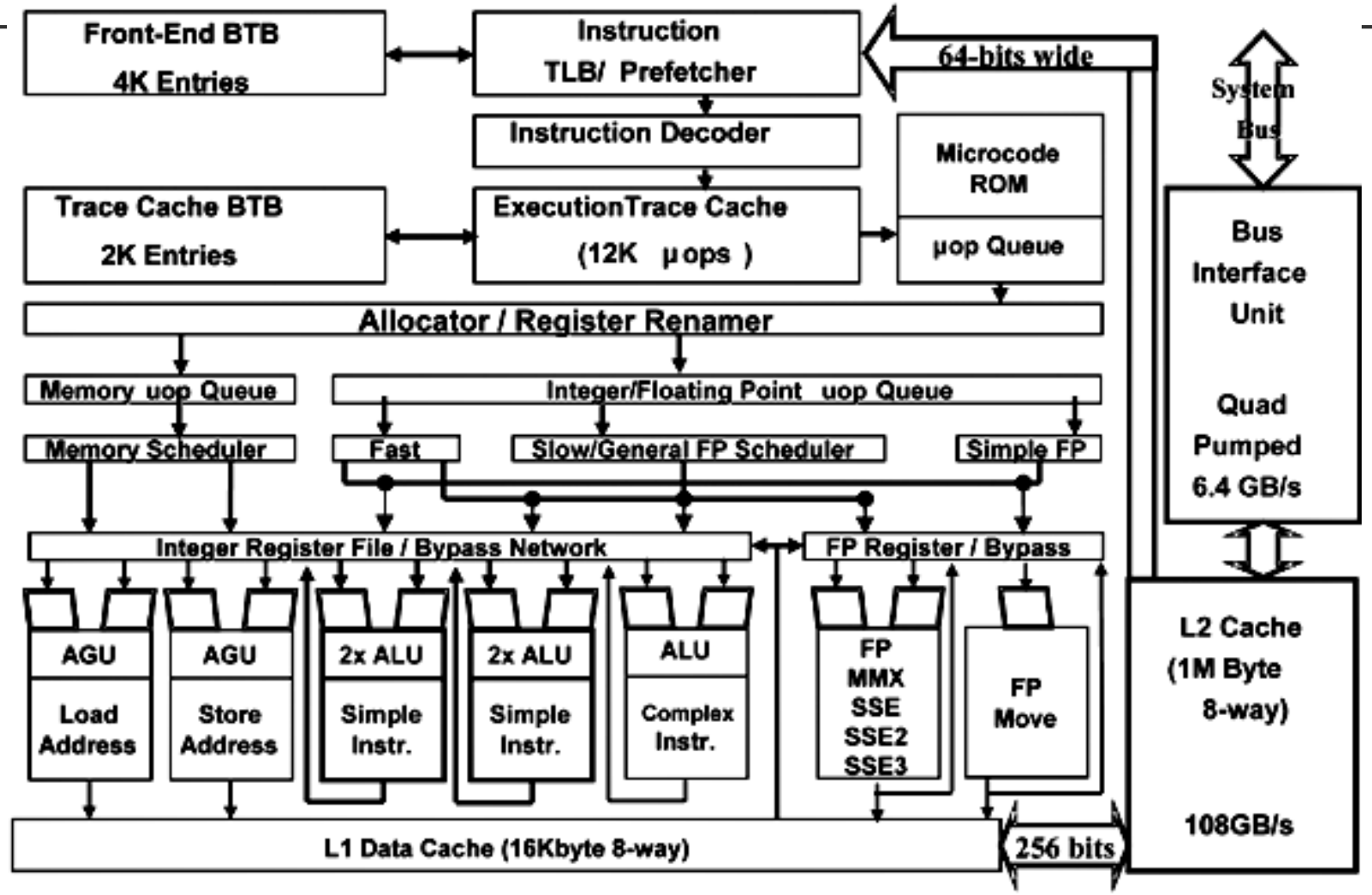
Siêu pipeline (superpipelining)

- 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

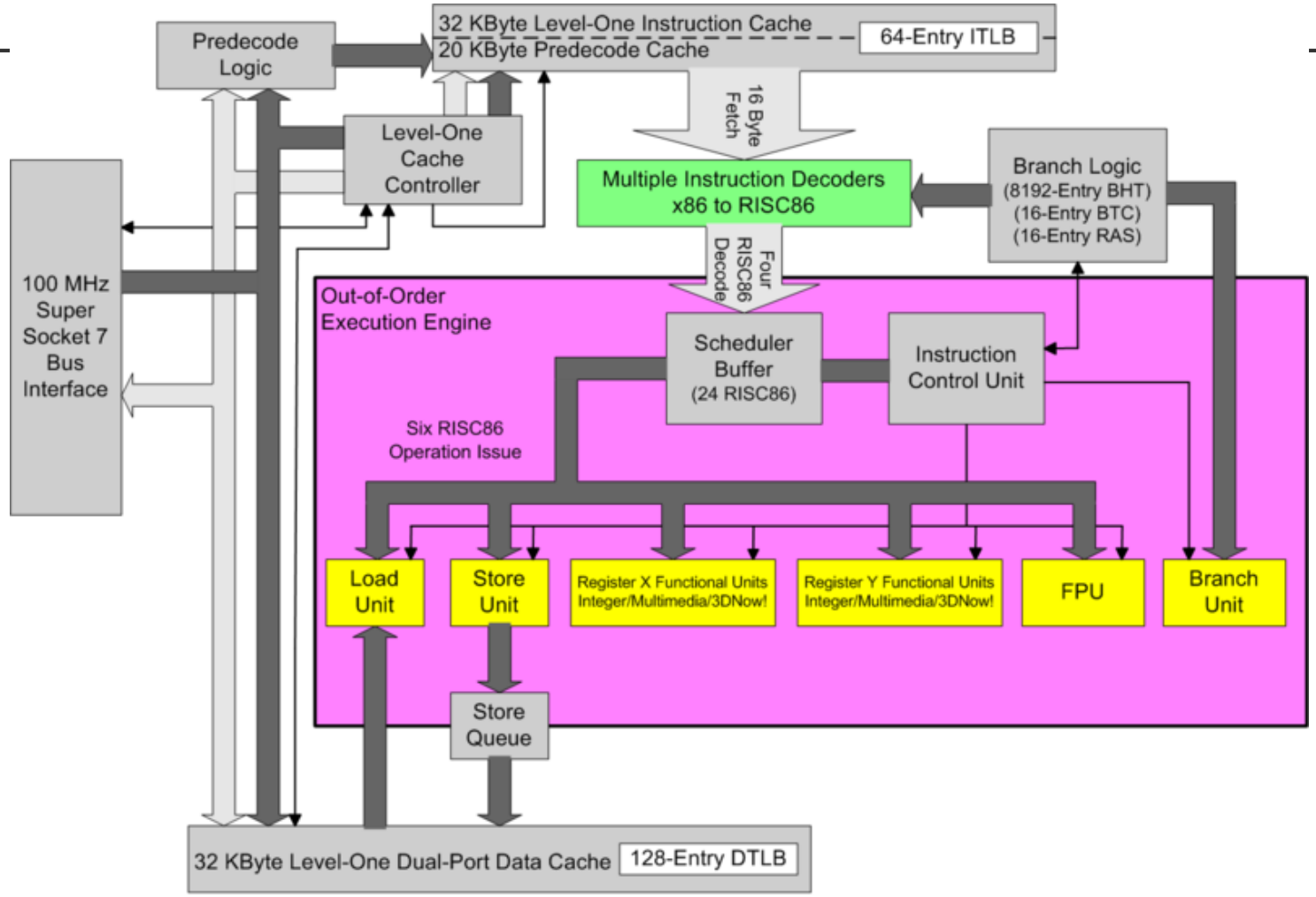
Pentium 4 siêu ống với 20 giai đoạn



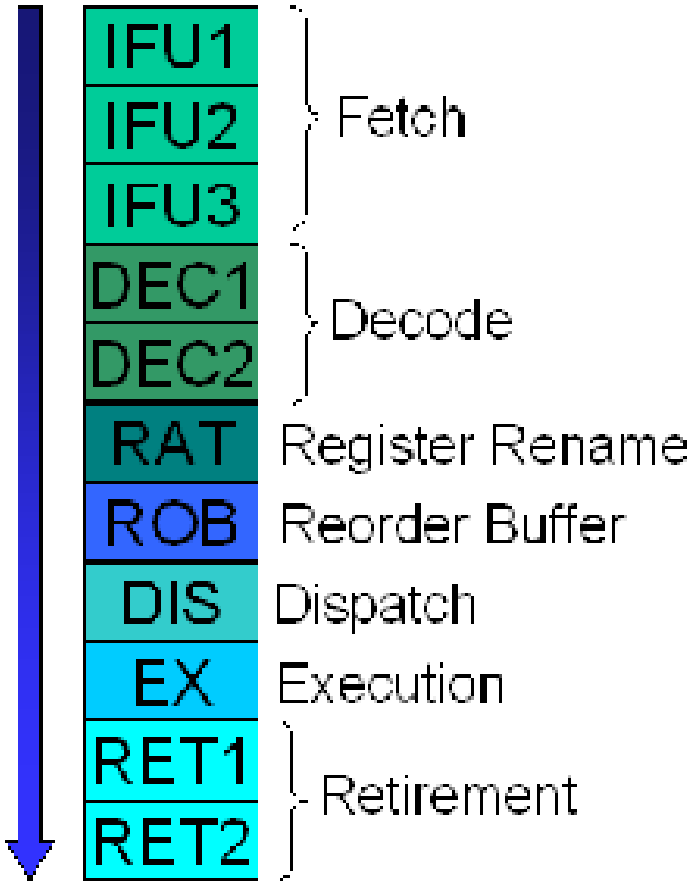
Branch Prediction – Intel P4



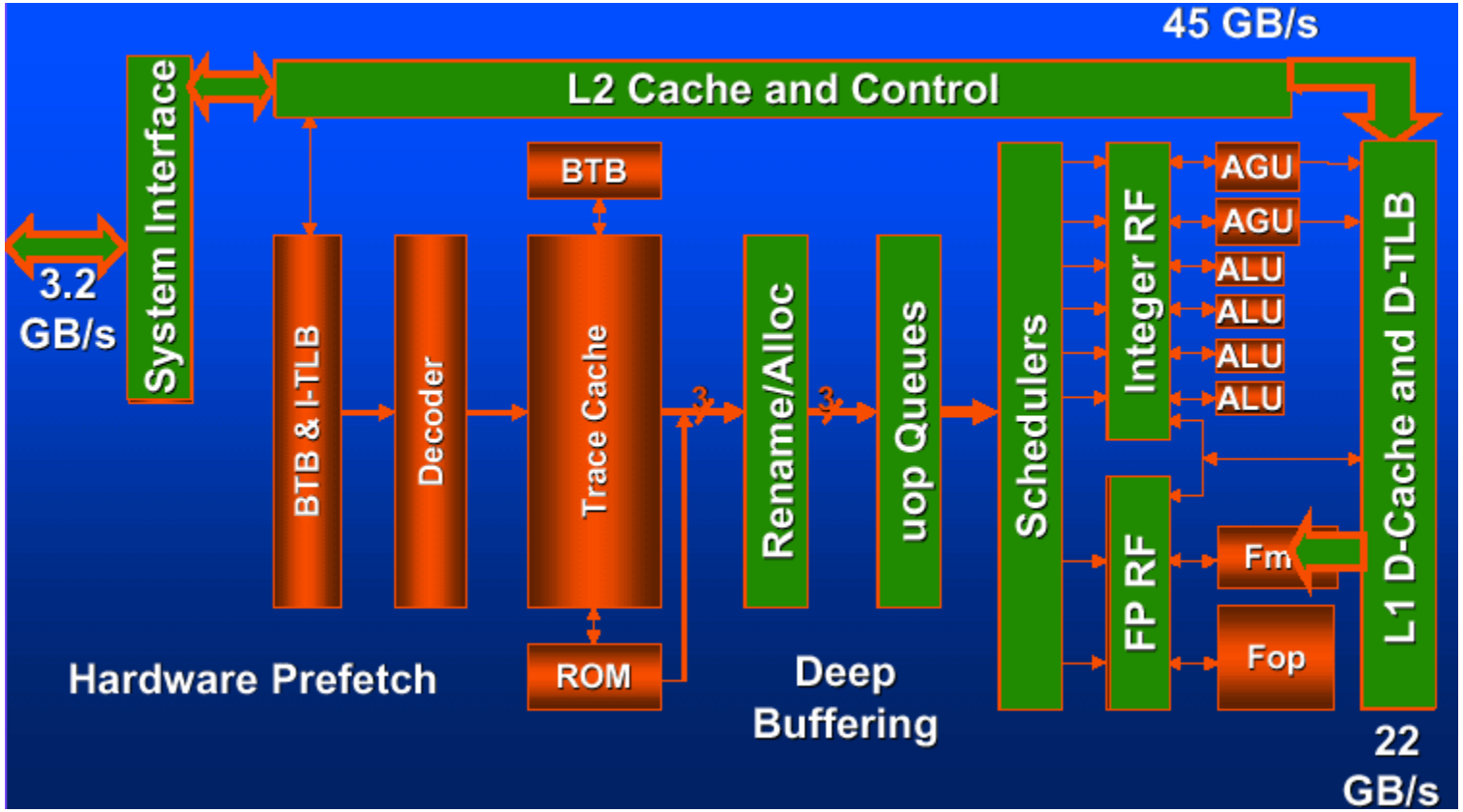
AMD K6-2 pipeline



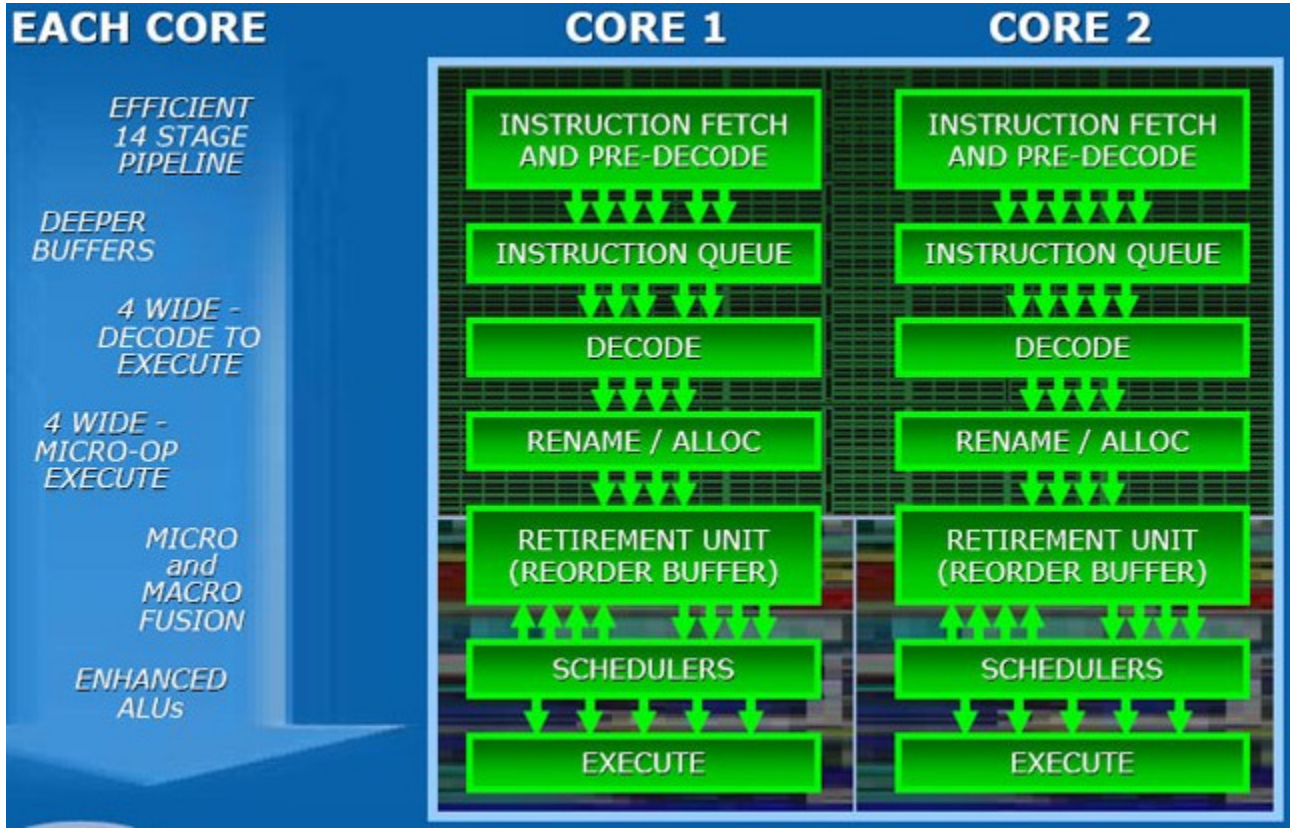
Pipeline –Pen III, M



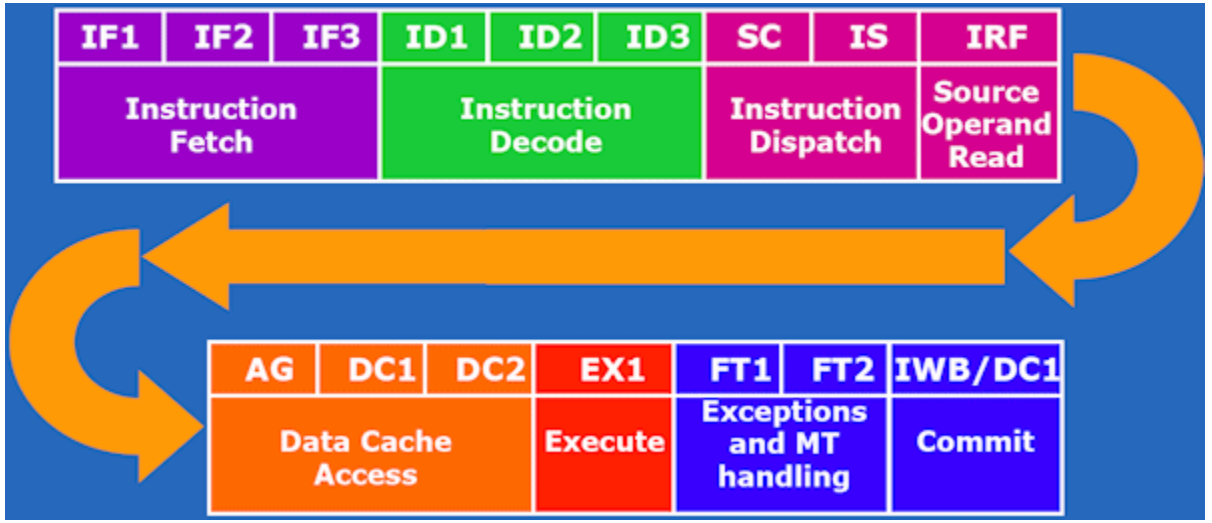
Intel Pen 4 Pipeline



Intel Core 2 Duo pipeline



Intel Atom 16-stage pipeline



Intel Core 2 Duo – Super Pipeline