

Chương 2: CPU Pipeline

Kiến trúc máy tính

ThS. Đinh Xuân Trường
truongdx@ptit.edu.vn



Posts and Telecommunications
Institute of Technology
Faculty of Information Technology 1



CNTT1
Học viện Công nghệ Bưu chính Viễn thông

January 15, 2023

Mục tiêu Buổi 5

Giới thiệu về CPU Pipeline

Các vấn đề của cơ chế ống lệnh và hướng giải quyết

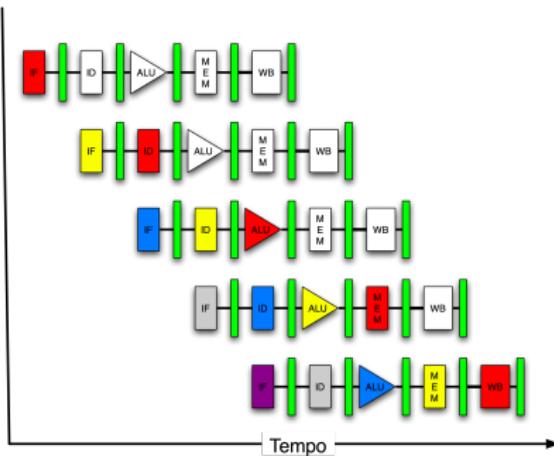
Xử lý xung đột dữ liệu và tài nguyên

Xử lý rẽ nhánh - branch

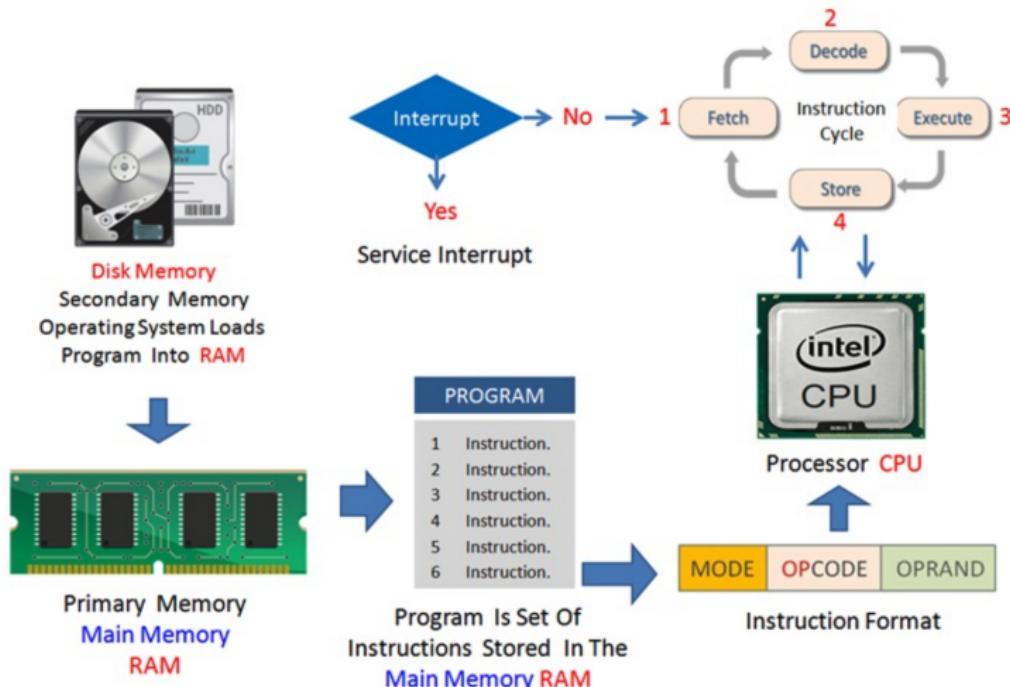
Super pipeline

Nội dung chương 2 (Tiếp)

1. Giới thiệu về CPU pipeline
2. Các vấn đề của cơ chế ông lệnh và hướng giải quyết
 - Xử lý xung đột dữ liệu và tài nguyên
 - Xử lý rẽ nhánh - branch
3. Super pipeline



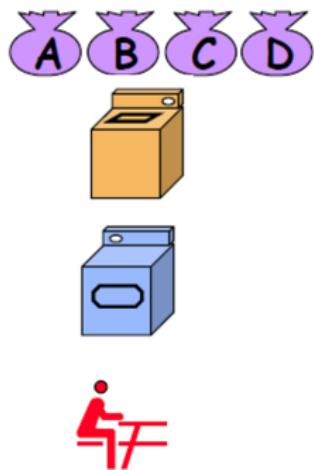
Nhắc lại về Chu kỳ xử lý lệnh của CPU



Ví dụ thực tế:

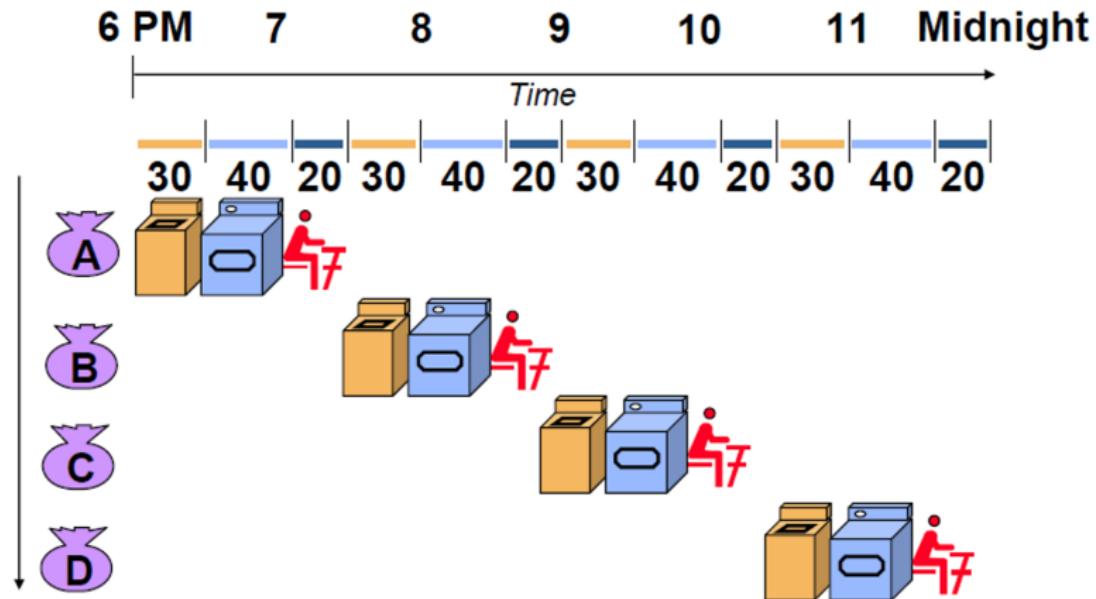
Bài toán giặt: A, B, C, D có 4 túi quần áo cần giặt, làm khô, gấp

- ▶ Giặt tốn 30 phút
- ▶ Sấy khô: 40 phút
- ▶ Gấp: 20 phút

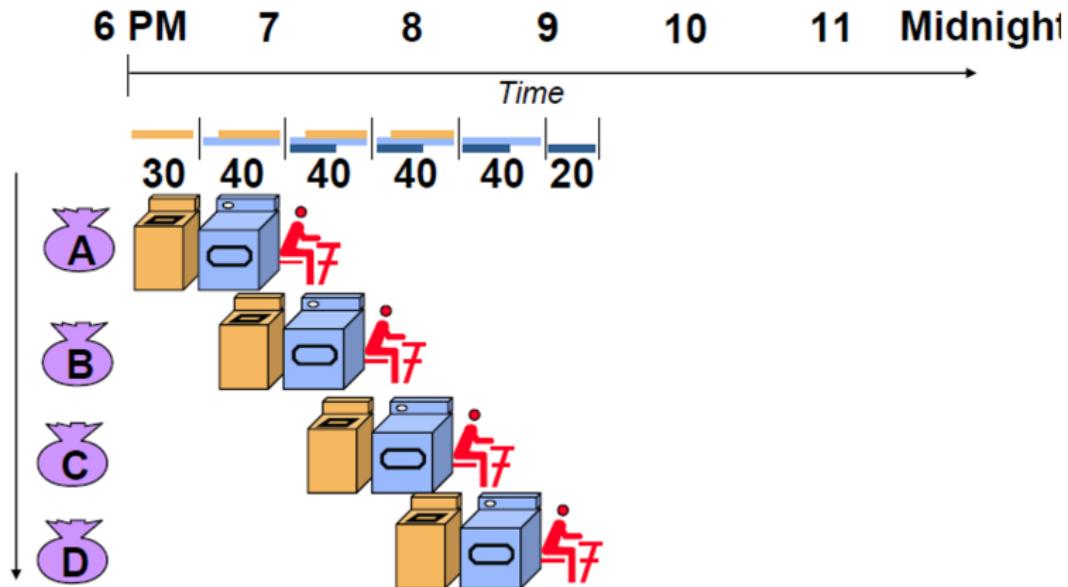


Giới thiệu về CPU pipeline (cont.)

Thực hiện tuần tự:



Thực hiện pipeline - đường ống:



Giới thiệu về Nguyên lý CPU Pipeline:

Quá trình thực hiện lệnh được chia thành các pha hay giai đoạn (stage). Mỗi lệnh có thể được thực hiện theo 5 giai đoạn của hệ thống load – store:

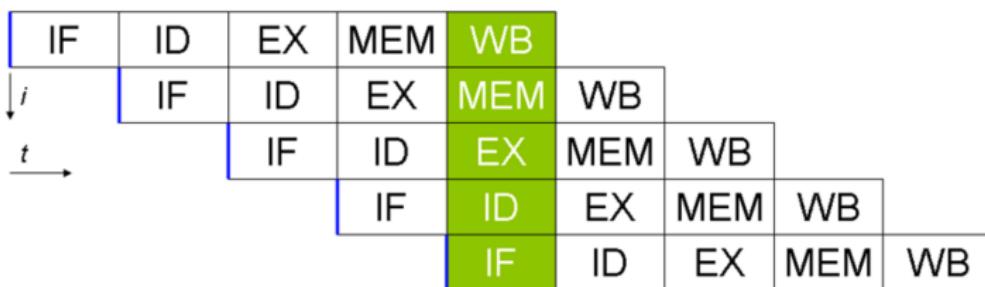


- ▶ **IF** Đọc lệnh (Instruction Fetch): (IF): Lấy lệnh từ bộ nhớ (hoặc cache)
- ▶ **ID** Giải mã lệnh (Instruction Decode): Thực hiện giải mã lệnh và lấy các toán hạng
- ▶ **IE** Thực thi lệnh (Instruction Execution): Nếu là lệnh truy cập bộ nhớ thì tính toán địa chỉ bộ nhớ
- ▶ (Memory access) Đọc - ghi bộ nhớ **MEM** : Đọc và ghi bộ nhớ nếu không truy cập bộ nhớ thì không có giai đoạn này
- ▶ **WB** Ghi (Write Back): kết quả (nếu có) CPU xử lý được ghi vào thanh ghi/bộ nhớ lưu

Giới thiệu về CPU pipeline (cont.)

Giới thiệu về Nguyên lý CPU Pipeline

Cải thiện hiệu năng bằng cách tăng số lượng lệnh vào xử lý:



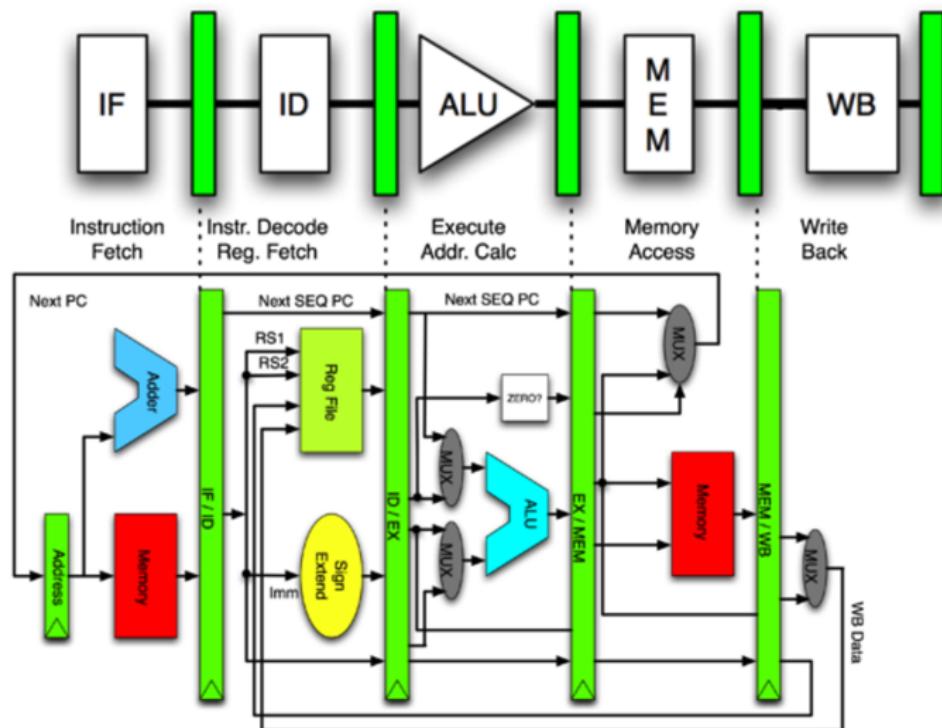
Nhận xét: Chia thực hiện lệnh thành 5 bước → Hiệu quả theo Pipeline hơn tương đương 5 lần

Giới thiệu về CPU pipeline (cont.)

Giới thiệu về Nguyên lý CPU Pipeline



Các thành phần phần cứng tham gia Pipeline:



Giới thiệu về CPU pipeline (cont.)

Giới thiệu về Nguyên lý CPU Pipeline



- ▶ Pipeline là kỹ thuật song song ở mức lệnh (ILP: Instruction Level Parallelism)
- ▶ Một pipeline đầy đủ luôn nhận 1 lệnh mới tại mỗi chu kỳ đồng hồ
- ▶ Một pipeline không đầy đủ có các 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
- ▶ Tại sao không tăng số giai đoạn của Pipeline là một số lớn hơn → tăng hiệu năng?
 - Tăng thành phần phần cứng → Đắt
 - Chip lớn → Tiêu tốn điện năng

Giới thiệu về CPU pipeline (cont.)

Giới thiệu về Nguyên lý CPU Pipeline

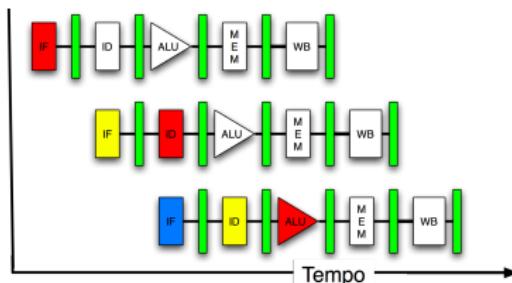
Số lượng giai đoạn phổ biến hiện nay từ 16 - 26 giai đoạn:

► 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 của pipeline

Pipeline tăng hiệu năng của CPU vậy có vấn đề xảy ra khi sử dụng kỹ thuật này không?

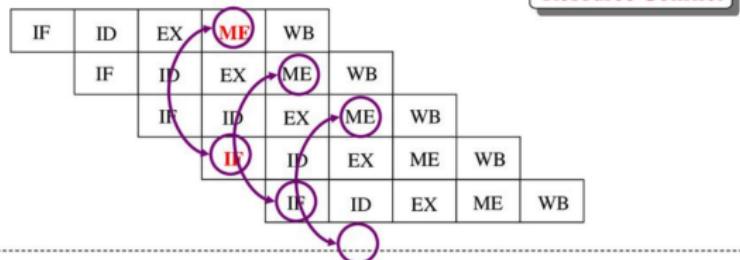
- ▶ Vấn đề xung đột tài nguyên (resource conflict)
 - Xung đột truy cập bộ nhớ
 - Xung đột truy cập thanh ghi
- ▶ Xung đột/ tranh chấp dữ liệu (data hazard)
 - Hầu hết là RAW hay Read After Write Hazard
- ▶ Các lệnh rẽ nhánh (Branch Instruction)
 - Không điều kiện
 - Có điều kiện
 - Gọi thực hiện và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

Các vấn đề của pipeline

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
- ▶ 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

Resource Conflict

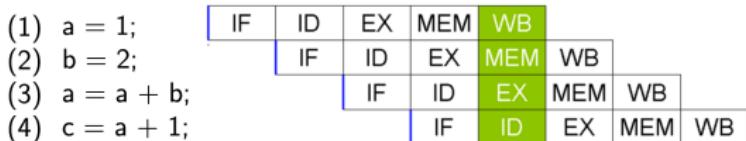


Xử lý xung đột dữ liệu và tài nguyên

Xét đoạn lệnh sau:

- (1) $a = 1;$
- (2) $b = 2;$
- (3) $a = a + b;$
- (4) $c = a + 1;$

- ▶ Câu hỏi đặt ra: Sau lệnh thứ 3 thì giá trị của $a = ?$ và $b = ?$; sau lệnh thứ 4 thì $c = ?$
- ▶ Nếu làm tuần tự bình thường - không sử dụng pipeline : Thực thi hết lệnh lành sang lệnh khác
- ▶ Thực hiện pipeline thì như thế nào?



Xử lý xung đột dữ liệu và tài nguyên (cont.)

- (1) $a = 1;$
- (2) $b = 2;$
- (3) $a = a + b;$
- (4) $c = a + 1;$

	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		

Nhận xét:

- ▶ Lệnh (3) đã được thực hiện khi a chưa bằng 1 và b chưa bằng 2
 - ▶ Kết quả $a + b$ chưa bằng 3
 - ▶ Tương tự, lệnh (4) thực hiện lúc này a = 1 rồi nhưng chưa được cập nhật trong bộ nhớ
 - ▶ Máy tính chạy nhanh nhưng kết quả đúng \neq Nhanh mà kết quả sai
- ⇒ Vấn đề Dữ liệu chưa sẵn sàng cho các lệnh phụ thuộc tiếp theo được gọi là **Data Hazard** - xung đột dữ liệu hay sự phụ thuộc dữ liệu giữa các lệnh.

Xử lý xung đột dữ liệu và tài nguyên (cont.)

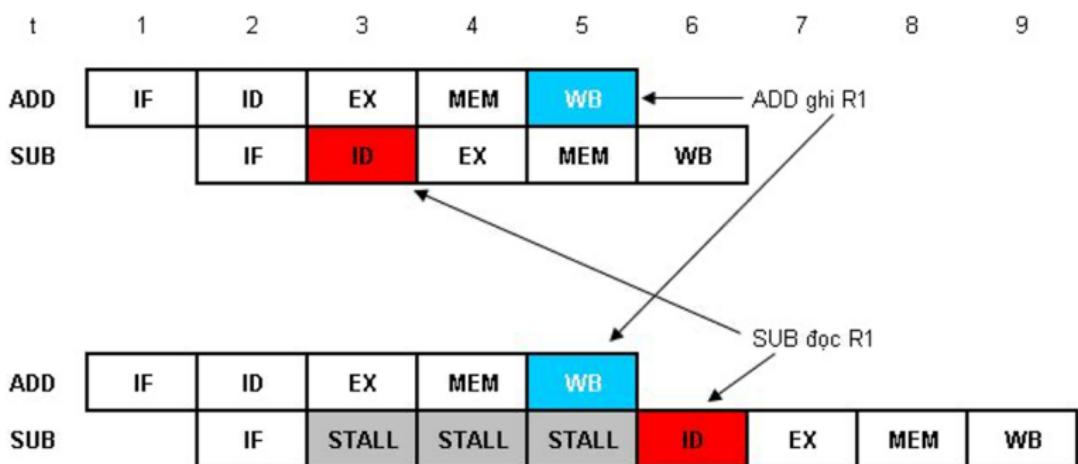
Xét ví dụ sau:

ADD R1, R1, R3;

$R1 \leftarrow R1 + R3$

SUB R4, R1, R2;

$R4 \leftarrow R1 - R2$



Hướng khắc phục xung đột dữ liệu

- ▶ Nhận biết các lệnh phụ thuộc dữ liệu
- ▶ Các lệnh phụ thuộc cần thực thi (EX) sau khi các lệnh mà nó phụ thuộc thực hiện xong
 - ⇒ 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
 - ▶ Lệnh NO-OP lệnh không thay đổi trạng thái CPU mà chỉ lấp đi 1 chu kỳ lệnh - thời gian 1 chu kỳ
 - 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:

t	1	2	3	4	5	6	7	8	9
ADD	IF	ID	EX	MEM	WB				
NO-OP		NO-OP	NO-OP	NO-OP	NO-OP	NO-OP			
NO-OP			NO-OP	NO-OP	NO-OP	NO-OP	NO-OP		
NO-OP				NO-OP	NO-OP	NO-OP	NO-OP	NO-OP	
SUB					IF	ID	EX	MEM	WB

Hướng khắc phục xung đột dữ liệu

Chèn 3 lệnh đọc lập dữ liệu vào giữa ADD và SUB:

	t	1	2	3	4	5	6	7	8	9
ADD R1, R1, R3		IF	ID	EX	MEM	WB				
LOAD R4, [1000]			IF	ID	EX	MEM	WB			
ADD R5, 500				IF	ID	EX	MEM	WB		
STORE [2000], R6					IF	ID	EX	MEM	WB	
SUB R4, R1, R2						IF	ID	EX	MEM	WB

Hướng khắc phục xung đột dữ liệu

Bài tập 1: Dịch đoạn code C dưới đây, với b, c, e, f đang được lưu vào R2, R3, R5, R6; a và d lưu vào R1 và R4. Cho biết cần chèn thêm mấy lệnh NO-OP trong trường hợp thiết kế CPU pipeline:

$$\begin{aligned}a &= b + c; \\d &= a - e + f;\end{aligned}$$

Bài tập 2: Xác định lỗi và bố trí lại các câu lệnh tránh trì hoãn khi thiết kế pipeline cho các câu lệnh sau:

*LOAD R1, 0(R0)
LOAD R2, 4(R0)
ADD R3, R1,R2
STORE R3, 12(R0)
LOAD R4, 8(R0)
ADD R5,R1,R4
STORE R5,16(R0)*

Xử lý rẽ nhánh - branch

Xét đoạn lệnh sau:

- (1) $if(a > b)$
- (2) $a = a - b ;$
- (3) $else$
- (4) $a = b - a;$

Nhận xét:

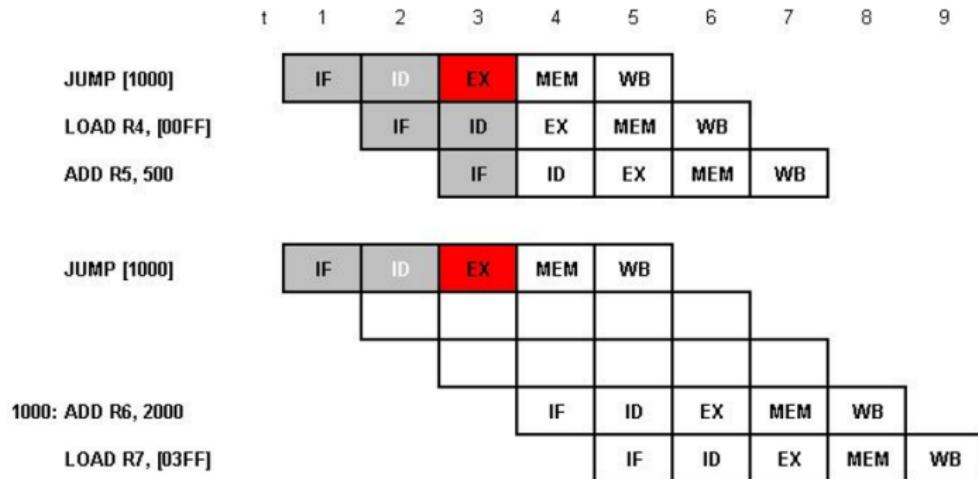
- ▶ Thực hiện pipeline Lệnh (1) chưa kiểm tra xong $a > b$ hay không thì lệnh (2) đã được thực hiện
- ▶ Vấn đề trong kỹ thuật pipeline khi kiểm tra các điều kiện đối với các lệnh rẽ nhánh hay vòng lặp (ví dụ: *while do*)
- ▶ Trường hợp này được gọi là **Branch Hazard** khi gặp câu lệnh rẽ nhánh và lặp.

Xử lý rẽ nhánh - branch (cont.)

- ▶ 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 gấp lệnh rẽ nhánh
 - Tải mới các lệnh từ địa chỉ rẽ nhánh vào pipeline. Tiêu tốn nhiều thời gian để điền đầy pipeline

Xử lý rẽ nhánh - branch (cont.)

Khi 1 lệnh rẽ nhánh được thực hiện, các lệnh tiếp theo bị đẩy ra khỏi pipeline và các lệnh mới được tải \Rightarrow chèn các lệnh NO-OP đến khi nào câu lệnh rẽ nhánh / điều kiện thực hiện xong.



Xử lý rẽ nhánh - branch (cont.)

Xét đoạn lệnh sau:

JUMP <Address>

ADD R1, R2

Address: *SUB R3, R4*

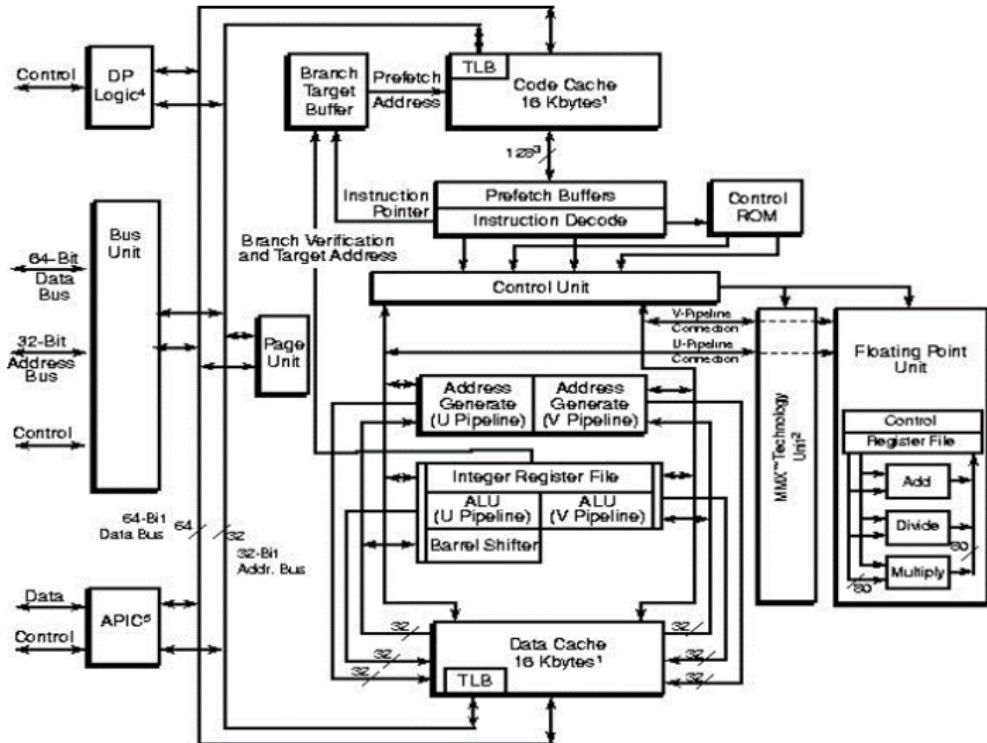
► Đích rẽ nhánh (branch target)

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

► Rẽ nhánh có điều kiện (conditional branches)

- Làm chậm rẽ nhánh (delayed branching)
- Dự báo rẽ nhánh (branch prediction)

Dịch rẽ nhánh của PIII:



A6105-01

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 tối khi lệnh rẽ nhánh được thực hiện xong
 - Sử dụng BTB riêng rẽ không hiệu quả vì phải đợi tối khi có thể xác định được lệnh đích.
- ▶ Chiến lược xử lý rẽ nhánh:
 - Làm chậm rẽ nhánh
 - Dự đoán rẽ nhánh

► 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

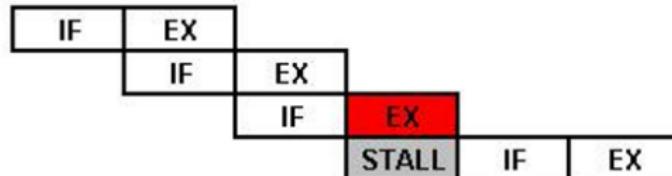
► Cài đặt:

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

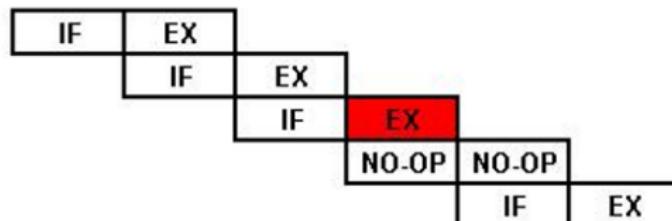
Xử lý rẽ nhánh - branch (cont.)

Làm chậm rẽ nhánh

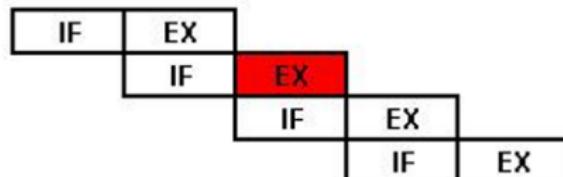
ADD R2, R3, R4
CMP R1,0
JNE somewhere
SUB R5, R6, R7



ADD R2, R3, R4
CMP R1,0
JNE somewhere
NO-OP
SUB R5, R6, R7



CMP R1,0
JNE somewhere
ADD R2, R3, R4
SUB R5, R6, R7



Ưu nhược điểm của làm chậm rẽ nhánh:

► **Ưu điểm:**

- 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%

► **Nhược điểm:**

- 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

► 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

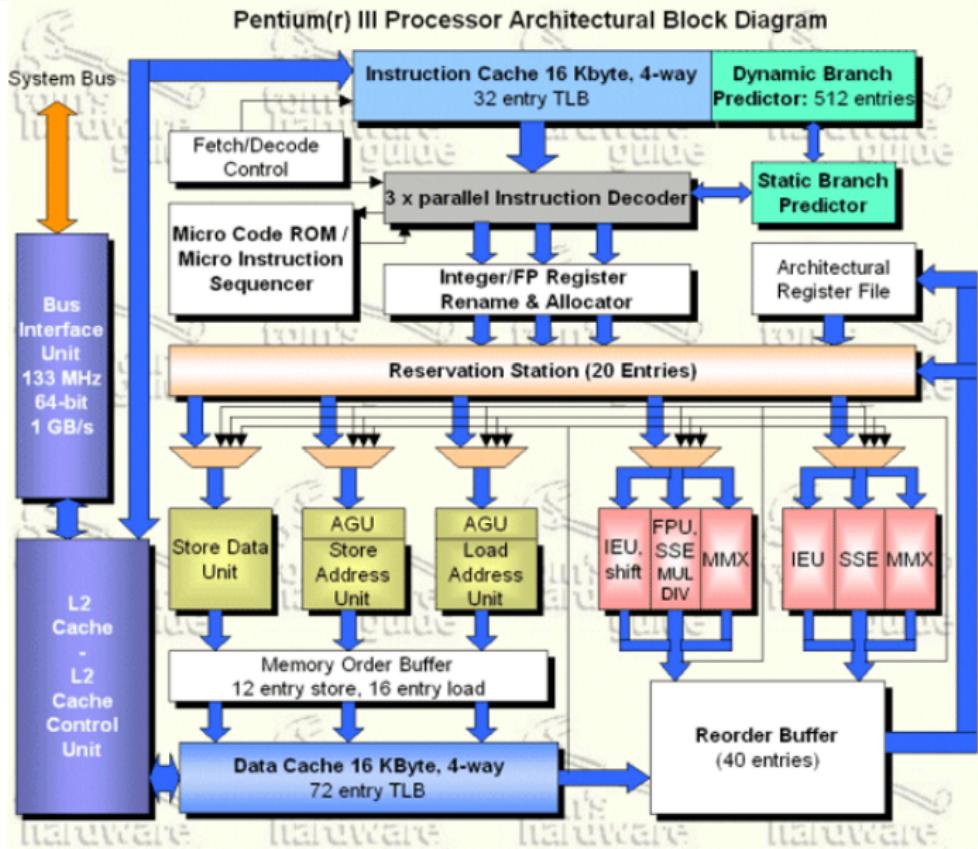
► Các cơ sở để dự đoán:

- 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

Xử lý rẽ nhánh - branch (cont.)

Dự đoán rẽ nhánh



Siêu pipeline (Superpipelining)

Siêu pipeline là kỹ thuật cho phép:

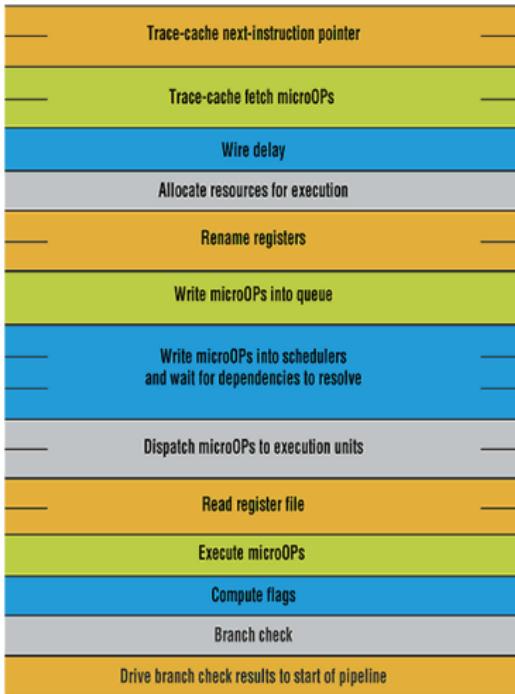
- ▶ Tăng độ sâu ống lệnh
- ▶ Tăng tốc độ đồng hồ
- ▶ Giảm thời gian trễ cho từng giai đoạn thực hiện lệnh

Ví dụ: nếu giai đoạn thực hiện lệnh bởi ALU kéo dài \rightarrow chia thành một số giai đoạn nhỏ \rightarrow giảm thời gian chờ cho các giai đoạn ngắn

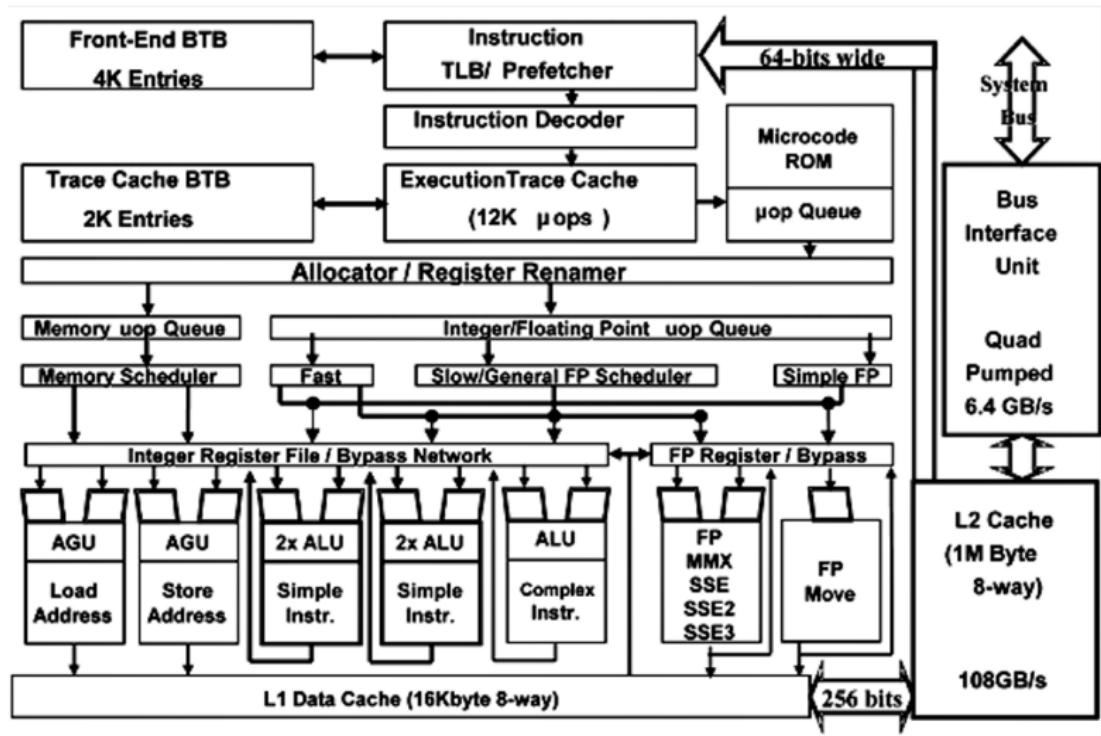
Pentium 4 siêu ống với 20 giai đoạn:



Pentium 4 siêu ông với 20 giai đoạn:

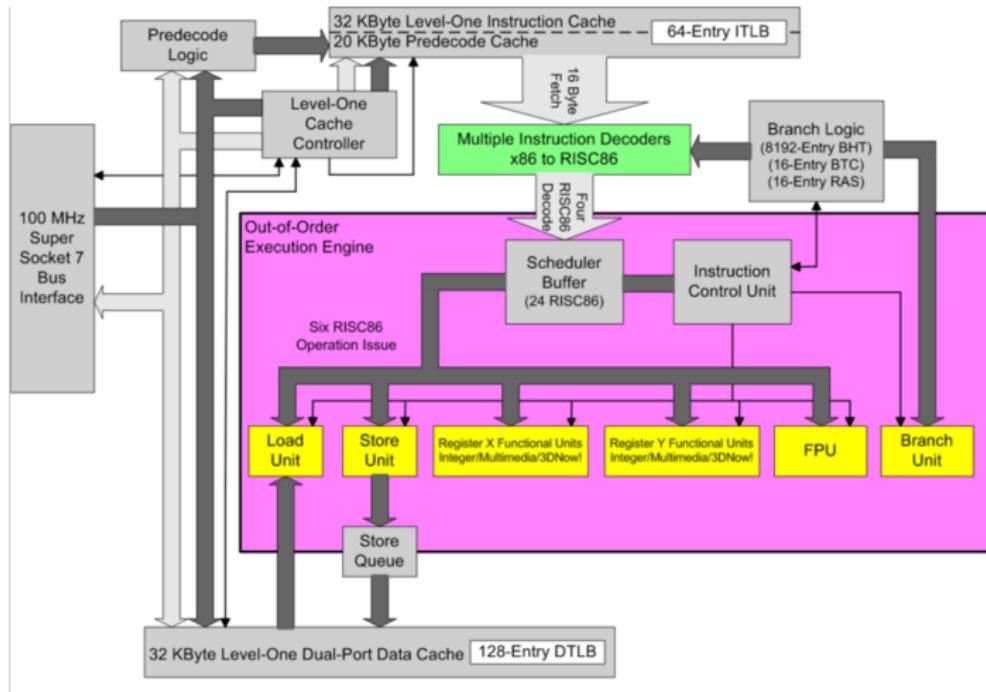


Branch Prediction – Intel P4:

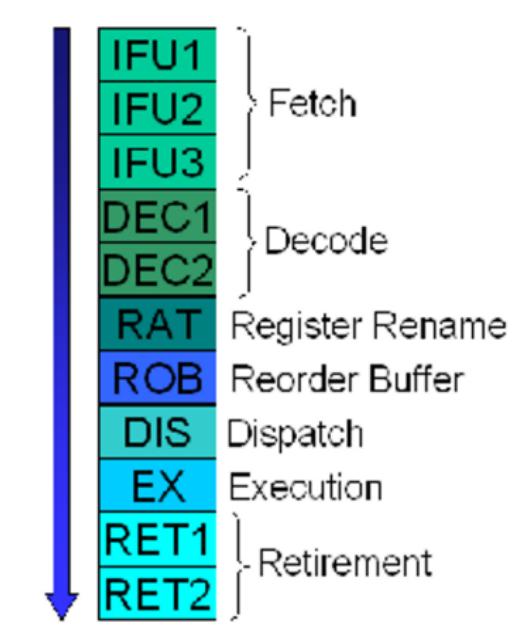


Siêu pipeline (Superpipelining) (cont.)

AMD K6-2 pipeline:

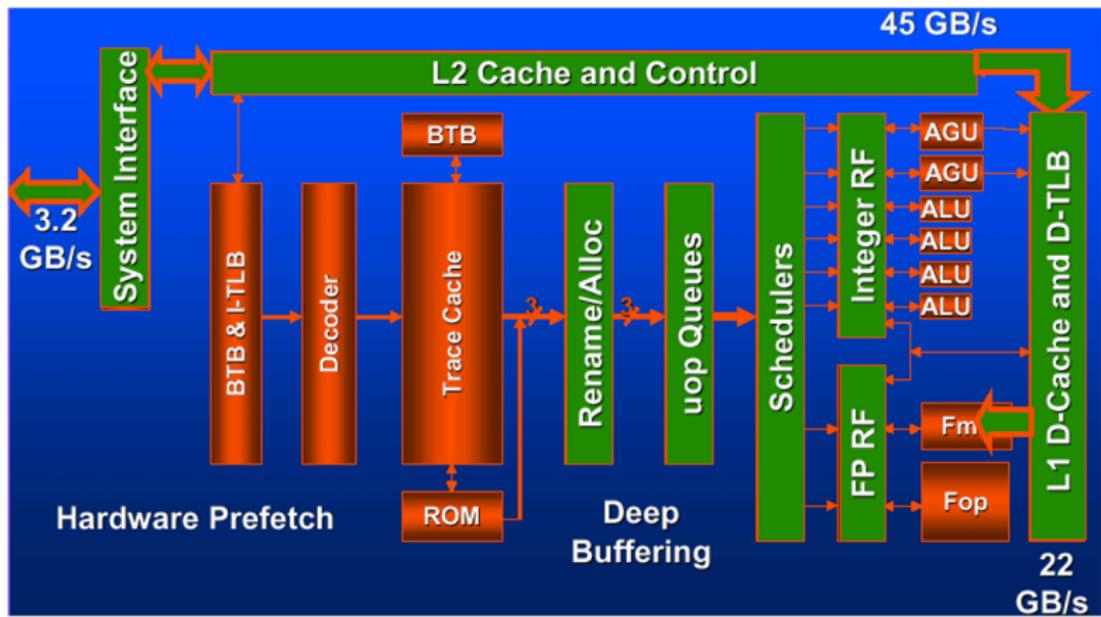


Pipeline – Pen III, M:



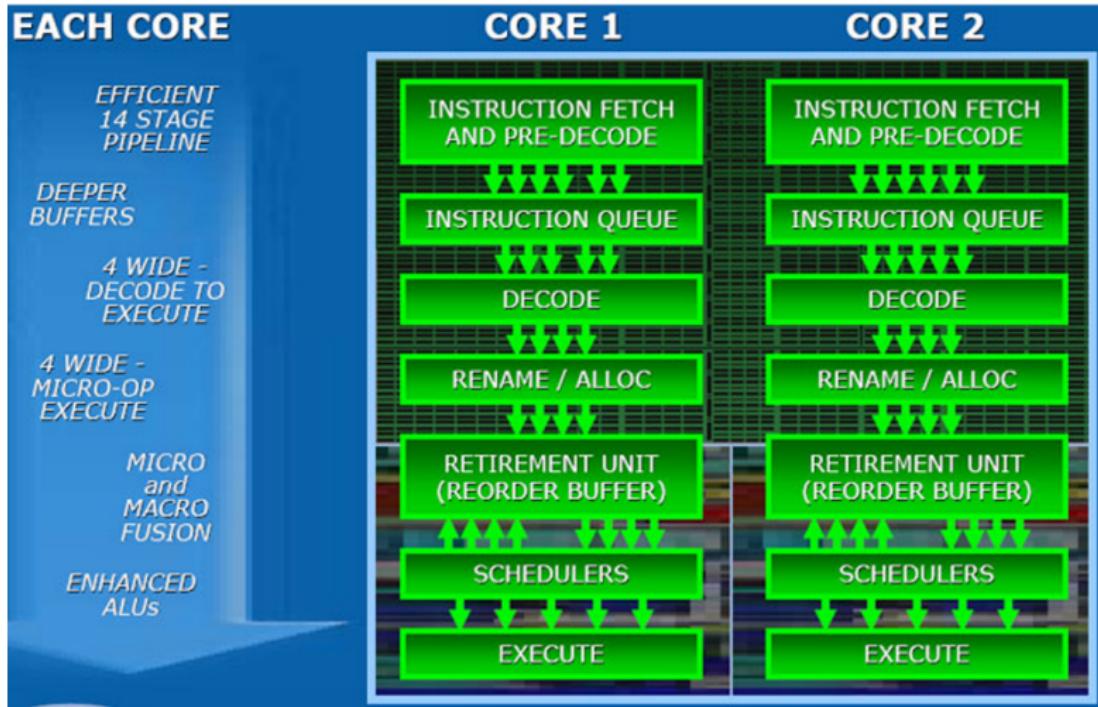
Siêu pipeline (Superpipelining) (cont.)

Intel Pen 4 Pipeline:



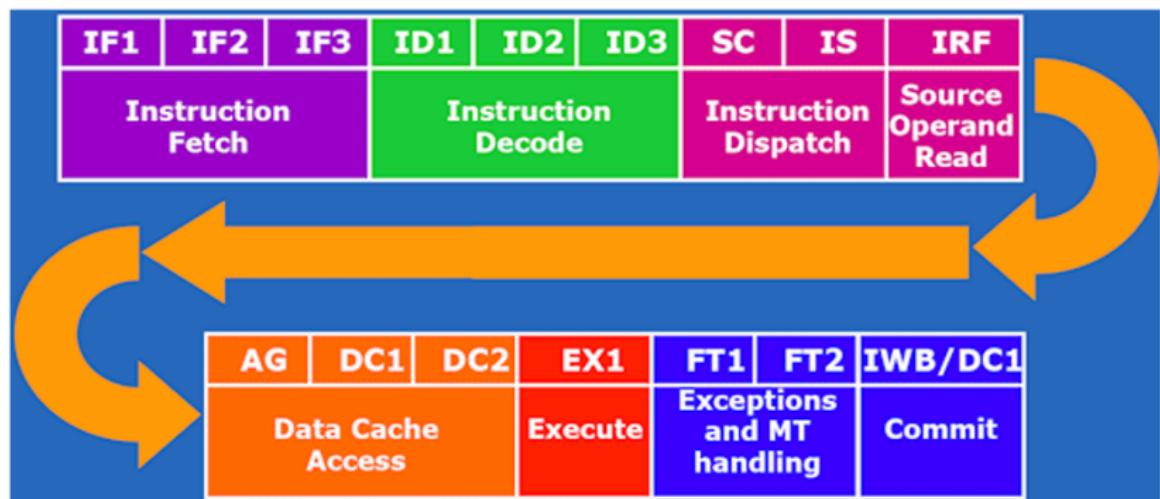
Siêu pipeline (Superpipelining) (cont.)

Intel Core 2 Duo pipeline:

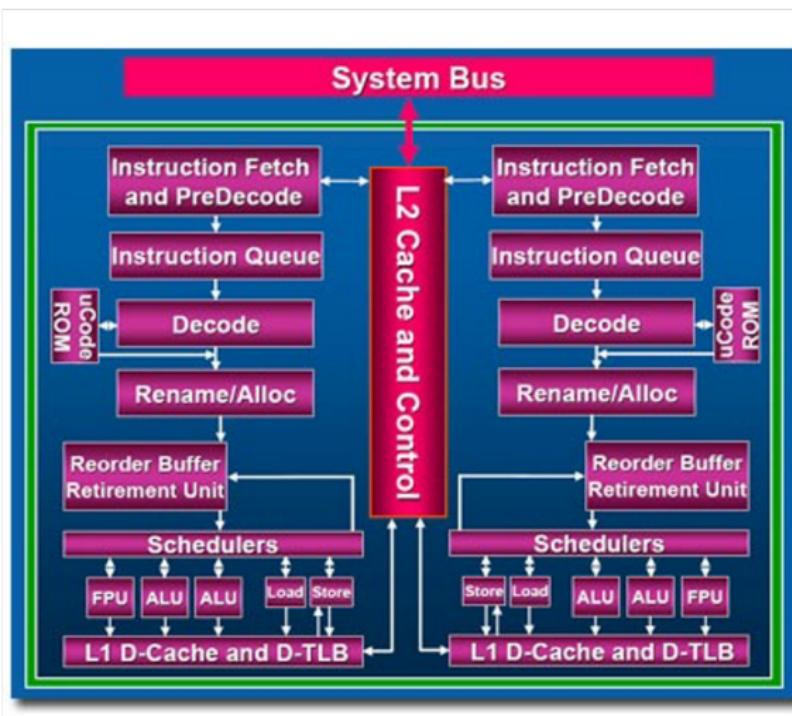


Siêu pipeline (Superpipelining) (cont.)

Intel Atom 16-stage pipeline:



Intel Core 2 Duo – Super Pipeline:



Tổng kết Buổi 5

Chương 2

- ▶ 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

Tiếp theo Chương 3 - Bộ nhớ và Bộ nhớ Cache

- ▶ Bộ nhớ trong và mô hình phân cấp bộ nhớ
- ▶ Phân loại bộ nhớ và tổ chức mạch nhớ
- ▶ ROM
- ▶ RAM
- ▶ Bộ nhớ cache

Câu hỏi và bài tập

- **Bài 1:** Cơ chế ống lệnh (pipeline) của CPU thường gặp phải những vấn đề gì? Nếu một hướng giải quyết xung đột dữ liệu trong pipeline khi thực hiện đoạn chương trình sau:

ADD R1, R2, R3 ;	R1 <== R2+R3
ADD R4, R4, #300 ;	R4 <== R4+300
CMP R1, #100 ;	so sánh R1 với 100
SUB R5, #2000 ;	R5 <== R5 + 2000

- Niết rằng mỗi lệnh được chia thành 5 giai đoạn trong pipeline: Đọc lệnh (IF), giải mã & đọc toán hạng (ID), truy nhập bộ nhớ (MEM), thực hiện (EX) và lưu kết quả (WB).

Câu hỏi và bài tập (cont.)

- **Bài 2:** Cơ chế ống lệnh (pipeline) của CPU thường gặp phải những vấn đề gì? Nêu một hướng giải quyết xung đột dữ liệu trong pipeline khi thực hiện đoạn chương trình sau:

```
ADD R4, R4, #300 ;      R4 <== R4+300  
ADD R1, R1, R3 ;      R1 <== R1+R3  
SUB R5, #2000 ;      R5 <== R5 + 2000  
SUB R1, R1, #100 ;      R1 <== R1 - 100
```

- biết rằng mỗi lệnh được chia thành 5 giai đoạn trong pipeline: Đọc lệnh (IF), giải mã & đọc toán hạng (ID), truy nhập bộ nhớ (MEM), thực hiện (EX) và lưu kết quả (WB).

Câu hỏi và bài tập (cont.)

► **D20 – Đề 1:** Cho đoạn chương trình sau (R1, R2 là các thanh ghi và lệnh quy ước theo dạng LÊNH <ĐÍCH> <GỐC>):

- (1) MOVE R0, #400
- (2) LOAD R1, #2000
- (3) STORE (R1), R0
- (4) SUBSTRACT R0, #20
- (5) ADD 2000, #10
- (6) ADD R0, (R1)

1. Nêu ý nghĩa của từng lệnh và xác định giá trị R0 sau khi thực hiện xong lệnh số (6)
2. Nêu một hướng giải quyết xung đột dữ liệu trong pipeline khi thực hiện đoạn chương trình trên biết rằng mỗi lệnh được chia thành 5 giai đoạn trong pipeline: Đọc lệnh (IF), giải mã & đọc toán hạng (ID), truy nhập bộ nhớ (MEM), thực hiện (EX) và lưu kết quả (WB).

Câu hỏi và bài tập (cont.)

- **D20 – Đề 2:** Cho đoạn chương trình sau (R1, R2 là các thanh ghi và lệnh quy ước theo dạng LỆNH <ĐÍCH> <GỐC>):

- (1) STORE -100(R2), R1
- (2) LOAD R1, (00FF)
- (3) COMPARE R3, R4
- (4) JUMP-IF-EQUAL Label
- (5) ADD R3, R4
- (6) ADD R2, 2
- (7) Label:

1. Xác định chế độ địa chỉ và ý nghĩa của từng lệnh;
2. Nêu hướng giải quyết xung đột dữ liệu trong pipeline khi thực hiện đoạn chương trình trên biết mỗi lệnh được chia thành 5 giai đoạn.
3. Giả thiết $R3 \neq R4$ và mỗi giai đoạn thực hiện lệnh đều thực hiện trong thời gian là 0.1ns, so sánh thời gian CPU chạy hết 6 lệnh đầu tiên trong trường hợp không sử dụng cơ chế pipeline và có sử dụng cơ chế pipeline trong ý 2.

Câu hỏi và bài tập (cont.)

► **D20 – Đề 5:** Cho đoạn chương trình sau (R1, R2 là các thanh ghi và lệnh quy ước theo dạng LÊNH <ĐÍCH> <GỐC>):

- (1) LOAD R2, #400
- (2) LOAD R1, #1200
- (3) STORE (R1), R2
- (4) SUBSTRACT R2, #20
- (5) ADD 1200, #10
- (6) ADD R2, (R1)

1. Nêu ý nghĩa của từng lệnh;
2. Xác định giá trị của thanh ghi R2 sau khi thực hiện xong lệnh số (6)
3. Nêu một hướng giải quyết xung đột dữ liệu trong pipeline khi thực hiện đoạn chương trình trên biết rằng mỗi lệnh được chia thành 5 giai đoạn trong pipeline: Đọc lệnh (IF), giải mã & đọc toán hạng (ID), truy nhập bộ nhớ (MEM), thực hiện (EX) và lưu kết quả (WB).