Nội dung

Quan điểm về máy tính từ góc nhìn nhà thiết kế HĐH

HĐH là một tầng phần mềm tạo ra máy ảo

HĐH cũng quản lý tài nguyên của máy tính nhưng chủ yếu là quản lý các chính sách chia sẻ tài nguyên, chúng ta sẽ thảo luận ở các bài học sau

Bài giảng hôm nay tập trung

Cách làm việc bên trong máy tính

Các cơ chế mở rộng của phần cứng cần thiết cho việc ảo hóa

Các chủ đề

Kiến trúc von Neumann

Thật ra là 1 thiết bị tính toán lớn (big calculator) …

Phần cứng hỗ trợ tổng quát hóa máy tính cơ bản

Sự kiện (Exceptions, Traps) và ngắt (Interrupts), modes

Nhập và xuất (Input and Output)

Mạng, thiết bị lưu trữ và màn hình

Máy tính von Neumann

Những máy vi tính đầu tiên (cuối 40's) thực sự là máy tính

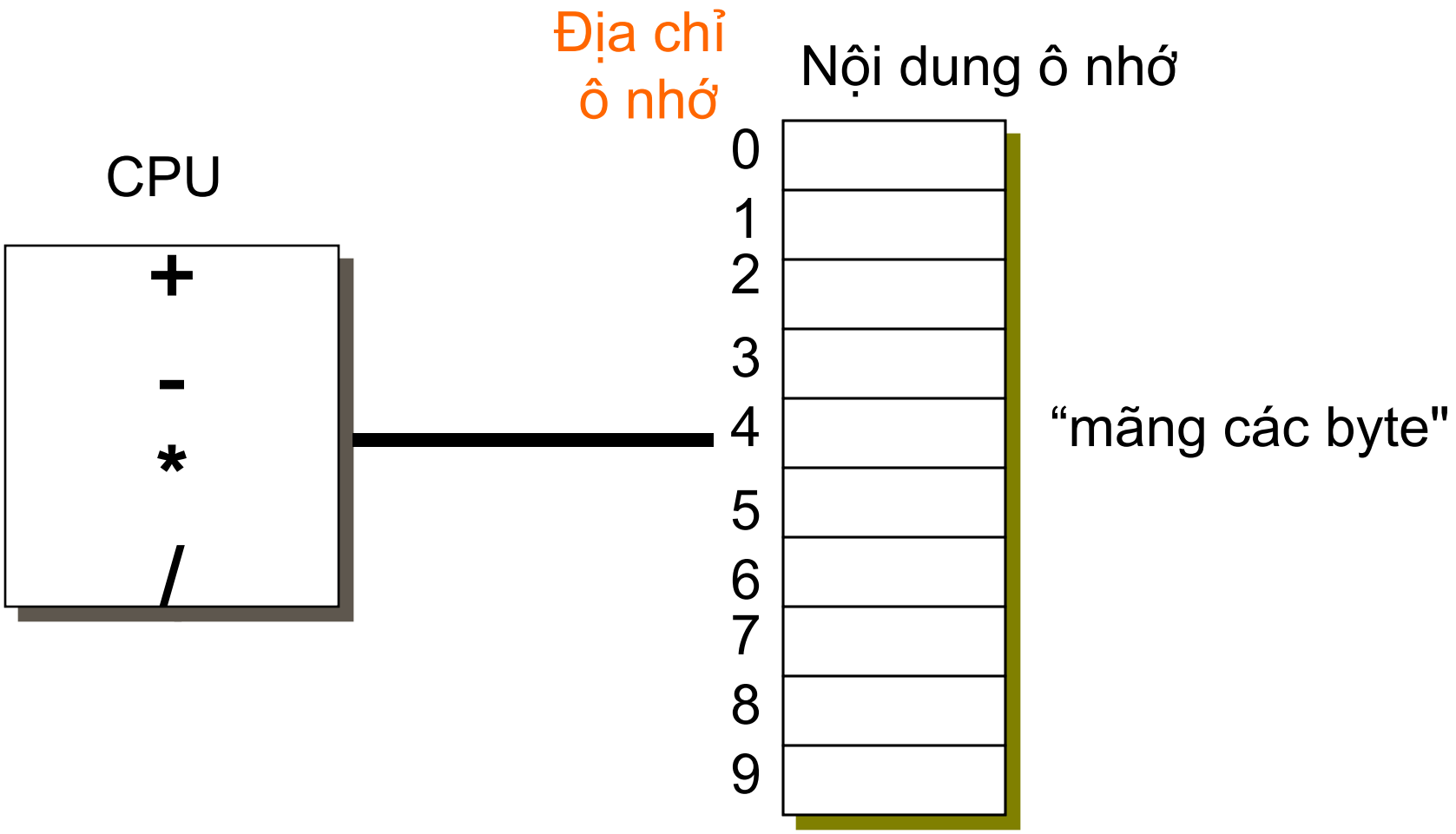
Sự cải tiến là lưu lại những dòng lệnh (mã hóa như là các số) cùng dữ liệu trên cùng một bộ nhớ

Khó khăn khi phân biệt:

Central Processing Unit (CPU) và

Bộ nhớ (Memory)

Mô hình quan niệm



Góc nhìn từ HĐH

Máy tính là một thiết bị phần cứng cứ thực hiện xoay vòng "lấy lệnh-giải mã- thực thi" (fetch-decode-execute)

Slide tiếp theo: sơ lược qua một máy tính đơn giản để minh họa

- Sự tổ chức của máy tính
  - Gồm các thành phần nào và nối kết nhau như thế nào
- Một minh họa "lấy lệnh-giải mã- thực thi"
- Các lệnh cấp cao được chuyển thành cấp thấp (mã máy) như thế nào!

Trong phần lõi của HĐH được xây dựng như một máy tính phức tạp trên nền phần cứng cơ bản

Lấy lệnh-Giải mã-Thực thi

Máy tính là một thiết bị tính toán lớn có nhiều chức năng

Lập trình để nó hỗ trợ nhiều chức năng

Tất cả máy tính von Neumann làm giống nhau:

Lấy lệnh kế tiếp từ bộ nhớ

Giải mã lệnh, chỉ ra phải làm gì

Thực thi lệnh và lưu kết quả

Các lệnh khá đơn giản. Ví dụ:

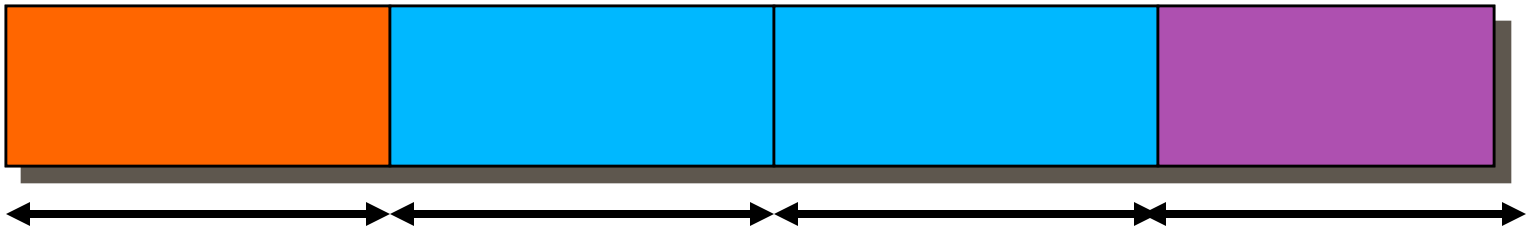
Tăng giá trị của ô nhớ lên 1

Cộng nội dung ô nhớ X và Y rồi lưu vào ô nhớ Z

Nhân nội dung ô nhớ A và B rồi lưu vào ô nhớ B

Giải mã lệnh

Làm sao biểu diễn lệnh bằng các con số?



8 bits
Phép toán

+: 1

-: 2

*: 3

/: 4

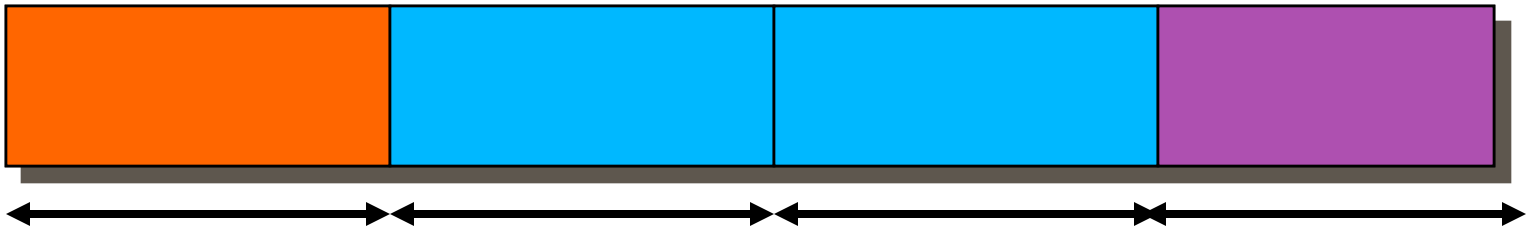
8 bits
Toán hạng

8 bits
Toán hạng

8 bits
Kết quả

Ví dụ giải mã

Cộng ô 28 với ô 63 và lưu kết quả vào ô 100:




```
Các lệnh dưới dạng số:
Thập phân: 1:28:63:100
Nhị phân:  00000001:00011100:00111111:01100100
Thập lục:  01:1C:3F:64
```

Con trỏ lệnh (The Program Counter, Instruction Pointer)

Lệnh kế tiếp lưu ở đâu trong máy tính?

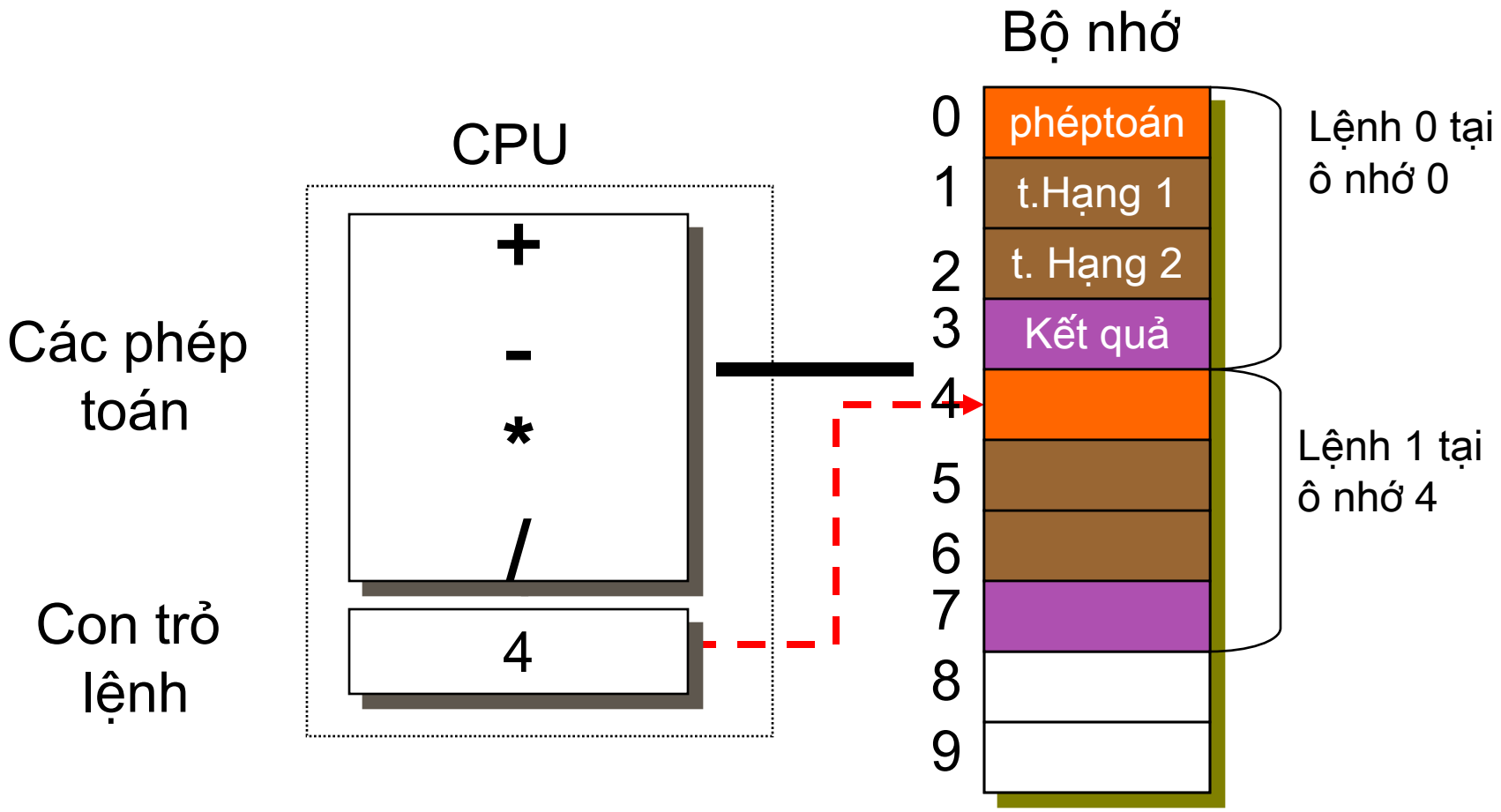
Có một ô nhớ (thanh ghi) đặc biệt trong CPU gọi là bộ đếm“program counter" (PC)

Bộ nhớ cho các mục đích đặc biệt trong CPU và thiết bị gọi là thanh ghi

Tìm nhận lệnh kế tiếp: tăng PC bằng chiều dài lệnh (4) sau khi thực thi mỗi lệnh

Giả thiết tất cả các lệnh cùng chiều dài

Mô hình quan niệm



Bộ nhớ gián tiếp

Làm sao để truy cập các phần tử trong mãng hiệu quả khi mà mọi việc chúng ta có thể làm là đặt tên cho ô nhớ?

Hiệu chỉnh phép toán để cho phép nhận một phép toán từ một ô nhớ

V.d.: LOAD [5], 2 nghĩa là lấy nội dung của ô nhớ có địa chỉ lưu trong ô nhớ 5 và gán vào ô nhớ có địa chỉ 2

Nếu ô 5 lưu số 100, thì chúng ta chép nội dung ô nhớ 100 vào ô nhớ 2

Như vậy gọi là gián tiếp

Nhận nội dung của một ô được trỏ tới bởi một ô khác

Phải tốn một bit trong phép toán để báo hiệu là bộ nhớ truy cập gián tiếp

Điều kiện và vòng lặp

Máy tính sơ khai chỉ cho phép các lệnh tuần tự

Đột phá đầu tiên là điều kiện và rẽ nhánh

Các lệnh làm thay đổi giá trị con trỏ lệnh(Program Counter)

Các lệnh điều kiện

Nếu nội dung ô nhớ là [> 0, <> 0, ...] thì thực thi một số lệnh khác hoặc là không..

Lệnh rẽ nhánh

Nếu nội dung của ô nhớ là [0, <>0, ...], thì gán giá trị PC một vị trí mới

Lệnh nhảy (jump) là một lệnh rẽ nhánh không điều kiện

Ví dụ: vòng lặp While

```
while (counter > 0) {
    sum = sum + Y[counter];
    counter--;
};
```

Các biến trong ô nhớ:

```
counter ở ô 1
sum ở 2
index ở ô 3
Y[0]= ô 4, Y[1]=ô 5…
```

Địa chỉ ô nhớ	Nhãn Assembler	Hợp ngữ	Ý nghĩa
100	LOOP:	BNZ 1,END	// nhảy tới địa chỉ nhãn END // nếu ô nhớ 1 khác 0.

104		ADD 2,[3],2	// Cộng ô 2 với ô
			// được trỏ tới từ
			// từ ô 3 và gán kết quả
			// vào ô 2

108		DEC 3	// giảm ô 3 xuống 1
112		DEC 1	// giảm ô 1 xuống 1
116		JUMP LOOP	// nhảy tới nhãn LOOP
120	END:	<block lệnh kế tiếp>	

Thanh ghi

Luật cơ bản: bộ nhớ lớn thì truy suất chậm, nhỏ thì nhanh

Nhưng ai cũng cần thêm bộ nhớ!

Giải pháp: đặt một lượng nhỏ bộ nhớ trong CPU để tăng tốc tính toán

Hầu hết chương trình chỉ làm việc trên một phần nhỏ bộ nhớ trong một khoảng nhỏ thời gian cho trước. Gọi là tính cục bộ.

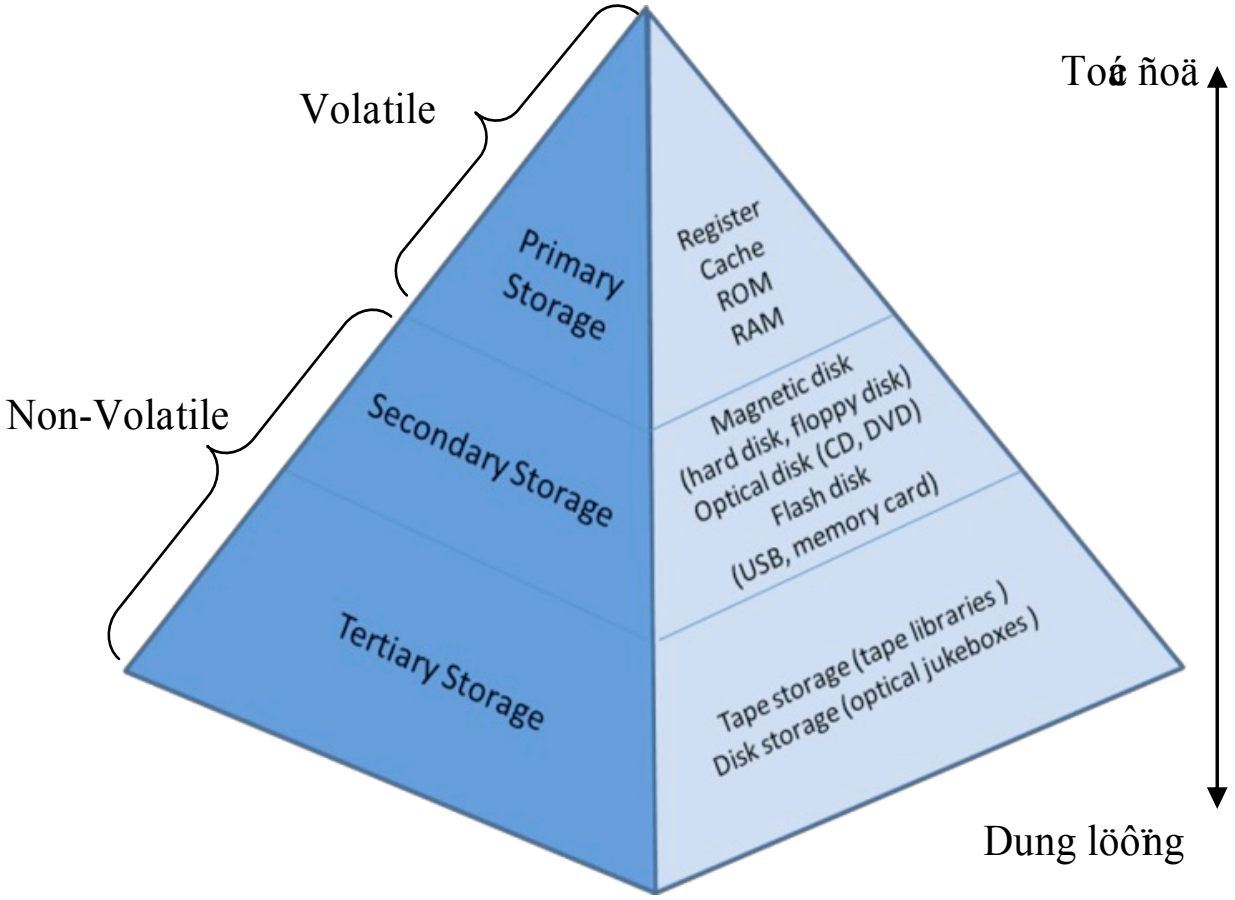
Vì vậy, nếu ta cache một số ô nhớ trong bộ nhớ của CPU, chúng ta có thể thực thi một tập lệnh trước khi phải truy cập bộ nhớ

Bộ nhớ nhỏ trong CPU nằm tách biệt với tập lệnh trong "bộ nhớ chính"

Bộ nhớ nhỏ trong CPU = thanh ghi

Bộ nhớ lớn = bộ nhớ chính

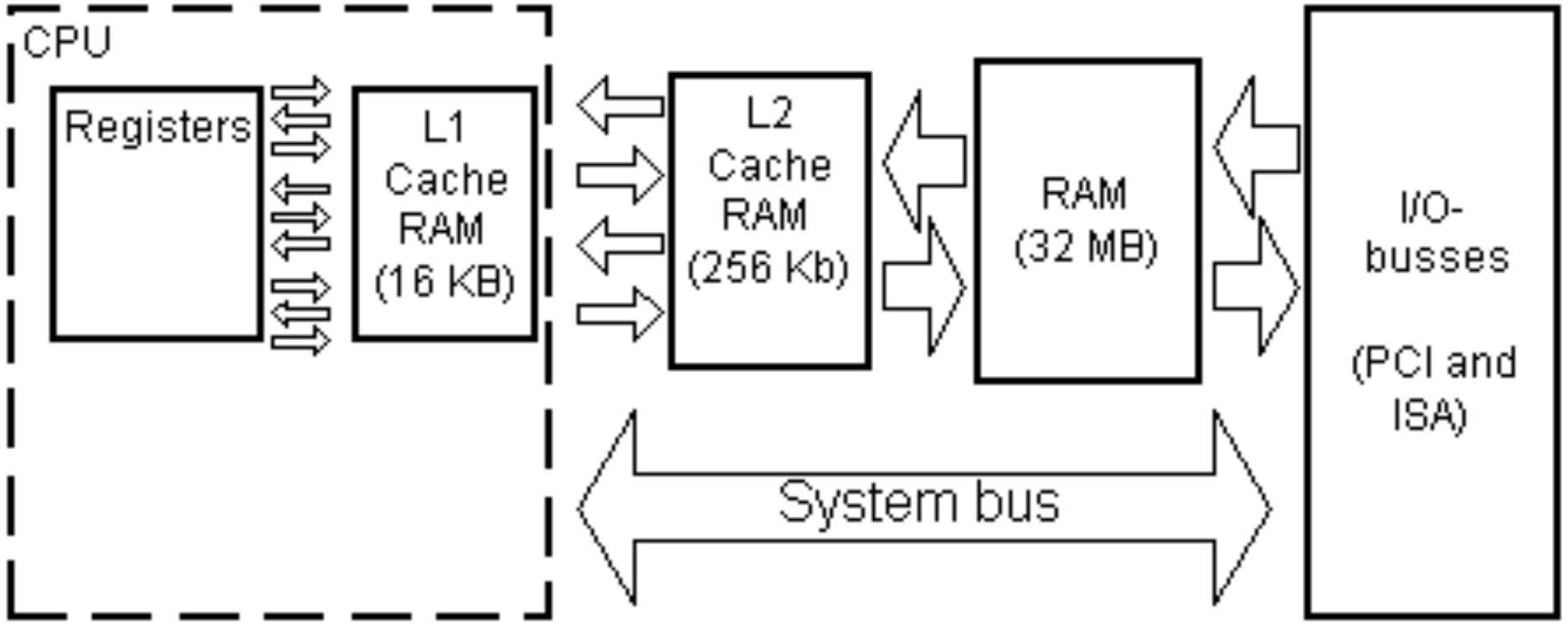
Phân loại bộ nhớ



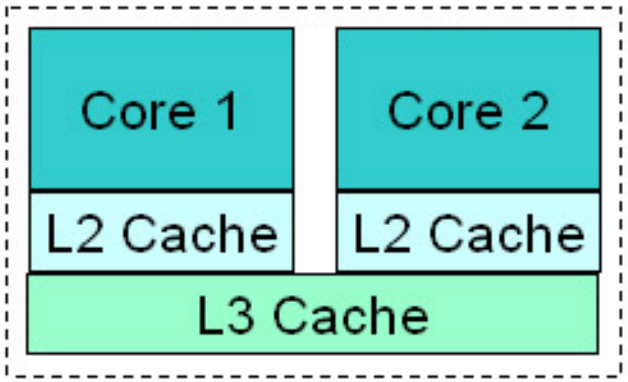
Các loại bộ nhớ khác nhau

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

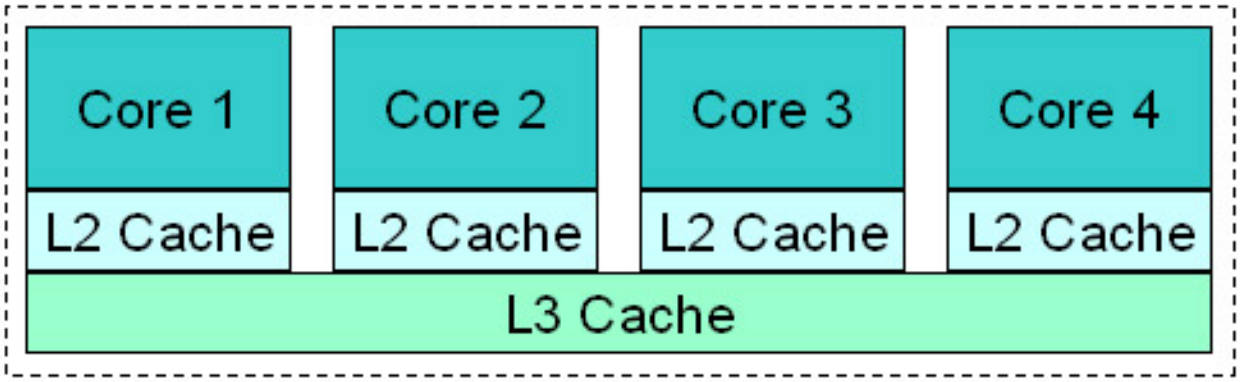
Kiến trúc cache của CPU 1 lõi 1 cache



Kiến trúc cache của CPU AMD nhiều lõi K10

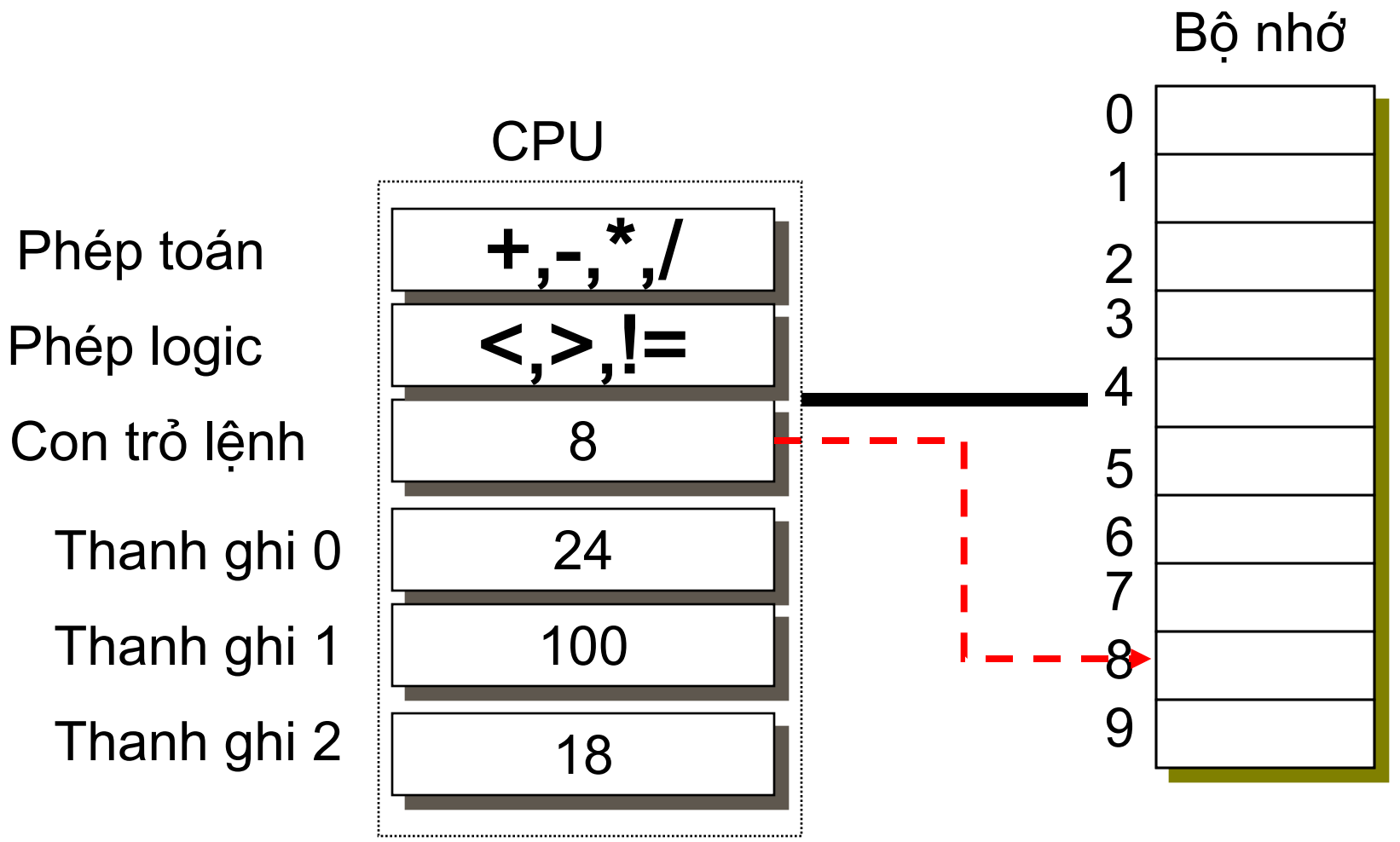


CPU dual-core kiến trúc K10



CPU quad-core kiến trúc K10

Mô hình các thanh ghi



Thanh ghi (tt)

Hầu hết CPU có 16-32 thanh ghi "dùng chung"

Tất cả trông "giống" nhau: kết hợp của phép toán, toán hạng và kết quả

Toán hạng và kết quả có thể ở:

Chỉ trên thanh ghi (Sparc, PowerPC, Mips, Alpha)

Thanh ghi & 1 vùng nhớ để lưu toán hạng (Intel x86 và các họ tương tự)

Kết hợp bất kì giữa thanh ghi và vùng nhớ (Vax)

Phép toán thực hiện nhanh hơn từ 100-1000 khi các toán hạng ở trên thanh ghi so với trong bộ nhớ

Và cũng lưu địa chỉ lệnh

Địa chỉ trong phạm vi 16-32 thanh ghi, không phải hàng GB bộ nhớ

Các lệnh cơ bản

Cộng nội dung thanh ghi 2 và thanh ghi 3 rồi ghi lên thanh ghi 5

ADD r2,r3,r5

Cộng 100 vào con trỏ lệnh(PC) nếu thanh ghi 2 khác 0

Liên quan rẻ nhánh

BNZ r2,100

Nạp nội dung vùng nhớ có địa chỉ lưu trong thanh ghi 5 vào thanh ghi 6

LDI r5,r6

Các đặc trưng kiến trúc cho HĐH

Chúng ta xem xét các đặc tính được thêm vào phần cứng để cho phép người thiết kế HĐH "mềm hóa" máy tính vật lý

- Processor modes
- Exceptions
- Traps
- Interrupts (ngắt)

Yêu cầu có sự thay đổi việc lặp lại "lấy lệnh-giải mã-thực thi" của phần cứng

Processor Modes

HĐH được lưu trong bộ nhớ … mô hình von Neumann?

Điều gì xảy ra nếu người dùng thay đổi mã HĐH hay dữ liệu?

Đưa ra khái niệm modes of operation(chế độ thực thi)

Các lệnh sẽ được thực thi trong user mode hay system mode

Một thanh ghi đặc biệt lưu mode hiện hành

Một số lệnh chỉ có thể được thực hiện trong system mode

Tương tự như vậy, một số vùng nhớ chỉ có thể ghi lên khi đang ở trong system mode

Chỉ có mã nguồn của HĐH được phép ở trong system mode

Chỉ có HĐH có thể thay đổi giá trị trong bộ nhớ của nó

Thanh ghi mode chỉ có thể được thay đổi trong system mode

Lược đồ bảo vệ đơn giản

Tất cả địa chỉ < 100 dành riêng cho HĐH

Thanh ghi mode cho biết

0 = CPU đang thực thi lệnh của HĐH (trong system mode)

1 = CPU đang thực thi trong user mode

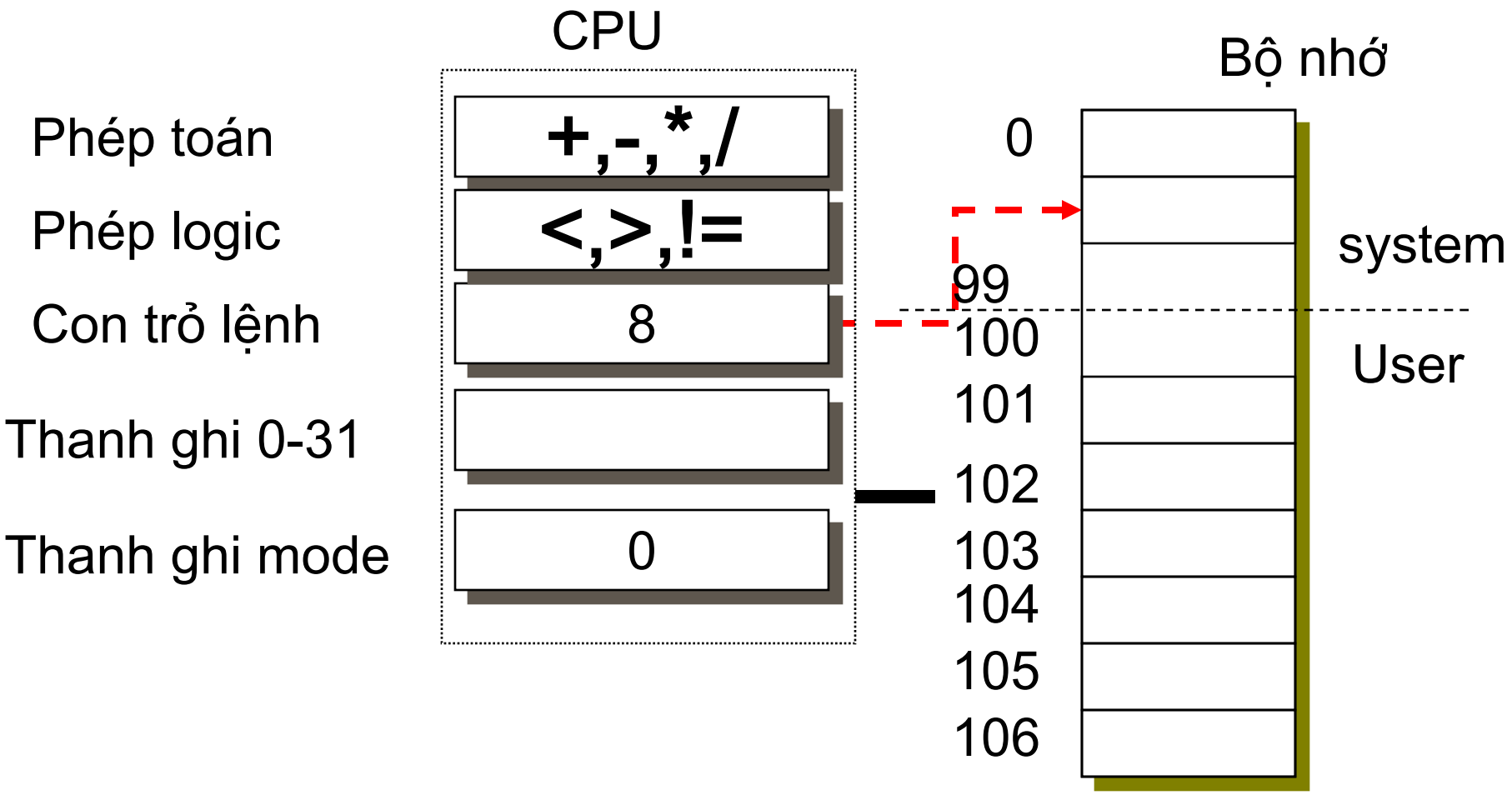
Phần cứng sẽ kiểm tra điều này:

Mỗi lần lấy lệnh, nếu mode bit = 1 và địa chỉ < 100, thì không thực hiện lệnh này

Khi truy cập các toán hạng, nếu mode bit = 1 và địa chỉ toán hạng < 100, thì không thực hiện lệnh này

Thanh ghi mode chỉ có thể được gán giá trị khi nó = 0

Mô hình bảo vệ đơn giản



Viết lại "lấy lệnh-giải mã-thực thi"

Lấy lệnh:

if ((the PC < 100) && (thanh ghi mode == 1)) then

Lỗi! Người dùng muốn truy cập HĐH

else

lấy lệnh tại vị trí PC

Giải mã:

if ((register kết quả == mode) && (thanh ghi mode == 1)) then

Lỗi! Người dùng muốn thay đổi thanh ghi mode

< …>

Thực thi:

if ((địa chỉ toán hạng < 100) && (thanh ghi mode == 1) then

Lỗi! Người dùng muốn truy cập HĐH

else

Thực thi lệnh

Exceptions

Điều gì xảy ra khi người dùng truy xuất vào mã hay dữ liệu của HĐH?

Trả lời: exceptions

Một exception xảy ra khi CPU phát hiện một lệnh mà nó không thể thi hành được

Nhảy tới một vùng xử lý trong HĐH khi exception xảy ra

Lỗi khác nhau sẽ được xử lý bởi những nơi khác nhau ("chuyển đến" ngôn ngữ HĐH)

“Lấy lệnh-giải mã-thực thi” và Exceptions

Lấy lệnh:

if ((the PC < 100) && (the mode bit == 1)) then

gán the PC = 60

gán the mode = 0

lấy lệnh tại địa chỉ PC đang lưu

60 xử lý cho lỗi truy cập vùng nhớ

Giải mã:

if ((thanh ghi kết quả == mode) && (thanh ghi mode == 1)) then

gán the PC = 64

gán the mode = 0

goto Lấy lệnh

64 xử lý cho lỗi việc thay đổi giá trị thanh ghi mode

< … >

Thực thi:

< kiểm tra các toán hạng có bi truy suất sai>

Vi phạm vùng truy cập

Chú ý là cả việc lấy lệnh hoặc dữ liệu từ bộ nhớ đều phải được kiểm tra

Trong khi thực thi phải kiểm tra tất cả các toán hạng

Đây là lược đồ quản lý sơ khai. Chúng ta sẽ học các lược đồ phức tạp hơn, đó là bộ nhớ ảo (*virtual memory)* trong phần sau của môn học

Phục hồi lại sau Exceptions

HĐH biết được nguyên nhân của exception từ vị trí PC

Nhưng làm sao để biết được chỗ nào trong chương trình người dùng phát lỗi?

Giải pháp: thêm một thanh ghi, PC’

- Khi có exception, lưu PC sang PC’ trước khi nạp giá trị mới vào PC

HĐH kiểm tra PC' và thực hiện các công việc phục hồi

- Dừng chương trình người dùng và in ra lỗi: lỗi tại địa chỉ PC'
- Chạy chương trình debugger

Exceptions & phục hồi

Lấy lệnh:

```
if (( the PC < 100) && ( the mode bit == 1)) then
        the PC' = PC
        the PC = 60
        the mode = 0
```

Giải mã:

```
if ((thanh ghi kết quả == mode) && ( thanh ghi mode == 1)) then
        the PC' = PC
        the PC = 64
        the mode = 0
        goto Lấy lệnh
< … >
```

Thực thi:

…

Traps

Bây giờ thi ta đã biết chuyện gì xảy ra khi người dùng truy cập bất hợp pháp vào mã hay dữ liệu của hệ điều hành

Khi chương trình người dùng truy cập một dịch vụ hợp pháp của HĐH thì sao?

Giải pháp: Trap

Trap là một lệnh đặc biệt nó gán PC trỏ tới 1 địa chỉ biết trước và đặt mode thành system mode

Không giống như exceptions, traps gửi thêm các tham số vào HĐH

System call ra đời

“Lấy lệnh-giải mã-thực thi” và traps

Lấy lệnh:

```
if (( the PC < 100) && ( the mode bit == 1)) then
        < exception>
```

Giải mã:

```
if (lệnh là 1 trap) then
        the PC' = PC
        the PC = 68
        the mode = 0
        goto Lấy lệnh
if (( thanh ghi kết quả == mode) && ( the mode bit == 1)) then
        < exeception >
```

Thực thi:

…

Traps

Làm sao HĐH biết dịch vụ nào mà chương trình người dùng muốn gọi?

Chương trình người dùng gửi cho HĐH 1 con số đại diện cho dịch vụ muốn yêu cầu

Ví dụ lệnh trap có chứa luôn trap ID:

Hầu hết CPUs thật có cơ chế ngầm để chuyển mã trap vào thanh ghi

V.d. chương trình người dùng gán thanh ghi 0 một mã trap, rồi thực thi lệnh trap

Trở về sau một Trap

Làm sao trở về user mode và mã chương trình người dùng sau một trap?

Gán thanh ghi mode = 0 rồi gán giá trị cho PC?

Nhưng sau khi mode bit gán sang user mode, -> exception!

Gán giá trị PC, rồi gán giá trị mode bit?

Nhảy tới "vùng người dùng", rồi gán thành user mode

Hầu hết máy tính có lệnh "trở về từ exception"

Một lệnh trong phần cứng:

Hoán đổi PC và PC'

Gán mode bit sang "user mode"

Traps và exceptions sử dụng cùng phương thức (RTE)

Ngắt (Interrupts)

Làm sao để chuyển CPU về system mode khi mà CT người dùng bị kết thúc?

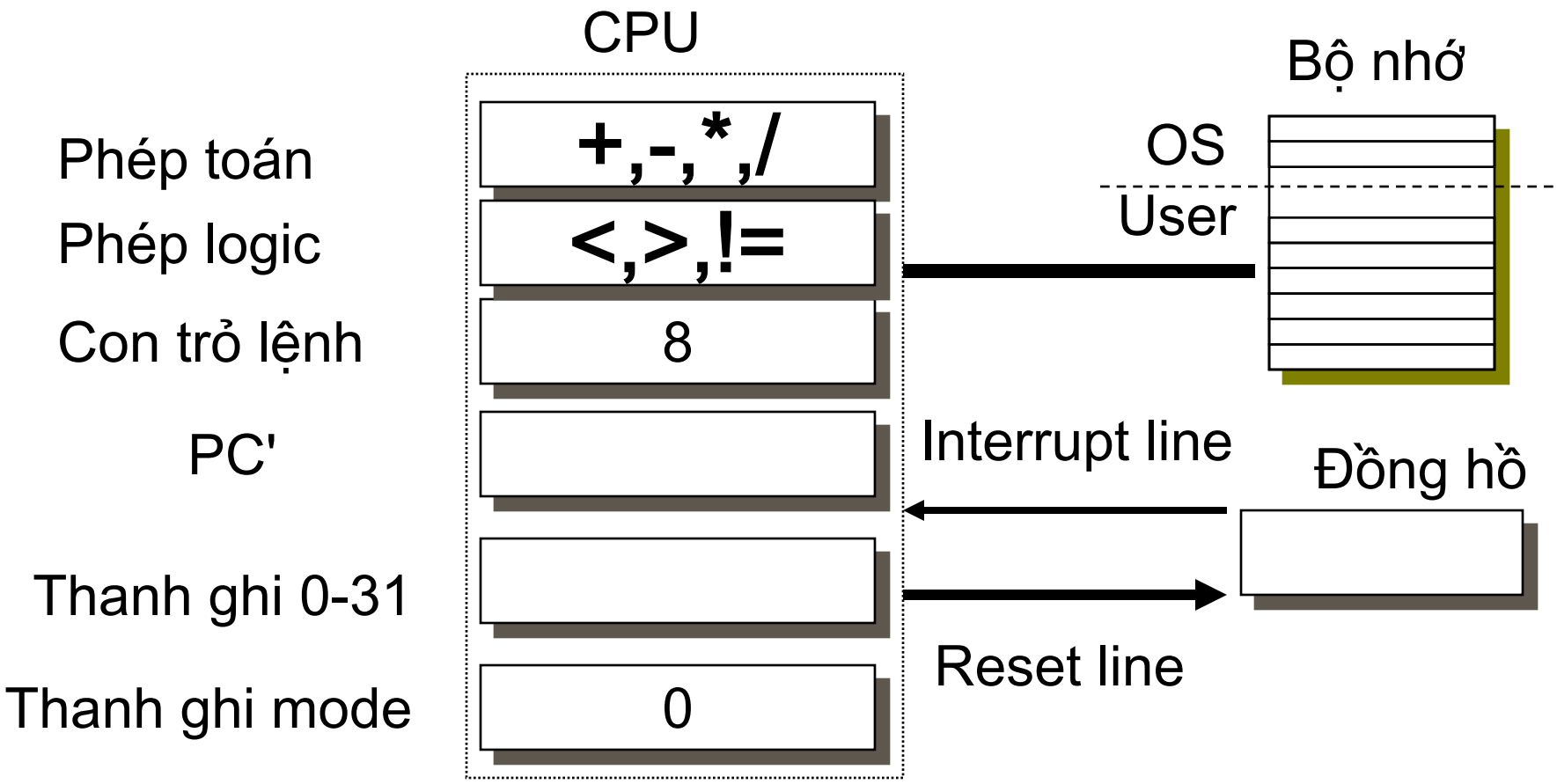
Giải pháp: Interrupts

Một interrupt là một sự kiện bên ngoài làm cho CPU nhảy tới một địa chỉ biết trước

Nối interrupt vào một đồng hồ định kì

Hiệu chỉnh "lấy lệnh-giải mã-thực thi" để kiểm tra các sự kiện bên ngoài theo chu kì của đồng hồ

Mô hình interrupt đơn giản



Đồng hồ

Đồng hồ bắt đầu đếm tới 10 milliseconds (Windows 7: 15,6 ms)

Đồng hồ gán interrupt line "high" (v.d. gán giá trị 1, có thể +5 volts)

Khi CPU gửi tín hiệu qua reset line, đồng hồ đặt interrupt line “low” và đếm lại tới 10 milliseconds

Interrupts

Lấy lệnh:

```
if (the clock interrupt line == 1) then
        the PC' = PC
        the PC = 72
        the mode = 0
        goto Lấy lệnh
if (( the PC < 100) && ( the mode bit == 1)) then
        < memory exception >
Lấy lệnh kế tiếp
```

Giải mã:

```
if (lệnh là một trap) then
        < trap exception >
if (( thanh ghi kết quả == mode) && ( the mode bit == 1)) then
         < mode exeception >
<…>
```

Thực thi: …

Cổng vào hệ thống (Entry Points)

"Entry points" là gì trong ví dụ máy tính đơn giản của chúng ta?

- 60: lỗi truy cập bộ nhớ
- 64: lỗi truy cập thanh ghi mode
- 68: người dùng gọi trap
- 72: ngắt đồng hồ

Mỗi entry point thật ra là một lệnh nhảy đến một đoạn mã trong HĐH

Tất cả các HĐH đều có tập các entry points cho exceptions, traps và interrupts

- Đôi khi chúng được kết hợp với nhau và cần phải có sự phân biệt

Lưu và phục hồi Context

Nhắc lại trạng thái của processor:

PC, PC', R0-R31, thanh ghi mode

Khi có lời gọi vào hệ thống, chúng ta muốn hệ thống làm đúng theo yêu cầu và trả về cho chương trình người dùng sao cho nó có thể thực thi bình thường tiếp tục

Không thể chỉ sử dụng các thanh ghi trong HĐH!

Giải pháp: lưu/phục hồi user context

Sử dụng bộ nhớ của HĐH để lưu toàn bộ trạng thái của CPU

Trước khi trả về cho người dùng, nạp lại giá trị các thanh ghi và rồi thực thi các lệnh để trả về từ exception.

Input và Output

Chúng ta nhận dữ liệu ra sao?

Làm sao nạp chương trình?

Điều gì xảy ra nếu chúng ta tắt máy?

Liệu tôi có thể gửi dữ liệu sang máy tính khác?

Giải pháp: thêm thiết bị để làm các nhiệm vụ này

- Bàn phím, chuột, màn hình
- Ổ đĩa
- Card mạng

Thiết bị I/O cơ bản

Card mạng có 2 thanh ghi:

Một để chứa dữ liệu gửi ra đường truyền.

Việc truyền thường được viết tắt TX (V.d. thanh ghi TX)

Một để chứa dữ liệu nhận từ đường truyền

Nhận thường được viết như là RX

CPU truy cập các thanh ghi này như thế nào?

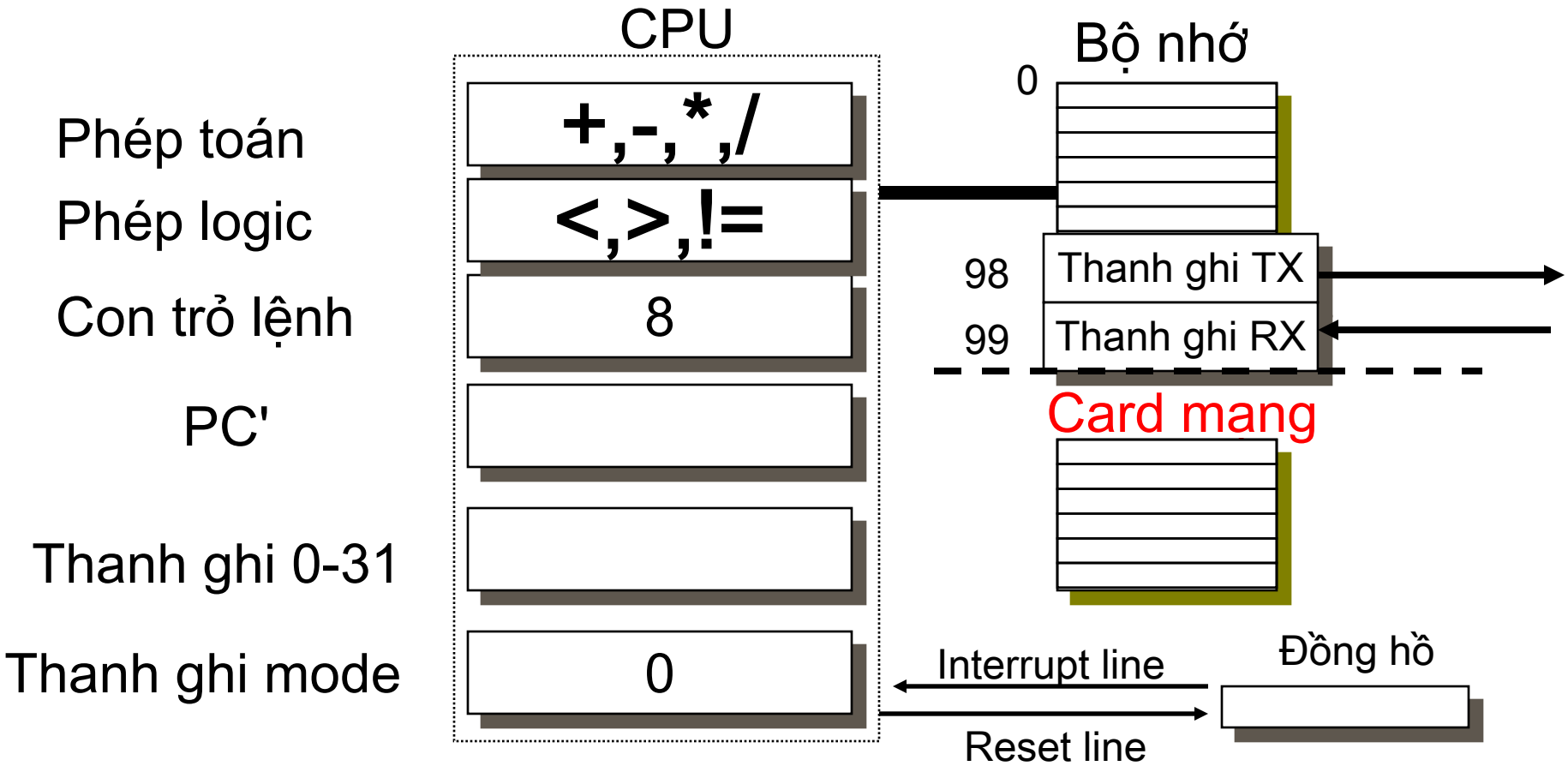
Giải pháp: ánh xạ chúng vào bộ nhớ

Một lệnh truy cập ô nhớ 98 nghĩa là truy cập thanh ghi TX

Một lệnh truy cập ô nhớ 99 nghĩa là truy cập thanh ghi RX

Các thanh ghi này được gọi là "ánh xạ bộ nhớ" (memory-mapped)

I/O đơn giản cho mạng



Tại sao cần "ánh xạ bộ nhớ"

Dùng bộ nhớ cho các thanh ghi thiết bị nhằm 2 tính năng:

- Bảo vệ truy cập --- chỉ HĐH mới có thể truy cập thiết bị.
  - Chương trình ứng dụng phải thông qua HĐH mới truy cập được thiết bị I/O vì các cơ chế bảo vệ đã thảo luận
  - Tại sao chúng ta ngăn chặn truy cập trực tiếp thiết bị từ chương trình người dùng?
- HĐH điều khiển thiết bị, ghi hay đọc dữ liệu vào thiết bị
  - Không cần sự thay đổi tập lệnh
  - Gọi là programmed I/O

Thanh ghi trạng thái

Làm sao HĐH biết có dữ liệu mới đến?

Làm sao HĐH biết byte cuối cùng đã gửi? (thì có thể gửi byte kế tiếp)

Giải pháp: các thanh ghi trạng thái

Một thanh ghi trạng thái lưu trạng thái của lệnh I/O cuối cùng

Card mạng chúng ta có 1 thanh ghi trạng thái

Để gửi, HĐH ghi 1 byte vào thanh ghi TX và chuyển bit thứ 0 trong thanh ghi trạng thái thành 1. Khi gửi xong byte này, nó lại gán bit thứ 0 của thanh ghi trạng thái trở về 0.

Khi card mạng nhận 1 byte, đặt byte này vào thanh ghi RX và đặt bit thứ 1 của thanh ghi trạng thái thành 1. Khi HĐH đọc xong dữ liệu, nó đặt bit thứ 1 của thanh ghi trạng thái trở về 0.

Polled I/O

Gửi:

```
While (bit 0 thanh ghi trạng thái == 1);          // đợi sẵn sàng gửi
Thanh ghi TX = data;
Thanh ghi trạng thái |= 0x1;                      // gán bit 0 thanh ghi trạng thái =1
```

Nhận:

```
While (bit 1 thanh ghi trạng thái != 1);          // đợi dữ liệu đến
Data = thanh ghi RX;
Thanh ghi trạng thái &=  0x01;                    // báo nhận xong
```

Không thể bắt HĐH dừng chờ dữ liệu đến!

Giải pháp: xét lại sau mỗi nhịp đồng hồ

```
If (bit 1 thanh ghi trạng thái == 1)
    Data = thanh ghi RX
    Thanh ghi trạng thái &= 0x01;
```

Dùng Interrupt điều khiển I/O

Polling có thể lãng phí nhiều vòng lặp của CPU

Khi gửi, CPU bị chậm lại theo tốc độ của thiết bị

Không thể ngăn việc nhận, kết hợp polling với đồng hồ, nhưng lãng phí nếu không có dữ liệu đến

Giải pháp: dùng interrupts

Khi có dữ liệu đến, phát tín hiệu bằng interrupt

Khi dữ liệu truyền xong, phát tín hiệu bằng interrupt.

Polling vs. Interrupts

Polling: chúng ta cứ 10 giây check email 1 lần thử có email mới không

Interrupt: nếu có email mới, trình duyệt sẽ phát âm thanh báo.

Nếu tất cả đều dùng polling!

Interrupts yêu cầu chi phí nhiều hơn cho quá trình xử lý:

- Dừng processor
- Chỉ ra loại interrupt
- Lưu trạng thái người dùng
- Yêu cầu xử lý

Nhân tố chính để chọn lựa là tầng số I/O so với chi phí Interrupt

Truy cập bộ nhớ trực tiếp (Direct Memory Access-DMA)

Bài toán I/O: CPU phải ghi/đọc dữ liệu dữ liệu vào các thanh ghi của thiết bị I/O.

Dữ liệu có thể là trong bộ nhớ!

Giải pháp: nhiều thiết bị có thể đọc/ghi trực tiếp vào bộ nhớ tương tự như CPU

- Base + bound hoặc base + count registers trong thiết bị
- Gán giá trị base + count register
- Bật bit trạng thái của thanh ghi truyền
- Thiết bị I/O đọc/ghi bộ nhớ từ địa chỉ base
- Interrupt khi đã xong

Polled IO vs. Truy cập bộ nhớ trực tiếp (DMA)

PIO làm ít việc hơn DMA

- PIO chỉ kiểm tra thanh ghi trạng thái, rồi gửi hay nhận dữ liệu
- DMA phải khởi tạo giá trị base, count, kiểm tra trạng thái và dùng interrupt

DMA hiệu quả hơn trong việc truyền

- PIO trói buộc CPU trong suốt thời gian truyền

Kích thước của dữ liệu **truyền** trở thành yếu tố quyết định để lựa chọn PIO hay DMA

Ví dụ PIO vs. DMA

Cho biết:

Việc NẠP tốn 100 CPU "cycles"

GHI tốn 50 cycles

Xử lý một interrupt tốn 2000 lệnh, trung bình 1 lệnh tốn 2 cycles

Để gửi 1 gói tin dùng PIO tốn 1 NẠP(setup) + 1 GHI cho 1 byte

Nếu dùng DMA tốn 4 NẠP(setup) + 1 interrupt

Tìm kích thước gói tin sao cho gửi bằng DMA tốn ít CPU cycles hơn PIO

Ví dụ PIO vs. DMA

Tìm số byte sao cho chi phí PIO==DMA (làm tròn)

cycles để NẠP: N

cycles để GHI: G

Số byte trong gói tin: B

Số CPU cycles cho mỗi loại:

Số cycles PIO = setup + G*B = N + G*B

Số cycles cho DMA = setup + interrupt

= 4N + 4000

Điều kiện PIO cycles = DMA cycles:

N + G*B = 4N+4000

100+50B = 4(100)+4000

B = 86 bytes (làm tròn)

Khi kích thước gói tin lớn hơn >86 bytes, DMA ít tốn CPU cycles hơn PIO.

Các thiết bị I/O tiêu biểu

Ổ đĩa:

CPU hiểu như một mãng các blocks cùng kích thước.

Card mạng:

Cho phép CPU gửi và nhận các gói tin thông qua mạng (có dây hay không dây)

Kích thước gói tin thông thường 64-8000 bytes

Card màn hình:

CPU hiểu là một vùng nhớ mà có thể chuyển thành điểm ảnh trên màn hình

Tóm tắt: thiết kế I/O

Polling vs. interrupts

Bằng cách nào thiết bị báo cho processor khi có sự kiện mới?

- Polling: thiết bị bị động, CPU phải đọc/ghi vào thanh ghi
- Interrupt: báo hiệu cho CPU bằng interrupt

Programmed I/O vs. DMA

Thiết bị gửi/nhận dữ liệu như thế nào

- Programmed I/O: CPU phải đọc/ghi trên thiết bị
- DMA: thiết bị đọc và ghi trên bộ nhớ

Quá trình khởi động máy tính (1)

- Hệ điều hành nằm ở đâu ?
- Làm sao để máy tính nạp và chạy HĐH lúc khởi động?
  - Quá trình để khởi động HĐH gọi là booting
- Quá trình khởi động của các máy hiện đại gồm 3 giai đoạn
  - CPU thực thi lệnh từ địa chỉ cố định biết trước (boot ROM)
  - Firmware nạp boot loader
  - Boot loader nạp HĐH
- (1) CPU thực thi lệnh từ địa chỉ biết trước trong bộ nhớ
  - Địa chỉ vùng nhớ này thường trỏ tới vùng nhớ chỉ đọc (ROM – read-only memory)
  - Với x86, địa chỉ này là 0xFFFF0, trỏ tới địa chỉ chương trình BIOS (basic input-output system) trong ROM

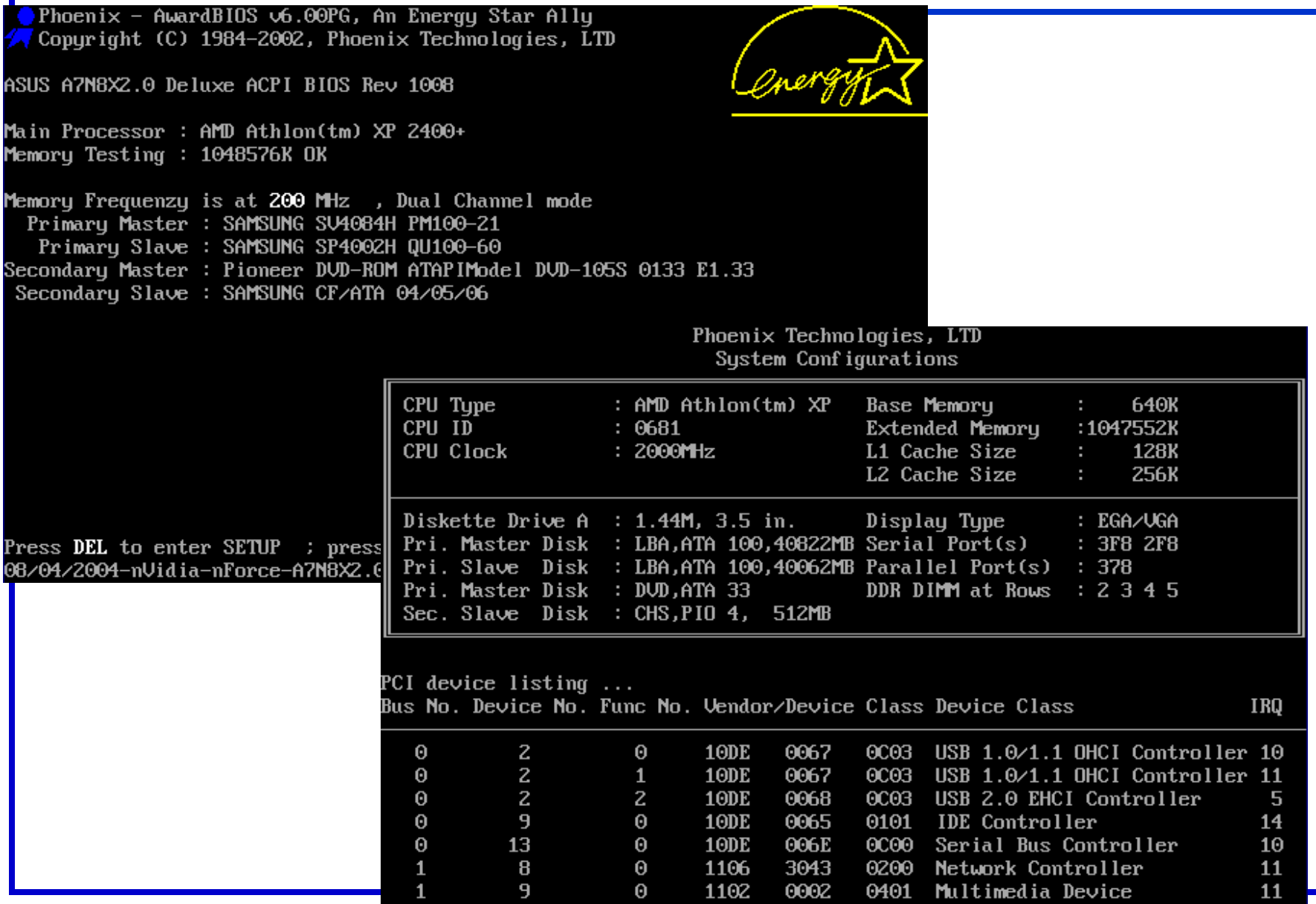
57

Quá trình khởi động máy tính (2)

- (2) ROM chứa mã nguồn "boot"
  - –Loại phần mềm chỉ đọc này gọi là firmware
  - –Với x86, chương trình BIOS thực hiện lần lượt các công việc:
    - •Kiểm tra cấu hình trong CMOS (complementary metal oxide semiconductor)
    - •Nạp trình quản lý ngắt (interrupt handler) và các trình điều khiển thiết bị
    - •Khởi tạo các thanh ghi và quản lý nguồn năng lượng(power management)
    - •Thực hiện quá trình kiểm tra phần cứng (POST – power-on self-test)
    - •Hiển thị các thiết lập hệ thống
    - •Xác định các thiết bị có khả năng khởi động
    - •Tiếp tục quá trình khởi động
  - –Nạp và thực thi chương trình boot loader.

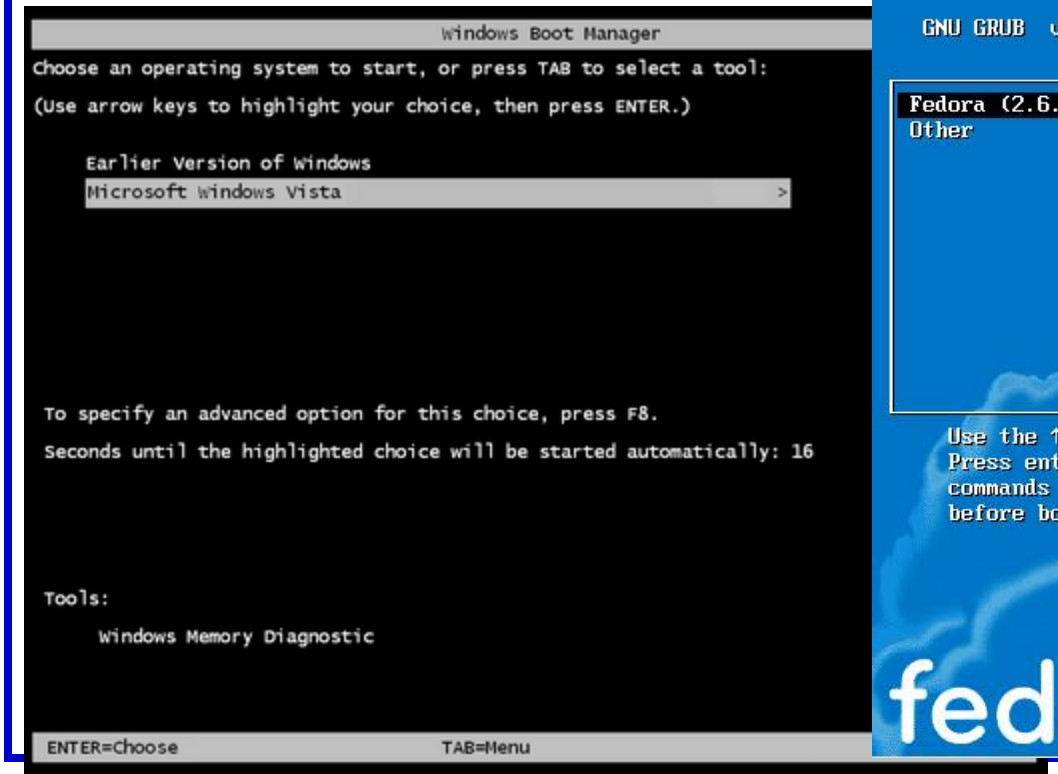
58

Thực thi firmware



Quá trình khởi động máy tính (3)

- (3) Boot loader sau đó nạp phần còn lại của HĐH. Chú ý rằng tại thời điểm này HĐH vẫn chưa chạy
  - Boot loader hiểu được nhiều hệ điều hành khác nhau
  - Boot loader hiểu được nhiều phiên bản khác nhau của các HĐH
  - Đã bao giờ nghe "*dual boot*" ?



Nạp hệ điều hành

Tại sao phải cần 1 chương trình Boot?

Tại sao ta không lưu HĐH vào trong ROM?

Tách HĐH ra khỏi phần cứng

- Nhiều HĐH hay Các phiên bản HĐH khác nhau

Muốn boot từ nhiều thiết bị khác nhau

- V.d. bảo mật thông qua network boot

HĐH thường khá lớn (4-8MB). Không nên làm giống như firmware