

HỆ THỐNG MÁY TÍNH

02 – Biểu diễn số nguyên

Hệ cơ số q tổng quát

2

- Tổng quát số nguyên có n chữ số thuộc hệ cơ số q bất kỳ được biểu diễn:

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot q^{n-1} + \dots + x_1 \cdot q^1 + x_0 \cdot q^0$$

(mỗi chữ số x_i lấy từ tập X có q phần tử)

- Ví dụ:

- Hệ cơ số 10: $A = 123 = 100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$

- $q = 2, X = \{0, 1\}$: hệ nhị phân (binary)

- $q = 8, X = \{0, 1, 2, \dots, 7\}$: hệ bát phân (octal)

- $q = 10, X = \{0, 1, 2, \dots, 9\}$: hệ thập phân (decimal)

- $q = 16, X = \{0, 1, 2, \dots, 9, A, B, \dots, F\}$: hệ thập lục phân (hexadecimal)

- Chuyển đổi: $A = 123_{10} = 01111011_2 = 173_{16} = 7B_h$

- Hệ cơ số thường được biểu diễn trong máy tính là hệ cơ số 2

Chuyển đổi giữa các hệ cơ số

3

□ Đặc điểm

- Con người sử dụng hệ thập phân
- Máy tính sử dụng hệ nhị phân, bát phân, thập lục phân

□ Nhu cầu

- Chuyển đổi qua lại giữa các hệ đếm ?
 - Hệ khác sang hệ thập phân (... \rightarrow dec)
 - Hệ thập phân sang hệ khác (dec \rightarrow ...)
 - Hệ nhị phân sang hệ khác và ngược lại (bin \leftrightarrow ...)
 - ...

Chuyển đổi giữa các hệ cơ số

[1] Decimal (10) → Binary (2)

4

- Lấy số cơ số 10 chia cho 2
 - Số dư đưa vào kết quả
 - Số nguyên đem chia tiếp cho 2
 - Quá trình lặp lại cho đến khi số nguyên = 0

□ Ví dụ: A = 123

- $123 : 2 = 61 \text{ dư } 1$
- $61 : 2 = 30 \text{ dư } 1$
- $30 : 2 = 15 \text{ dư } 0$
- $15 : 2 = 7 \text{ dư } 1$
- $7 : 2 = 3 \text{ dư } 1$
- $3 : 2 = 1 \text{ dư } 1$
- $1 : 2 = 0 \text{ dư } 1$

Kết quả: 1111011, vì 123 là số dương,
thêm 1 bit hiển dấu vào đầu là 0 vào

→ Kết quả cuối cùng: **01111011**

Chuyển đổi giữa các hệ cơ số

[2] Decimal (10) → Hexadecimal (16)

5

- Lấy số cơ số 10 chia cho 16
 - Số dư đưa vào kết quả
 - Số nguyên đem chia tiếp cho 16
 - Quá trình lặp lại cho đến khi số nguyên = 0
 - Ví dụ: A = 123
 - $123 : 16 = 7 \text{ dư } 12 \text{ (B)}$
 - $7 : 16 = 0 \text{ dư } 7$
- Kết quả cuối cùng: **7B**

Chuyển đổi giữa các hệ cơ số

[3] Binary (2) → Decimal (10)

6

- Khai triển biểu diễn và tính giá trị biểu thức

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot 2^{n-1} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

- Ví dụ:

$$\blacksquare 1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$$

Chuyển đổi giữa các hệ cơ số

[4] Binary (2) → Hexadecimal (16)

7

- Nhóm từng **bộ 4 bit** trong biểu diễn nhị phân rồi chuyển sang ký số tương ứng trong hệ thập lục phân (0000 → 0, ..., 1111 → F)

- Ví dụ

$$\blacksquare 1001011_2 = 0100\ 1011 = 4B_{16}$$

HEX	BIN	HEX	BIN	HEX	BIN	HEX	BIN
0	0000	4	0100	8	1000	C`	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Chuyển đổi giữa các hệ cơ số

[5] Hexadecimal (16) → Binary (2)

8

- Sử dụng bảng dưới đây để chuyển đổi:

HEX	BIN	HEX	BIN	HEX	BIN	HEX	BIN
0	0000	4	0100	8	1000	C`	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Ví dụ:

$$\square 4B_{16} = 1001011_2$$

Chuyển đổi giữa các hệ cơ số

[6] Hexadecimal (16) → Decimal (10)

9

- Khai triển biểu diễn và tính giá trị biểu thức

$$x_{n-1}...x_1x_0 = x_{n-1}.16^{n-1} + ... + x_1.16^1 + x_0.16^0$$

- Ví dụ:

- $7B_{16} = 7.16^1 + 12 (B).16^0 = 123_{10}$

Hệ nhị phân

10

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot 2^{n-1} + \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$

- Được dùng nhiều trong máy tính để biểu diễn các giá trị lưu trong các thanh ghi hoặc trong các ô nhớ. Thanh ghi hoặc ô nhớ có kích thước 1 byte (8 bit) hoặc 1 word (16 bit).
- n được gọi là chiều dài bit của số đó
- Bit trái nhất x_{n-1} là bit có giá trị (nặng) nhất **MSB** (Most Significant Bit)
- Bit phải nhất x_0 là bit ít giá trị (nhẹ) nhất **LSB** (Less Significant Bit)

Ý tưởng nhị phân

11

- Số nhị phân có thể dùng để biểu diễn bất kỳ việc gì mà bạn muốn!
- Một số ví dụ:
 - Giá trị logic: 0 → False; 1 → True
 - Ký tự:
 - 26 ký tự (A → Z): 5 bits ($2^5 = 32$)
 - Tính cả trường hợp viết hoa/thường + ký tự lạ → 7 bits (ASCII)
 - Tất cả các ký tự ngôn ngữ trên thế giới → 8, 16, 32 bits (Unicode)
 - Màu sắc: Red (00), Green (01), Blue (11)
 - Vị trí / Địa chỉ: (0, 0, 1)...
 - Bộ nhớ: N bits → Lưu được tối đa 2^N đối tượng

Số nguyên không dấu

12

□ Đặc điểm

- Biểu diễn các đại lượng luôn dương
 - Ví dụ: chiều cao, cân nặng, mã ASCII...
- Tất cả bit đều được sử dụng để biểu diễn giá trị (không quan tâm đến dấu âm, dương)
- Số nguyên không dấu 1 byte lớn nhất là $1111\ 1111_2 = 2^8 - 1 = 255_{10}$
- Số nguyên không dấu 1 word lớn nhất là $1111\ 1111\ 1111\ 1111_2 = 2^{16} - 1 = 65535_{10}$
- Tùy nhu cầu có thể sử dụng số 2, 3... word.
- **LSB = 1** thì số đó là số đó là **số lẻ**

Số nguyên có dấu

13

- Lưu các số dương hoặc âm (số có dấu)
- Có 4 cách phổ biến:
 - [1] Dấu lượng
 - [2] Bù 1
 - [3] Bù 2
 - [4] Số quá (thừa) K
- Số có dấu trong máy tính được biểu diễn ở dạng số bù 2

Số nguyên có dấu

[1] Dấu lượng

14

- **Bit trái nhất (MSB):** bit đánh dấu âm / dương
 - ▣ 0: số dương
 - ▣ 1: số âm
- **Các bit còn lại:** biểu diễn độ lớn của số (hay giá trị tuyệt đối của số)
- **Ví dụ:**
 - ▣ **Một byte 8 bit:** sẽ có 7 bit (trừ đi bit dấu) dùng để biểu diễn giá trị tuyệt đối cho các số có giá trị từ 0000000 (0_{10}) đến 1111111 (127_{10})
→ Ta có thể biểu diễn các số từ **-127_{10} đến $+127_{10}$**
 - ▣ $-N$ và N chỉ khác giá trị bit MSB (bit dấu), phần độ lớn (giá trị tuyệt đối) hoàn toàn giống nhau

Số nguyên có dấu

[2] Bù 1

15

- Tương tự như phương pháp [1], bit MSB dùng làm bit dấu
 - 0: Số dương
 - 1: Số âm
- Các bit còn lại (*) dùng làm độ lớn
- **Số âm: Thực hiện phép đảo bit tất cả các bit của (*)**
- **Ví dụ:**
 - Dạng bù 1 của 00101011 (43) là 11010100 (−43)
 - **Một byte 8 bit:** biểu diễn từ -127_{10} đến $+127_{10}$
 - Bù 1 có hai dạng biểu diễn cho số 0, bao gồm: 00000000 (+0) và 11111111 (−0) (mẫu 8 bit, giống phương pháp [1])
 - Khi thực hiện phép cộng, cũng thực hiện theo quy tắc cộng nhị phân thông thường, tuy nhiên, **nếu còn phát sinh bit nhớ thì phải tiếp tục cộng bit nhớ này vào kết quả vừa thu được**

Số nguyên có dấu

[3] Bù 2

16

- Biểu diễn giống như số bù 1 + **ta phải cộng thêm số 1 vào kết quả (dạng nhị phân)**
- Số bù 2 ra đời khi người ta gặp vấn đề với hai phương pháp dấu lượng [1] và bù 1 [2], đó là:
 - Có hai cách biểu diễn cho số 0 (+0 và -0) → không đồng nhất
 - Bit nhớ phát sinh sau khi đã thực hiện phép tính phải được cộng tiếp vào kết quả → dễ gây nhầm lẫn
- **Phương pháp số bù 2 khắc phục hoàn toàn 2 vấn đề đó**
- **Ví dụ:**
 - **Một byte 8 bit:** biểu diễn từ -128_{10} đến $+127_{10}$ (được lợi 1 số vì chỉ có 1 cách biểu diễn số 0)

Số bù 1 và Số bù 2

17

Số 5 (8 bit)

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

Số bù 1 của 5

1	1	1	1	1	0	1	0
---	---	---	---	---	---	---	---

+

1

Số bù 2 của 5

1	1	1	1	1	0	1	1
---	---	---	---	---	---	---	---

+ Số 5

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

Kết quả

1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---

Nhận xét số bù 2

18

- (Số bù 2 của x) + x = một dãy toàn bit 0 (không tính bit 1 cao nhất do vượt quá phạm vi lưu trữ)

→ Do đó số bù 2 của x chính là giá trị âm của x hay $-x$ (Còn gọi là phép lấy đối)

- Đổi số thập phân âm -5 sang nhị phân?

→ Đổi 5 sang nhị phân rồi lấy số bù 2 của nó

- Thực hiện phép toán $a - b$?

→ $a - b = a + (-b)$ → Cộng với số bù 2 của b .

Số nguyên có dấu

[4] Số quá (thừa) K

19

- Còn gọi là biểu diễn số dịch (*biased representation*)
- Chọn một số nguyên dương K cho trước làm giá trị dịch
- Biểu diễn số N:
 - **+N (dương):** có được bằng cách lấy $K + N$, với K được chọn sao cho tổng của K và một số âm bất kỳ trong miền giá trị luôn luôn dương
 - **-N (âm):** có được bằng cách lấy $K - N$ (hay lấy bù hai của số vừa xác định)
- **Ví dụ:**
 - Dùng 1 Byte (8 bit): biểu diễn từ -128_{10} đến $+127_{10}$
 - Trong hệ 8 bit, biểu diễn $N = 25$, chọn số thừa $k = 128$, :
 - $+25_{10} = 10011001_2$
 - $-25_{10} = 01100111_2$
 - Chỉ có một giá trị 0: $+0 = 10000000_2$, $-0 = 10000000_2$

Nhận xét

20

- **Số bù 2 [3]** → lưu trữ số có dấu và các phép tính của chúng trên máy tính (**thường dùng nhất**)
 - ▣ Không cần thuật toán đặc biệt nào cho các phép tính cộng và tính trừ
 - ▣ Giúp phát hiện dễ dàng các trường hợp bị tràn.
- **Dấu lượng [1] / số bù 1 [2]** → dùng các thuật toán phức tạp và bất lợi vì luôn có hai cách biểu diễn của số 0 (+0 và -0)
- **Dấu lượng [1]** → phép nhân của số có dấu chấm động
- **Số thừa K [4]** → dùng cho số mũ của các số có dấu chấm động

Biểu diễn số âm (số bù 2)

21

$$x_{n-1} \dots x_1 x_0 = x_{n-1} \cdot (-2^{n-1}) + x_{n-2} \cdot 2^{n-2} \dots + x_1 \cdot 2^1 + x_0 \cdot 2^0$$



Phạm vi lưu trữ: $[-2^{n-1}, 2^{n-1} - 1]$

□ Ví dụ:

$$\begin{aligned} \square \text{ 1101 0110}_2 &= -2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 \\ &= -128 + 64 + 16 + 4 + 2 = \\ &= -42_{10} \end{aligned}$$

Ví dụ (số bù 2)

22

$$+123 = 01111011b$$

$$-123 = 10000101b$$

$$0 = 00000000b$$

$$-1 = 11111111b$$

$$-2 = 11111110b$$

$$-3 = 11111101b$$

$$-127 = 10000001b$$

$$-128 = 10000000b$$

Tính giá trị không dấu và có dấu

23

- Tính giá trị không dấu và có dấu của 1 số?
 - Ví dụ số word (16 bit): 1100 1100 1111 0000
 - Số nguyên không dấu ?
 - Tất cả 16 bit lưu giá trị → giá trị là 52464
 - Số nguyên có dấu ?
 - Bit MSB = 1 do đó số này là số âm
 - Áp dụng công thức → giá trị là -13072

Tính giá trị không dấu và có dấu

24

□ Nhận xét

- Bit **MSB** = 0 thì giá trị có dấu bằng giá trị không dấu.
- Bit **MSB** = 1 thì giá trị có dấu bằng giá trị không dấu trừ đi 256 (2^8 nếu tính theo byte) hay 65536 (2^{16} nếu tính theo word).

□ Tính giá trị không dấu và có dấu của 1 số?

- Ví dụ số word (16 bit): **1**100 1100 1111 0000
- Giá trị không dấu = **52464**
- Giá trị có dấu: vì bit **MSB** = 1 nên giá trị có dấu = $52464 - 65536 = -13072$

Phép dịch bit và phép xoay

25

- **Shift left (SHL):** 1100 1010 → 1001 0100
 - Chuyển tất cả các bit sang trái, bỏ bit trái nhất, thêm 0 ở bit phải nhất
- **Shift right (SHR):** 1001 0101 → 0100 1010
 - Chuyển tất cả các bit sang phải, bỏ bit phải nhất, thêm 0 ở bit trái nhất
- **Rotate left (ROL):** 1100 1010 → 1001 0101
 - Chuyển tất cả các bit sang trái, bit trái nhất thành bit phải nhất
- **Rotate right (ROR):** 1001 0101 → 1100 1010
 - Chuyển tất cả các bit sang phải, bit phải nhất thành bit trái nhất

Phép toán Logic

AND, OR, NOT, XOR

26

AND	0	1
0	0	0
1	0	1

“Phép nhân”

OR	0	1
0	0	1
1	1	1

“Phép cộng”

XOR	0	1
0	0	1
1	1	0

“Phép so sánh khác”

NOT	0	1
	1	0

“Phép phủ định”

$$\begin{array}{r} \text{AND} \quad 11010011 \\ \quad \quad 00001111 \\ \hline \quad \quad 00000011 \end{array}$$

$$\begin{array}{r} \text{OR} \quad 00000011 \\ \quad \quad 01100000 \\ \hline \quad \quad 01100011 \end{array}$$

$$\begin{array}{r} \text{XOR} \quad 01100011 \\ \quad \quad 01100011 \\ \hline \quad \quad 00000000 \end{array}$$

$$\begin{array}{r} \text{NOT} \quad 11010011 \\ \hline = \quad 00101100 \end{array}$$

Ví dụ

27

□ $X = 0000\ 1000b = 8d$

→ $X\ \text{shl}\ 2 = 0010\ 0000b = 32d = 8 \cdot 2^2$

→ $(X\ \text{shl}\ 2)\ \text{or}\ X = 0010\ 1000b = 40d = 32 + 8$

□ $Y = 0100\ 1010b = 74d$

→ $((Y\ \text{and}\ 0Fh)\ \text{shl}\ 4) = 1010\ 0000$

OR

OR

→ $((Y\ \text{and}\ F0h)\ \text{shr}\ 4) = 0000\ 0100$

= $1010\ 0100 = 164d$ (không dấu)

= $(164 - 2^8) = -92d$ (có dấu)

Một số nhận xét

28

- $x \text{ SHL } y = x \cdot 2^y$
- $x \text{ SHR } y = x / 2^y$
- **AND** dùng để tắt bit (AND với 0 luôn = 0)
- **OR** dùng để bật bit (OR với 1 luôn = 1)
- **XOR, NOT** dùng để đảo bit (XOR với 1 = đảo bit đó)
- $x \text{ AND } 0 = 0$
- $x \text{ XOR } x = 0$

- **Mở rộng:**
 - Lấy giá trị tại bit thứ i của x : $(x \text{ SHR } i) \text{ AND } 1$
 - Gán giá trị 1 tại bit thứ i của x : $(1 \text{ SHL } i) \text{ OR } x$
 - Gán giá trị 0 tại bit thứ i của x : $\text{NOT}(1 \text{ SHL } i) \text{ AND } x$
 - Đảo bit thứ i của x : $(1 \text{ SHL } i) \text{ XOR } x$

Các phép toán tử

29

- Phép Cộng (+)
- Phép Trừ (-)
- Phép Nhân (*)
- Phép Chia (/)

Phép cộng

30

- Nguyên tắc cơ bản:

$$\begin{array}{r|rr} + & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 10 \end{array}$$

- Ví dụ:

$$\begin{array}{r} \\ \\ \\ \\ \\ \hline 1 \end{array}$$

Phép cộng

31

$$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$$

(a) $(-7) + (+5)$

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$$

(b) $(-4) + (+4)$

$$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$$

(c) $(+3) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$$

(d) $(-4) + (-1)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$$

(e) $(+5) + (+4)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$$

(f) $(-7) + (-6)$

Phép trừ

32

- Nguyên tắc cơ bản: Đưa về phép cộng

$$A - B = A + (-B) = A + (\text{số bù 2 của } B)$$

- Ví dụ: $11101 - 10011 = 11101 + 01101$

					1
	1	1	1	0	1
+	0	1	1	0	1
<hr/>					
1	0	1	0	1	0

Phép trừ

33

$$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$$

(a) $M = 2 = 0010$
 $S = 7 = 0111$
 $-S = 1001$

$$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$$

(b) $M = 5 = 0101$
 $S = 2 = 0010$
 $-S = 1110$

$$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$$

(c) $M = -5 = 1011$
 $S = 2 = 0010$
 $-S = 1110$

$$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$$

(d) $M = 5 = 0101$
 $S = -2 = 1110$
 $-S = 0010$

$$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$$

(e) $M = 7 = 0111$
 $S = -7 = 1001$
 $-S = 0111$

$$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$$

(f) $M = -6 = 1010$
 $S = 4 = 0100$
 $-S = 1100$

Phép nhân

34

- Nguyên tắc cơ bản:

\times	0	1
0	0	0
1	0	1

Phép nhân

35

$$\begin{array}{r} \begin{array}{r} \times \\ 10001 \\ 110 \end{array} \\ \hline 00000 \\ 10001 \\ 10001 \\ \hline 1100110 \end{array}$$

Phép nhân

36

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

$= 11$
 $= 13$
 $= 143$

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 00000000 \\ + 1011 \\ \hline 00001011 \\ + 0000 \\ \hline 00001011 \\ + 1011 \\ \hline 00110111 \\ + 1011 \\ \hline 10001111 \end{array}$$

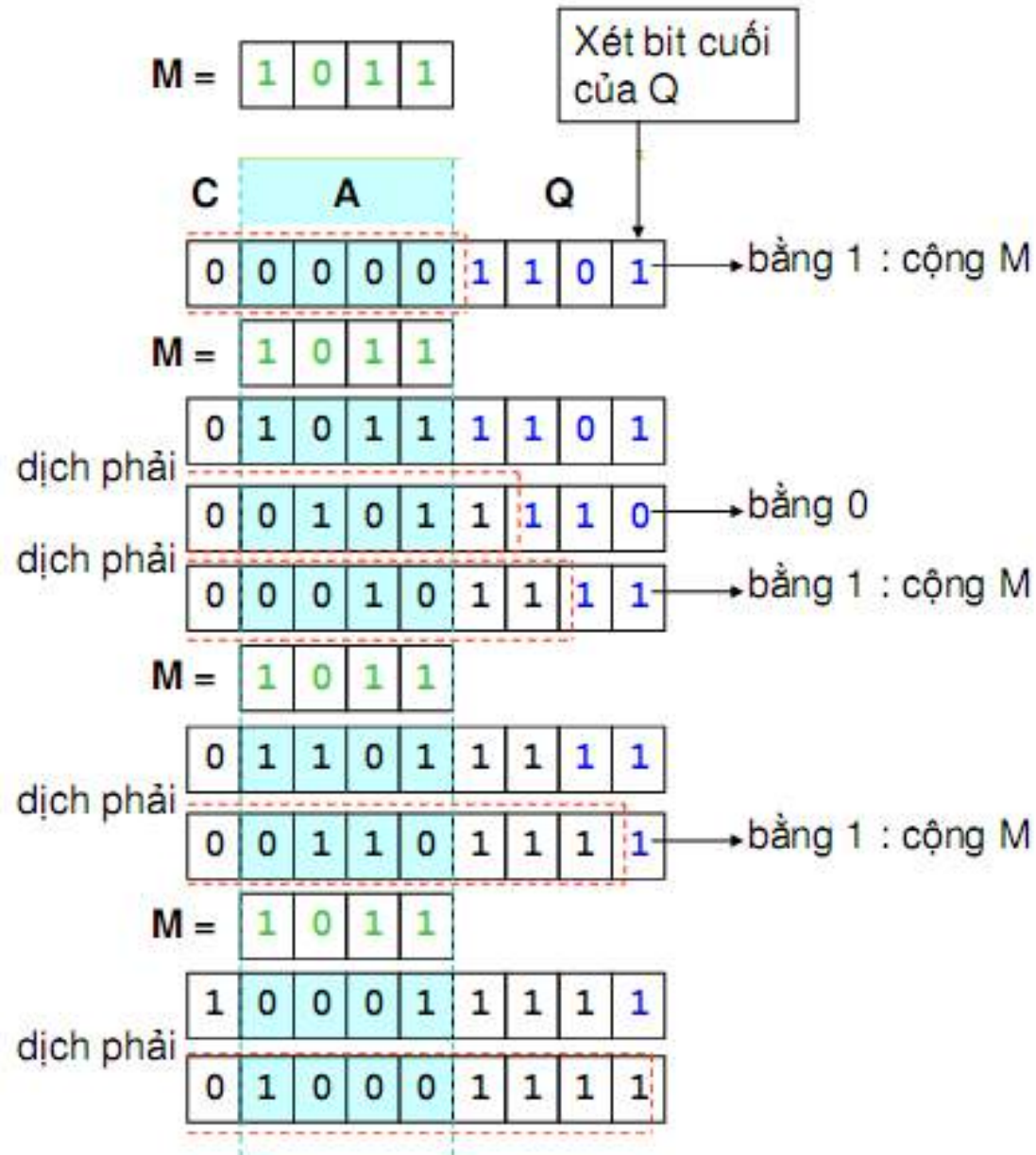
Thuật toán nhân

37

- Giả sử ta muốn thực hiện phép nhân $M \times Q$ với
 - ▣ Q có n bit
- Ta định nghĩa các biến:
 - ▣ C (1 bit): đóng vai trò bit nhớ
 - ▣ A (n bit): đóng vai trò 1 phần kết quả nhân ($[C, A, Q]$: kết quả nhân)
 - ▣ $[C, A]$ ($n + 1$ bit) ; $[C, A, Q]$ ($2n + 1$ bit): coi như các thanh ghi ghép
- Thuật toán:

```
Khởi tạo:  $[C, A] = 0$ ;  $k = n$ 
Lặp khi  $k > 0$ 
{
    Nếu bit cuối của  $Q = 1$  thì
        Lấy  $(A + M) \rightarrow [C, A]$ 

    Shift right  $[C, A, Q]$ 
     $k = k - 1$ 
}
```



Thuật toán nhân cải tiến (số không/có dấu)

39

Khởi tạo: $A = 0$; $k = n$; $Q_{-1} = 0$ (thêm 1 bit = 0 vào cuối Q)

Lặp khi $k > 0$

{

Nếu 2 bit cuối của Q_0Q_{-1}

{

= 10 thì $A - M \rightarrow A$

= 01 thì $A + M \rightarrow A$

= 00, 11 thì A không thay đổi

}

Shift right $[A, Q, Q_{-1}]$

$k = k - 1$

}

Kết quả: $[A, Q]$

Ví dụ

$M = 7, Q = -3, n = 4$

40

	A	Q	Q_{-1}	M
Khởi đầu	0000	1101	0	0111
Bước 0: $A=A-M$	1001	1101	0	0111
shift	1100	1110	1	0111
Bước 1: $A=A+M$	0011	1110	1	0111
shift	0001	1111	0	0111
Bước 2: $A=A-M$	1010	1111	0	0111
shift	1101	0111	1	0111
Bước 3: shift	1110	1011	1	0111

Kết quả $11101011 = -21$

Phép chia

41

□ Giả sử ta muốn thực hiện Q / M với

Khởi tạo: $A = n$ bit 0 nếu $Q > 0$; $A = n$ bit 1 nếu $Q < 0$; $k = n$

Lặp khi $k > 0$

{

Shift left (SHL) $[A, Q]$

$A - M \rightarrow A$

Nếu $A < 0$: $Q_0 = 0$ và $A + M \rightarrow A$

Ngược lại: $Q_0 = 1$

$k = k - 1$

}

Kết quả: Q là thương, A là số dư

Ví dụ phép chia

42

A	Q	M = 0011	A	Q	M = 1101
0000	0111	Initial value	0000	0111	Initial value
0000	1110	Shift	0000	1110	Shift
1101		Subtract	1101		Add
0000	1110	Restore	0000	1110	Restore
0001	1100	Shift	0001	1100	Shift
1110		Subtract	1110		Add
0001	1100	Restore	0001	1100	Restore
0011	1000	Shift	0011	1000	Shift
0000		Subtract	0000		Add
0000	1001	Set $Q_0 = 1$	0000	1001	Set $Q_0 = 1$
0001	0010	Shift	0001	0010	Shift
1110		Subtract	1110		Add
0001	0010	Restore	0001	0010	Restore

(a) (7)/(3)

(b) (7)/(-3)

Prefix in byte (Chuẩn IEC)

43

- International Electrotechnical Commission (IEC)

Name	Abbr	Factor
kibi	Ki	$2^{10} = 1,024$
mebi	Mi	$2^{20} = 1,048,576$
gibi	Gi	$2^{30} = 1,073,741,824$
tebi	Ti	$2^{40} = 1,099,511,627,776$
pebi	Pi	$2^{50} = 1,125,899,906,842,624$
exbi	Ei	$2^{60} = 1,152,921,504,606,846,976$
zebi	Zi	$2^{70} = 1,180,591,620,717,411,303,424$
yobi	Yi	$2^{80} = 1,208,925,819,614,629,174,706,176$

Prefix in byte (Chuẩn SI)

44

□ International System of Units (SI)

Name	Abbr	Factor	SI size
Kilo	K	$2^{10} = 1,024$	$10^3 = 1,000$
Mega	M	$2^{20} = 1,048,576$	$10^6 = 1,000,000$
Giga	G	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$
Tera	T	$2^{40} = 1,099,511,627,776$	$10^{12} = 1,000,000,000,000$
Peta	P	$2^{50} = 1,125,899,906,842,624$	$10^{15} = 1,000,000,000,000,000$
Exa	E	$2^{60} = 1,152,921,504,606,846,976$	$10^{18} = 1,000,000,000,000,000,000$
Zetta	Z	$2^{70} = 1,180,591,620,717,411,303,424$	$10^{21} = 1,000,000,000,000,000,000,000$
Yotta	Y	$2^{80} = 1,208,925,819,614,629,174,706,176$	$10^{24} = 1,000,000,000,000,000,000,000,000$

- **Chú ý:** khi nói “kilobyte” chúng ta nghĩ là 1024 byte nhưng thực ra nó là 1000 bytes theo chuẩn SI, 1024 bytes là kibibyte (IEC)
- Hiện nay chỉ có các nhà sản xuất đĩa cứng và viễn thông mới dùng chuẩn SI
 - **30 GB** → $30 * 10^9 \sim \mathbf{28} * 2^{30}$ bytes
 - 1 Mbit/s → 10^6 b/s

Homework

45

- Đọc chương 9, sách của W.Stalling
- Đọc trước slide bài giảng số thực

HỆ THỐNG MÁY TÍNH

Đặt vấn đề

2

- Biểu diễn số 123.375_{10} sang hệ nhị phân?
 - Ý tưởng đơn giản: Biểu diễn phần nguyên và phần thập phân riêng lẻ
 - Với phần nguyên: Dùng 8 bit ($[0_{10}, 255_{10}]$)
 $123_{10} = 64 + 32 + 16 + 8 + 2 + 1 = 0111\ 1011_2$
 - Với phần thập phân: Tương tự dùng 8 bit
 $0.375 = 0.25 + 0.125 = 2^{-2} + 2^{-3} = 0110\ 0000_2$
- $123.375_{10} = 0111\ 1011.0110\ 0000_2$
- Tổng quát công thức khai triển của số thập phân hệ nhị phân:

$$\underbrace{x_{n-1}x_{n-2}\dots x_0}_{\text{Phần nguyên}} \cdot \underbrace{x_{-1}x_{-2}\dots x_{-m}}_{\text{Phần thập phân}} = \underbrace{x_{n-1}.2^{n-1} + x_{n-2}.2^{n-2} \dots + x_0.2^0}_{\text{Phần nguyên}} + \underbrace{x_{-1}.2^{-1} + x_{-2}.2^{-2} + \dots + x_{-m}.2^{-m}}_{\text{Phần thập phân}}$$

Đặt vấn đề

3

□ Tuy nhiên...với 8 bit:

- Phần nguyên lớn nhất có thể biểu diễn: 255

- Phần thập phân nhỏ nhất có thể biểu diễn: $2^{-8} \sim 10^{-3} = 0.001$

→ Biểu diễn số nhỏ như 0.0001 (10^{-4}) hay 0.000001 (10^{-5})?

→ Một giải pháp: Tăng số bit phần thập phân

- Với 16 bit cho phần thập phân: $\min = 2^{-16} \sim 10^{-5}$

- Có vẻ không hiệu quả...Cách tốt hơn ?

→ Floating Point Number (Số thực dấu chấm động)

Floating Point Number ?

4

- Giả sử ta có số (ở dạng nhị phân)

$$X = 0.\underbrace{00000000000000}_{14 \text{ số } 0}11_2 = (2^{-15} + 2^{-16})_{10}$$

→ $X = 0.11_2 * (2^{-14})_{10} (= (2^{-1} + 2^{-2}).2^{-14} = 2^{-15} + 2^{-16})$

- Thay vì dùng 16 bit để lưu trữ phần thập phân, ta có thể chỉ cần 6 bit:

$$X = 0.11 \text{ } 1110$$

- Cách làm: Di chuyển vị trí dấu chấm sang phải 14 vị trí, dùng 4 bit để lưu trữ số 14 này
- Đây là ý tưởng **cơ bản** của số thực dấu chấm động (floating point number)

Chuẩn hóa số thập phân

5

- Trước khi các số được biểu diễn dưới dạng số chấm động, chúng cần được chuẩn hóa về dạng: $\pm 1.F * 2^E$
 - ▣ F : Phần thập phân không dấu (định trị - Significant)
 - ▣ E : Phần số mũ (Exponent)
- Ví dụ:
 - ▣ $+0.09375_{10} = 0.00011_2 = +1.1 * 2^{-4}$
 - ▣ $-5.25_{10} = 101.01_2 = -1.0101 * 2^2$

Biểu diễn số chấm động

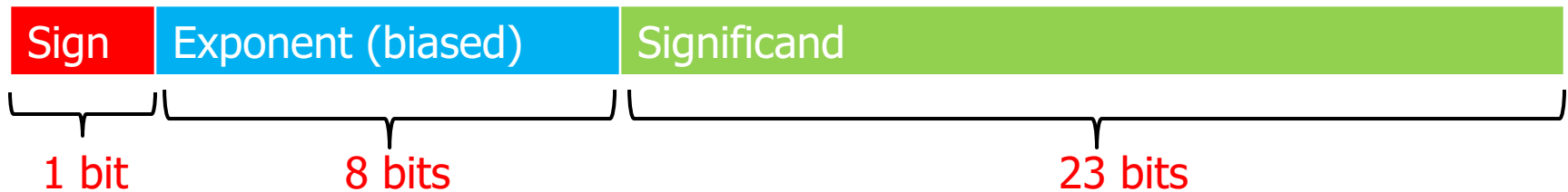
6

- Có nhiều chuẩn nhưng hiện nay chuẩn IEEE 754 được dùng nhiều nhất để lưu trữ số thập phân theo dấu chấm động trong máy tính, gồm 2 dạng:
(slide sau)

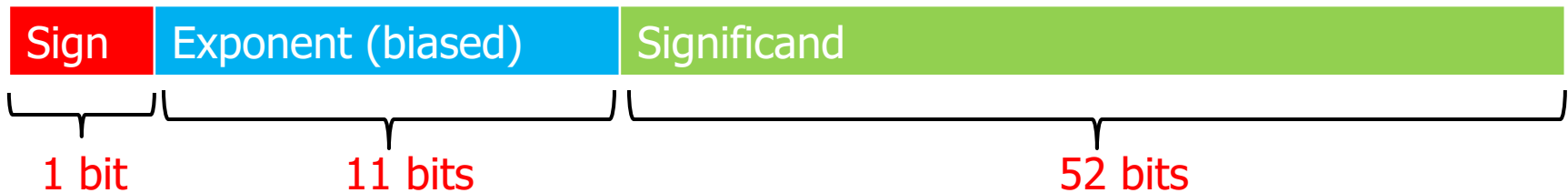
Biểu diễn số chấm động

7

- Số chấm động chính xác đơn (32 bits):



- Số chấm động chính xác kép (64 bits):



- **Sign:** Bit dấu (1: Số âm, 0: Số dương)
- **Exponent:** Số mũ (Biểu diễn dưới dạng số quá K (Biased) với
 - Chính xác đơn: $K = 127$ ($2^{n-1} - 1 = 2^{8-1} - 1$) với n là số bit lưu trữ Exponent
 - Chính xác kép: $K = 1023$ ($2^{n-1} - 1 = 2^{11-1} - 1$)
- **Significand (Fraction):** Phần định trị (phần lẻ sau dấu chấm)

Ví dụ

8

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **X = -5.25**

- **Bước 1:** Đổi X sang hệ nhị phân

$$X = -5.25_{10} = -101.01_2$$

- **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -5.25 = -101.01 = -1.0101 * 2^2$$

- **Bước 3:** Biểu diễn Floating Point

- Số âm: bit dấu Sign = 1

- Số mũ E = 2 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 2 + 127 = 129_{10} = 1000\ 0001_2$$

- Phần định trị = 0101 0000 0000 0000 0000 000 (Thêm 19 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 1 1000 0001 0101 0000 0000 0000 0000 000

Thảo luận về exponent

9

- Vì sao phần số mũ exponent không giữ nguyên lại phải lưu trữ dưới dạng số quá K (Dạng biased)?
- Giả sử trong số chấm động chính xác đơn (32 bits), ta dùng 8 bits để lưu giá trị exponent (biểu diễn dưới dạng số quá K), vậy miền giá trị của nó là [0, 255]
- Với $K = 127$, số mũ gốc ban đầu có miền giá trị [-127, 128]
- Miền giá trị này **khá vô lý**, vậy tại sao chúng ta không chọn số $K = 128$ để miền giá trị gốc là [-128, 127] như bình thường?

Câu hỏi 1 - Đáp án

10

- Sở dĩ Exponent được lưu trữ dưới dạng Biased vì ta muốn chuyển từ miền giá trị **số có dấu** sang **số không dấu** (vì trong biased, số k được chọn để sau khi cộng số bất kỳ trong miền giá trị gốc, kết quả là số luôn dương)
→ Dễ dàng so sánh, tính toán

Câu hỏi 2 - Đáp án

11

- Số K được chọn là 127 mà không phải là 128 vì tại bước 2 trước khi biểu diễn thành số chấm động, chúng ta cần phải chuẩn hóa thành dạng $\pm 1.F * 2^E$
 - Tức là chúng ta sẽ **luôn luôn để dành 1 bit** (số 1) phía trước dấu chấm chứ không đẩy sang trái hết
- Với 8 bit, số mũ gốc ban đầu không thể đạt mức nhỏ nhất là -128 mà chỉ là -127
- Do vậy ta chỉ cần chọn $K = 127$ là được

Vậy thì...

12

- Khi muốn biểu diễn số 0 thì ta không thể tìm ra bit trái nhất có giá trị = 1 để đẩy dấu chấm động, vậy làm sao chuẩn hóa về dạng $\pm 1.F * 2^E$?
 - Với số dạng $\pm 0.F * 2^{-127}$ thì chuẩn hóa được nữa không?
 - Với $K = 127$, exponent lớn nhất sẽ là 255
- Số mũ gốc ban đầu lớn nhất là $255 - 127 = +128$
- **Vô lý** vì với 8 bit có dấu ta không thể biểu diễn được số +128 ?

Trả lời

13

- Vì đó là những **số thực đặc biệt**, ta không thể biểu diễn bằng dấu chấm động 😊

Số thực đặc biệt

14

- Số 0 (zero)
 - Exponent = 0, Significand = 0
- Số không thể chuẩn hóa (denormalized)
 - Exponent = 0, Significand \neq 0
- Số vô cùng (infinity)
 - Exponent = 111...1 (toàn bit 1), Significand = 0
- Số báo lỗi (NaN – Not a Number)
 - Exponent = 111...1 (toàn bit 1), Significand \neq 0

Normalized number

15

- Largest positive normalized number: $+1.[23 \text{ số } 1] * 2^{127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	1111 1110	1111 1111 1111 1111 1111 111

- Smallest positive normalized number: $+1.[23 \text{ số } 0] * 2^{-126}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0001	0000 0000 0000 0000 0000 000

- Tương tự cho số negative (số âm)

Denormalized number

16

- Largest positive denormalized number: $+0.[23 \text{ số } 1] * 2^{-127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0000	1111 1111 1111 1111 1111 111

Tuy nhiên IEEE 754 quy định là $+0.[23 \text{ số } 1] * 2^{-126}$ vì muốn tiến gần hơn với "Smallest positive normalized number = $+1.[23 \text{ số } 0] * 2^{-126}$ "

- Smallest positive denormalized number: $+1.[22 \text{ số } 0]1 * 2^{-127}$

S	Exp	Significand (Fraction)
-	-----	-----
0	0000 0000	0000 0000 0000 0000 0000 001

Tuy nhiên IEEE 754 quy định là $+0.[22 \text{ số } 0]1 * 2^{-126}$

- Tương tự cho số negative (số âm)

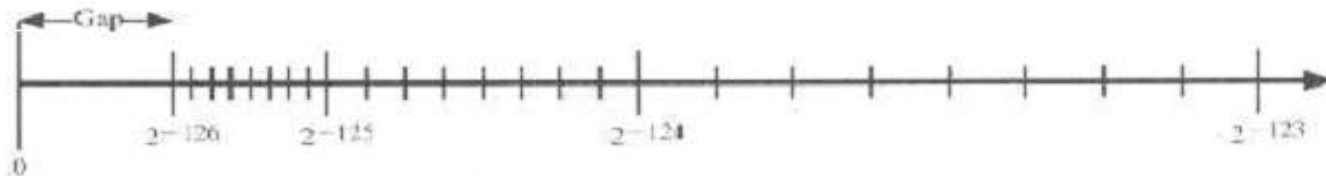
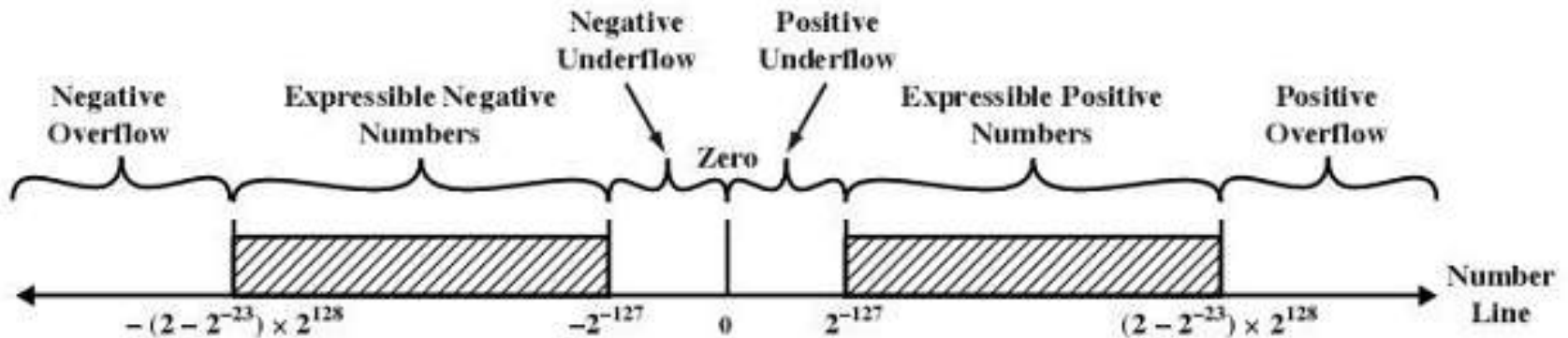
Ví dụ: $n = 4, m = 3, \text{bias} = 7$

17

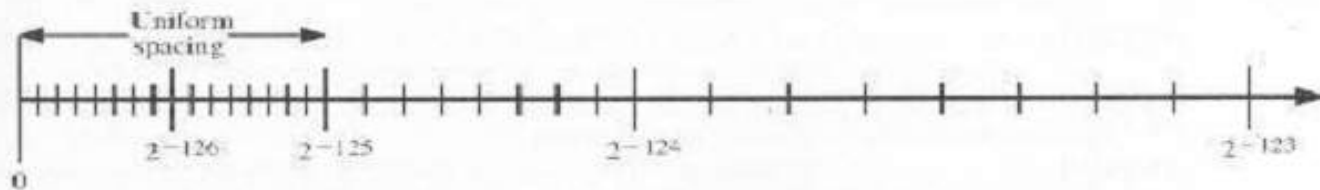
	s	exp	frac	E	Value
Denormalized numbers	0	0000	000	-6	0
	0	0000	001	-6	$1/8 * 1/64 = 1/512$ ← closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$
	...				
	0	0000	110	-6	$6/8 * 1/64 = 6/512$
	0	0000	111	-6	$7/8 * 1/64 = 7/512$ ← largest denorm
				
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$ ← smallest norm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$
	...				
	0	0110	110	-1	$14/8 * 1/2 = 14/16$
	0	0110	111	-1	$15/8 * 1/2 = 15/16$ ← closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$
	0	0111	001	0	$9/8 * 1 = 9/8$ ← closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$
	...				
	0	1110	110	7	$14/8 * 128 = 224$
	0	1110	111	7	$15/8 * 128 = 240$ ← largest norm
				
	0	1111	000	n/a	inf

Phân bố các số thực (32 bits)

18



Without denormalized numbers



With denormalized numbers

Chuẩn IEEE 754

19

Parameter	Format			
	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥ 43	64	≥ 79
Exponent width (bits)	8	≥ 11	11	≥ 15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥ 1023	1023	≥ 16383
Minimum exponent	-126	≤ -1022	-1022	≤ -16382
Number range (base 10)	$10^{-38}, 10^{+38}$	unspecified	$10^{-308}, 10^{+308}$	unspecified
Significand width (bits)*	23	≥ 31	52	≥ 63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

* not including implied bit

Bài tập 1

20

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **X = +12.625**

- **Bước 1:** Đổi X sang hệ nhị phân

$$X = -12.625_{10} = -1100.101_2$$

- **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -12.625_{10} = -1100.101_2 = -1.100101 * 2^3$$

- **Bước 3:** Biểu diễn Floating Point

- Số dương: bit dấu Sign = 0

- Số mũ E = 3 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 3 + 127 = 130_{10} = 1000\ 0010_2$$

- Phần định trị = 1001 0100 0000 0000 0000 000 (Thêm 17 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 0 1000 0010 1001 0100 0000 0000 0000 000

Bài tập 2

21

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): **X = -3050**

- **Bước 1:** Đổi X sang hệ nhị phân

$$X = -3050_{10} = -1011\ 1110\ 1010_2$$

- **Bước 2:** Chuẩn hóa theo dạng $\pm 1.F * 2^E$

$$X = -3050_{10} = -1011\ 1110\ 1010_2 = -1.01111101010 * 2^{11}$$

- **Bước 3:** Biểu diễn Floating Point

- Số âm: bit dấu Sign = 1

- Số mũ E = 11 → Phần mũ exponent với số thừa K=127 được biểu diễn:

$$\rightarrow \text{Exponent} = E + 127 = 11 + 127 = 138_{10} = 1000\ 1010_2$$

- Phần định trị = 0111 1101 0100 0000 0000 000 (Thêm 12 số 0 cho đủ 23 bit)

→ Kết quả nhận được: 1 1000 1010 0111 1101 0100 0000 0000

Bài tập 3

22

- Biểu diễn số thực sau theo dạng số chấm động chính xác đơn (32 bit): $X = +1.1 * 2^{-128}$
 - **Lưu ý:**
 - Số X: positive number
 - $X < \text{Smallest positive normalized number: } +1.[23 \text{ số } 0] * 2^{-126}$
 - số X là số không thể chuẩn hóa (denormalized number)
 - **Chuyển X về dạng: $X = +0.011 * 2^{-126}$**
 - **Bước 3:** Biểu diễn Floating Point
 - Số dương: bit dấu Sign = 0
 - Vì đây là số không thể chuẩn hóa → Phần mũ exponent được biểu diễn: $0000\ 0000_2$
 - Phần định trị = $0110\ 0000\ 0000\ 0000\ 0000\ 000$
- Kết quả nhận được: $0\ 0000\ 0000\ 0110\ 0000\ 0000\ 0000\ 0000\ 000$

Homework

23

- Sách W.Stalling – Computer Arithmetic, đọc chương 9
- Đọc file 04_FloatingPoint.doc
- Trả lời các câu hỏi:
 - ▣ Overflow, underflow?
 - ▣ Cộng trừ nhân chia trên số thực?
 - ▣ Quy tắc làm tròn?
 - ▣ NaN: nguyên tắc phát sinh?
 - ▣ Quiet NaN và Signaling NaN?

HỆ THỐNG MÁY TÍNH

04 – BỘ LỆNH MIPS 32 bit

Giới thiệu

2

- Nhiệm vụ cơ bản nhất của CPU là phải thực hiện các lệnh được yêu cầu, gọi là **instruction**
- Các CPU sẽ sử dụng các tập lệnh (instruction set) khác nhau để có thể giao tiếp với nó

Kích thước lệnh

3

- Kích thước lệnh bị ảnh hưởng bởi:
 - ▣ Cấu trúc đường truyền bus
 - ▣ Kích thước và tổ chức bộ nhớ
 - ▣ Tốc độ CPU
- Giải pháp tối ưu lệnh:
 - ▣ Dùng lệnh có kích thước ngắn, mỗi lệnh chỉ nên được thực thi trong đúng 1 chu kỳ CPU
 - ▣ Dùng bộ nhớ cache

Bộ lệnh MIPS

4

- Chúng ta sẽ làm quen với tập lệnh cho kiến trúc MIPS (PlayStation 1, 2; PSP; Windows CE, Routers...)
- Được xây dựng theo kiến trúc (RISC) với 4 nguyên tắc:
 - ▣ Càng đơn giản, càng ổn định
 - ▣ Càng nhỏ gọn, xử lý càng nhanh
 - ▣ Tăng tốc xử lý cho những trường hợp thường xuyên xảy ra
 - ▣ Thiết kế đòi hỏi sự thỏa hiệp tốt

Cấu trúc cơ bản của 1 chương trình hợp ngữ trên MIPS

5

```
.data                                # khai báo các data label (có thể hiểu là các biến)  
                                     # sau chỉ thị này
```

```
label1: <kiểu lưu trữ> <giá trị khởi tạo>
```

```
label2: <kiểu lưu trữ> <giá trị khởi tạo>
```

```
...
```

```
.text                                # viết các lệnh sau chỉ thị này
```

```
.globl <các text label toàn cục, có thể truy xuất từ các file khác>
```

```
.globl main                          # Đây là text label toàn cục bắt buộc của program
```

```
...
```

```
main:                               # điểm text label bắt đầu của program
```

```
...
```

Hello.asm

6

```
.data                                # data segment
str:  .asciiz "Hello asm !"

.text                                # text segment
.globl  main

main:                                # starting point of program
    addi $v0, $0, 4                  # $v0 = 0 + 4 = 4 → print str syscall
    la $a0, str                      # $a0 = address(str)
    syscall                          # excute the system call
```

Bộ lệnh MIPS – Thanh ghi

7

- Là đơn vị lưu trữ data duy nhất trong CPU
- Trong kiến trúc MIPS:
 - Có tổng cộng 32 thanh ghi đánh số từ \$0 → \$31
 - Càng ít càng dễ quản lý, tính toán càng nhanh
 - Có thể truy xuất thanh ghi qua tên của nó (slide sau)
 - Mỗi thanh ghi có kích thước cố định 32 bit
 - Bị giới hạn bởi khả năng tính toán của chip xử lý
 - Kích thước toán hạng trong các câu lệnh MIPS bị giới hạn ở 32 bit, nhóm 32 bit gọi là từ (word)

Thanh ghi toán hạng

8

- Như chúng ta đã biết khi lập trình, biến (variable) là khái niệm rất quan trọng khi muốn biểu diễn các toán hạng để tính toán
- Trong kiến trúc MIPS không tồn tại khái niệm biến, thay vào đó là thanh ghi toán hạng

Thanh ghi toán hạng

9

- Ngôn ngữ cấp cao (C, Java...): toán hạng = biến (variable)
 - ▣ Các biến lưu trong bộ nhớ chính
- Ngôn ngữ cấp thấp (Hợp ngữ): toán hạng chứa trong các thanh ghi
 - ▣ Thanh ghi không có kiểu dữ liệu
 - ▣ Kiểu dữ liệu thanh ghi được quyết định bởi thao tác trên thanh ghi
- So sánh:
 - ▣ **Ưu:** Thanh ghi truy xuất nhanh hơn nhiều bộ nhớ chính
 - ▣ **Khuyết:** Không như bộ nhớ chính, thanh ghi là phần cứng có số lượng giới hạn và cố định → Phải tính toán kỹ khi sử dụng

Một số thanh ghi toán hạng quan tâm

10

□ Save register:

- MIPS lấy ra 8 thanh ghi (\$16 - \$23) dùng để thực hiện các phép tính số học, được đặt tên tương ứng là **\$s0 - \$s7**
- Tương ứng trong C, để chứa giá trị biến (variable)

□ Temporary register:

- MIPS lấy ra 8 thanh ghi (\$8 - \$15) dùng để chứa kết quả trung gian, được đặt tên tương ứng là **\$t0 - \$t7**
- Tương ứng trong C, để chứa giá trị biến tạm (temporary variable)

Bảng danh sách thanh ghi MIPS

11

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Thanh ghi 1 (\$at) để dành cho assembler. Thanh ghi 26 – 27 (\$k0 - \$k1) để dành cho OS

Bộ lệnh MIPS – 4 thao tác chính

12

- Phần 1: Phép toán số học (Arithmetic)
- Phần 2: Di chuyển dữ liệu (Data transfer)
- Phần 3: Thao tác luận lý (Logical)
- Phần 4: Rẽ nhánh (Un/Conditional branch)

Phần 1: Phép toán số học

13

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

- ▣ **opt** (operator): Tên thao tác (toán tử, tác tử)
- ▣ **opr** (operand): Thanh ghi (toán hạng, tác tố đích)
chứa kết quả
- ▣ **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)
- ▣ **opr2** (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Ví dụ

14

- Giả sử xét câu lệnh sau:

add a, b, c

- ▣ Chỉ thị cho CPU thực hiện phép cộng

$a \leftarrow b + c$

- ▣ a, b, c được gọi là thanh ghi toán hạng
- ▣ Phép toán trên chỉ có thể thực hiện với đúng 3 toán hạng (không nhiều cũng không ít hơn)

Cộng, trừ số nguyên

15

□ Cộng (Add):

- Cộng có dấu: `add $s0, $s1, $s2`
- Cộng không dấu: `addu $s0, $s1, $s2` (u: unsigned)
- Diễn giải: $\$s0 \leftarrow \$s1 + \$s2$
C/C++: $(a = b + c)$

□ Trừ (Subtract):

- Trừ có dấu: `sub $s0, $s1, $s2`
- Trừ không dấu: `subu $s0, $s1, $s2` (u: unsigned)
- Diễn giải: $\$s0 \leftarrow \$s1 - \$s2$
C/C++: $(a = b - c)$

Nhận xét

16

- Toán hạng trong các lệnh trên phải là thanh ghi
- Trong MIPS, lệnh thao tác với số nguyên có dấu được biểu diễn dưới dạng bù 2
- Làm sao biết 1 phép toán được biên dịch từ C (ví dụ $a = b + c$) là thao tác có dấu hay không dấu? → Dựa vào trình biên dịch
- Có thể dùng 1 toán hạng vừa là nguồn vừa là đích
add \$s0, \$s0, \$s1
- Cộng, trừ với hằng số? → \$s2 sẽ đóng vai trò là hằng số
 - Cộng: addi \$s0, \$s1, 3 (addi = add immediate)
 - Trừ: addi \$s0, \$s1, -3

Ví dụ 1

17

- Chuyển thành lệnh MIPS từ lệnh C:

$$\mathbf{a = b + c + d - e}$$

- Chia nhỏ thành nhiều lệnh MIPS:

`add $s0, $s1, $s2` `# a = b + c`

`add $s0, $s0, $s3` `# a = a + d`

`sub $s0, $s0, $s4` `# a = a - e`

- Tại sao dùng nhiều lệnh hơn C?

→ Bị giới hạn bởi số lượng cổng mạch toán tử và thiết kế bên trong cổng mạch

- Ký tự “`#`” dùng để chú thích trong hợp ngữ cho MIPS

Ví dụ 2

18

- Chuyển thành lệnh MIPS từ lệnh C:

$$\mathbf{f = (g + h) - (i + j)}$$

- Chia nhỏ thành nhiều lệnh MIPS:

add	\$t0, \$s1, \$s2	# temp1 = g + h
add	\$t1, \$s3, \$s4	# temp2 = i + j
sub	\$s0, \$t0, \$t1	# f = temp1 - temp2

Lưu ý: Phép gán ?

19

- Kiến trúc MIPS không có cổng mạch dành riêng cho phép gán

→ Giải pháp: Dùng thanh ghi zero (\$0 hay \$zero) luôn mang giá trị 0

- Ví dụ:

`add $s0, $s1, $zero`

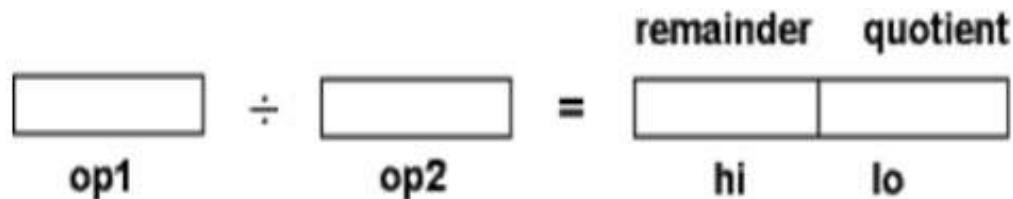
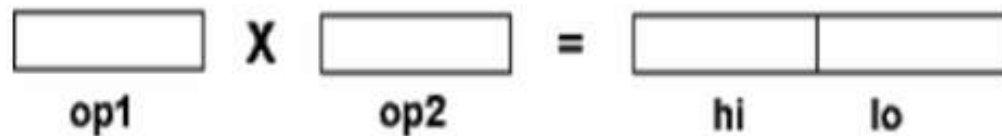
Tương đương: $\$s0 = \$s1 + 0 = \$s1$ (gán)

- Lệnh `add $zero, $zero, $s0` có hợp lệ ?

Phép nhân, chia số nguyên

20

- Thao tác nhân / chia của MIPS có kết quả chứa trong cặp 2 thanh ghi tên là \$hi và \$lo
Bit 0-31 thuộc \$lo và 32-63 thuộc \$hi



Phép nhân

21

- Cú pháp:

mult **\$s0, \$s1**

- Kết quả (64 bit) chứa trong 2 thanh ghi

- ▣ **\$lo (32 bit)** = $((\$s0 * \$s1) \ll 32) \gg 32$

- ▣ **\$hi (32 bit)** = $(\$s0 * \$s1) \gg 32$

- Câu hỏi: Làm sao truy xuất giá trị 2 thanh ghi \$lo và \$hi?

→ Dùng 2 cặp lệnh **mflo** (move from lo), **mfhi** (move from hi) - **mtlo** (move to lo), **mthi** (move to high)

- ▣ **mflo** **\$s0** ($\$s0 = \lo)

- ▣ **mfhi** **\$s0** ($\$s0 = \hi)

Phép chia

22

- Cú pháp:

`div $s0, $s1`

- Kết quả (64 bit) chứa trong 2 thanh ghi
 - ▣ `$lo (32 bit)` = `$s0 / $s1` (thương)
 - ▣ `$hi (32 bit)` = `$s0 % $s1` (số dư)

Thao tác số dấu chấm động

23

- MIPS sử dụng 32 thanh ghi dấu phẩy động để biểu diễn độ chính xác đơn của số thực. Các thanh ghi này có tên là : \$f0 – \$f31.
- Để biểu diễn độ chính xác kép (double precision) thì MIPS sử dụng sự ghép đôi của 2 thanh ghi có độ chính xác đơn.

Vấn đề tràn số

24

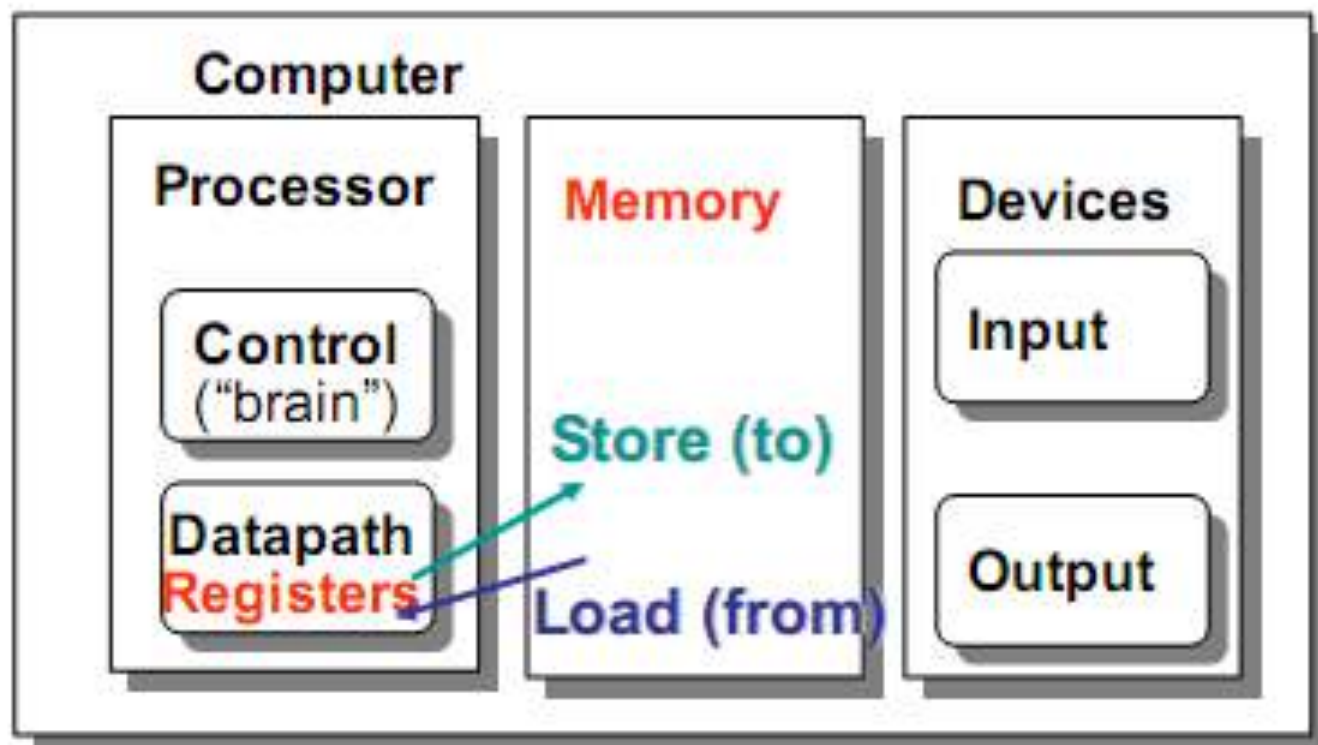
- Kết quả phép tính vượt qua miền giá trị cho phép → Tràn số xảy ra
- Một số ngôn ngữ có khả năng phát hiện tràn số (Ada), một số không (C)
- MIPS cung cấp 2 loại lệnh số học:
 - ▣ **add, addi, sub**: Phát hiện tràn số
 - ▣ **addu, addiu, subu**: Không phát hiện tràn số
- Trình biên dịch sẽ lựa chọn các lệnh số học tương ứng
 - ▣ Trình biên dịch C trên kiến trúc MIPS sử dụng **addu, addiu, subu**

Phần 2: Di chuyển dữ liệu

25

□ Một số nhận xét:

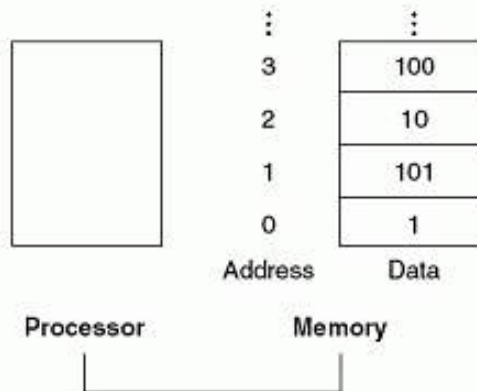
- Ngoài các biến đơn, còn có các **biến phức tạp** thể hiện nhiều kiểu cấu trúc dữ liệu khác nhau, ví dụ như **array**
- Các cấu trúc dữ liệu phức tạp có số phần tử dữ liệu nhiều hơn số thanh ghi của CPU → làm sao lưu ??
 - **Lưu phần nhiều data trong RAM, chỉ load 1 ít vào thanh ghi của CPU khi cần xử lý**
- Vấn đề lưu chuyển dữ liệu giữa thanh ghi và bộ nhớ ?
 - Nhóm lệnh **lưu chuyển dữ liệu** (data transfer)



Bộ nhớ chính

27

- Có thể được xem như là array 1 chiều rất lớn, mỗi phần tử là 1 ô nhớ có kích thước bằng nhau
- Các ô nhớ được đánh số thứ tự từ 0 trở đi
→ Gọi là **địa chỉ (address) ô nhớ**
- Để truy xuất dữ liệu trong ô nhớ cần phải cung cấp địa chỉ ô nhớ đó



Cấu trúc lệnh

28

□ Cú pháp:


opt **opr**, **opr1** (**opr2**)

- **opt** (operator): Tên thao tác (Load / Save)
- **opr** (operand): Thanh ghi lưu từ nhớ (word)
- **opr1** (operand 1): Hằng số nguyên
- **opr2** (operand 2): Thanh ghi chứa địa chỉ vùng nhớ cơ sở (địa chỉ nền)

Hai thao tác chính


29

- **lw**: Nạp 1 từ dữ liệu, từ bộ nhớ, vào 1 thanh ghi trên CPU (Load Word - lw)


lw \$t0, 12 (\$s0)

Nạp từ nhớ có địa chỉ ($\$s0 + 12$) chứa vào thanh ghi \$t0

- **sw**: Lưu 1 từ dữ liệu, từ thanh ghi trên CPU, ra bộ nhớ (Store Word – sw)


sw \$t0, 12 (\$s0)

Lưu giá trị trong thanh ghi \$t0 vào ô nhớ có địa chỉ ($\$s0 + 12$)

Lưu ý 1

30

- **\$s0** được gọi là **thanh ghi cơ sở (base register)** thường dùng để lưu địa chỉ bắt đầu của mảng / cấu trúc
- **12** gọi là **độ dời (offset)** thường dùng để truy cập các phần tử mảng hay cấu trúc

Lưu ý 2

31

- Một thanh ghi có lưu bất kỳ giá trị 32 bit nào, có thể là số nguyên (có dấu / không dấu), có thể là địa chỉ của 1 vùng nhớ trên RAM
- Ví dụ:
 - ▣ `add $t2, $t1, $t0` → \$t0, \$t1 lưu giá trị
 - ▣ `lw $t2, 4 ($t0)` → \$t0 lưu địa chỉ (C: con trỏ)

Lưu ý 3

32

- Số biến cần dùng của chương trình nếu nhiều hơn số thanh ghi của CPU?
- Giải pháp:
 - Thanh ghi chỉ chứa các biến đang xử lý hiện hành và các biến thường sử dụng
 - Kỹ thuật **spilling register**

Ví dụ 1

33

- Giả sử **A** là 1 array gồm 100 từ với **địa chỉ bắt đầu (địa chỉ nền – base address) chứa trong thanh ghi \$s3**. Giá trị các biến **g, h** lần lượt chứa trong các thanh ghi **\$s1** và **\$s2**
- Hãy chuyển thành mã hợp ngữ MIPS:

$$g = h + A[8]$$

- Trả lời:

lw \$t0, 32(\$s3) # Chứa A[8] vào \$t0

add \$s1, \$s2, \$t0

Ví dụ 2

34

- Hãy chuyển thành mã hợp ngữ MIPS:

$$A[12] = h - A[8]$$

- Trả lời:

lw \$t0, 32(\$s3) # Chứa A[8] vào \$t0

sub \$t0, \$s2, \$t0

sw \$t0, 48(\$s3) # Kết quả vào A[12]

Nguyên tắc lưu dữ liệu trong bộ nhớ

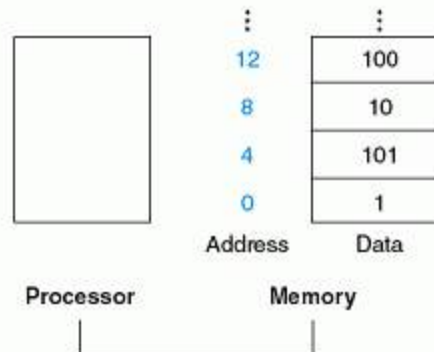
35

- MIPS thao tác và lưu trữ dữ liệu trong bộ nhớ theo 2 nguyên tắc:
 - ▣ Alignment Restriction
 - ▣ Big Endian

Alignment Restriction

36

- MIPS lưu dữ liệu trong bộ nhớ theo nguyên tắc **Alignment Restriction**
 - Các đối tượng lưu trong bộ nhớ (từ nhớ) phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng
 - Mà mỗi từ nhớ có kích thước là 32 bit = 4 byte = kích thước lưu trữ của 1 thanh ghi trong CPU
- Như vậy, từ nhớ phải bắt đầu tại địa chỉ là bội số của 4



Big Endian

37

- MIPS lưu trữ thứ tự các byte trong 1 word trong bộ nhớ theo nguyên tắc **Big Endian** (Kiến trúc x86 sử dụng Little Endian)
- Ví dụ: Lưu trữ giá trị 4 byte: **12345678h** trong bộ nhớ

Địa chỉ byte	Big Endian	Little Endian
0	12	78
1	34	56
2	56	34
3	78	12

Lưu ý

38

- Để truy xuất vào 1 từ nhớ sau 1 từ nhớ thì cần **tăng 1 lượng 4 byte** chứ không phải 1 byte
- Do đó luôn nhớ rằng các lệnh **lw** và **sw** thì **độ dời (offset) phải là bội số của 4**
- Tuy nhiên bộ nhớ các máy tính cá nhân ngày nay lại được đánh địa chỉ theo từng byte (8 bit)

Mở rộng: Load, Save 1 byte

39

- Ngoài việc hỗ trợ load, save 1 từ (lw, sw), MIPS còn hỗ trợ **load, save từng byte** (ASCII)

- ▣ **Load byte: lb**

- ▣ **Save byte: sb**

- ▣ Cú pháp lệnh tương tự lw, sw

- Ví dụ:

lb \$s0, 3 (\$s1)

Lệnh này nạp giá trị byte nhớ có địa chỉ ($\$s1 + 3$) vào byte thấp của thanh ghi \$s0

Nguyên tắc

40

- Giả sử nạp 1 byte có giá trị **xzzz zzzz** vào thanh ghi trên CPU (x: bit dấu của byte đó)
- Giá trị thanh ghi trên CPU (32 bit) sau khi nạp có dạng:
xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz
- Tất cả các bit từ phải sang sẽ có giá trị = bit dấu của giá trị 1 byte vừa nạp (sign-extended)
- Nếu muốn các bit còn lại từ phải sang có giá trị không theo bit dấu (=0) thì dùng lệnh:

lbu (load byte unsigned)

Mở rộng: Load, Save 2 byte (1/2 Word)

41

- MIPS còn hỗ trợ **load, save 1/2 word (2 byte)** (Unicode)
 - ▣ **Load half: lh** (nạp 2 byte nhớ vào 2 byte thấp của thanh ghi \$s0)
 - ▣ **Store half: sh**
 - ▣ Cú pháp lệnh tương tự lw, sw
- Ví dụ:

lh \$s0, 3 (\$s1)

Lệnh này nạp giá trị 2 byte nhớ có địa chỉ ($\$s1 + 3$) vào 2 byte thấp của thanh ghi \$s0

Phần 3: Thao tác luận lý

42

- Chúng ta đã xem xét các thao tác số học (+, -, *, /)
 - ▣ Dữ liệu trên thanh ghi như **1 giá trị đơn** (số nguyên có dấu / không dấu)
- Cần **thao tác trên từng bit** của dữ liệu → **Thao tác luận lý**
 - ▣ Các thao tác luận lý xem dữ liệu trong thanh ghi là dãy 32 bit riêng lẻ thay vì 1 giá trị đơn
- Có 2 loại thao tác luận lý:
 - ▣ **Phép toán luận lý**
 - ▣ **Phép dịch luận lý**

Phép toán luận lý

43

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

▣ **opt** (operator): Tên thao tác

▣ **opr** (operand): Thanh ghi (toán hạng đích)
chứa kết quả

▣ **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)

▣ **opr2** (operand 2): Thanh ghi / hằng số
(toán hạng nguồn 2)

Phép toán luận lý

44

- MIPS hỗ trợ 2 nhóm lệnh cho các phép toán luận lý trên bit:
 - ▣ **and, or, nor:** Toán hạng nguồn thứ 2 (opr2) phải là thanh ghi
 - ▣ **andi, ori:** Toán hạng nguồn thứ 2 (opr2) là hằng số
- Lưu ý: MIPS không hỗ trợ lệnh cho các phép luận lý NOT, XOR, NAND...
- Lý do: Vì với 3 phép toán luận lý and, or, nor ta có thể tạo ra tất cả các phép luận lý khác → Tiết kiệm thiết kế cổng mạch
- Ví dụ:
$$\text{not}(A) = \text{not}(A \text{ or } 0) = A \text{ nor } 0$$

Phép dịch luận lý

45

□ Cú pháp:

opt **opr**, **opr1**, **opr2**

- ▣ **opt** (operator): Tên thao tác
- ▣ **opr** (operand): Thanh ghi (toán hạng đích)
chứa kết quả
- ▣ **opr1** (operand 1): Thanh ghi (toán hạng nguồn 1)
- ▣ **opr2** (operand 2): Hằng số < 32 (Số bit dịch)

Phép dịch luận lý

46

- MIPS hỗ trợ 2 nhóm lệnh cho các phép dịch luận lý trên bit:
 - ▣ Dịch luận lý
 - Dịch trái (**sll** – shift left logical): Thêm vào các bit 0 bên phải
 - Dịch phải (**srl** – shift right logical): Thêm vào các bit 0 bên trái
 - ▣ Dịch số học
 - Không có dịch trái số học
 - Dịch phải (**sra** – shift right arithmetic): Thêm các bit = giá trị bit dấu bên trái

Ví dụ

47

- **sll \$s1, \$s2, 2** # dịch trái luận lý \$s2 2 bit

\$s2 = 0000 0000 0000 0000 0000 0000 0101 0101 = 85

\$s1 = 0000 0000 0000 0000 0000 0001 0101 01**00** = 340 (85 * 2²)

- **srl \$s1, \$s2, 2** # dịch phải luận lý \$s2 2 bit

\$s2 = 0000 0000 0000 0000 0000 0000 0101 0101 = 85

\$s1 = **00**00 0000 0000 0000 0000 0000 0001 0101 = 21 (85 / 2²)

- **sra \$s1, \$s2, 2** # dịch phải số học \$s2 2 bit

\$s2 = 1111 1111 1111 1111 1111 1111 1111 0000 = -16

\$s1 = **11**11 1111 1111 1111 1111 1111 1111 1100 = -4 (-16 / 2²)

Phần 4: Rẽ nhánh

48

- Tương tự lệnh if trong C: Có 2 loại

- if (condition) clause

- if (condition)

clause1

else

clause2

- Lệnh if thứ 2 có thể diễn giải như sau:

if (condition) goto L1 // if → Làm clause1

clause2 // else → Làm clause2

goto L2 // Làm tiếp các lệnh khác

L1: clause1

L2: ...

Rẽ nhánh trong MIPS

49

□ Rẽ nhánh có điều kiện

▣ `beq opr1, opr2, label`

- `beq`: Branch if (register are) equal
- if (`opr1 == opr2`) goto `label`

▣ `bne opr1, opr2, label`

- `bne`: Branch if (register are) not equal
- if (`opr1 != opr2`) goto `label`

□ Rẽ nhánh không điều kiện

▣ `j label`

- Jump to label
- Tương ứng trong C: goto label
- Có thể viết lại thành: `beq $0, $0, label`

Ví dụ

50

- Biên dịch câu lệnh sau trong C thành lệnh hợp ngữ MIPS:

if (i == j) f = g + h;

else f = g - h;

- Ánh xạ biến f, g, h, i, j tương ứng vào các thanh ghi: \$s0, \$s1, \$s2, \$s3, \$s4
- Lệnh hợp ngữ MIPS:

beq \$s3, \$s4, TrueCase # branch (i == j)

sub \$s0, \$s1, \$s2 # f = g - h (false)

j Fin # goto "Fin" label

TrueCase: **add \$s0, \$s1, \$s2** # f = g + h (true)

Fin: ...

Xử lý vòng lặp

51

- Xét mảng `int A[]`. Giả sử ta có vòng lặp trong C:

do {

`g = g + A[i];`

`i = i + j;`

} while (i != h);

- Ta có thể viết lại:

Loop: `g = g + A[i];`

`i = i + j;`

`if (i != h) goto Loop;`

→ Sử dụng lệnh rẽ có điều kiện để biểu diễn vòng lặp!

Xử lý vòng lặp

52

- Ánh xạ biến vào các thanh ghi như sau:

g	h	i	j	base address of A
\$s1	\$s2	\$s3	\$s4	\$s5

- Trong ví dụ trên có thể viết lại thành lệnh MIPS như sau:

```
Loop:      sll    $t1, $s3, 2           # $t1 = i * 22
           add    $t1, $t1, $s5        # $t1 = addr A[i]
           lw     $t1, 0($t1)          # $t1 = A[i]
           add    $s1, $s1, $t1        # g = g + A[i]
           add    $s3, $s3, $s4        # i = i + j
           bne    $s3, $s2, Loop       # if (i != j) goto Label
```


Xử lý vòng lặp

53

- Tương tự cho các vòng lặp phổ biến khác trong C:
 - ▣ while
 - ▣ for
 - ▣ do...while
- Nguyên tắc chung:
 - ▣ Viết lại vòng lặp dưới dạng goto
 - ▣ Sử dụng các lệnh MIPS rẽ nhánh có điều kiện

So sánh không bằng ?

54

- **beq** và **bne** được sử dụng để **so sánh bằng** (== và != trong C)
- Muốn so sánh lớn hơn hay nhỏ hơn?
- MIPS hỗ trợ lệnh **so sánh không bằng**:
 - **slt** **opr1**, **opr2**, **opr3**
 - **slt**: Set on Less Than
 - if (**opr2** < **opr3**)
 opr1 = 1;
else
 opr1 = 0;

So sánh không bằng

55

- Trong C, câu lệnh sau:

if (g < h) goto Less; # g: \$s0, h: \$s1

- Được chuyển thành lệnh MIPS như sau:

slt \$t0, \$s0, \$s1 # if (g < h) then \$t0 = 1

bne \$t0, \$0, Less # if (\$t0 != 0) goto Less

if (g < h) goto Less

- **Nhận xét:** Thanh ghi \$0 luôn chứa giá trị 0, nên lệnh **bne** và **bep** thường dùng để so sánh sau lệnh **slt**

Các lệnh so sánh khác?

56

- Các phép so sánh còn lại như $>$, \geq , \leq thì sao?
- MIPS không trực tiếp hỗ trợ cho các phép so sánh trên, tuy nhiên dựa vào các lệnh `slt`, `bne`, `beq` ta hoàn toàn có thể biểu diễn chúng!

a: \$s0, b: \$s1

57

□ **a < b**

```
slt    $t0, $s0, $s1    # if (a < b) then $t0 = 1
bne    $t0, $0, Label   # if (a < b) then goto Label
<do something>          # else then do something
```

□ **a > b**

```
slt    $t0, $s1, $s0    # if (b < a) then $t0 = 1
bne    $t0, $0, Label   # if (b < a) then goto Label
<do something>          # else then do something
```

□ **a ≥ b**

```
slt    $t0, $s0, $s1    # if (a < b) then $t0 = 1
beq    $t0, $0, Label   # if (a ≥ b) then goto Label
<do something>          # else then do something
```

□ **a ≤ b**

```
slt    $t0, $s1, $s0    # if (b < a) then $t0 = 1
beq    $t0, $0, Label   # if (b ≥ a) then goto Label
<do something>          # else then do something
```

Nhận xét

58

- So sánh $==$ \rightarrow Dùng lệnh **beq**
- So sánh \neq \rightarrow Dùng lệnh **bne**
- So sánh $<$ và $>$ \rightarrow Dùng cặp lệnh (**slt** \rightarrow **bne**)
- So sánh \leq và \geq \rightarrow Dùng cặp lệnh (**slt** \rightarrow **beq**)

So sánh với hằng số

59

- So sánh bằng: beq / bne
- So sánh không bằng: MIPS hỗ trợ sẵn lệnh **slti**
 - ▣ `slti opr, opr1, const`
 - ▣ Thường dùng cho switch...case, vòng lặp for

Ví dụ: switch...case trong C

60

□ **switch (k) {**

case 0: f = i + j; break;

case 1: f = g + h; break;

case 2: f = g - h; break;

}

□ Ta có thể viết lại thành các lệnh if lồng nhau:

if (k == 0) f = i + j;

else if (k == 1) f = g + h;

else if (k == 2) f = g - h;

□ Ánh xạ giá trị biến vào các thanh ghi:

f	g	h	i	j	k
\$s0	\$s1	\$s2	\$s3	\$s4	\$s5

Ví dụ: switch...case trong C

61

- Chuyển thành lệnh hợp ngữ MIPS:

	bne	\$s5, \$0, L1	# if (k != 0) then goto L1
	add	\$s0, \$s3, \$s4	# else (k == 0) then f = i + j
	j	Exit	# end of case → Exit (break)
L1:	addi	\$t0, \$s5, -1	# \$t0 = k - 1
	bne	\$t0, \$0, L2	# if (k != 1) then goto L2
	add	\$s0, \$s1, \$s2	# else (k == 1) then f = g + h
	j	Exit	# end of case → Exit (break)
L2:	addi	\$t0, \$s5, -2	# \$t0 = k - 2
	bne	\$t0, \$0, Exit	# if (k != 2) then goto Exit
	sub	\$s0, \$s1, \$s2	# else (k == 2) then f = g - h
Exit:		

Trình con (Thủ tục)

62

- Hàm (function) trong C → (Biên dịch) → Trình con (Thủ tục) trong hợp ngữ
- Giả sử trong C, ta viết như sau:

```
void main()
```

```
{
```

```
    int a, b;
```

```
    ...
```

```
    sum(a, b);
```

```
    ...
```

```
}
```

- Hàm được chuyển thành lệnh hợp ngữ như thế nào ?
- Dữ liệu được lưu trữ ra sao ?

```
int sum(int x, int y)
```

```
{
```

```
    return (x + y);
```

```
}
```

C

... sum (a, b); ...

/* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {
 return x + y;
}

MIPS	Địa chỉ	Lệnh	
	1000	add	\$a0, \$s0, \$zero # x = a
	1004	add	\$a1, \$s1, \$zero # y = b
	1008	addi	\$ra, \$zero, 1016 # lưu địa chỉ lát sau quay về vào \$ra = 1016
	1012	j	sum # nhảy đến nhãn sum
	1016	[Làm tiếp thao tác khác...]	
		
	2000	sum:	add \$v0, \$a0, \$a1 # thực hiện thủ tục "sum"
	2024	jr	\$ra # nhảy tới địa chỉ trong \$ra

Thanh ghi lưu trữ dữ liệu trong thủ tục

64

- MIPS hỗ trợ 1 số thanh ghi để lưu trữ dữ liệu cho thủ tục:
 - ▣ Đối số input (argument input): \$a0 \$a1 \$a2 \$a3
 - ▣ Kết quả trả về (return ...): \$v0 \$v1
 - ▣ Biến cục bộ trong thủ tục: \$s0 \$s1 ... \$s7
 - ▣ Địa chỉ quay về (return address): \$ra
- Nếu có nhu cầu lưu nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên?
 - Bao nhiêu thanh ghi là đủ ?
 - Sử dụng ngăn xếp (stack)

C

... sum (a, b); ...

/* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {

return x + y;

}

NHẬN XÉT 1

M

I

P

S

Địa chỉ

Lệnh

1000

add

\$a0, \$s0, \$zero

x = a

1004

add

\$a1, \$s1, \$zero

1008

addi

\$ra, \$zero, 1016

1012

j

sum

1016

[Làm tiếp thao tác khác...]

....

2000

sum:

add

\$v0, \$a0, \$a1

thực hiện thủ tục "sum"

2024

jr

\$ra

nhảy tới địa chỉ trong \$ra

• Tại sao không dùng lệnh **j** cho đơn giản, mà lại dùng **jr** ?

→Thủ tục "sum" có thể được gọi ở nhiều chỗ khác nhau, do vậy vị trí quay về mỗi lần gọi sẽ khác nhau

→ Lệnh mới: **jr**

C

... sum (a, b); ...

/* a: \$s0, b: \$s1 */

[Làm tiếp thao tác khác...]

}

int sum (int x, int y) {
 return x + y;
}

NHẬN XÉT 2

MIPS	Địa chỉ	Lệnh
	1000	add \$a0, \$s0, \$zero
	1004	add \$a1, \$s1, \$zero
	1008	addi \$ra, \$zero, 1016
	1012	j sum
	1016	[Làm tiếp thao tác khác...]
	
	2000	sum: add \$v0, \$a0,
	2024	jr \$ra

• Thay vì dùng 2 lệnh để lưu địa chỉ quay về vào thanh ghi \$ra và nhảy đến thủ tục "sum":

1008 addi \$ra, \$zero, 1016 # \$ra = 1016

1012 j sum # goto sum

→MIPS hỗ trợ lệnh mới: jal (jump and link) để thực hiện 2 công việc trên:

1008 jal sum # \$ra = 1012, goto sum

→ Tại sao không cần xác định tường minh địa chỉ quay về trong \$ra ?

Các lệnh nhảy mới

67

- **jr (jump register)**
 - Cú pháp: **jr register**
 - Diễn giải: Nhảy đến địa chỉ nằm trong thanh ghi register thay vì nhảy đến 1 nhãn như lệnh j (jump)
- **jal (jump and link)**
 - Cú pháp: **jal label**
 - Diễn giải: Thực hiện 2 bước:
 - **Bước 1 (link):** Lưu địa chỉ của lệnh kế tiếp vào thanh ghi \$ra (Tại sao không phải là địa chỉ của lệnh hiện tại ?)
 - **Bước 2 (jump):** Nhảy đến nhãn label
- Hai lệnh này được sử dụng hiệu quả trong thủ tục
 - **jal:** tự động lưu địa chỉ quay về chương trình chính vào thanh ghi \$ra và nhảy đến thủ tục con
 - **jr \$ra:** Quay lại thân chương trình chính bằng cách nhảy đến địa chỉ đã được lưu trước đó trong \$ra

Bài tập

68

- Chuyển đoạn chương trình sau thành mã hợp ngữ MIPS:

```
void main()  
{  
    int i, j, k, m;  
    ...  
    i = mult (j, k); ...  
    m = mult (i, i); ...  
}
```

```
int mult (int mcand, int mlier)  
{  
    int product = 0;  
    while (mlier > 0)  
    {  
        product = product + mcand;  
        mlier = mlier - 1;  
    }  
    return product;  
}
```


Thủ tục lồng nhau

69

- Vấn đề đặt ra khi chuyển thành mã hợp ngữ của đoạn lệnh sau:

```
int sumSquare (int x, int y)
{
    return mult (x, x) + y;
}
```
 - Thủ tục `sumSquare` sẽ gọi thủ tục `mult` trong thân hàm của nó
 - Vấn đề:
 - Địa chỉ quay về của thủ tục `sumSquare` lưu trong thanh ghi `$ra` sẽ bị ghi đè bởi địa chỉ quay về của thủ tục `mult` khi thủ tục này được gọi!
 - Như vậy cần phải lưu lại (`backup`) trong bộ nhớ chính địa chỉ quay về của thủ tục `sumSquare` (trong thanh ghi `$ra`) **trước khi gọi thủ tục `mult`**
- Sử dụng ngăn xếp (Stack)

Ngăn xếp (Stack)

70

- Là ngăn xếp gồm nhiều ô nhớ kết hợp (vùng nhớ) nằm trong bộ nhớ chính
- Cấu trúc dữ liệu lý tưởng để chứa tạm các giá trị trong thanh ghi
 - Thường chứa **địa chỉ trả về**, các **biến cục bộ của trình con**, nhất là các biến có cấu trúc (array, list...) không chứa vừa trong các thanh ghi trong CPU
- Được định vị và quản lý bởi **stack pointer**
- Có 2 tác vụ hoạt động cơ bản:
 - **push**: Đưa dữ liệu từ thanh ghi vào stack
 - **pop**: Lấy dữ liệu từ stack chép vào thanh ghi
- Trong MIPS dành sẵn 1 thanh ghi **\$sp** để lưu trữ stack pointer
- Để sử dụng Stack, cần khai báo kích vùng Stack bằng cách tăng (push) giá trị con trỏ ngăn xếp stack pointer (lưu trữ trong thanh ghi **\$sp**)
 - Lưu ý: Stack pointer tăng theo chiều **giảm địa chỉ**

C

```
int sumSquare (int x, int y) { return mult (x, x) + y; }  
/* x: $a0, y: $a1 */
```

M

I

P

S

sumSquare:
init
push
push
pop
free

addi \$sp, \$sp, -8
sw \$ra, 4 (\$sp)
sw \$a1, 0 (\$sp)
add \$a1, \$a0, \$zero
jal mult
lw \$a1, 0 (\$sp)
add \$v0, \$v0, \$a1
lw \$ra, 4 (\$sp)
addi \$sp, \$sp, 8
jr \$ra

mult:
...
jr \$ra

khai báo kích thước stack cần dùng = 8 byte
cất địa chỉ quay về của thủ tục sumSquare đưa vào stack
cất giá trị y vào stack
gán tham số thứ 2 là x (ban đầu là y) để phục vụ cho thủ tục mult sắp gọi
nhảy đến thủ tục mult
sau khi thực thi xong thủ tục mult , khôi phục lại tham số thứ 2 = y
dựa trên giá trị đã lưu trước đó trong stack
mult() + y
khôi phục địa chỉ quay về của thủ tục sumSquare từ stack, đưa lại vào \$ra
khôi phục 8 byte giá trị \$sp ban đầu đã "mượn", kết thúc stack
nhảy đến đoạn lệnh ngay sau khi gọi thủ tục sumSquare trong chương trình chính, để thao tác tiếp các lệnh khác.

Tổng quát: Thao tác với stack

72

- Khởi tạo stack (init)
- Lưu trữ tạm các dữ liệu cần thiết vào stack (push)
- Gán các đối số (nếu có)
- Gọi lệnh jal để nhảy đến các thủ tục con
- Khôi phục các dữ liệu đã lưu tạm từ stack (pop)
- Khôi phục bộ nhớ, kết thúc stack (free)

Cụ thể hóa

73

- **Đầu thủ tục:**

Procedure_Label:

```
addi $sp, $sp, -framesize    # khởi tạo stack, dịch chuyển stack pointer $sp lùi
```

sw \$ra, framesize - 4 (\$sp) # cất \$ra (kích thước 4 byte) vào stack (push)

Lưu tạm các thanh ghi khác (nếu cần)

- **Thân thủ tục:**

```
jal other_procedure # Gọi các thủ tục khác (nếu cần)
```

- **Cuối thủ tục:**

```
lw $ra, frame_size - 4($sp)      # khôi phục $ra từ stack (pop)
```

```
lw ... # khôi phục các thanh ghi khác (nếu cần)
```

```
addi    $sp, $sp, framesize    # khôi phục $sp, giải phóng stack
```

```
jr $ra # nhảy đến lệnh tiếp theo "Procedure Label"
```

trong chương trình chính

Một số nguyên tắc khi thực thi thủ tục

74

- Nhảy đến thủ tục bằng lệnh **jal** và quay về nơi trước đó đã gọi nó bằng lệnh **jr \$ra**
- 4 thanh ghi chứa đối số của thủ tục: **\$a0, \$a1, \$a2, \$a3**
- Kết quả trả về của thủ tục chứa trong thanh ghi **\$v0** (và **\$v1** nếu cần)
- Phải tuân theo **nguyên tắc sử dụng các thanh ghi** (register conventions)

Nguyên tắc sử dụng thanh ghi

75

- **\$0**: (Không thay đổi) Luôn bằng 0
- **\$s0 - \$s7**: (Khôi phục lại nếu thay đổi) Rất quan trọng, nếu thủ tục được gọi (callee) thay đổi các thanh ghi này thì nó phải khôi phục lại giá trị các thanh ghi này trước khi kết thúc
- **\$sp**: (Khôi phục lại nếu thay đổi) Thanh ghi con trỏ stack phải có giá trị không đổi trước và sau khi gọi lệnh "jal", nếu không thủ tục gọi (caller) sẽ không quay về được.
- **Tip**: Tất cả các thanh ghi này đều bắt đầu bằng ký tự s !

Nguyên tắc sử dụng thanh ghi

76

- **\$ra**: (Có thể thay đổi) Khi gọi lệnh “jal” sẽ làm thay đổi giá trị thanh ghi này. Thủ tục gọi (caller) lưu lại (backup) giá trị của thanh ghi \$ra vào stack nếu cần
- **\$v0 - \$v1**: (Có thể thay đổi) Chứa kết quả trả về của thủ tục
- **\$a0 - \$a1**: (Có thể thay đổi) Chứa đối số của thủ tục
- **\$t0 - \$t9**: (Có thể thay đổi) Đây là các thanh ghi tạm nên có thể bị thay đổi bất cứ lúc nào

Tóm tắt

77

- Nếu thủ tục R gọi thủ tục E:
 - R phải lưu vào stack các thanh ghi tạm có thể bị sử dụng trong E trước khi gọi lệnh **jal E** (goto E)
 - E phải lưu lại giá trị các thanh ghi lưu trữ (**\$s0 - \$s7**) nếu nó muốn sử dụng các thanh ghi này → trước khi kết thúc E sẽ khôi phục lại giá trị của chúng
 - **Nhớ:** Thủ tục gọi **R (caller)** và Thủ tục được gọi **E (callee)** chỉ cần lưu các thanh ghi tạm / thanh ghi lưu trữ mà nó muốn dùng, không phải tất cả các thanh ghi!

Bảng tóm tắt

78

Name	Register number	Usage	Preserved on call?
\$zero	0	the constant value 0	n.a.
\$v0-\$v1	2-3	values for results and expression evaluation	no
\$a0-\$a3	4-7	arguments	no
\$t0-\$t7	8-15	temporaries	no
\$s0-\$s7	16-23	saved	yes
\$t8-\$t9	24-25	more temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

System call

79

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_character	11	\$a0 = integer	
read_character	12		char (in \$v0)

Hello.asm

80

`.data` # data segment

`str: .asciiz "Hello asm !"`

`.text` # text segment

`.globl main`

main: # starting point of program

`addi $v0, $0, 4` # $\$v0 = 0 + 4 = 4 \rightarrow$ print str syscall

`la $a0, str` # $\$a0 = \text{address(str)}$

`syscall` # excute the system call

PHỤ LỤC

\$0	0	\$zero		
\$1		\$at	Reserved for assembler use	
\$2		\$v0	Procedure results	
\$3		\$v1		
\$4		\$a0	Procedure arguments	Saved
\$5		\$a1		
\$6		\$a2		
\$7		\$a3		
\$8		\$t0	Temporary values	
\$9		\$t1		
\$10		\$t2		
\$11		\$t3		
\$12		\$t4		
\$13		\$t5		
\$14		\$t6		
\$15		\$t7		
\$16		\$s0	Operands	Saved across procedure calls
\$17		\$s1		
\$18		\$s2		
\$19		\$s3		
\$20		\$s4		
\$21		\$s5		
\$22		\$s6		
\$23		\$s7		
\$24		\$t8	More temporaries	
\$25		\$t9		
\$26		\$k0	Reserved for OS (kernel)	
\$27		\$k1		
\$28		\$gp	Global pointer	Saved
\$29		\$sp	Stack pointer	
\$30		\$fp	Frame pointer	
\$31		\$ra	Return address	

Phụ lục 1: 40 lệnh cơ bản MIPS

Instruction	Usage
Load upper immediate	lui rt,imm
Add	add rd,rs,rt
Subtract	sub rd,rs,rt
Set less than	slt rd,rs,rt
Add immediate	addi rt,rs,imm
Set less than immediate	slti rd,rs,imm
AND	and rd,rs,rt
OR	or rd,rs,rt
XOR	xor rd,rs,rt
NOR	nor rd,rs,rt
AND immediate	andi rt,rs,imm
OR immediate	ori rt,rs,imm
XOR immediate	xori rt,rs,imm
Load word	lw rt,imm(rs)
Store word	sw rt,imm(rs)
Jump	j L
Jump register	jr rs
Branch less than 0	bltz rs,L
Branch equal	beq rs,rt,L
Branch not equal	bne rs,rt,L

Instruction	Usage
Move from Hi	mfhi rd
Move from Lo	mflo rd
Add unsigned	addu rd,rs,rt
Subtract unsigned	subu rd,rs,rt
Multiply	mult rs,rt
Multiply unsigned	multu rs,rt
Divide	div rs,rt
Divide unsigned	divu rs,rt
Add immediate unsigned	addiu rs,rt,imm
Shift left logical	sll rd,rt,sh
Shift right logical	srl rd,rt,sh
Shift right arithmetic	sra rd,rt,sh
Shift left logical variable	sllv rd,rt,rs
Shift right logical variable	srlv rd,rt,rs
Shift right arith variable	srav rd,rt,rs
Load byte	lb rt,imm(rs)
Load byte unsigned	lbu rt,imm(rs)
Store byte	sb rt,imm(rs)
Jump and link	jal L
System call	syscall

Phụ lục 2: Pseudo Instructions

84

- “Lệnh giả”: Mặc định không được hỗ trợ bởi MIPS
- Là những lệnh cần phải biên dịch thành rất nhiều câu lệnh thật trước khi được thực hiện bởi phần cứng
→ Lệnh giả = Thủ tục
- Dùng để hỗ trợ lập trình viên thao tác nhanh chóng với những thao tác phức tạp gồm nhiều bước

Ví dụ: Tính $\$s1 = |\$s0|$

85

- Để tính được trị tuyệt đối của $\$s0 \rightarrow \$s1$, ta có lệnh giả là: **abs \$s1, \$s0**
- Thực sự MIPS không có lệnh này, khi chạy sẽ biên dịch lệnh này thành các lệnh thật sau:

Trị tuyệt đối của X là $-X$ nếu $X < 0$, là X nếu $X \geq 0$

abs:

sub \$s1, \$zero, \$s0

slt \$t0, \$s0, \$zero

bne \$t0, \$zero, done

add \$s1, \$s0, \$zero

done:

jr \$ra

Một số lệnh giả phổ biến của MIPS

86

Name	instruction syntax	meaning
Move	move rd, rs	rd = rs
Load Address	la rd, rs	rd = address (rs)
Load Immediate	li rd, imm	rd = 32 bit Immediate value
Branch greater than	bgt rs, rt, Label	if(R[rs]>R[rt]) PC=Label
Branch less than	blt rs, rt, Label	if(R[rs]<R[rt]) PC=Label
Branch greater than or equal	bge rs, rt, Label	if(R[rs]>=R[rt]) PC=Label
branch less than or equal	ble rs, rt, Label	if(R[rs]<=R[rt]) PC=Label
branch greater than unsigned	bgtu rs, rt, Label	if(R[rs]<=R[rt]) PC=Label
branch greater than zero	bgtz rs, Label	if(R[rs] >=0) PC=Label

Phụ lục 3: Biểu diễn lệnh trong ngôn ngữ máy

87

- Chúng ta đã học 1 số nhóm lệnh hợp ngữ thao tác trên CPU tuy nhiên...
 - CPU có hiểu các lệnh hợp ngữ đã học này không?
- Tất nhiên là không vì nó chỉ hiểu được ngôn ngữ máy gồm toàn bit 0 và 1
- Dãy bit đó gọi là **lệnh máy (machine language instruction)**
 - Mỗi lệnh máy có kích thước **32 bit**, được chia thành các **nhóm bit, gọi là trường (field)**, mỗi nhóm có 1 vai trò trong lệnh máy
 - Lệnh máy có 1 cấu trúc xác định gọi là **cấu trúc lệnh (Instruction Format)**

MIPS Instruction Format

88

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, <i>imm.</i> format
J-format	op	target address					Jump instruction format

- Có 3 format lệnh trong MIPS:
 - **R-format:** Dùng trong các lệnh tính toán số học (add, sub, and, or, nor, sll, srl, sra...)
 - **I-format:** Dùng trong các lệnh thao tác với hằng số, chuyển dữ liệu với bộ nhớ, rẽ nhánh
 - **J-format:** Dùng trong các lệnh nhảy (jump – C: goto)

R-format

89

6 bits	5	5	5	5	6
opcode	rs	rt	rd	shmat	funct

- ▣ **opcode** (operation code): mã thao tác, cho biết lệnh làm gì
- ▣ **funct** (function code): kết hợp với opcode để xác định lệnh làm gì (trường hợp các lệnh có cùng mã thao tác với opcode)
- ▣ **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- ▣ **rt** (target register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 2
- ▣ **rd** (destination register): thanh ghi đích, thường chứa kết quả lệnh
- ▣ **shamt**: chứa số bit cần dịch trong các lệnh dịch, nếu không phải lệnh dịch thì trường này có giá trị 0

Nhận xét

90

- Các trường lưu địa chỉ thanh ghi **rs, rt, rd** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để biểu diễn 32 thanh ghi của MIPS

- Trường lưu số bit cần dịch **shamt** có kích thước **5 bit**
 - Có khả năng biểu diễn các số từ 0 đến 31
 - Đủ để dịch hết 32 bit lưu trữ của 1 thanh ghi

Ví dụ R-format (1)

91

- Biểu diễn machine code của lệnh: **add \$t0, \$t1, \$t2**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	9	10	8	0	32
000000	01001	01010	01000	00000	100000

- opcode = 0
 - funct = 32
- } Xác định thao tác cộng
(tất cả các lệnh theo cấu trúc R-format đều có opcode = 0)
- rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)
 - rt = 10 (toán hạng nguồn thứ 2 là \$t2 ~ \$10)
 - rd = 8 (toán hạng đích là \$t0 ~ \$8)
 - shmat = 0 (không phải lệnh dịch)

Ví dụ R-format (2)

92

- Biểu diễn machine code của lệnh: **sll \$t2, \$s0, 4**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	rd	shmat	funct
0	0	16	10	4	0
000000	00000	10000	01010	00100	000000

- opcode = 0
 - funct = 0
- } Xác định thao tác dịch trái luận lý
(tất cả các lệnh theo cấu trúc R-format đều có opcode = 0)
- rs = 0 (không dùng trong phép dịch)
 - rt = 16 (toán hạng nguồn là \$s0 ~ \$16)
 - rd = 10 (toán hạng đích là \$t2 ~ \$10)
 - shmat = 4 (số bit dịch = 4)

Vấn đề của R-format

93

- Làm sao giải quyết trường hợp nếu câu lệnh đòi hỏi trường dành cho toán hạng phải lớn hơn 5 bit?
- Ví dụ:
 - Lệnh **addi** cộng giá trị thanh ghi với 1 hằng số, nếu giới hạn trường hằng số ở 5 bit → hằng số không thể lớn hơn $2^5 = 32$
→ Giới hạn khả năng tính toán số học!
 - Lệnh **lw, sw** cần biểu diễn 2 thanh ghi và 1 hằng số offset, nếu giới hạn ở 5 bit
→ Giới hạn khả năng truy xuất dữ liệu trong bộ nhớ
 - Lệnh **beq, bne** cần biểu diễn 2 thanh ghi và 1 hằng số chứa địa chỉ (nhãn) cần nhảy, nếu giới hạn ở 5 bit
→ Giới hạn lưu trữ chương trình trong bộ nhớ
- Giải pháp: Dùng **I-format** cho các lệnh thao tác hằng số, truy xuất dữ liệu bộ nhớ và rẽ nhánh

I-format

94

6 bits	5	5	16
opcode	rs	rt	immediate

- **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format, chỉ khác không cần thêm trường funct)
 - Đây cũng là lý do tại sao R-format có 2 trường 6 bit để xác định lệnh làm gì thay vì 1 trường 12 bit → Để nhất quán với các cấu trúc lệnh khác (I-format) trong khi kích thước mỗi trường vẫn hợp lý
- **rs** (source register): thanh ghi nguồn, thường chứa toán hạng nguồn thứ 1
- **rt** (target register): thanh ghi đích, thường chứa kết quả lệnh
- **immediate**: 16 bit, có thể biểu diễn số nguyên từ -2^{15} đến $(2^{15} - 1)$
 - I-format đã có thể lưu hằng số 16 bit (thay vì 5 bit như R-format)

Ví dụ I-format

95

- Biểu diễn machine code của lệnh: **addi \$s0, \$s1, 10**
- Biểu diễn lệnh với R-format theo từng trường:

opcode	rs	rt	immediate
8	17	16	10
001000	10001	10000	0000 0000 0000 0000 1010

- opcode = 8: Xác định thao tác cộng hằng số
- rs = 17 (toán hạng nguồn thứ 1 là \$s1 ~ \$17)
- rt = 16 (toán hạng đích là \$s0 ~ \$16)
- immediate = 10 (toán hạng nguồn thứ 2 = hằng số = 10)

Vấn đề I-format

96

- Trường hằng số (immediate) có kích thước 16 bit
- Nếu muốn thao tác với các hằng số 32 bit?
 - Tăng kích thước trường immediate thành 32 bit?
 - Tăng kích thước các lệnh thao tác với hằng số có cấu trúc I-format
 - Phá vỡ cấu trúc lệnh 32 bit của MIPS

Vấn đề I-format (tt)

97

- Giải pháp: MIPS cung cấp lệnh mới “lui”
 - ▣ lui register, immediate
 - ▣ Load Upper Immediate
 - ▣ Đưa hằng số 16 bit vào 2 byte cao của 1 thanh ghi
 - ▣ Giá trị 2 byte thấp của thanh ghi đó gán = 0
 - ▣ Lệnh này có cấu trúc I-format

Ví dụ

98

- Muốn cộng giá trị 32 bit **0xABABCD** vào thanh ghi **\$t0** ?

- ▣ Không thể dùng:

```
addi $t0, $t0, 0xABABCD
```

- ▣ Giải pháp dùng lệnh **lui**:

```
lui   $at, 0xABAB
```

```
ori   $at, $at, 0xCDCD
```

```
add   $t0, $0, $at
```

Vấn đề rẽ nhánh có điều kiện trong I-format

99

- Các lệnh rẽ nhánh có điều kiện có cấu trúc I-format

6 bits	5	5	16
opcode	rs	rt	immediate

- **opcode**: xác định lệnh beq hay bne
 - **rs, rt**: chứa các giá trị của thanh ghi cần so sánh
 - **immediate** chứa địa chỉ (nhãn) cần nhảy tới?
 - immediate chỉ có 16 bit → chỉ có thể nhảy tới địa chỉ từ 0 – 2^{16} (65535)
?
- Chương trình bị giới hạn không gian rất nhiều
- **Câu trả lời: immediate KHÔNG phải chứa địa chỉ cần nhảy tới**

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

100

- Trong MIPS, thanh ghi **PC** (Program Counter) sẽ chứa địa chỉ của lệnh **đang được thực hiện**
- **immediate**: số có dấu, chứa khoảng cách so với địa chỉ lệnh đang thực hiện nằm trong thanh ghi PC
 - ▣ $\text{immediate} + \text{PC} \rightarrow$ địa chỉ cần nhảy tới
- Cách xác định địa chỉ này gọi là **PC-Relative Addressing** (định vị theo thanh ghi PC)
 - ▣ Xem slide "Addressing Mode" (phần sau) để biết thêm về các Addressing mode trong MIPS

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

101

- Mỗi lệnh trong MIPS có kích thước 32 bit (1 word – 1 từ nhớ)
- MIPS truy xuất bộ nhớ theo nguyên tắc Alignment Restriction
- Đơn vị của **immediate**, khoảng cách so với PC, là từ nhớ (**word = 4 byte**) chứ không phải là byte
- Các lệnh rẽ nhánh có thể nhảy tới các địa chỉ có khoảng cách $\pm 2^{15}$ word tính từ địa chỉ lưu trong PC ($\pm 2^{17}$ byte)

Vấn đề rẽ nhánh có điều kiện trong I-format (tt)

102

- Cách tính địa chỉ rẽ nhánh:

- Nếu không rẽ nhánh:

$PC = PC + 4 = \text{địa chỉ lệnh kế tiếp trong bộ nhớ}$

- Nếu thực hiện rẽ nhánh:

$PC = (PC + 4) + (\text{immediate} * 4)$

- Vì sao cộng immediate với $(PC + 4)$ thay vì PC ? → Khi rẽ nhánh bị delayed 1 lệnh kể với lệnh rẽ nhánh
- Nhận xét: immediate cho biết số lệnh cần nhảy qua để đến được nhãn

Ví dụ I-format

103

□ **Loop:** **beq** **\$t1, \$0, End**

add **\$t0, \$t0, \$t2**

addi **\$t1, \$t1, -1**

j **Loop**

End: ...

opcode	rs	rt	immediate
4	9	0	3
000100	01001	00000	0000 0000 0000 0000 0011

- opcode = 4: Xác định thao tác của lệnh beq
- rs = 9 (toán hạng nguồn thứ 1 là \$t1 ~ \$9)
- rt = 0 (toán hạng nguồn thứ 2 là \$0 ~ \$0)
- immediate = 3 (nhảy qua 3 lệnh kể từ lệnh rẽ nhánh có điều kiện)

Vấn đề I-format

104

- Mỗi lệnh trong MIPS có kích thước 32 bit
- Mong muốn: Có thể nhảy đến bất kỳ lệnh nào (MIPS hỗ trợ các hàm nhảy không điều kiện như **j**)
 - Nhảy trong khoảng 2^{32} (4 GB) bộ nhớ
 - I-format bị hạn chế giới hạn vùng nhảy
 - Dùng **J-format**
- Tuy nhiên, dù format nào cũng phải **cần tối thiểu 6 bit cho opcode** để nhất quán lệnh với các format khác
 - **J-format** chỉ có thể dùng $32 - 6 = 26$ bit để biểu diễn khoảng cách nhảy

J-format

105

6 bits	26
opcode	target address

- ▣ **opcode** (operation code): mã thao tác, cho biết lệnh làm gì (tương tự opcode của R-format và I-format)
 - ▣ Để nhất quán với các cấu trúc lệnh khác (R-format và I-format)
- ▣ **target address**: Lưu địa chỉ đích của lệnh nhảy
 - ▣ Tương tự lệnh rẽ nhánh, địa chỉ đích của lệnh nhảy tính theo đơn vị word

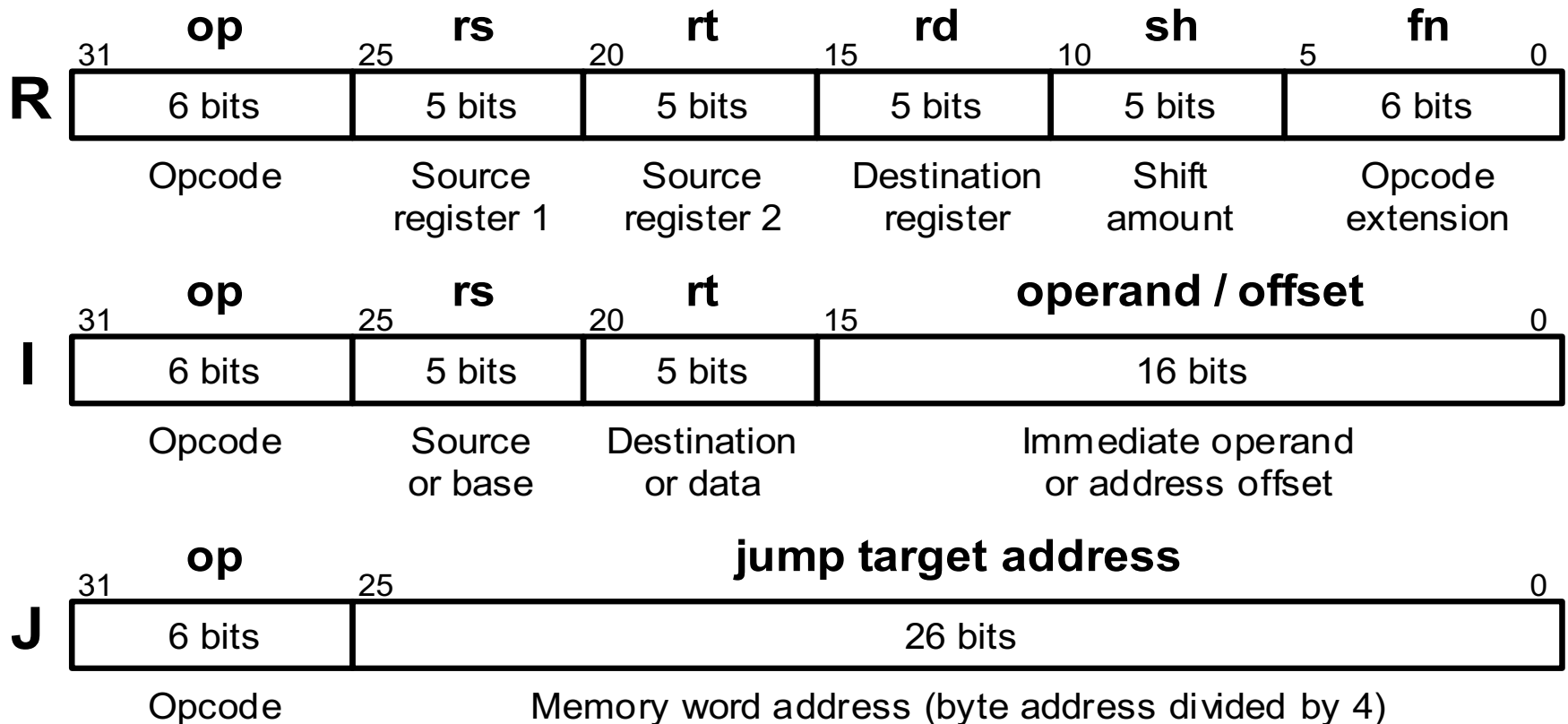
Nhận xét

106

- Trong **J-format**, các lệnh nhảy có thể nhảy tới các lệnh có địa chỉ trong khoảng 2^{26}
- Muốn nhảy tới các lệnh có địa chỉ lớn hơn từ 2^{27} đến 2^{32} ?
 - MIPS hỗ trợ lệnh **jr** (đọc trong phần thủ tục)
 - Tuy nhiên nhu cầu này không cần thiết lắm vì chương trình thường không quá lớn như vậy

Bảng tóm tắt Format

107



Phụ lục 4: Addressing mode

108

- Là phương thức định vị trí (địa chỉ hóa) các toán hạng trong kiến trúc MIPS
- Có 5 phương pháp chính:
 - **Immediate addressing** (Vd: `addi $t0, $t0, 5`)
Toán hạng = hằng số 16 bit trong câu lệnh
 - **Register addressing** (Vd: `add $t0, $t0, $t1`)
Toán hạng = nội dung thanh ghi
 - **Base addressing** (Vd: `lw $t1, 8($t0)`)
Toán hạng = nội dung ô nhớ (địa chỉ ô nhớ = nội dung thanh ghi + hằng số 16 bit trong câu lệnh)
 - **PC-relative addressing** (Vd: `beq $t0, $t1, Label`)
Toán hạng = địa chỉ đích lệnh nhảy = nội dung thanh ghi PC + hằng số 16 bit trong câu lệnh
 - **Pseudodirect addressing** (Vd: `j 2500`)
Toán hạng = địa chỉ đích lệnh nhảy = các bit cao thanh ghi PC + hằng số 26 bit trong câu lệnh

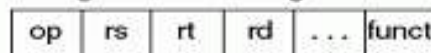
Addressing mode

109

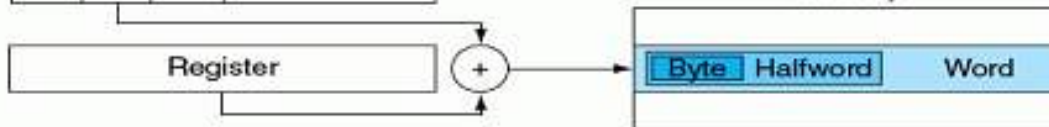
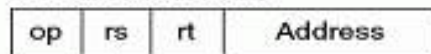
1. Immediate addressing



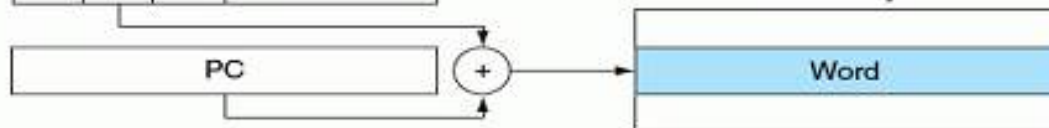
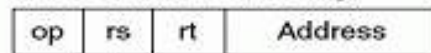
2. Register addressing



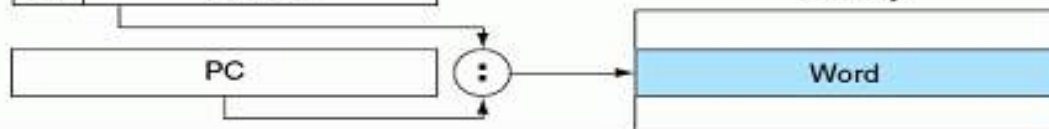
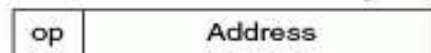
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Homework

110

- Sách Petterson & Hennessy: Đọc hết chương 2
- Tài liệu tham khảo: Đọc "08_HP_AppA.pdf"