ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

KIẾN TRÚC MÁY TÍNH

Computer Architecture

Course ID: IT3030, IT3034, IT3283

Version: CA.RISCV.2024.1

Nội dung học phần

Chương 1. Giới thiệu chung

Chương 2. Hệ thống máy tính

Chương 3. Kiến trúc tập lệnh

Chương 4. Số học máy tính

Chương 5. Bộ xử lý

Chương 6. Bộ nhớ máy tính

Chương 7. Hệ thống vào-ra

Chương 8. Các kiến trúc song song

Chương 3
KIẾN TRÚC TẬP LỆNH

Nội dung

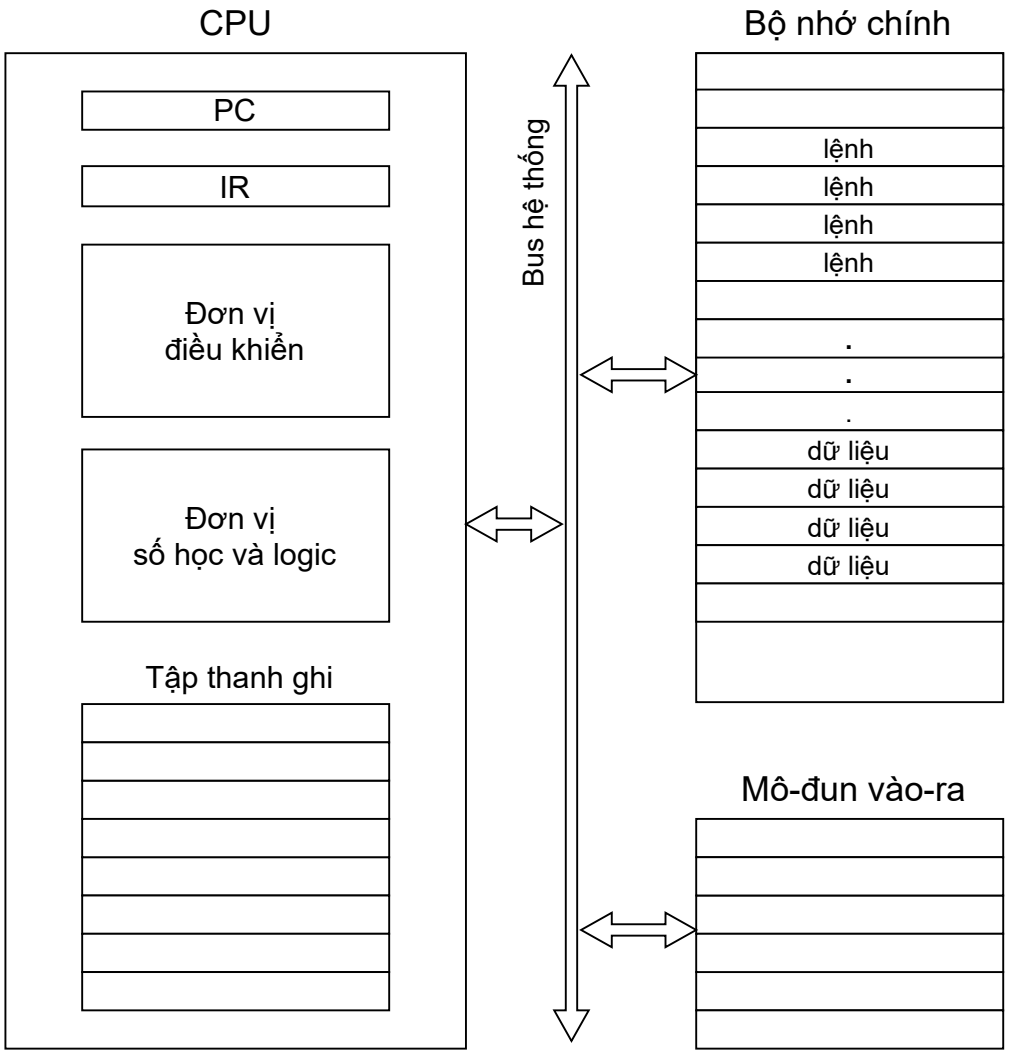
3.1. Giới thiệu chung

- Kiến trúc tập lệnh (Instruction Set Architecture): cách nhìn máy tính bởi người lập trình
  - Giao diện giữa phần mềm và phần cứng
  - Được xác định bởi tập lệnh và vị trí toán hạng
  - Các kiến trúc thông dụng: Intel x86, ARM, **RISC-V**, ...
- Vi kiến trúc (Microarchitecture): cách thực hiện kiến trúc tập lệnh bằng phần cứng
  - Được trình bày trong Chương 5

Ngôn ngữ trong máy tính

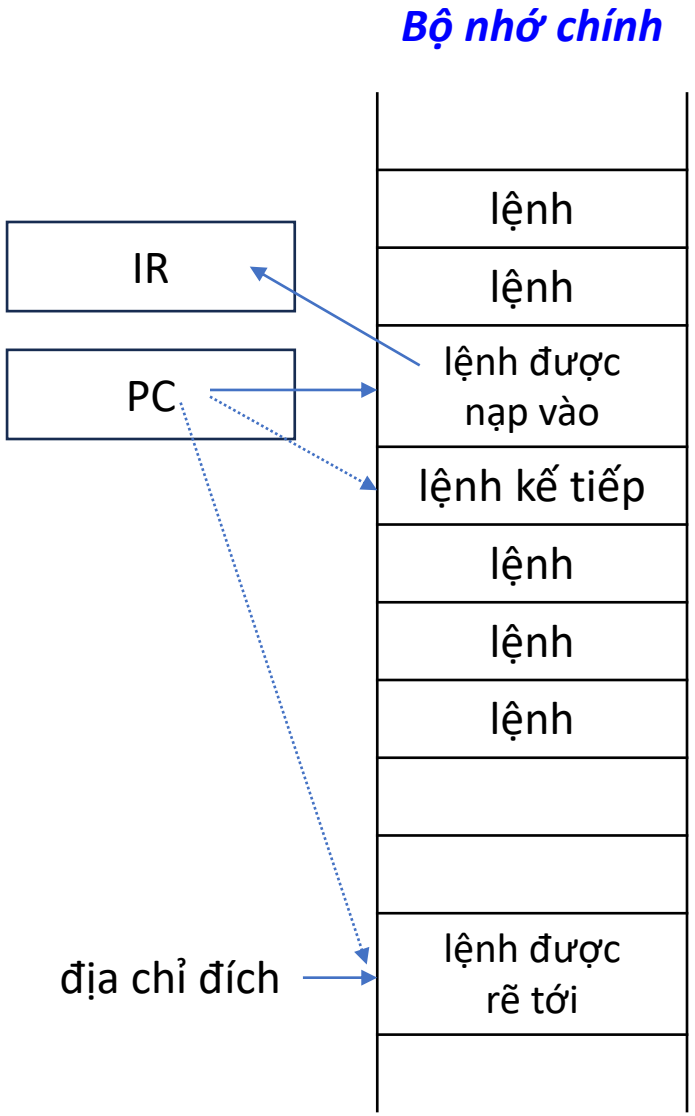
- Ngôn ngữ máy (machine language):
  - Ngôn ngữ để phần cứng máy tính hiểu được
  - Còn gọi là mã máy (machine code)
  - Các lệnh được mã hoá bằng các bit 0 và 1
  - Bộ xử lý đọc và thực hiện các lệnh mã máy
- Hợp ngữ (assembly language):
  - Mô tả các lệnh máy về dạng các ký hiệu để con người dễ dàng đọc được

Mô hình lập trình của máy tính



CPU tìm nạp lệnh từ bộ nhớ

- Bộ đếm chương trình PC (Program Counter) là thanh ghi của CPU giữ địa chỉ của lệnh cần nạp vào để thực hiện
- Bộ xử lý phát địa chỉ từ PC đến bộ nhớ, lệnh được nạp vào thanh ghi lệnh IR (Instruction Register)
- Bộ xử lý thay đổi nội dung PC để trỏ đến lệnh tiếp theo:
  - Tuần tự: PC tăng lên để trỏ đến lệnh kế tiếp
  - Rẽ nhánh hoặc nhảy: PC nhận địa chỉ đích, trỏ đến lệnh được rẽ tới



Giải mã và thực hiện lệnh

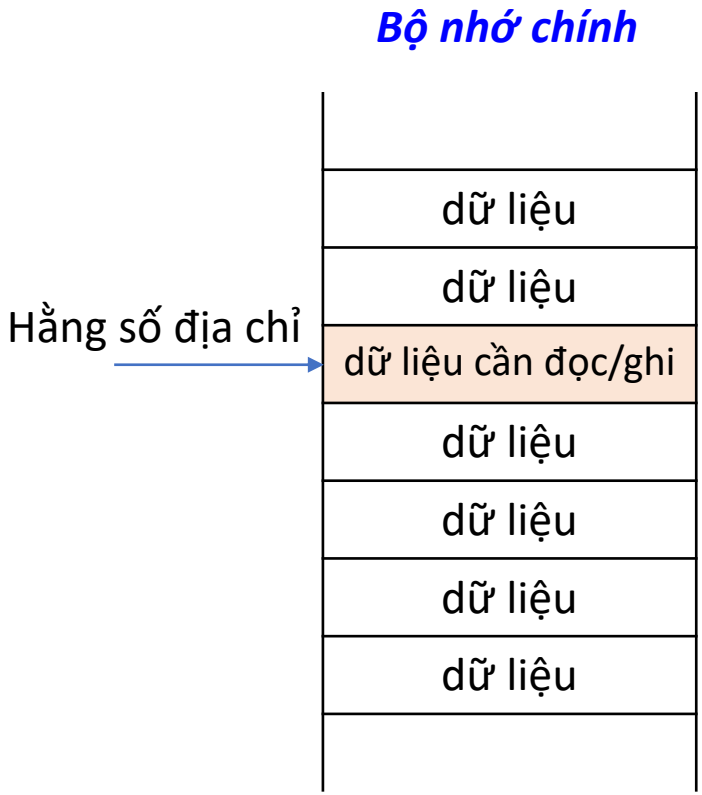
- Bộ xử lý giải mã lệnh đã được nạp vào và phát các tín hiệu điều khiển thực hiện thao tác mà lệnh yêu cầu
- Các kiểu thao tác chính của lệnh:
  - Trao đổi dữ liệu giữa CPU với bộ nhớ chính hoặc với cổng vào-ra
  - Thực hiện các phép toán số học hoặc phép toán logic với các dữ liệu (được thực hiện bởi ALU)
  - Chuyển điều khiển trong chương trình (rẽ nhánh, nhảy)

CPU đọc/ghi dữ liệu bộ nhớ

- Với các lệnh trao đổi dữ liệu với bộ nhớ, CPU cần biết và phát ra địa chỉ của ngăn nhớ cần đọc/ghi
- Địa chỉ đó có thể là:
  - Hằng số địa chỉ được cho trực tiếp trong lệnh
  - Giá trị địa chỉ nằm trong thanh ghi con trỏ
  - Địa chỉ = Địa chỉ cơ sở + giá trị dịch chuyển

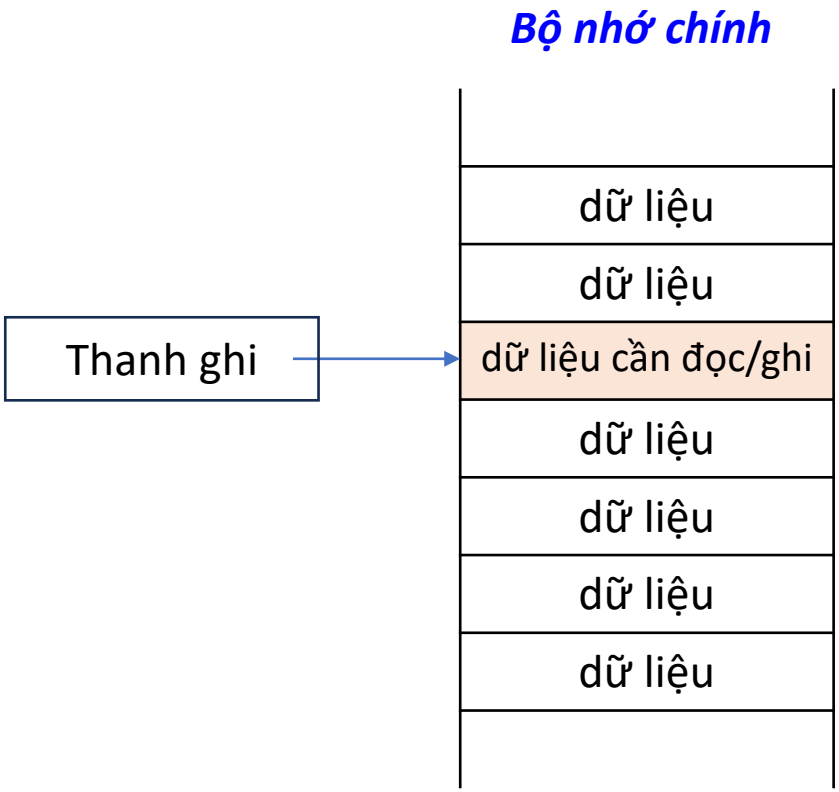
Hằng số địa chỉ

- Trong lệnh cho hằng số địa chỉ cụ thể
- Bộ xử lý phát giá trị địa chỉ này đến bộ nhớ để tìm ra ngăn nhớ dữ liệu cần đọc/ghi



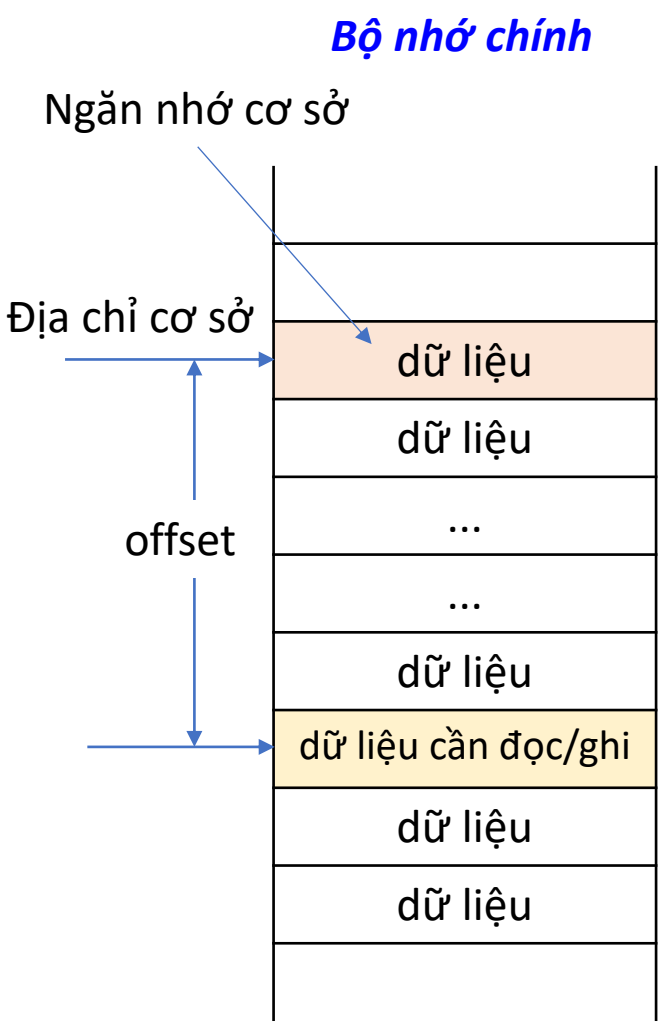
Sử dụng thanh ghi con trỏ

- Trong lệnh cho biết tên thanh ghi con trỏ
- Thanh ghi con trỏ chứa giá trị địa chỉ
- Bộ xử lý phát địa chỉ này ra để tìm ra ngăn nhớ dữ liệu cần đọc/ghi



Sử dụng địa chỉ cơ sở và dịch chuyển

- Địa chỉ cơ sở (base address): địa chỉ của ngăn nhớ cơ sở
- Dịch chuyển địa chỉ (offset): gia số địa chỉ giữa ngăn nhớ cần đọc/ghi so với ngăn nhớ cơ sở
- Địa chỉ của ngăn nhớ cần đọc/ghi = (địa chỉ cơ sở) + (offset)
- Có thể sử dụng các thanh ghi để quản lý các tham số này
- Trường hợp riêng:
  - Địa chỉ cơ sở = 0 → offset chính là địa chỉ ngăn nhớ
  - Offset = 0 → ngăn nhớ cần đọc/ghi là ngăn nhớ cơ sở

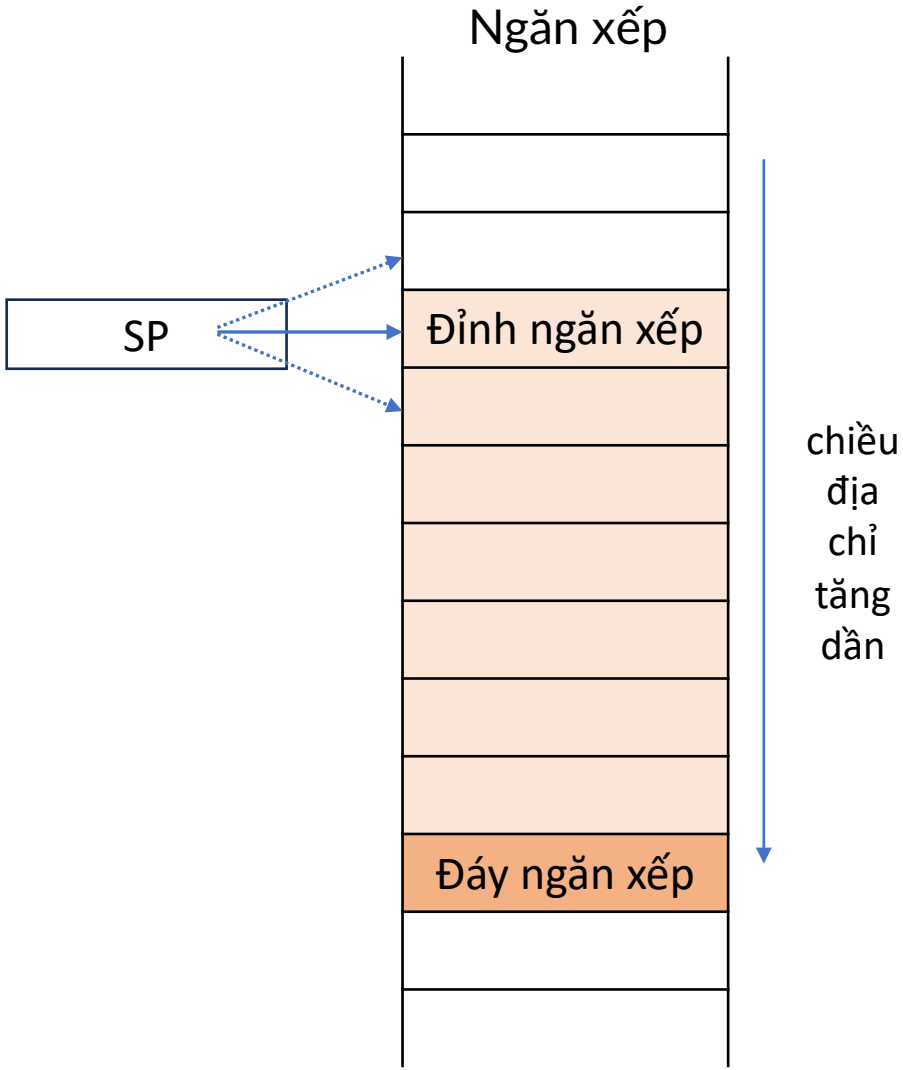


Ngăn xếp (Stack)

- Ngăn xếp là vùng nhớ dữ liệu có cấu trúc LIFO (Last In - First Out vào sau - ra trước)
- Ngăn xếp thường dùng để phục vụ cho chương trình con
- Đáy ngăn xếp là một ngăn nhớ xác định
- Đỉnh ngăn xếp là thông tin nằm ở vị trí trên cùng trong ngăn xếp
- Đỉnh ngăn xếp có thể bị thay đổi
- Ngăn xếp được dùng trên các kiến trúc phổ biến (VD: Intel x86)

Con trỏ ngăn xếp SP (Stack Pointer)

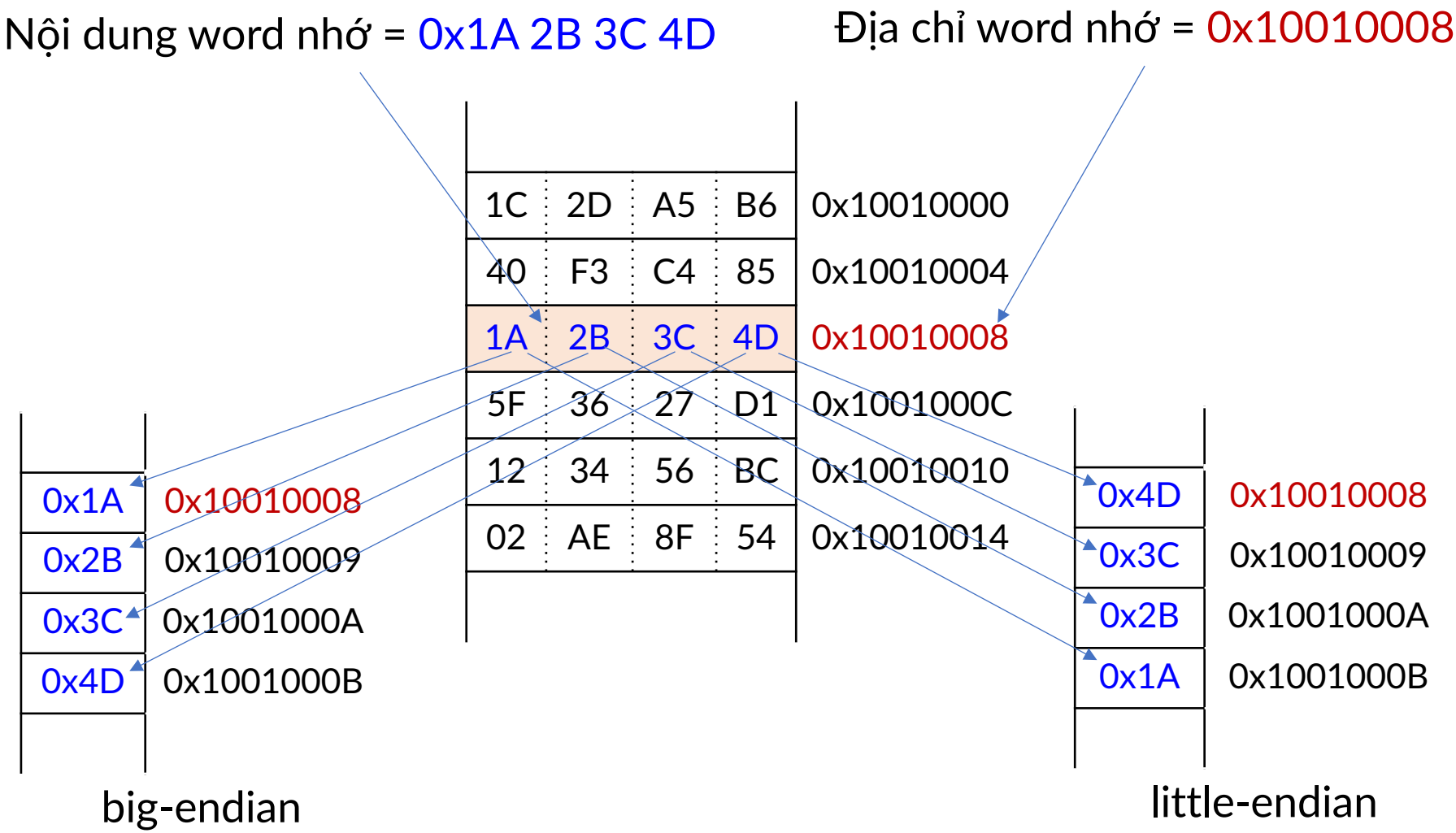
- SP là thanh ghi chứa địa chỉ của ngăn nhớ đỉnh ngăn xếp
- Khi cất thêm thông tin vào ngăn xếp:
  - Giảm nội dung của SP
- Khi lấy thông tin ra khỏi ngăn xếp:
  - Tăng nội dung của SP
- Khi ngăn xếp rỗng, SP trỏ vào đáy



Thứ tự lưu trữ các byte trong bộ nhớ chính

- Bộ nhớ chính được đánh địa chỉ cho từng byte
- Hai cách lưu trữ thông tin nhiều byte:
  - Đầu nhỏ (Little-endian): Byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ lớn.
  - Đầu to (Big-endian): Byte có ý nghĩa cao được lưu trữ ở ngăn nhớ có địa chỉ nhỏ, byte có ý nghĩa thấp được lưu trữ ở ngăn nhớ có địa chỉ lớn.
- Các sản phẩm thực tế:
  - Intel x86, RISC-V: little-endian
  - Motorola 680x0, SunSPARC: big-endian
  - ARM: little-endian hoặc big-endian (tuỳ phiên bản)

Minh họa lưu trữ trong bộ nhớ chính



Lưu ý: Địa chỉ word nhớ là địa chỉ của byte nhớ đầu tiên của word nhớ đó

Tập lệnh

- Mỗi bộ xử lý được thiết kế theo một tập lệnh xác định
- Tập lệnh thường có hàng chục đến hàng trăm lệnh
- Mỗi lệnh máy (mã máy) là một chuỗi các bit (0,1) mà bộ xử lý hiểu được để thực hiện một thao tác xác định.
- Các lệnh được mô tả bằng các ký hiệu gợi nhớ dạng text, đó chính là các lệnh của hợp ngữ (assembly language)

Dạng lệnh hợp ngữ

- Mã C:

```
a = b + c;
```

- Ví dụ lệnh hợp ngữ:

```
add a, b, c  # a = b + c
```

trong đó:

- add: ký hiệu gợi nhớ chỉ ra thao tác (phép toán) cần thực hiện.
  - Chú ý: mỗi lệnh chỉ thực hiện một thao tác
- b, c: các toán hạng nguồn *(source operands)*
- a: toán hạng đích *(destination operand)*
- phần sau dấu # là lời giải thích (chỉ có tác dụng đến hết dòng)

Các thành phần của lệnh máy

Mã thao tác	Địa chỉ toán hạng

- Mã thao tác (operation code hay opcode): mã hóa cho thao tác mà bộ xử lý phải thực hiện
  - Các thao tác chuyển dữ liệu
  - Các phép toán số học
  - Các phép toán logic
  - Các thao tác chuyển điều khiển (rẽ nhánh, nhảy)
- Địa chỉ toán hạng: chỉ ra nơi chứa các toán hạng mà thao tác sẽ tác động
  - Toán hạng có thể là:
    - Hằng số nằm ngay trong lệnh
    - Nội dung của thanh ghi → cần biết tên (số hiệu) thanh ghi
    - Nội dung của ngăn nhớ (hoặc cổng vào-ra) → cần biết địa chỉ

Số lượng địa chỉ toán hạng trong lệnh

- Ba địa chỉ toán hạng:
  - `add  r1, r2, r3 # r1 = r2 + r3`
  - Sử dụng phổ biến trên các kiến trúc hiện nay
- Hai địa chỉ toán hạng:
  - `add  r1, r2      # r1 = r1 + r2`
  - Sử dụng trên Intel x86, Motorola 680x0
- Một địa chỉ toán hạng:
  - `add  r1          # Acc = Acc + r1`
  - Được sử dụng trên kiến trúc thế hệ trước
- 0 địa chỉ toán hạng:
  - `add`
  - Các toán hạng đều được ngầm định ở ngăn xếp
  - Không thông dụng

Các kiến trúc tập lệnh CISC và RISC

- CISC: Complex Instruction Set Computer
  - Máy tính với tập lệnh phức tạp
  - Các bộ xử lý: Intel x86, Motorola 680x0
- RISC: Reduced Instruction Set Computer
  - Máy tính với tập lệnh thu gọn
  - MIPS, ARM, RISC-V, ...
  - RISC đối nghịch với CISC
  - Kiến trúc tập lệnh tiên tiến

Các đặc trưng của kiến trúc RISC

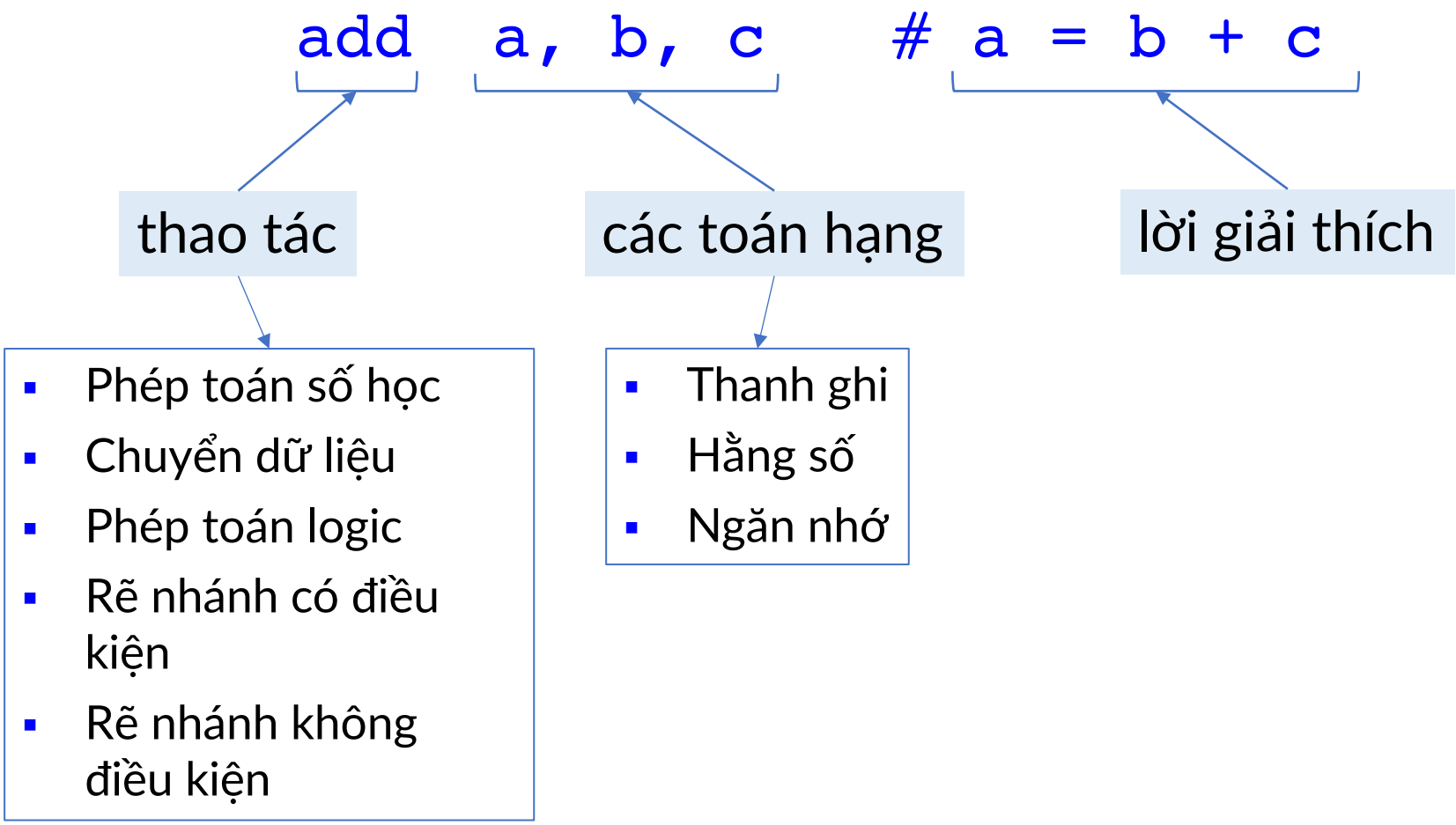
- Số lượng lệnh ít
- Hầu hết các lệnh truy nhập toán hạng ở các thanh ghi
- Truy nhập bộ nhớ bằng các lệnh LOAD/STORE (nạp/lưu)
- Thời gian thực hiện các lệnh là như nhau
- Các lệnh có độ dài cố định (thường là 32 bit)
- Số lượng dạng lệnh ít
- Có ít phương pháp định địa chỉ toán hạng
- Có nhiều thanh ghi
- Hỗ trợ các thao tác của ngôn ngữ bậc cao

Kiến trúc tập lệnh RISC-V

- RISC-V (RISC Five) - thế hệ thứ 5 của kiến trúc RISC
- Được phát triển bởi UC Berkeley (2010)
- Được quản lý bởi RISC-V Foundation (riscv.org) (2015)
- Kiến trúc tập lệnh mở (open standard ISA), điển hình cho kiến trúc tập lệnh hiện đại
- Kiến trúc 32-bit, 64-bit, và 128-bit
- Các phần tiếp theo trong chương này sẽ nghiên cứu kiến trúc tập lệnh RISC-V 32-bit
- Tài liệu:
  - Chapter 2 – COD RISC-V Edition
  - RISC-V Reference Data

3.2. Lệnh hợp ngữ và toán hạng

- Ví dụ lệnh hợp ngữ:



Tập thanh ghi của RISC-V 32

- RISC-V 32 có tập 32 thanh ghi 32-bit
  - từ x0 đến x31 (mã hóa bằng 5-bit)
- Qui ước gọi dữ liệu trong RISC-V:
  - Dữ liệu 32-bit được gọi là “word”
  - Dữ liệu 16-bit được gọi là “halfword”
  - Dữ liệu 64-bit được gọi là “doubleword”

Tập thanh ghi của RISC-V

Tên thanh ghi	Tên thanh ghi theo số hiệu	Công dụng
zero	x0	constant value 0, chứa hằng số = 0
ra	x1	return address, chứa địa chỉ trở về
sp	x2	stack pointer, con trỏ ngăn xếp
gp	x3	global pointer, con trỏ toàn cục
tp	x4	thread pointer, con trỏ luồng
t0-t2	x5-x7	temporaries, chứa các giá trị tạm thời
s0/fp	x8	saved register/frame pointer, lưu biến/con trỏ khung
s1	x9	saved register, thanh ghi lưu biến
a0-a1	x10-x11	function arguments / return values, tham số vào của thủ tục/các giá trị trả về của thủ tục
a2-a7	x12-x17	function arguments, tham số vào của thủ tục
s2-s11	x18-x27	saved registers, lưu các biến
t3-t6	x28-x31	temporaries, các giá trị tạm thời

Tập thanh ghi (tiếp)

- Các thanh ghi có thể được gọi theo tên: zero, ra, t0, t1, … hoặc x0, x1, x5, x6, …
- Các thanh ghi có chức năng xác định, ví dụ:
  - zero: luôn luôn giữ hằng số = 0
  - s0-s11: được sử dụng giữ các biến
  - t0-t6: được sử dụng giữ các giá trị tạm thời
  - sp: con trỏ ngăn xếp

Lệnh số học với toán hạng thanh ghi

- Lệnh **add**, lệnh **sub** (subtract) chỉ thao tác với toán hạng thanh ghi
  - `add  rd, rs1, rs2  # rd = rs1 + rs2`
  - `sub  rd, rs1, rs2  # rd = rs1 - rs2`
- Ví dụ mã C:

```
a = b - c;
f = (g + h) - (i + j);
```

  - giả thiết: a, b, c, f, g, h, i, j nằm ở s0, s1, s2, s3, s4, s5, s6, s7
- Mã hợp ngữ RISC-V:

```
sub  s0, s1, s2  # a = b - c
add  t0, s4, s5  # t0 = g + h
add  t1, s6, s7  # t1 = i + j
sub  s3, t0, t1  # f = (g+h)-(i+j)
```

Toán hạng hằng số (immediate)

- Toán hạng hằng số được xác định ngay trong lệnh

```
addi rd, rs1, imm  # rd = rs1+imm
```

`imm` là hằng số nguyên có dấu 12-bit [-2048, +2047]

- Mã C:

```
a = a + 4;
b = a - 12;
```

- Mã hợp ngữ RISC-V:

```
# s0 = a,  s1 = b
addi s0, s0, 4     # a = a+4
addi s1, s0, -12   # b = a-12
```

Lưu ý: Không có lệnh trừ (subi) với giá trị hằng số

Toán hạng ở bộ nhớ

- Muốn thực hiện phép toán số học với toán hạng ở bộ nhớ, cần phải:
  - Nạp (load) giá trị từ bộ nhớ vào thanh ghi
  - Thực hiện phép toán trên thanh ghi
  - Lưu (store) kết quả từ thanh ghi ra bộ nhớ
- Bộ nhớ được đánh địa chỉ theo byte
  - RISC-V sử dụng 32-bit để đánh địa chỉ cho các byte nhớ và các cổng vào-ra
  - Không gian địa chỉ: 0x00000000 – 0xFFFFFFFF
  - Mỗi word có độ dài 32-bit chiếm 4-byte trong bộ nhớ, địa chỉ của các word là bội của 4 (địa chỉ của byte đầu tiên)
- Với kiến trúc RISC-V, bộ nhớ lưu trữ theo thứ tự đầu nhỏ (little-endian)

Minh hoạ byte nhớ và word nhớ

Nội dung	Địa chỉ byte nhớ
0x01	**0x0000 0000**
0x23	0x0000 0001
0x45	0x0000 0002
0x67	0x0000 0003
0x89	**0x0000 0004**
0xAB	0x0000 0005
0xCD	0x0000 0006
0xEF	0x0000 0007
.	
.	
.	
	0xFFFF FFFB
0x1A	**0xFFFF FFFC**
0x2B	0xFFFF FFFD
0x3C	0xFFFF FFFE
0x4D	0xFFFF FFFF

2^{32} bytes

Nội dung (Hexa)				Địa chỉ word nhớ
67	45	23	01	0x0000 0000
EF	CD	AB	89	0x0000 0004
				0x0000 0008
				0x0000 000C
				0x0000 0010
				0x0000 0014
				0x0000 0018
.				
.				
.				
				0xFFFF FFF4
				0xFFFF FFF8
4D	3C	2B	1A	0xFFFF FFFC

2^{30} words

Địa chỉ của word nhớ là địa chỉ của byte đầu tiên trong word nhớ đó

Lệnh lw (load word)

- Nạp (đọc) word dữ liệu 32-bit từ bộ nhớ đưa vào thanh ghi

```
lw  rd, imm(rs1) # rd = mem[rs1+imm]
```

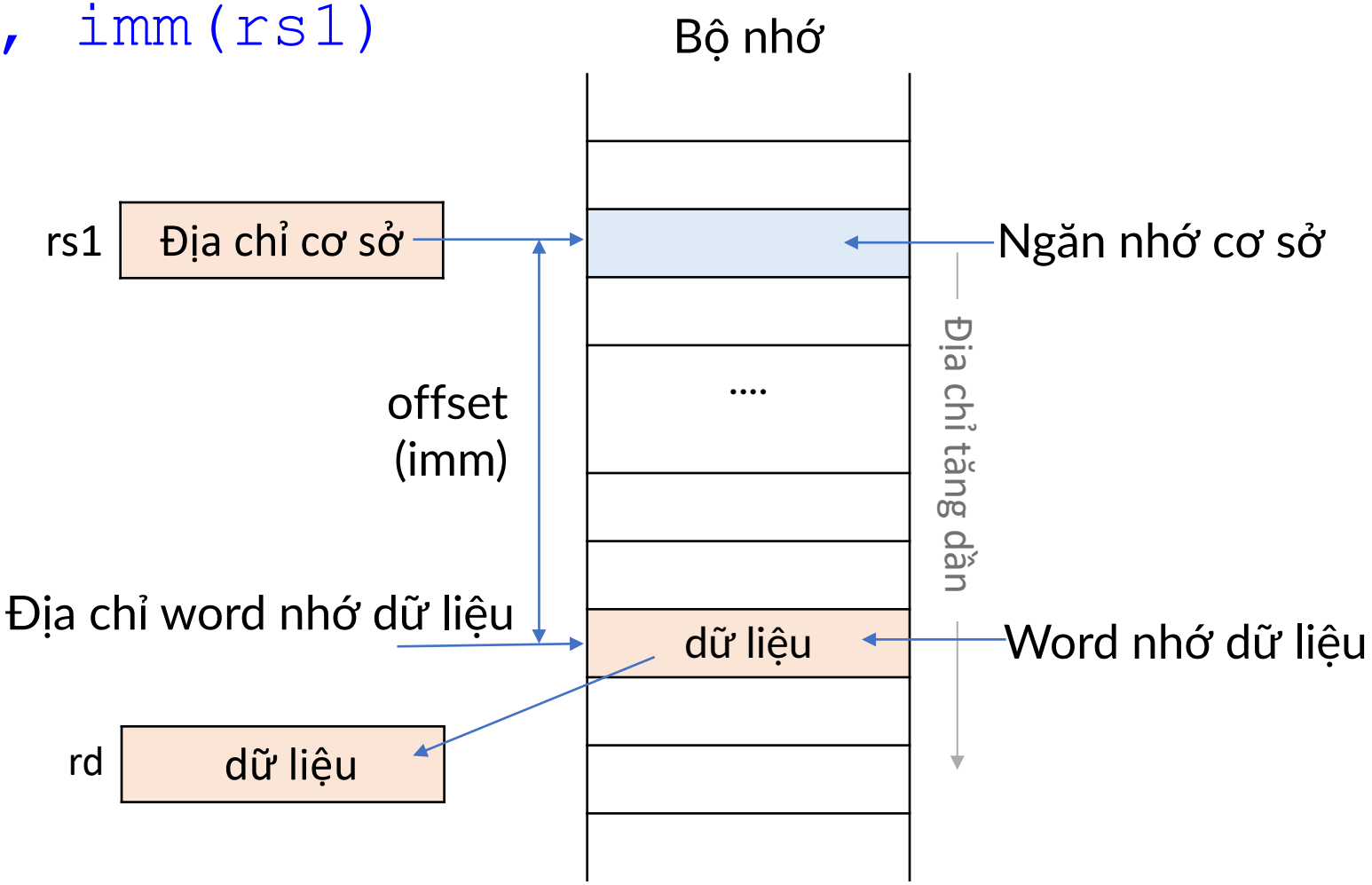
  - rs1: thanh ghi chứa địa chỉ cơ sở (base address)
  - imm (immediate): hằng số dịch chuyển địa chỉ (offset)

  → địa chỉ của word dữ liệu cần đọc = địa chỉ cơ sở + imm

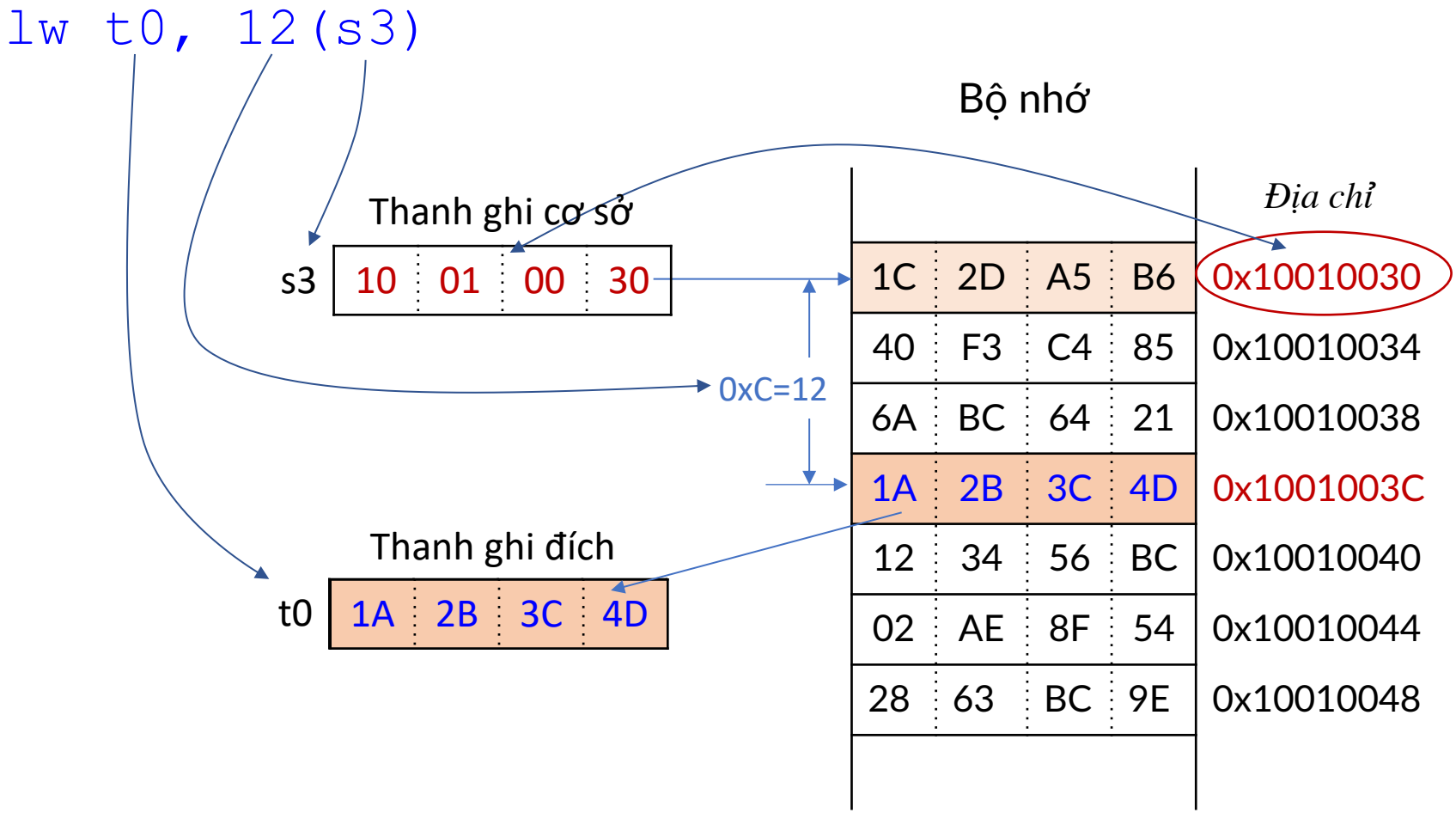
  - rd: thanh ghi đích, nơi chứa word dữ liệu được nạp vào
  - Giá trị địa chỉ cơ sở và imm đều chia hết cho 4

Minh hoạ lệnh lw (load word)

```
lw rd, imm(rs1)
```



Ví dụ lệnh lw


lw t0, 12(s3)
Bộ nhớ
Địa chỉ
Thanh ghi cơ sở
s3
10
01
00
30
0xC=12
Thanh ghi đích
t0
1A
2B
3C
4D
1C 2D A5 B6 0x10010030
40 F3 C4 85 0x10010034
6A BC 64 21 0x10010038
1A 2B 3C 4D 0x1001003C
12 34 56 BC 0x10010040
02 AE 8F 54 0x10010044
28 63 BC 9E 0x10010048

Lệnh sw (store word)

- Lưu (ghi) word dữ liệu 32-bit từ thanh ghi ra bộ nhớ

```
sw rs2, imm(rs1) # mem[rs1+imm] = rs2
```

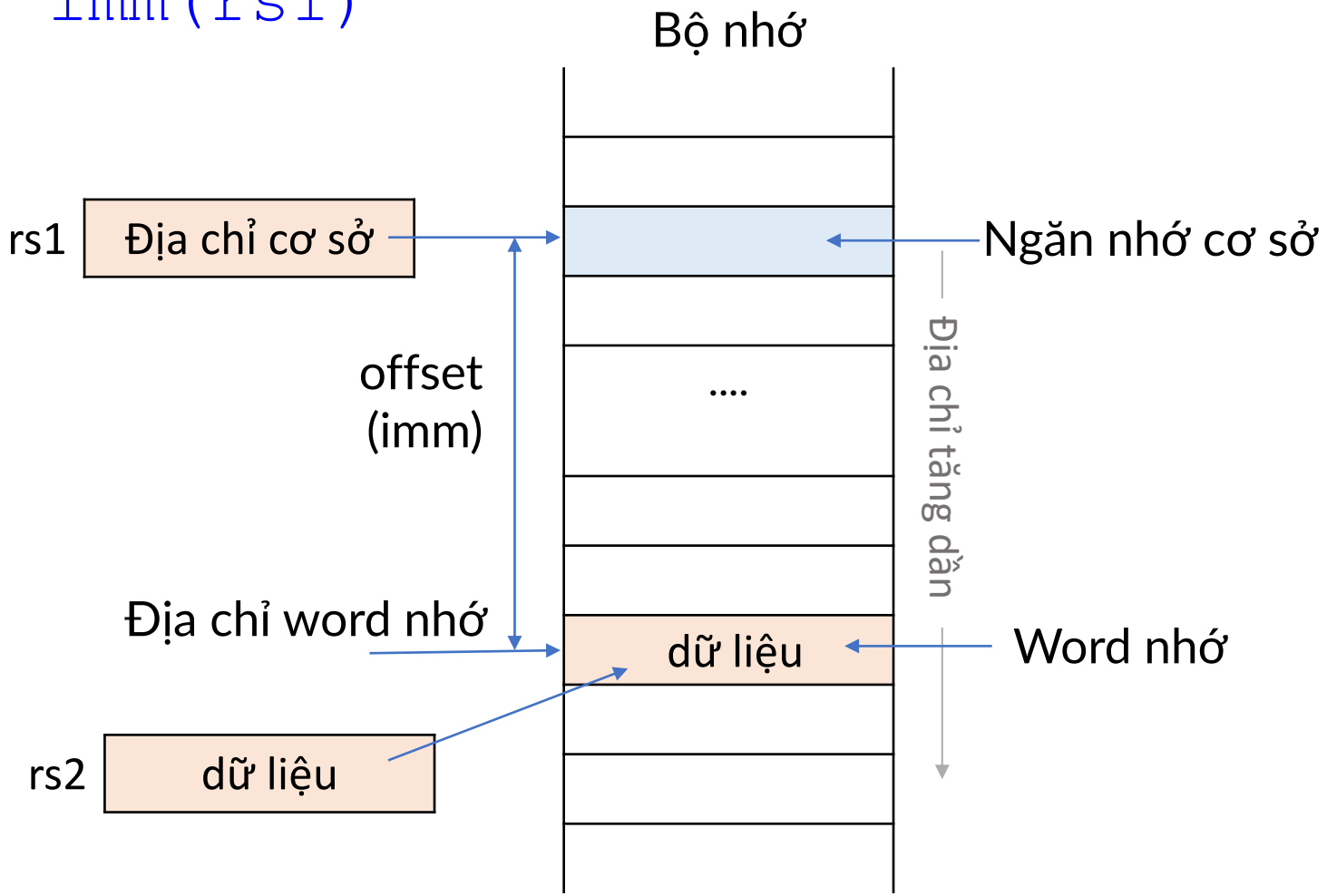
  - rs2: thanh ghi nguồn, chứa word dữ liệu cần ghi ra bộ nhớ
  - rs1: thanh ghi chứa địa chỉ cơ sở (base address)
  - imm (immediate): hằng số dịch chuyển địa chỉ (offset)

  → địa chỉ của nơi ghi word dữ liệu = địa chỉ cơ sở + imm

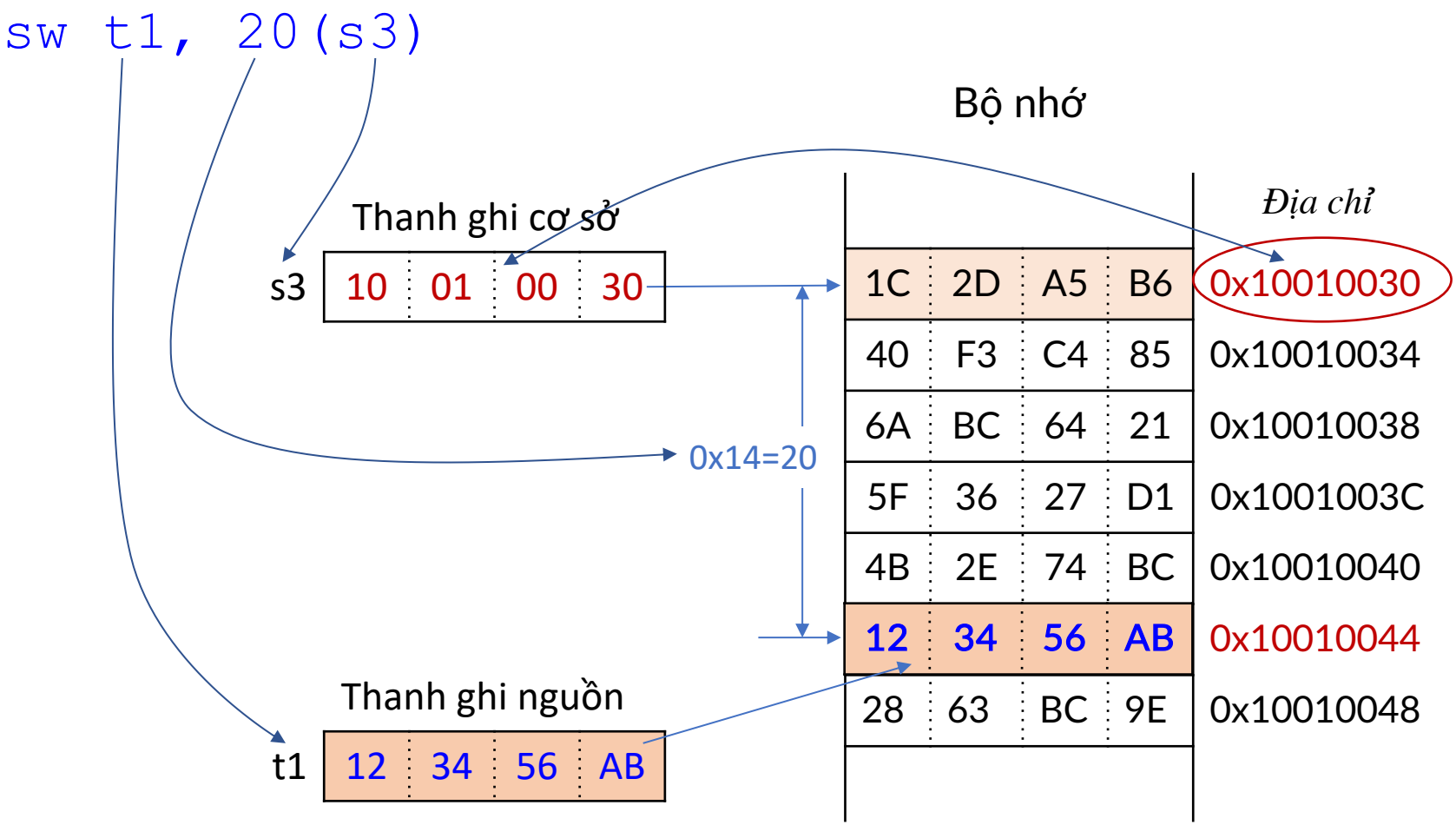
  - Giá trị địa chỉ cơ sở và imm đều chia hết cho 4

Minh hoạ lệnh sw

```
sw rs2, imm(rs1)
```



Ví dụ lệnh sw


sw t1, 20(s3)
Bộ nhớ
Địa chỉ
Thanh ghi cơ sở
s3
10 01 00 30
0x14=20
Thanh ghi nguồn
t1
12 34 56 AB
1C 2D A5 B6 0x10010030
40 F3 C4 85 0x10010034
6A BC 64 21 0x10010038
5F 36 27 D1 0x1001003C
4B 2E 74 BC 0x10010040
12 34 56 AB 0x10010044
28 63 BC 9E 0x10010048

Ví dụ toán hạng bộ nhớ (1)

- Mã C:

// A là mảng các phần tử số nguyên 32-bit

```
g = h + A[8];
```

  - Cho g ở s4, h ở s5
  - s6 chứa địa chỉ cơ sở của mảng A

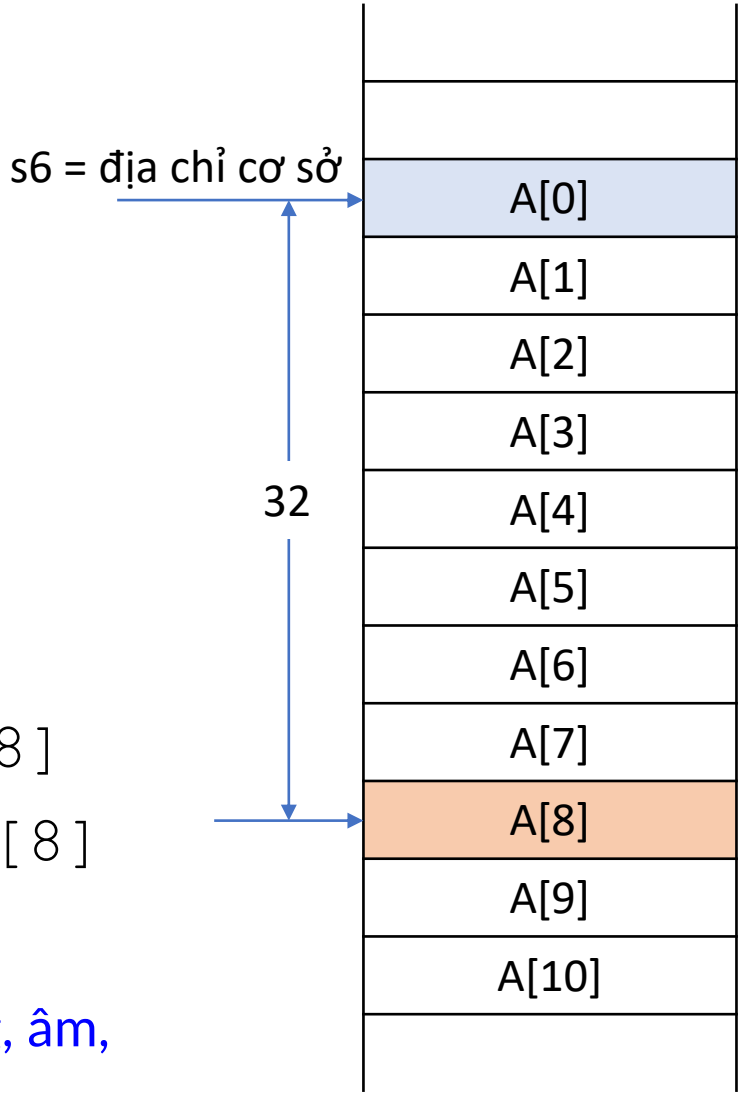
- Mã hợp ngữ RISC-V:

Chỉ số 8, do đó offset = 32

```
lw  t0, 32(s6)   # t0 = A[8]
add s4, s5, t0   # g = h+A[8]
```

offset

Thanh ghi cơ sở



Chú ý: offset phải là hằng số, có thể dương, âm, hoặc bằng 0

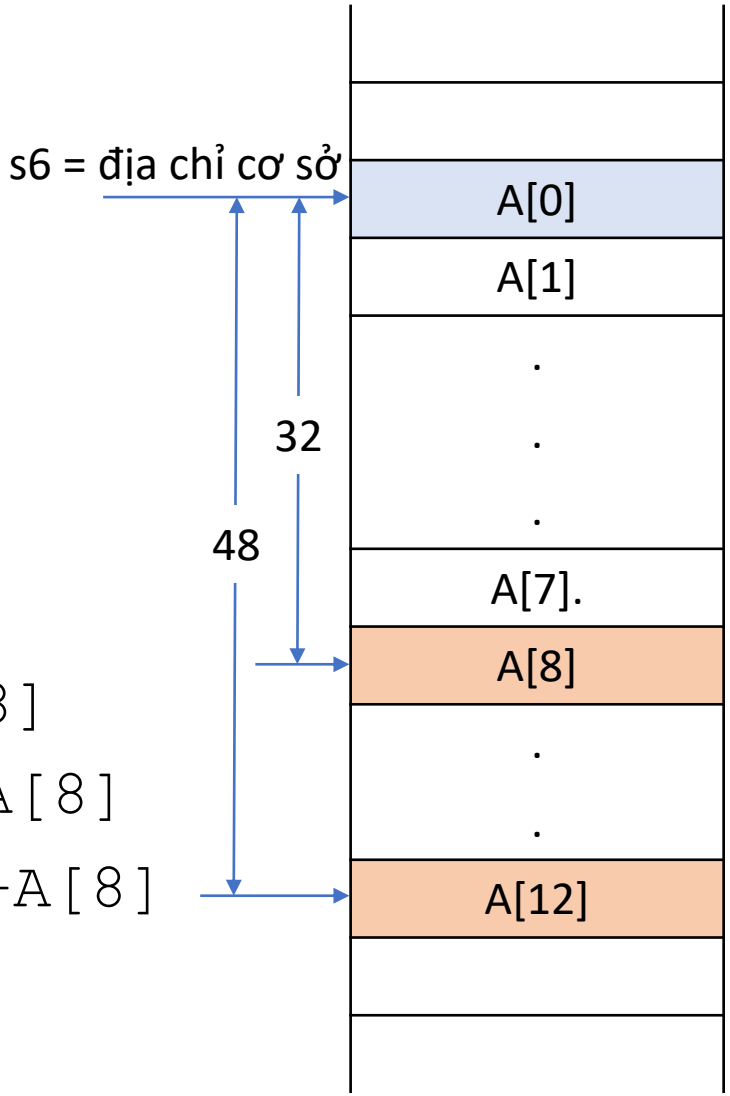
Ví dụ toán hạng bộ nhớ (2)

- Mã C:

// A là mảng các phần tử số nguyên 32-bit

```
A[12] = h + A[8];
```

  - Cho h ở s5
  - s6 chứa địa chỉ cơ sở của mảng A



- Mã hợp ngữ RISC-V:

```
lw   t0, 32(s6)   # t0 = A[8]
add  t0, s5, t0   # t0 = h+A[8]
sw   t0, 48(s6)   # A[12]=h+A[8]
```

Thanh ghi với Bộ nhớ

- Truy nhập thanh ghi nhanh hơn bộ nhớ
- Thao tác dữ liệu trên bộ nhớ yêu cầu nạp (load) và lưu (store)
  - Cần thực hiện nhiều lệnh hơn
- Chương trình dịch sử dụng các thanh ghi cho các biến nhiều nhất có thể
  - Chỉ sử dụng bộ nhớ cho các biến ít được sử dụng
  - Cần tối ưu hóa sử dụng thanh ghi

Nạp hằng số vào thanh ghi

- Trường hợp hằng số nguyên có dấu 12-bit (nằm trong khoảng [-2048, +2047])→ sử dụng lệnh addi
- Mã C:

// int là số nguyên có dấu 32-bit

```
int a = 5;
int b = -240;
```

- Mã hợp ngữ RISC-V

```
addi s0, zero, 5     # s0 = 5
addi s1, zero, -240  # s1 = -240
```

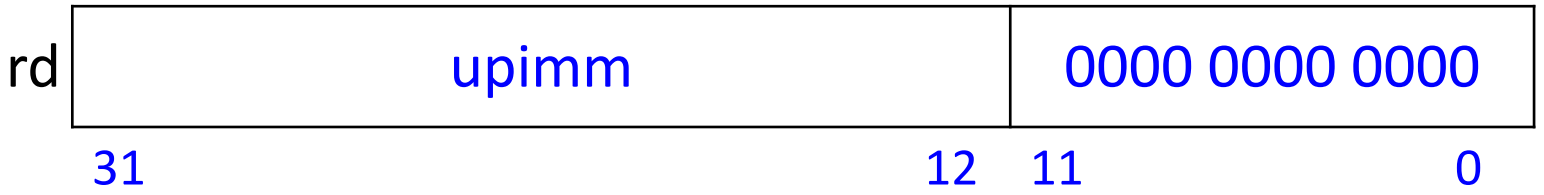
- Trong trường hợp hằng số cần nhiều hơn 12-bit → không thể sử dụng cách này

Nạp hằng số 32-bit vào thanh ghi

- Trong trường hợp hằng số 32-bit → sử dụng lệnh lui và lệnh addi
- Lệnh lui (load upper immediate)

```
lui rd, upimm
```

  - Đưa hằng số upimm vào 20 bit cao của thanh ghi đích
  - 12 bit thấp cuả thanh ghi đích = 0



Ví dụ khởi tạo thanh ghi 32-bit

- Mã C:

```
int  a = 0x21AB563C;
```

- Mã hợp ngữ RISC-V:

```
lui  s0,0x21AB5     # s0 = 0x21AB5000
addi s0,s0,0x63C    # s0 = 0x21AB563C
```

2 1 A B 5

0010 0001 1010 1011 0101 0000 0000 0000

2 1 A B 5 6 3 C

0010 0001 1010 1011 0101 0110 0011 1100

Ví dụ khởi tạo thanh ghi 32-bit

- Nếu bit 11 của hằng số 32-bit là 1, cần tăng giá trị phần bên trái (20 bit cao) thêm 1 khi sử dụng lệnh `lui`

- Mã C:

```
int  b = 0x12345EDC; //lưu ý: 0xEDC=-292
                     // 1110 1101 1100
```

- Mã hợp ngữ RISC-V:

```
lui  s1,0x12346    #s1=0x12346000
addi s1,s1,-292    #s1=0x12346000+0xFFFFFEDC
                   #  =0x12345EDC
```

3.3. Các lệnh logic

Các lệnh logic: Thao tác trên các bit của dữ liệu

Phép toán logic	Toán tử trong C	Lệnh của RISC-V
Shift left	<<	`slli`
Shift right	>>	`srli`
Bitwise AND	&	`and, andi`
Bitwise OR	|	`or, ori`
Bitwise XOR	^	`xor, xori`
Bitwise NOT	~	`xor, xori`

Lệnh logic: and, or, xor

```
and  rd, rs1, rs2 # rd = rs1 & rs2
or   rd, rs1, rs2 # rd = rs1 | rs2
xor  rd, rs1, rs2 # rd = rs1 ^ rs2
```

Nội dung các thanh ghi nguồn

s1	0100	0110	1010	0001	1100	0000	1011	0111
s2	1111	1111	1111	1111	0000	0000	0000	0000

Mã hợp ngữ	Kết quả thanh ghi đích								
and s3, s1, s2	s3								
or s4, s1, s2	s4								
xor s5, s1, s2	s5								

Lệnh logic: and, or, xor

```
and  rd, rs1, rs2 # rd = rs1 & rs2
or   rd, rs1, rs2 # rd = rs1 | rs2
xor  rd, rs1, rs2 # rd = rs1 ^ rs2
```

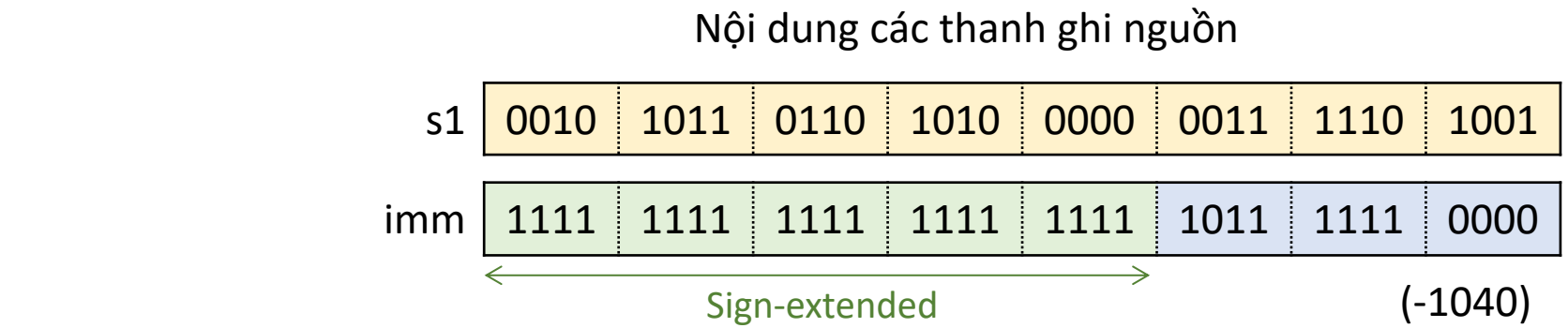
Nội dung các thanh ghi nguồn

s1	0100	0110	1010	0001	1100	0000	1011	0111
s2	1111	1111	1111	1111	0000	0000	0000	0000

Mã hợp ngữ	Kết quả thanh ghi đích								
and s3, s1, s2	s3	0100	0110	1010	0001	0000	0000	0000	0000
or s4, s1, s2	s4	1111	1111	1111	1111	1100	0000	1011	0111
xor s5, s1, s2	s5	1011	1001	0101	1110	1100	0000	1011	0111

Lệnh logic: andi, ori, xori

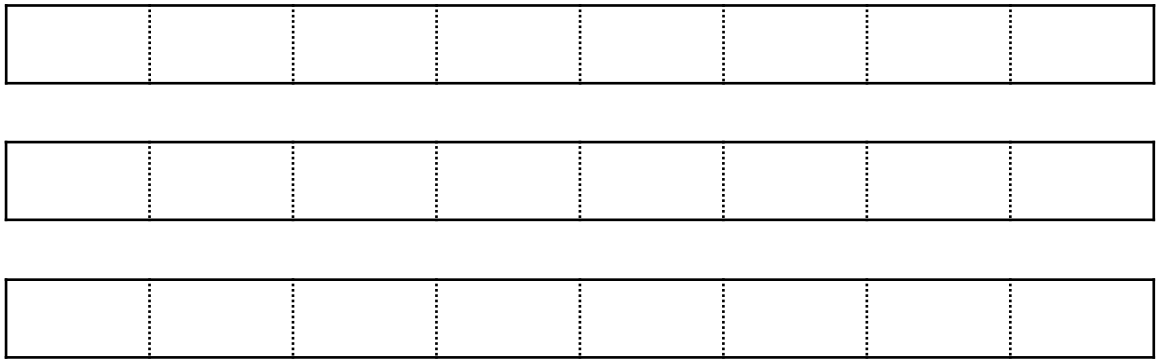
```
# imm: hằng số nguyên 12-bit có dấu
  andi  rd, rs1, imm   # rd = rs1 & SignExt(imm)
  ori   rd, rs1, imm   # rd = rs1 | SignExt(imm)
  xori  rd, rs1, imm   # rd = rs1 ^ SignExt(imm)
```



Mã hợp ngữ

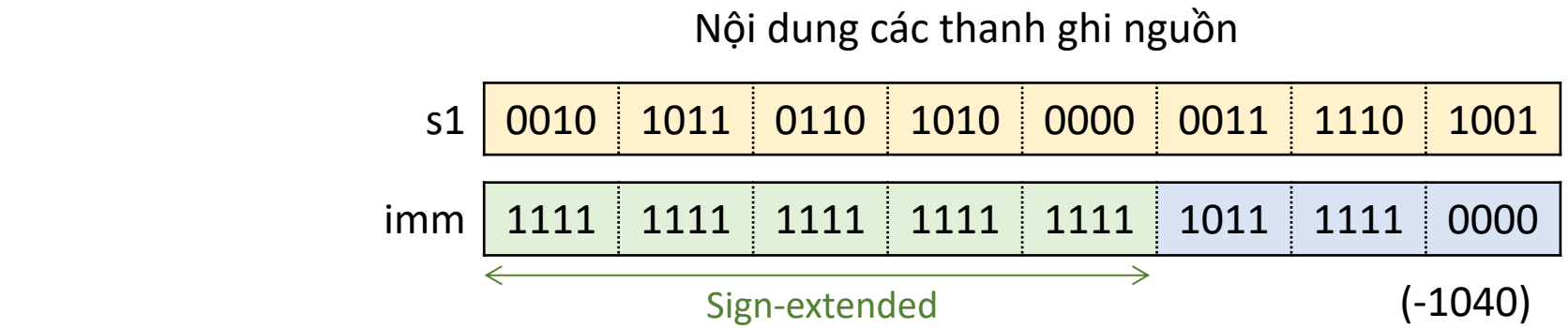
Kết quả thanh ghi đích

```
andi  s6, s1, -1040
ori   s7, s1, -1040
xori  s8, s1, -1040
```



Lệnh logic: andi, ori, xori

```
# imm: hằng số nguyên 12-bit có dấu
  andi  rd, rs1, imm   # rd = rs1 & SignExt(imm)
  ori   rd, rs1, imm   # rd = rs1 | SignExt(imm)
  xori  rd, rs1, imm   # rd = rs1 ^ SignExt(imm)
```



Mã hợp ngữ	Kết quả thanh ghi đích							
andi s6, s1, -1040	s6	0010	1011	0110	1010	0000	0011	1110	0000
ori s7, s1, -1040	s7	1111	1111	1111	1111	1111	1011	1111	1001
xori s8, s1, -1040	s8	1101	0100	1001	0101	1111	1000	0001	1001

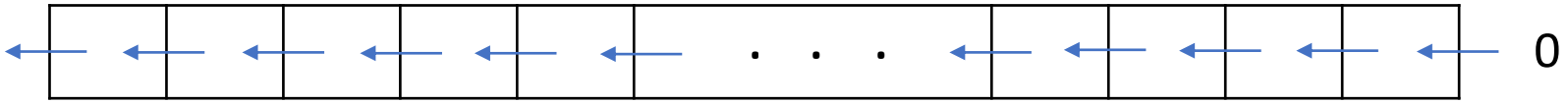
Ý nghĩa của các phép toán logic

- Phép AND: giữ nguyên một số bit trong word, xóa các bit còn lại về 0
- Phép OR: giữ nguyên một số bit trong word, thiết lập các bit còn lại lên 1
- Phép XOR: giữ nguyên một số bit trong word, đảo giá trị các bit còn lại
- Phép NOT: đảo các bit trong word
  - Đổi 0 thành 1, và đổi 1 thành 0
  - RISC-V không có lệnh NOT, dùng lệnh `xori` để đảo bit
  - A XOR -1 = NOT A
  - `xori s1, s0, -1 # s1 = not s0`

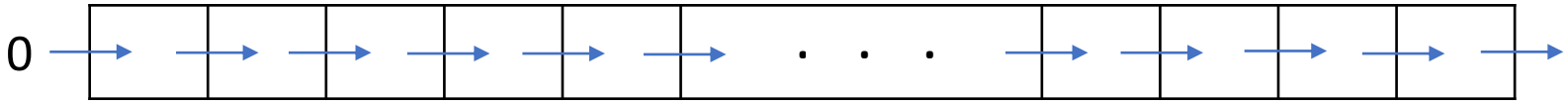
Lưu ý: -1 = 0xFFF được mở rộng thành 0xFFFFFFFF

Phép dịch bit

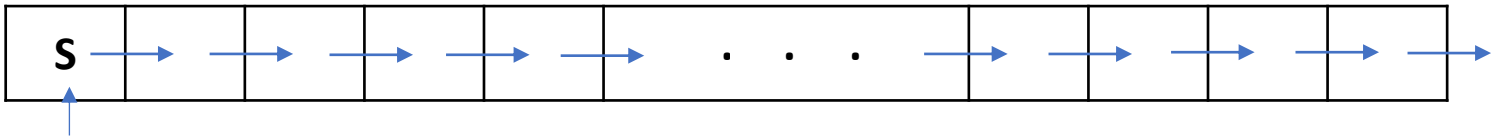
Dịch trái logic



Dịch phải logic



Dịch phải số học



giữ nguyên bit dấu S

- Dịch trái *k* bit → nhân với 2^k (kết quả trong 32-bit)
- Dịch phải *k* bit → chia cho 2^k lấy phần nguyên

Lệnh dịch bit: sll, srl, sra

- `sll` shift left logical (dịch trái logic)
  - `sll rd, rs1, rs2  # rd = rs1 << rs2`
  - Dịch trái nội dung rs1 và cất kết quả sang rd
  - rs2: chứa số vị trí được dịch
- `srl` shift right logical (dịch phải logic)
  - `srl rd, rs1, rs2  # rd = rs1 >> rs2`
  - Dịch phải nội dung rs1 và cất kết quả sang rd
  - rs2: chứa số vị trí được dịch
- `sra` shift right arithmetic (dịch phải số học)
  - `sra rd, rs1, rs2  # rd = rs1 >>> rs2`
  - Giữ nguyên bit dấu

Ví dụ

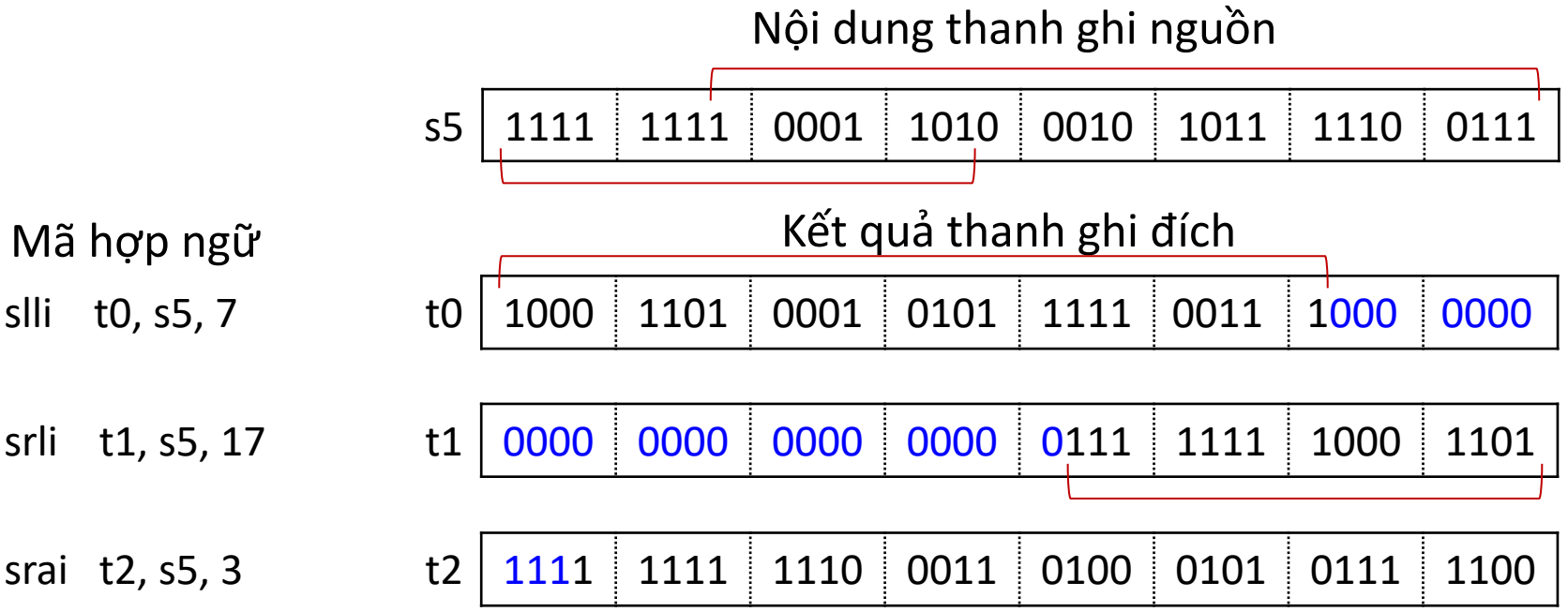
- Giả sử cho nội dung các thanh ghi:
  - s0 = 0x00000013 (=19_{10}) ; s1 = 0xFFFFFFF0 (=-16)
  - t0 = 3; t1 = 2
- Tìm nội dung s2, s3, s4 sau khi thực hiện các lệnh:
  - `sll  s2, s0, t0     # s2 = 19x8 = 152 = 0x00000098`
  - `srl  s3, s0, t1     # s3 = phần nguyên của 19/4 = 4`
  - `sra  s4, s1, t1     # s4 = -16/4 = -4 = 0xFFFFFFFC`

	s0	0000	0000	0000	0000	0000	0000	0001	0011	19
sll s2, s0, t0	s2	0000	0000	0000	0000	0000	0000	1001	1000	152
srl s3, s0, t1	s3	0000	0000	0000	0000	0000	0000	0000	0100	4
	s1	1111	1111	1111	1111	1111	1111	1111	0000	-16
srai s4, s1, t1	s4	1111	1111	1111	1111	1111	1111	1111	1100	-4

Lệnh dịch bit: slli, srli, srai

- `slli  rd, rs1, uimm  # rd = rs1 << uimm`
- `srli  rd, rs1, uimm  # rd = rs1 >> uimm`
- `srai  rd, rs1, uimm  # rd = rs1 >>> uimm`

`uimm` là hằng số nguyên không dấu 5-bit, chỉ ra dịch bao nhiêu bit



3.4. Dịch các câu lệnh điều khiển

- Các câu lệnh điều khiển rẽ nhánh
  - `if`
  - `if/else`
  - `switch/case`
- Các câu lệnh lặp
  - `while`
  - `do while`
  - `for`

Các lệnh rẽ nhánh

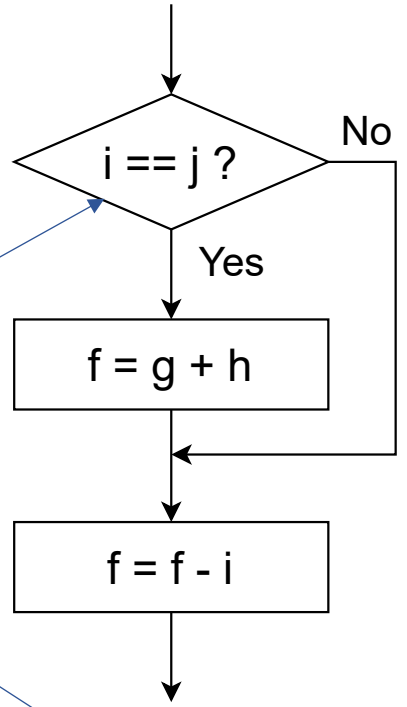
- Các lệnh rẽ nhánh (có điều kiện): Nếu điều kiện đúng, rẽ nhánh đến lệnh được đánh nhãn tương ứng; ngược lại chuyển sang thực hiện lệnh kế tiếp (tuần tự)
  - `beq rs1, rs2, L1`
    - branch if equal
    - nếu (rs1 == rs2) rẽ nhánh đến lệnh ở nhãn L1
  - `bne rs1, rs2, L1`
    - branch if not equal
    - nếu (rs1 != rs2) rẽ nhánh đến lệnh ở nhãn L1
  - `blt rs1, rs2, L1`
    - branch if less than
    - nếu (rs1 < rs2) rẽ nhánh đến lệnh ở nhãn L1
  - `bge rs1, rs2, L1`
    - branch if greater than or equal
    - nếu (rs1 >= rs2) rẽ nhánh đến lệnh ở nhãn L1

Dịch câu lệnh `if`

- Mã C:

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

  - f, g, h, i, j ở s0, s1, s2, s3, s4



- Mã hợp ngữ RISC-V:

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
     bne s3, s4, L1      # Nếu i=j
     add s0, s1, s2      # thì f=g+h
L1:  sub s0, s0, s3      # f=f-i
```

Điều kiện ngược nhau

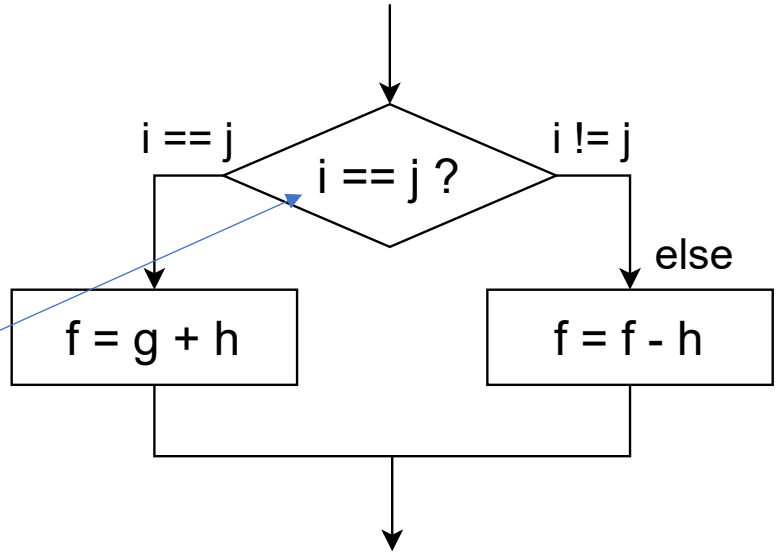
(Chú ý: Tên NHÃN là do người lập trình tự đặt)

Dịch câu lệnh `if/else`

- Mã C:

```
if (i==j) f = g+h;
else f = f-h;
```

  - f, g, h, i, j ở s0, s1, s2, s3, s4



- Mã hợp ngữ RISC-V:

```
      bne s3, s4, else    # Nếu i=j
      add s0, s1, s2      # thì f=g+h
      beq x0, x0, done    # rẽ đến done
else: sub s0, s0, s2      # nếu i!=j thì f=f-h
done: …
```

nhảy qua nhánh else

điều kiện ngược nhau

Dịch câu lệnh switch/case

Mã C:

```
switch (button) {
   case 1:  amt = 20; break;
   case 2:  amt = 50; break;
   case 3:  amt = 100; break;
   default: amt = 0;
}
```

// tương đương với sử dụng các câu lệnh if/else

```
if       (button == 1) amt = 20;
else if  (button == 2) amt = 50;
else if  (button == 3) amt = 100;
else                   amt = 0;
```

Dịch câu lệnh switch/case

Mã hợp ngữ RISC-V

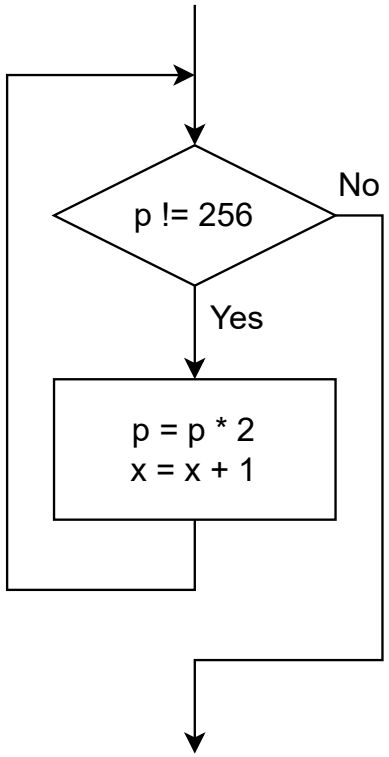
```
# s0 = button, s1 = amt
case1:
    addi    t0, zero, 1         # t0 = 1
    bne     s0, t0, case2       # button == 1? if not, skip to case2
    addi    s1, zero, 20        # if yes, amt = 20
    beq     x0, x0, done        # and break out of case
case2:
    addi    t0, zero, 2         # t0 = 2
    bne     s0, t0, case3       # button == 2? if not, skip to case3
    addi    s1, zero, 50        # if yes, amt = 50
    beq     x0, x0, done        # and break out of case
case3:
    addi    t0, zero, 3         # t0 = 3
    bne     s0, t0, default     # button == 3? if not, skip to default
    addi    s1, zero, 100       # if yes, amt = 100
    beq     x0, x0, done        # and break out of case
default:
   add      s1, zero, zero      # amt = 0
done:
```

cuối mỗi nhánh (trừ nhánh cuối cùng) có một lệnh nhảy về nhãn done

Dịch vòng lặp `while`

- Mã C:

```
// xác định giá trị của x
// sao cho 2^x = 256
    int p = 1;
    int x = 0;
    while (p != 256) {
      p = p * 2;
      x = x + 1;
    }
```

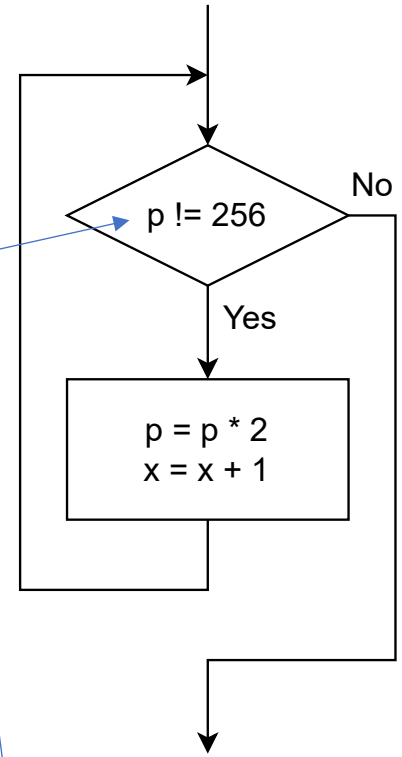


giả thiết: s0 = p, s1 = x

Dịch vòng lặp `while`

- Mã hợp ngữ RISC-V:

```
# s0 = p, s1 = x
  addi s0, zero, 1       # p = 1
  add  s1, zero, zero    # x = 0
  addi t0, zero, 256     # t0 = 256
loop:
  beq  s0, t0, done      # p=256, thoát
  slli s0, s0, 1         # p = p*2
  addi s1, s1, 1         # x = x+1
  beq  zero, zero, loop  # lặp lại
done:
```



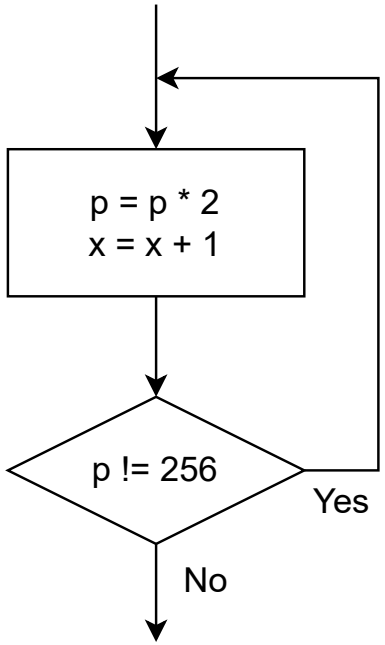
rẽ nhánh không điều kiện để quay lên

điều kiện ngược nhau

Dịch vòng lặp do...while

- Mã C:

```
// xác định giá trị của x
// sao cho 2^x = 256
    int p = 1;
    int x = 0;
    do {
      p = p * 2;
      x = x + 1;
    } while (p != 256);
```

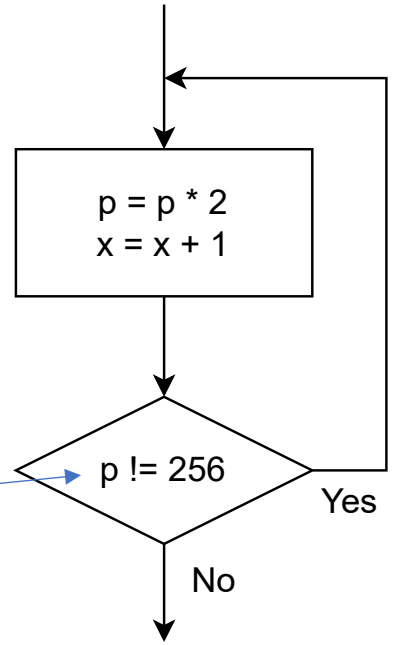


giả thiết: s0 = p, s1 = x

Dịch vòng lặp do...while

- Mã hợp ngữ RISC-V:

```
# s0 = p, s1 = x
   addi s0, zero, 1     # p = 1
   add  s1, zero, zero  # x = 0
   addi t0, zero, 256   # t0 = 256
loop:
   slli s0, s0, 1       # p = p*2
   addi s1, s1, 1       # x = x+1
   bne  s0, t0, loop    # p!=256, lặp lại
done:
```



điều kiện giống nhau

Dịch vòng lặp for – ví dụ 1 – viết kiểu while

- Mã C:

```
  // add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i++) {
  sum = sum + i;
}
```

- Mã hợp ngữ RISC-V:

```
# s0 = i, s1 = sum
      addi   s1, zero, 0    # sum = 0
      addi   s0, zero, 0    # i = 0
      addi   t0, zero, 10   # t0 = 10
for:  beq    s0, t0, done   # Nếu i=10, thoát
      add    s1, s1, s0     # Nếu i<10 thì sum = sum+i
      addi   s0, s0, 1      # tăng i thêm 1
      beq    x0, x0, for    # quay lại for
done: ...
```

Dịch vòng lặp for – ví dụ 1 – viết kiểu do...while

- Mã C:

```
  // add the numbers from 0 to 9
int sum = 0;
int i;
for (i=0; i!=10; i++) {
  sum = sum + i;
}
```

- Mã hợp ngữ RISC-V:

```
# s0 = i, s1 = sum
      addi   s1, zero, 0    # sum = 0
      addi   s0, zero, 0    # i = 0
      addi   t0, zero, 10   # t0 = 10
for:  add    s1, s1, s0     # sum = sum+i
      addi   s0, s0, 1      # tăng i thêm 1
      bne    s0, t0, for    # i<10 quay lại for
done: ...
```

Vòng lặp for – ví dụ 2

- Mã C

```
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

Ví dụ 2 cho vòng lặp for

- Mã hợp ngữ RISC-V

```
# s0 = i, s1 = sum
        addi s1, zero, 0        # sum = 0
        addi s0, zero, 1        # i = 1
        addi t0, zero, 101      # t0 = 101
loop:   bge  s0, t0, done       # Nếu i>= 101, thì thoát
        add  s1, s1, s0         # nếu i<101 thì sum=sum+i
        slli s0, s0, 1          # i= 2*i
        beq x0, x0, loop        # lặp lại
done:
```

3.5. Lập trình mảng dữ liệu

- Truy cập số lượng lớn các dữ liệu cùng loại
- Chỉ số (Index): truy cập từng phần tử của mảng
- Kích thước mảng (Size): số phần tử của mảng

Truy cập các phần tử 32-bit của mảng với chỉ số cố định

Nạp phần tử B[6] vào thanh ghi t3

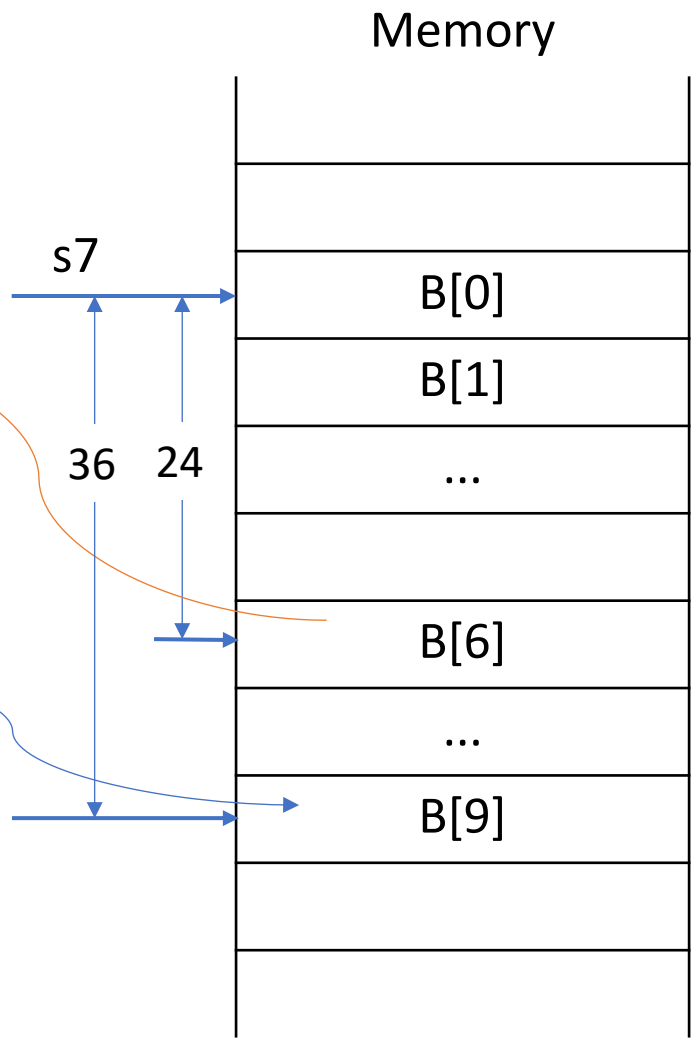
```
lw      t3, 24(s7)      #  t3 = B[6]
```

t3 

Cất nội dung t4 ra phần tử B[9]

```
sw      t4, 36(s7)      # B[9]= t4
```

t4 



Truy cập các phần tử 32-bit của mảng với chỉ số dạng biến

Nạp A[i] vào thanh ghi t5: (s3 = i)

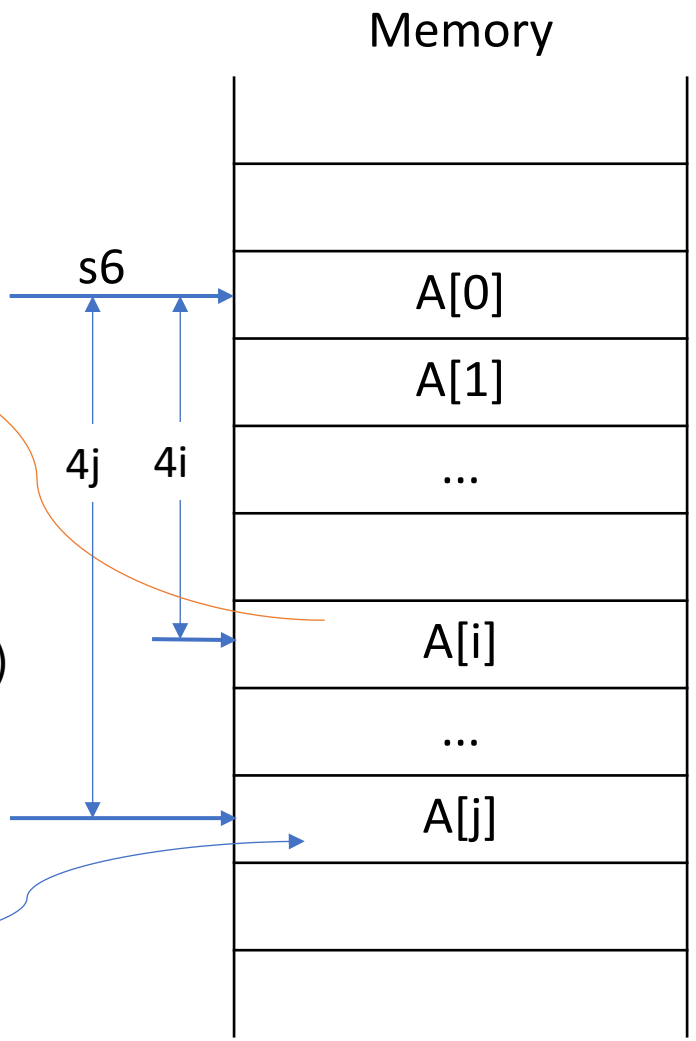
```
slli    t0, s3, 2       # t0 = 4*i
add     t0, s6, t0      # t0 = địa chỉ của  A[i]
lw      t5, 0(t0)       # t5 = A[i]
```

t5

Cất nội dung thanh ghi t6 ra phần từ A[j] (s4 = j)

```
slli    t1, s4, 2       # t1 = 4*j
add     t1, s6, t1      # t1 = địa chỉ của  A[j]
sw      t6, 0(t1)       # A[j]= t6
```

t6



Ví dụ vòng lặp truy cập mảng dữ liệu

- Mã C

```
int M[200];
int i;

for (i=0; i < 200; i = i + 1)
    M[i] = M[i] + 5;

// giả sử địa chỉ cơ sở của mảng =
  0x10023400
```

Ví dụ vòng lặp truy cập mảng dữ liệu (tiếp)

Mã hợp ngữ RISC-V

```
# s0 = array base address (0x10023400), s1 = i
# khởi tạo các thanh ghi
      lui  s0, 0x10023      # s0 = 0x10023000
      addi s0, s0, 0x400    # s0 = 0x10023400
      addi s1, zero, 0      # i = 0
      addi t2, zero, 200    # t2 = 200
# vòng lặp
loop: bge  s1, t2, done     # i >= 200 then done
      slli t0, s1, 2        # t0 = i*4
      add  t0, t0, s0       # address of M[i]
      lw   t1, 0(t0)        # t1 = M[i]
      addi t1, t1, 5        # t1 = M[i]+5
      sw   t1, 0(t0)        # M[i] = M[i]+5
      addi s1, s1, 1        # i = i + 1
      beq  x0, x0, loop     # repeat
done:
```

Truy cập byte và ký tự

- Các tập ký tự được mã hóa theo byte
  - ASCII: 128 ký tự
    - 95 ký thị hiển thị , 33 mã điều khiển
  - Latin-1: 256 ký tự
    - ASCII và các ký tự mở rộng

Bộ mã ASCII

- Do ANSI (American National Standard Institute) thiết kế
- Bộ mã 8-bit → có thể mã hóa được 2^8 ký tự, có mã từ: $00_{16} \div FF_{16}$, trong đó:
  - 128 ký tự chuẩn của ASCII có mã từ $00_{16} \div 7F_{16}$
  - 128 ký tự mở rộng có mã từ $80_{16} \div FF_{16}$

Các ký tự chuẩn

- Các ký tự hiển thị chuẩn có mã từ $20_{(16)} \div 7E_{(16)}$
  - Các chữ cái Latin: 'A' – 'Z' ; 'a' – 'z'
  - Các chữ số thập phân: '0' – '9'
  - các dấu câu: . , : ; ...
  - các dấu phép toán: + - * / % ...
  - một số ký hiệu thông dụng: &, $, @, #
  - dấu cách
- Các mã điều khiển có mã từ $00_{(16)} \div 1F_{(16)}$ và $7F_{(16)}$

Mã ASCII của các ký tự hiển thị chuẩn

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Các mã điều khiển

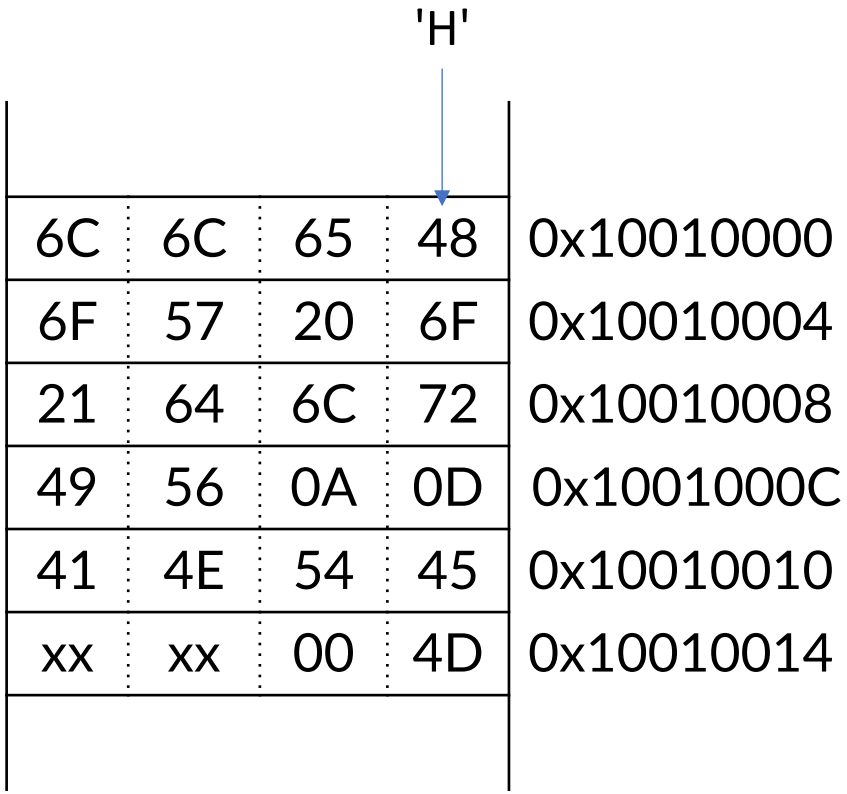
Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of TeXt	12	DC2	Device Control 2
3	ETX	End Of TeXt	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELl	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

Ví dụ:

- CR - mã điều khiển chuyển con trỏ về đầu dòng
- LF - mã điều khiển chuyển con trỏ xuống dòng tiếp theo
- BS - mã điều khiển lùi trái con trỏ một vị trí
- FF - mã điều khiển chuyển con trỏ sang đầu trang tiếp theo

Ví dụ mã hóa text

' Hello World ! (↵)
VIETNAM '



Các lệnh thao tác với byte

- RISC-V có các lệnh để Nạp/Lưu từng byte:

`lb rd,imm(rs1)`

  - Nạp 1 byte từ bộ nhớ vào bên phải thanh ghi đích rd
  - Phần còn lại của thanh ghi rd được mở rộng có dấu thành 32-bit (Sign-extended)

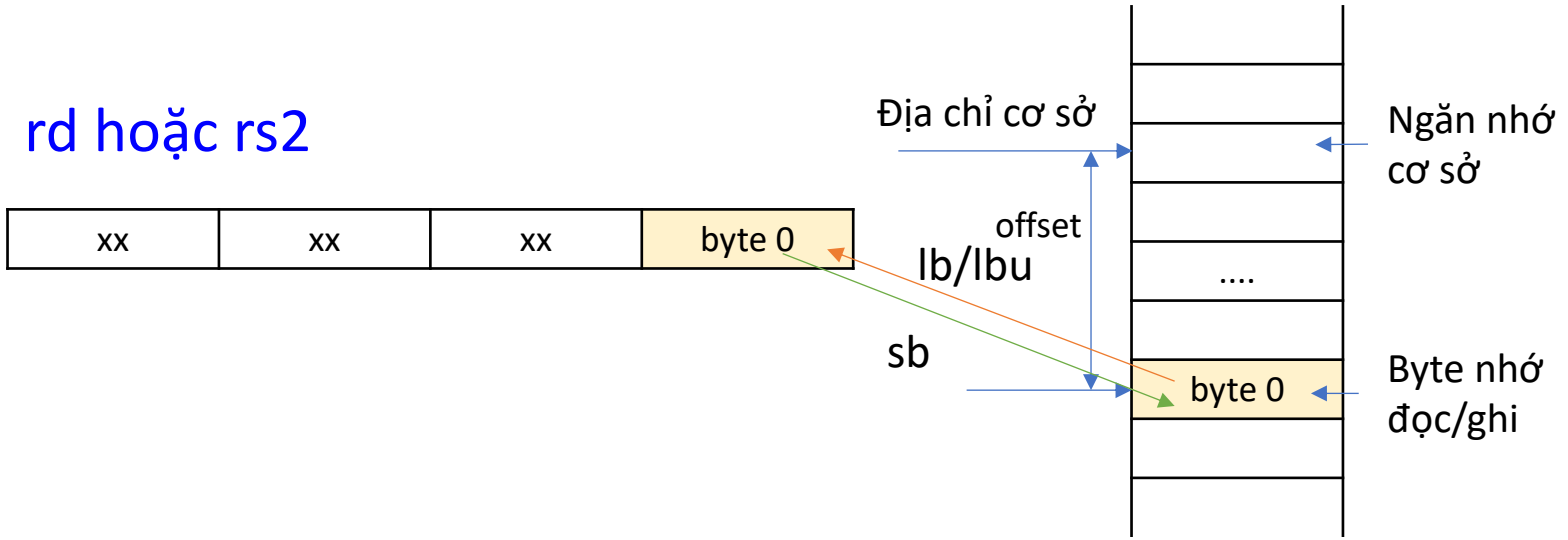
`lbu rd,imm(rs1)`

  - Nạp 1 byte từ bộ nhớ vào bên phải thanh ghi đích rd
  - Phần còn lại của thanh ghi rd được mở rộng không dấu thành 32-bit (Zero-extended)

`sb rs2,imm(rs1)`

  - Chỉ lưu byte bên phải thanh ghi rs2 ra bộ nhớ

Lệnh lb (load byte), lbu và lệnh sb (store byte)




```
lb    rd, imm(rs1)      # nạp 1 byte từ bộ nhớ vào thanh ghi
                        # 3 byte bên trái được mở rộng theo số có dấu
lbu   rd, imm(rs1)      # nạp 1 byte từ bộ nhớ vào thanh ghi
                        # 3 byte bên trái được mở rộng theo số không dấu
sb    rs2, imm(rs1)     # cất 1 byte bên phải của  thanh ghi ra bộ nhớ
```

(Nội dung rs1 và imm có thể chẵn hay lẻ)

Các lệnh thao tác với halfword

`lh rd,imm(rs1)`

- Nạp 2 byte (halfword) từ bộ nhớ vào bên phải thanh ghi đích rd
- Phần còn lại của thanh ghi rd được mở rộng có dấu thành 32-bit (Sign-extended)

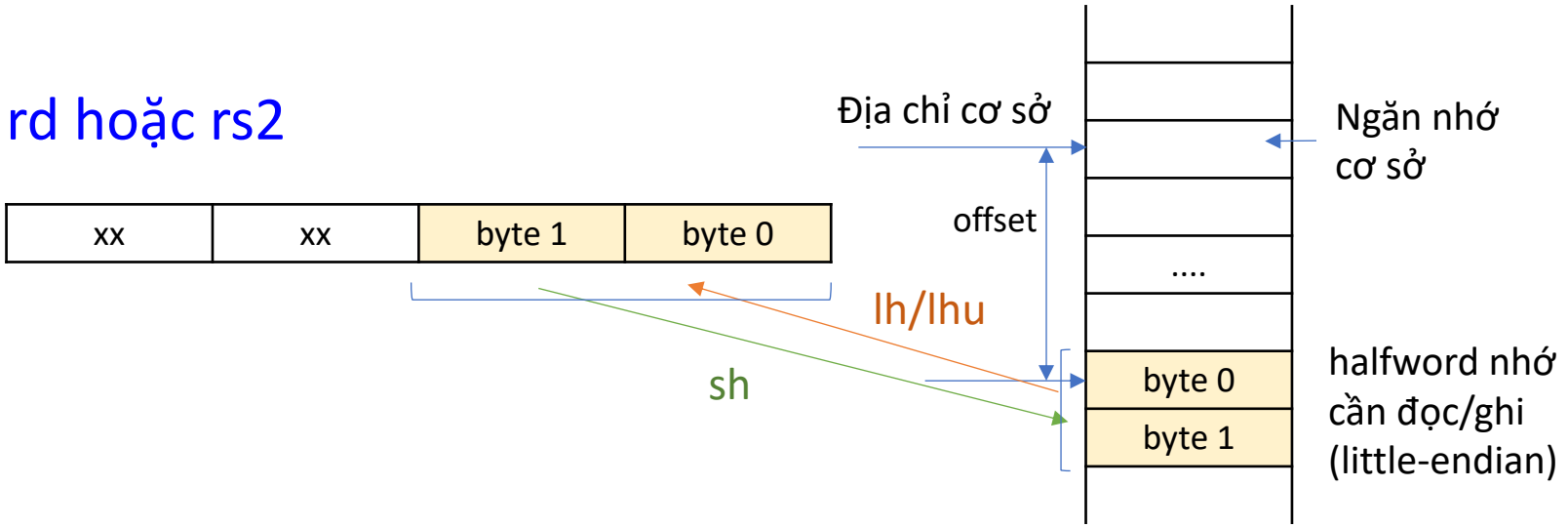
`lhu rd,imm(rs1)`

- Nạp 2 byte (halfword) từ bộ nhớ vào bên phải thanh ghi đích rd
- Phần còn lại của thanh ghi rd được mở rộng không dấu thành 32-bit (Zero-extended)

`sh rs2,imm(rs1)`

- Lưu 2 byte bên phải thanh ghi rs2 ra bộ nhớ

Lệnh lh (load halfword), lhu và lệnh sh (store halfword)




```
lh    rd, imm(rs1)      # nạp 2 byte từ bộ nhớ vào thanh ghi
                        # 2 byte bên trái được mở rộng theo số có dấu
lhu   rd, imm(rs1)      # nạp 2 byte từ bộ nhớ vào thanh ghi
                        # 2 byte bên trái được mở rộng theo số không dấu
sh    rs2, imm(rs1)     # cất 2 byte bên phải của  thanh ghi ra bộ nhớ
```

(Nội dung rs1 và imm phải là chẵn)

Ví dụ

```
# s6=0x10010040
lb   s1, 1(s6)
lb   s2, 7(s6)
lbu  s3, 7(s6)
lh   s4, 2(s6)
lh   s5, 6(s6)
```

little-endian

4C	6D	65	48	0x10010040
BE	9A	20	6F	0x10010044
21	64	6C	72	0x10010048
49	56	0A	0D	0x1001004C
41	4E	54	45	0x10010050
xx	xx	00	4D	0x10010054

Ví dụ truy nhập mảng ký tự

- Mã C:

// chuyển đổi mảng chữ cái thường thành chữ cái hoa

```
char CharArray[10] = "abcdefghij";
int i;
for (i = 0; i < 10; ++i)
    CharArray[i] = CharArray[i] - 32;
printf("%s", CharArray);
```

Ví dụ truy nhập mảng ký tự

- Mã hợp ngữ RISC-V:

```
. data
        CharArray: .ascii   "abcdefghij"
.text
# s0 = base address of CharArray, s1 = i
        addi   s1, zero, 0    # i = 0
        addi   t3, zero, 10   # t3 = 10
for:    bge    s1, t3, done   # i >= 10 ? thoát
        add    t4, s0, s1     # t4 = địa chỉ CharArray[i]
        lb     t5, 0(t4)      # t5 = CharArray[i]
        addi   t5, t5, -32    # t5 = CharArray[i]-32(&0xDF)
        sb     t5, 0(t4)      # CharArray[i] = t5
        addi   s1, s1, 1      # i = i + 1
        j      for            # lặp lại
done:
```

3.6. Chương trình con - thủ tục

- Các bước yêu cầu:
  1. Đặt các tham số vào các thanh ghi a0-a7 (x10-x17)
  2. Chuyển điều khiển đến thủ tục
  3. Dành bộ nhớ cho thủ tục
  4. Thực hiện các thao tác của thủ tục
  5. Đặt kết quả vào thanh ghi cho chương trình đã gọi thủ tục
  6. Trở về vị trí đã gọi

Các thanh ghi liên quan

- x10-x17 (a0–a7) : các thanh ghi để truyền tham số vào hoặc trả kết quả ra
- x1 (ra - return address) : thanh ghi chứa địa chỉ trở về
- x2 (sp - stack pointer): con trỏ ngăn xếp
- x3 (gp - global pointer): con trỏ toàn cục
- x4 (tp - thread pointer): con trỏ luồng

Các lệnh liên quan với thủ tục

- Gọi thủ tục: sử dụng lệnh jal (jump-and-link)

```
jal ra, ProcedureAddress
```

  - Địa chỉ của lệnh kế tiếp (địa chỉ trở về) được cất sang `ra`
  - Nhảy đến địa chỉ của thủ tục: nạp vào PC địa chỉ của lệnh đầu tiên của Thủ tục được gọi
  - Lệnh nhảy không điều kiện:

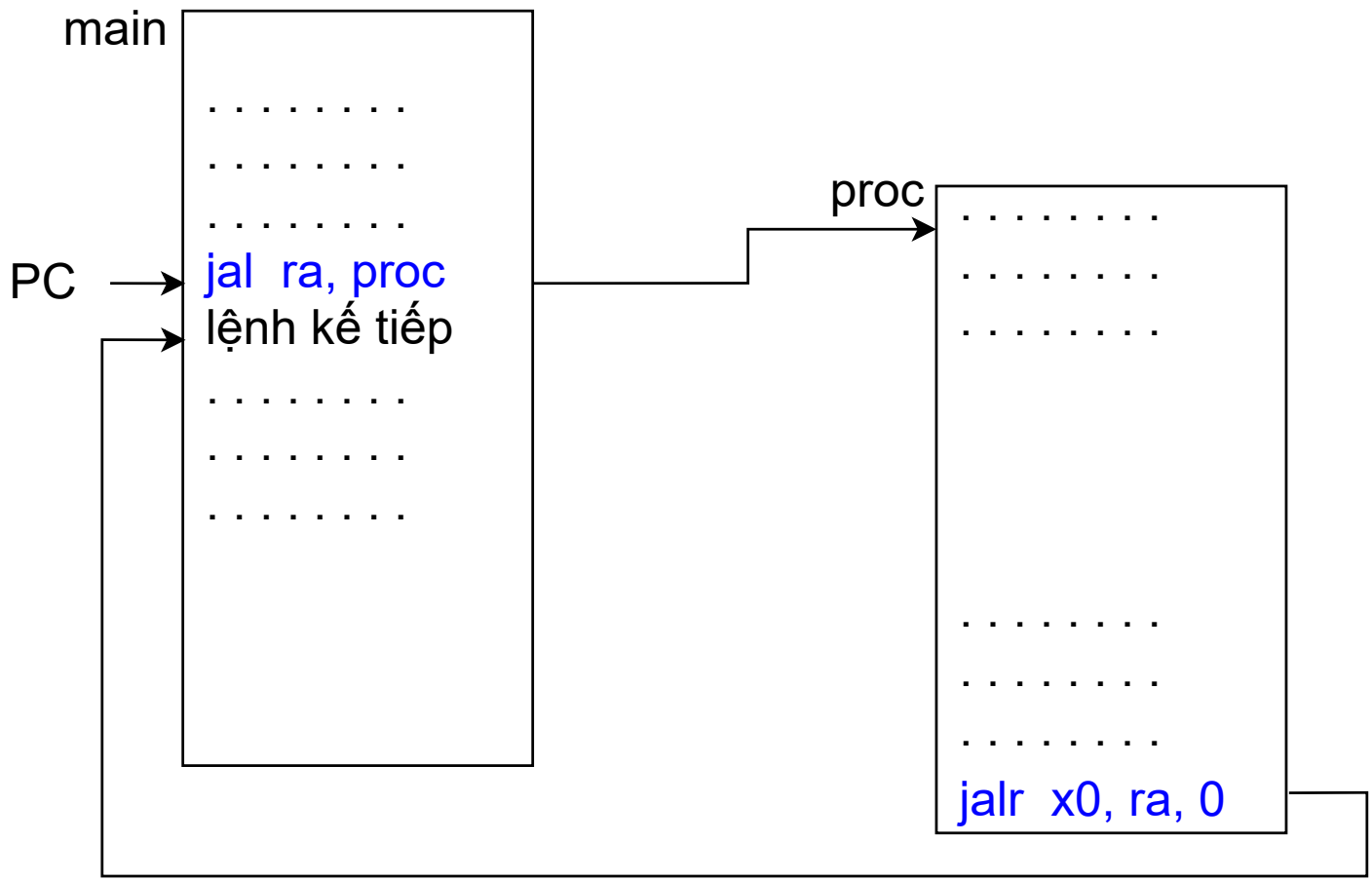
```
jal   x0, Label
```

- Trở về từ thủ tục: sử dụng lệnh jalr (jump-and-link register)

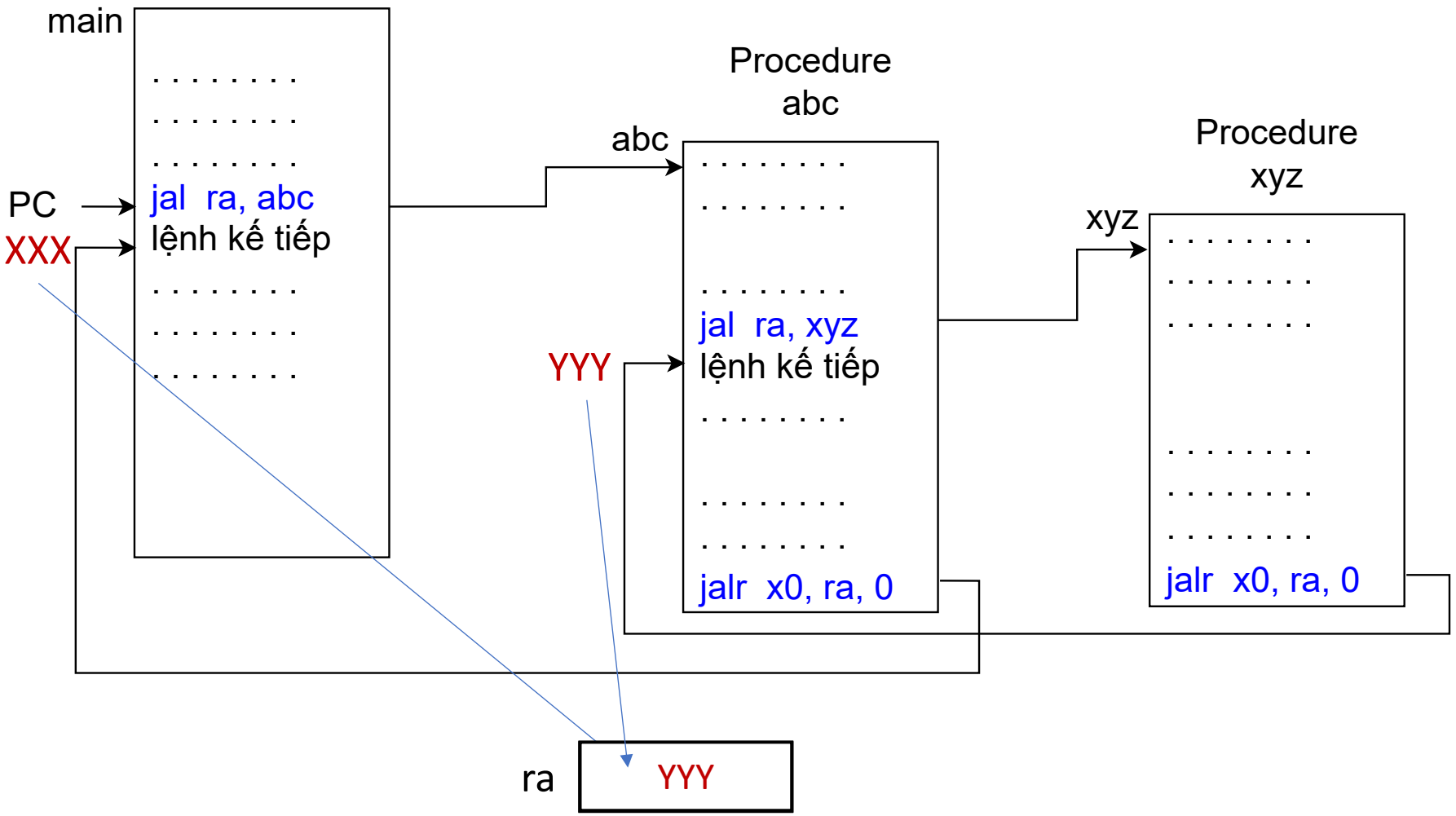
```
jalr zero, ra, 0
```
hoặc
```
jalr x0, x1, 0
```

  - Nhảy đến lệnh có địa chỉ đã được lưu ở thanh ghi `ra`

Minh họa gọi Thủ tục



Gọi thủ tục lồng nhau



Ví dụ Thủ tục lá

- Thủ tục lá là thủ tục không có lời gọi thủ tục khác
- Mã C:

```
int leaf_example (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

  - Các tham số g, h, i, j ở a0, a1, a2, a3
  - f ở s4 (do đó, cần cất s4 ra ngăn xếp)
  - t0 và t1 được thủ tục dùng để chứa các giá trị tạm thời, cũng cần cất trước khi sử dụng
  - Kết quả ở a0

Mã hợp ngữ RISC-V

```
leaf_example:
    addi   sp, sp, -12    # tạo 3 chỗ ở stack
    sw     t0, 8(sp)      # cất nội dung t0
    sw     t1, 4(sp)      # cất nội dung t1
    sw     s4, 0(sp)      # cất nội dung s4
    add    t0, a0, a1     # t0 = g+h
    add    t1, a2, a3     # t1 = i+j
    sub    s4, t0, t1     # f = (g+h)-(i+j)
    add    a0, s4, x0     # trả kết quả sang a0
    lw     s4, 0(sp)      # khôi phục s4
    lw     t1, 4(sp)      # khôi phục t1
    lw     t0, 8(sp)      # khôi phục t0
    addi   sp, sp, 12     # xóa 3 mục ở stack
    jalr   x0, ra, 0      # trở về nơi đã gọi
```

Ví dụ Thủ tục cành

- Là thủ tục có gọi thủ tục khác
- Mã C:

```
int fact (int n)
{
   if (n < 1) return (1);
   else return n * fact(n - 1);
}
```

  - Tham số n ở a0
  - Kết quả trả ra ở a1

Mã hợp ngữ RISC-V

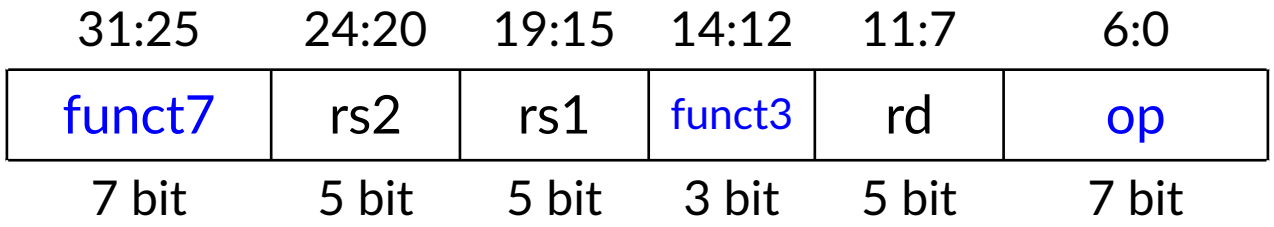
```
fact:
        addi   sp, sp, -8      # dành 2 chỗ ở stack
        sw     ra, 4(sp)       # cất địa chỉ trở về
        sw     a0, 0(sp)       # cất n
        addi   t0, x0, 1       # t0 = 1
        bgt    a0, t0, L1      # nếu n > 1, rẽ tới L1
    #   blt    t0, a0, L1      # chính là lệnh này
        addi   a1, x0, 1       # nếu không, kết quả là 1
        addi   sp, sp, 8       # khôi phục sp
        jalr   x0, ra, 0       # và trở về
L1:     addi   a0, a0, -1      # nếu n > 1 thì n = n-1
        jal    ra, fact        # gọi fact với (n-1)
YYY ──► lw     a0, 0(sp)       # khôi phục n
        lw     ra, 4(sp)       # khôi phục địa chỉ trở về
        addi   sp, sp, 8       # giải phóng 2 mục khỏi stack
        mul    a1, a0, a1      # a1 = n*(n-1)!
        jalr   x0, ra, 0       # trở về
```

3.7. Mã máy

- Các lệnh được mã hóa dưới dạng nhị phân được gọi là mã máy
- Bộ xử lý chỉ hiểu 0 và 1
- Tập lệnh của RISC-V:
  - Các lệnh được mã hóa 32-bit
  - Mỗi lệnh chiếm 4-byte trong bộ nhớ, do vậy địa chỉ của lệnh trong bộ nhớ là bội của 4
  - 4 kiểu dạng lệnh:
    - Kiểu R
    - Kiểu I
    - Kiểu S/B
    - Kiểu U/J

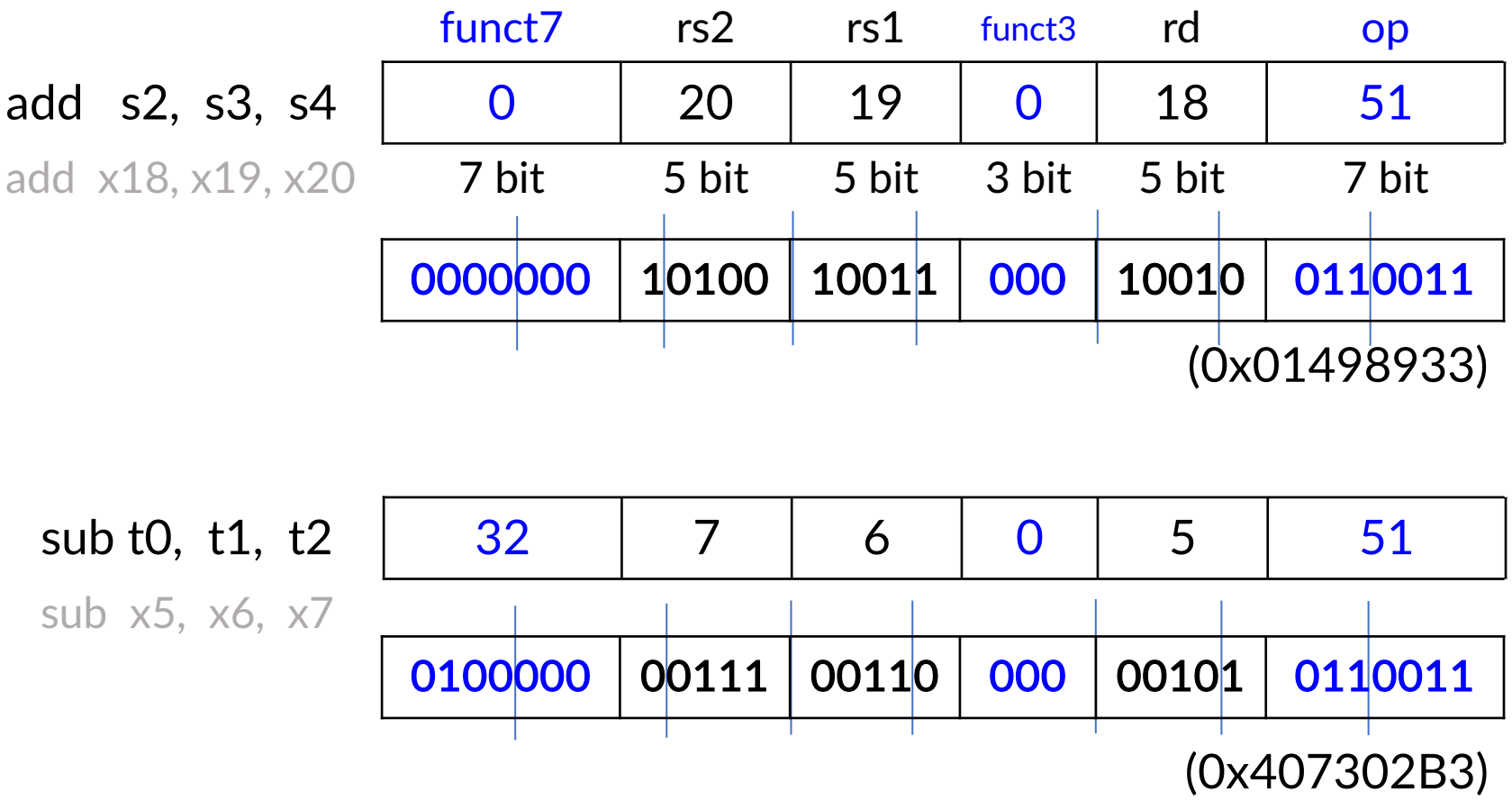
Kiểu R (Register-type)

- Kiểu thanh ghi



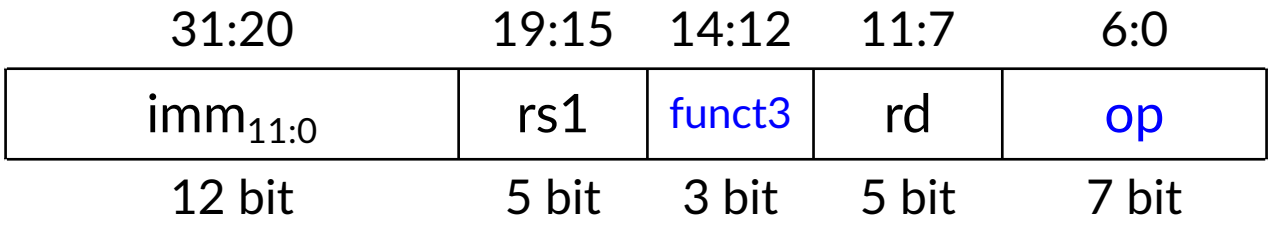
- 3 toán hạng thanh ghi
  - rs1, rs2: hai thanh ghi nguồn (source registers)
  - rd: thanh ghi đích (destination register)
- Các trường khác:
  - op: mã thao tác (opcode) = 0110011 = 0x33 = $51_{(10)}$
  - funct7, funct3: các trường function, cùng với opcode chỉ ra thao tác cần thực hiện

Ví dụ lệnh kiểu R



Kiểu I (Immediate-type)

- Kiểu tức thì

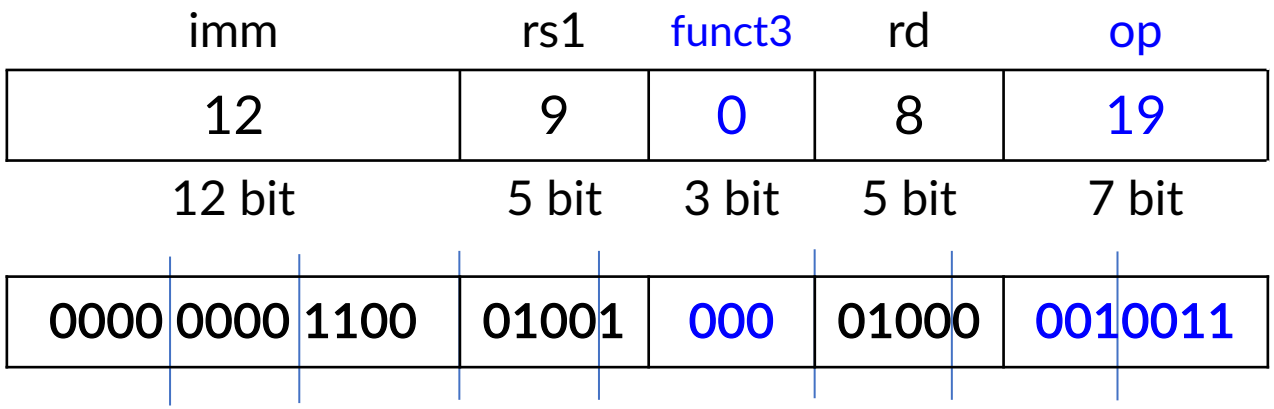


- 3 toán hạng:
  - rs1: toán hạng thanh ghi nguồn
  - rd: toán hạng thanh ghi đích
  - imm: hằng số 12-bit số nguyên có dấu theo mã bù hai
- Các trường khác:
  - op: mã thao tác
  - funct3: trường function, cùng với opcode chỉ ra thao tác cần thực hiện

Ví dụ lệnh kiểu I

addi s0, s1, 12

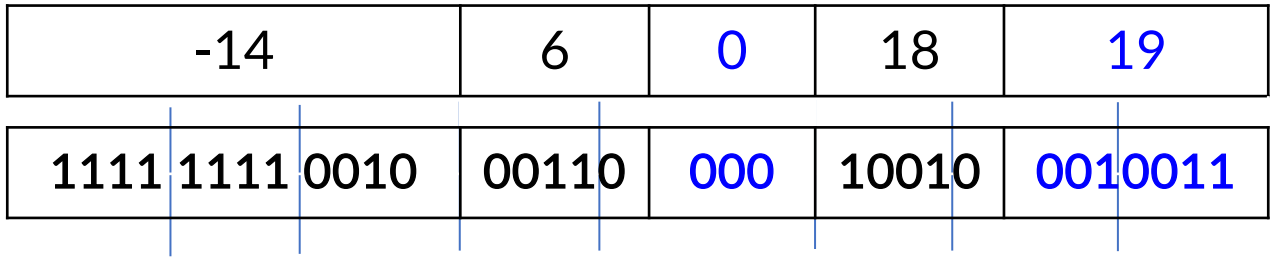
addi x8, x9, 12



(0x00C48413)

addi s2, t1, -14

addi x18, x6, -14

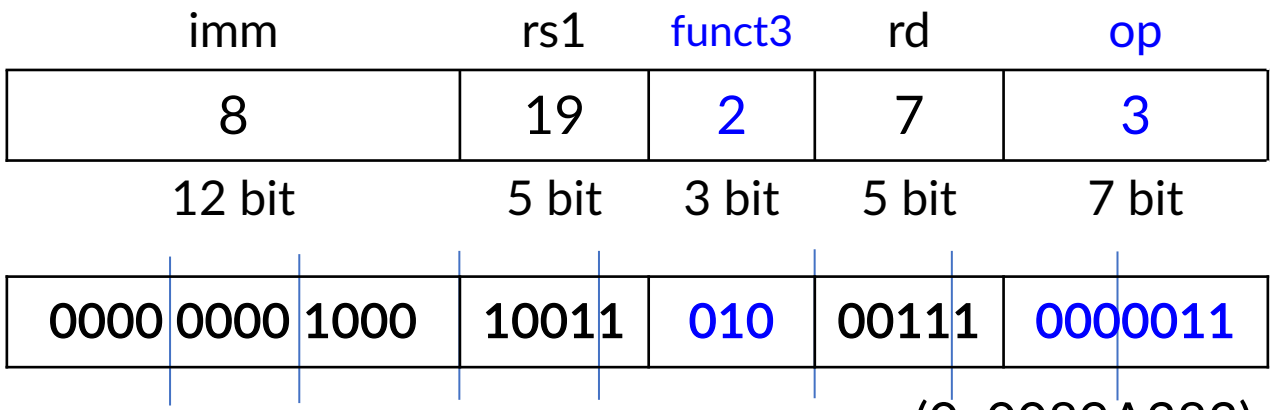


(0xFF230913)

Ví dụ lệnh kiểu I

lw t2, 8(s3)

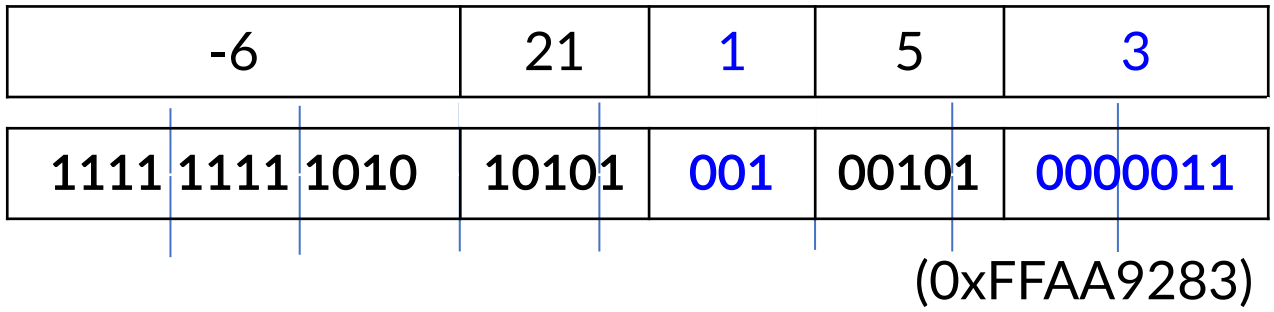
lw x7, 8(x19)



(0x0089A383)

lh t0, -6(s5)

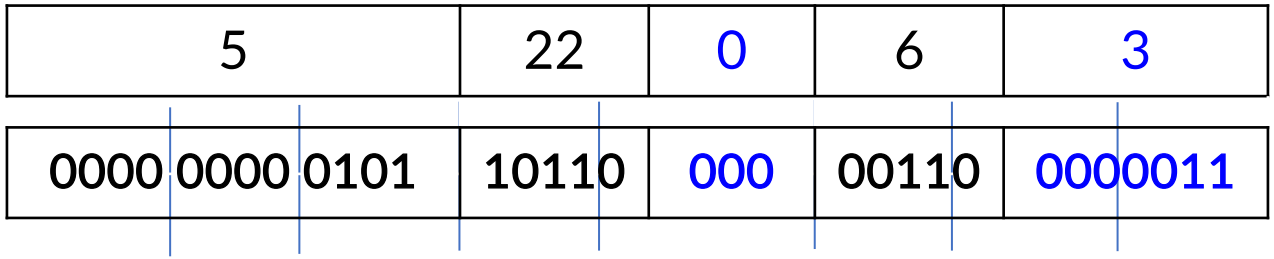
lh x5, -6(x21)



(0xFFAA9283)

lb t1, 5(s6)

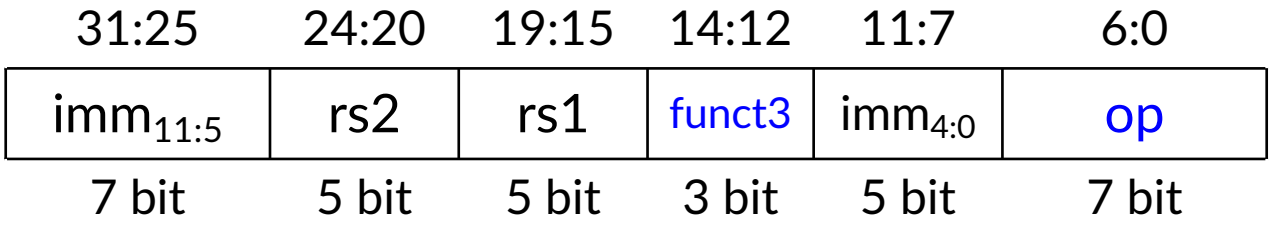
lh x6, 5(x22)



(0x005B0303)

Kiểu S (Store-Type)

- Các lệnh store: sw, sh, sb



- 3 toán hạng:
  - rs1: thanh ghi cơ sở
  - rs2: thanh ghi chứa giá trị sẽ được ghi đến bộ nhớ
  - imm: hằng số 12-bit số nguyên có dấu theo mã bù hai
- Các trường khác:
  - op: mã thao tác = 0100011 = 0x23 = $35_{(10)}$
  - funct3: trường function

Ví dụ lệnh máy kiểu S

-8= 1111 1111 1000

sw s0, -8(s6)

sw x8, -8(x22)

$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
1111 111	8	22	2	11000	35
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
1111 111	01000	10110	010	11000	0100011

(0xFE8B2C23)

14= 0000 0000 1110

sh s1, 14(s7)

sh x9, 14(x23)

0000 000	9	23	1	01110	35
0000 000	01001	10111	001	01110	0100011

(0x009B9723)

7= 0000 0000 0111

sb s2, 7(s8)

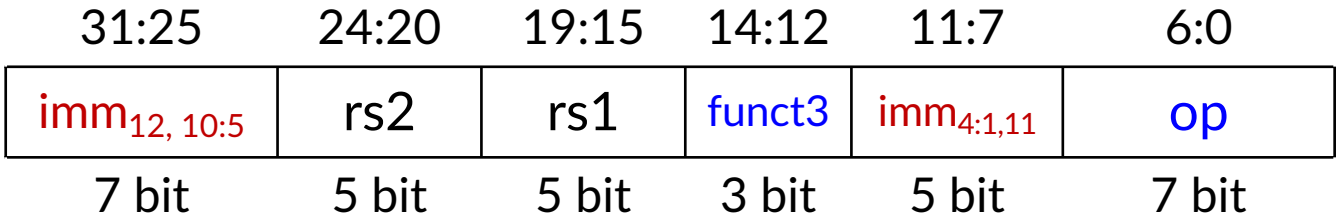
sb x18, 7(x24)

0000 000	18	24	0	00111	35
0000 000	10010	11000	000	00111	0100011

(0x012C03A3)

Kiểu B (Branch-Type)

- Các lệnh rẽ nhánh: beq, bne, blt, bge, bltu, bgeu



- 3 toán hạng:
  - rs1, rs2: thanh ghi nguồn 1 và 2
  - $imm_{12:1}$: 12-bit của hằng số imm số nguyên có dấu theo mã bù hai
- Các trường khác:
  - op: mã thao tác = 1100011 = 0x63 = $99_{(10)}$
  - funct3:

Lệnh	beq	bne	blt	bge	bltu	bgeu
funct3	000	001	100	101	110	111

Ví dụ lệnh máy kiểu B

```
# Địa chỉ              Hợp ngữ
0x00401050          beq s0, t5, L1
0x00401054          ...
...                 ...
...                 ...
0x00401060     L1:  addi s1, s1, 16
```

imm = 0x00400060 - 0x00400050 = 0x10 = 16

$imm_{12:0}$ = 16 = 0 0 000000 1000 0

vị trí bit 12 11 10:5 4:1 0

beq s0, t5, L1

beq x8, x30, 16

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
0 000000	30	8	0	1000 0	99
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
0 000000	11110	01000	000	1000 0	1100011

(0x01E40863)

Ví dụ lệnh máy kiểu B

```
 Địa chỉ              Hợp ngữ
0x00400070     L2: add s0, s1, s2
0x00400074         ...
...                ...
...                ...
0x0040008C         bne t0, s0, L2
```

imm = -(0x0040008C - 0x00400070) = - 0x1C = -28

$imm_{12:0}$ = 28 = 1 1 111111 0010 0

vị trí bit 12 11 10:5 4:1 0

bne t0, s0, L2

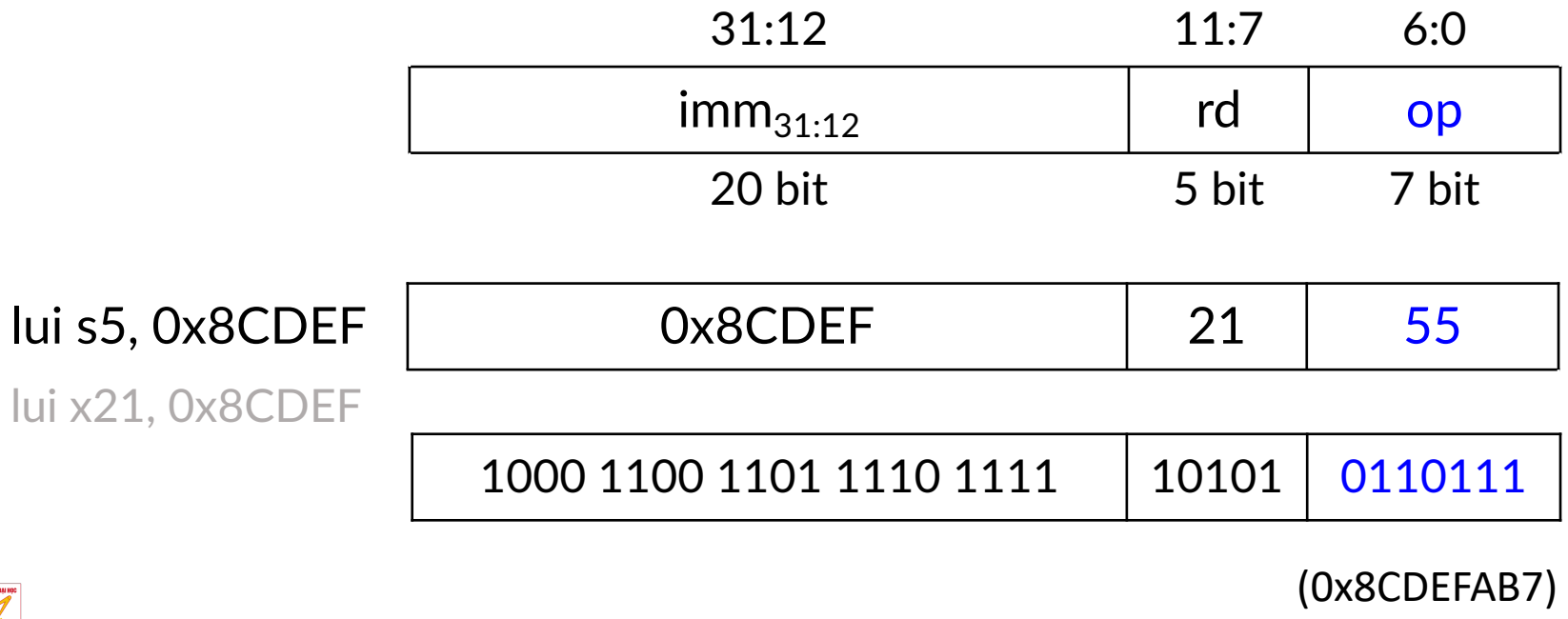
bne x5, x8, 28

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
1 111111	8	5	1	0010 1	99
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
1 111111	01000	00101	001	0010 1	1100011

(0xFE8292E3)

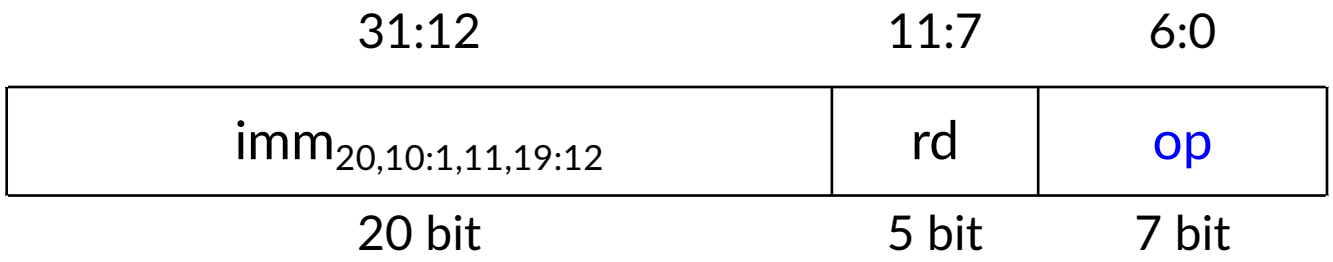
Kiểu U (Upper Immediate -Type)

- Sử dụng cho lệnh `lui`
- 2 toán hạng:
  - rd: thanh ghi đích
  - $imm_{31:12}$: 20-bit cao của hằng số imm 32-bit
- op: mã thao tác = 0110111 = 0x37 = $55_{(10)}$



Kiểu J (Jump -Type)

- Sử dụng cho lệnh `jal`
- 2 toán hạng:
  - rd: thanh ghi đích
  - $imm_{20,10:1,11,19:12}$: 20-bit (20:1) của hằng số imm 21-bit
- op: mã thao tác = 1101111 = 0x6F = $111_{(10)}$



- Chú ý: Lệnh jalr là lệnh kiểu I

Ví dụ lệnh máy kiểu J

```
# Địa chỉ                Hợp ngữ
0x00400050             jal ra, funct1
0x00400054             ...
...                    ...
...                    ...
0x004029A8  funct1:    addi s1, s1, 16
```

imm = 0x004029A8 - 0x00400050 = 0x02958

$imm_{20:0}$ = 0 00000010 1 0010101100 0

vị trí bit 20 19:12 11 10:1 0

	$imm_{20,10:1,11,19:12}$	rd	op
jal ra, funct1	0 0010101100 1 00000010	1	$111_{(10)}$
jal x1, 0x02958			
	0 0010101100 1 00000010	00001	1101111

(0x159020EF)

Ví dụ lệnh máy kiểu J

```
# Địa chỉ                Hợp ngữ
0x00403074     L1: addi s1, x0, 2
0x00403078         ...
...                ...
...                ...
0x004030AC         jal x0, L1
```

imm = -(0x004030AC - 0x00403074) = -0x0038 = -56 =0xFFFC8

$imm_{20:0}$ = 1 1111 1111 1 111 1100 100 0

vị trí bit 20 19:12 11 10:1 0

	$imm_{20,10:1,11,19:12}$	rd	op
jal x0, L1	1 1111100100 1 11111111	0	$111_{(10)}$
jal x0, 0xFFFC8			
	1 1111100100 1 11111111	00000	1101111

(0xFC9FF06F)

Tổng kết các kiểu lệnh máy

7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type
20 bit				5 bit	7 bit	

Dịch mã máy về hợp ngữ

- Viết lệnh máy dưới dạng nhị phân
- Bắt đầu với opcode (và funct3): sẽ chỉ ra các phần còn lại
- Phân tách các trường
- Các trường op, funct3, và funct7 chỉ ra thao tác

Ví dụ dịch mã máy về dạng hợp ngữ

```
0x41FE83B3 = 0100 0001 1111 1110 1000 0011 1011 0011
             op = 51, funct3 = 0: add or sub (R-type)
             funct7 = 0100000 : sub
```

funct7	rs2	rs1	funct3	rd	op
0100 000	11111	11101	000	00111	011 0011
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit
32	31	29	0	7	51

Lệnh hợp ngữ: `sub x7, x29, x31`

hoặc `sub t2, t4, t6`

Ví dụ dịch mã máy về dạng hợp ngữ (2)

```
0xFDA48393 = 1111 1101 1010 0100 1000 0011 1001 0011
             op = 19, funct3 = 0: addi (I-type)
```

funct7	rs1	funct3	rd	op
1111 1101 1010	01001	000	00111	001 0011
12 bit	5 bit	3 bit	5 bit	7 bit
-38	9	0	7	19

Lệnh hợp ngữ: `addi x7, x9, -38`

hoặc `addi t2, s1, -38`

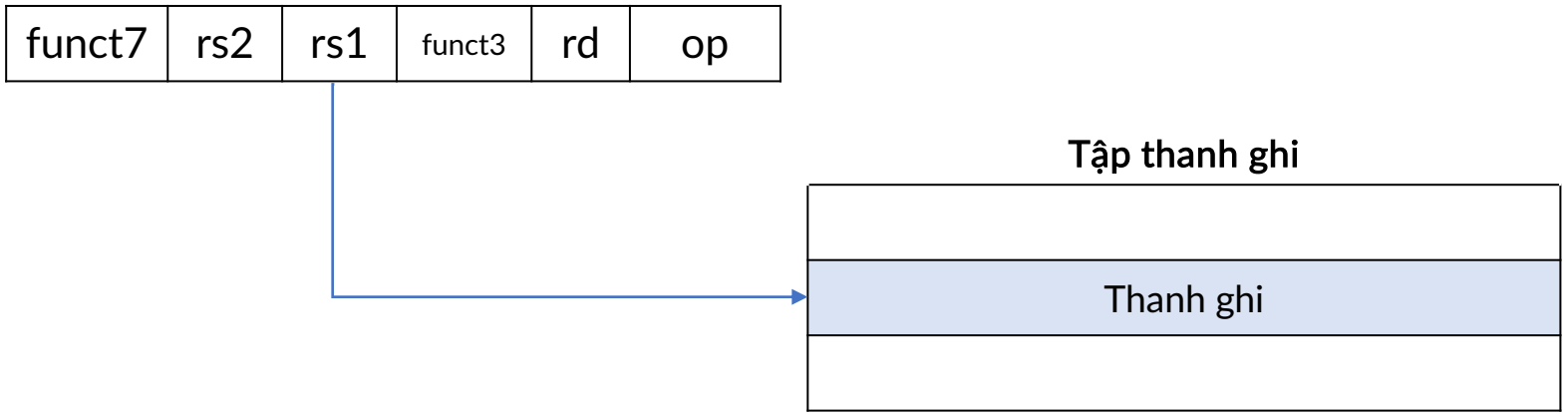
Các phương thức định địa chỉ

- Phương thức định địa chỉ (Addressing modes) là cách mã hóa trong lệnh để:
  - xác định nơi đọc/ghi toán hạng, hoặc
  - xác định địa chỉ của lệnh tiếp theo.

- RISC-V có 4 phương thức định địa chỉ
  - Định địa chỉ thanh ghi (Register Addressing)
  - Định địa chỉ tức thì (Immediate Addressing)
  - Định địa chỉ cơ sở (Base Addressing)
  - Định địa chỉ tương đối với PC (PC-relative Addressing)

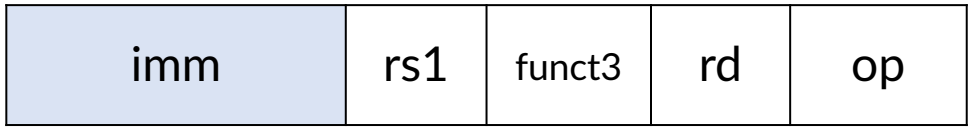
Định địa chỉ thanh ghi

- Các toán hạng nằm ở thanh ghi
- Tất cả các lệnh kiểu R sử dụng phương thức này
- Ví dụ:
  - `add s0, t2, t3`
  - `sub t8, s1, s0`



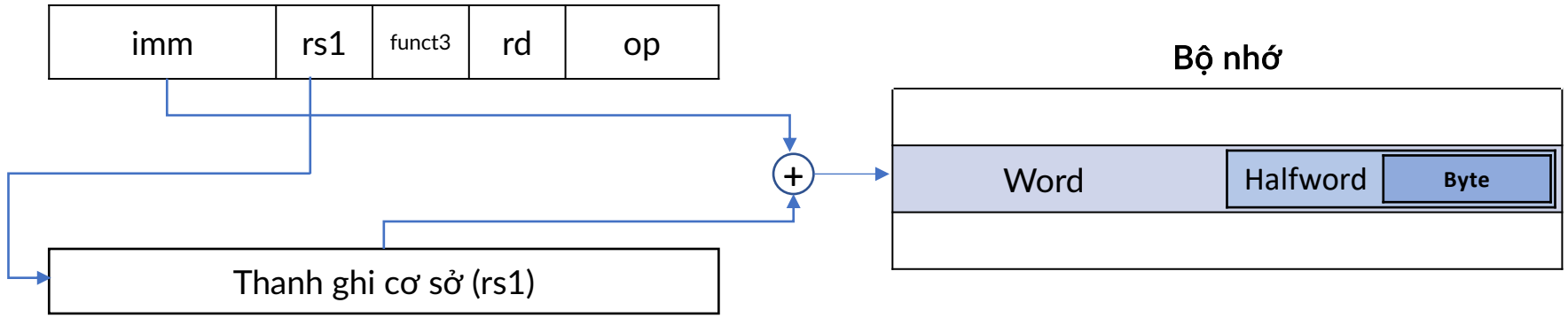
Định địa chỉ tức thì

- Toán hạng là hằng số imm (12-bit) trong lệnh
- Ví dụ:
  - `addi s3, t5, -20`
  - `ori  s4, t7, 0xFF`



Định địa chỉ cơ sở

- Toán hạng nằm ở bộ nhớ
- Các lệnh Load và Store
- Địa chỉ toán hạng = Địa chỉ cơ sở + imm
- Ví dụ:
  - `lw  s4, 12(s6)`
    - Địa chỉ = `(s6) + 12`
  - `sw  t2, -20(s7)`
    - Địa chỉ = `(s7) - 20`



Định địa chỉ tương đối với PC

- Xác định địa chỉ của lệnh tiếp theo
- Sử dụng cho các lệnh branch và lệnh `jal`

- Các lệnh branch: Khi điều kiện đúng thì rẽ nhánh tới thực hiện lệnh ở Địa chỉ đích rẽ nhánh.

  Địa chỉ đích rẽ nhánh = **PC + SignExt({$imm_{12:1}$,1'b0})**

  (PC chứa địa chỉ của lệnh branch)

- Lệnh `jal`: Nhảy tới thực hiện lệnh ở Địa chỉ đích

  Địa chỉ đích = PC + SignExt({$imm_{20:1}$,1'b0})

  (PC chứa địa chỉ của lệnh jal)

Ví dụ tìm địa chỉ đích của lệnh branch

- Giả sử có lệnh mã máy ở địa chỉ 0x00401020 là: 0x01338C63. Xác định địa chỉ đích ?

  0x01338C63 =0000 0001 0011 0011 1000 1100 0110 0011

  op = 110 0011 và funct3 = 000 → lệnh beq

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	Lệnh hợp ngữ
0 000000	1 0011	0011 1	000	1100 0	110 0011	beq x7, x19, L1
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	beq t2, s3, L1

Địa chỉ đích = 0x00401020 + 0 0 000000 1100 $0_{(2)}$

= 0x00401020 + 0x018 = 0x00401038

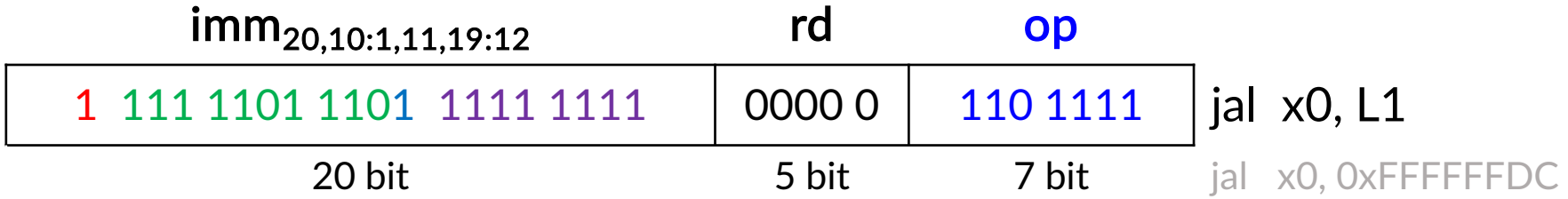
```
0x00401020          beq   t2, s3, L1
0x00400024          ...
...                 ...
0x00401038  L1:     <lệnh tiếp theo>
```

Ví dụ tìm địa chỉ đích của lệnh jal

- Giả sử có lệnh mã máy ở địa chỉ 0x004018C4 là: 0xFDDFF06F. Xác định địa chỉ đích ?

0xFDDFF06F =1111 1101 1101 1111 1111 0000 0110 1111

op = 110 1111 → lệnh jal



Địa chỉ đích = 0x004018C4 + 1 1111 1111 1 111 1101 110 $0_{(2)}$

= 0x004018C4 + 0xFFFFFFDC = 0x004018A0

```
0x004018A0    L1:   ...
0x004018A4          ...
...                 ...
0x004018C4          jal x0, L1
```

3.8. Một số lệnh khác

- Các lệnh so sánh
- Các lệnh nhân chia
- Các lệnh số dấu phẩy động
- Các lệnh nén

Các lệnh so sánh

Set less than: thiết lập nếu nhỏ hơn

- So sánh số nguyên có dấu:

```
slt   rd, rs1, rs2   # rd=(rs1<rs2)?1:0
slti  rd, rs1, imm   # rd=(rs1<imm)?1:0
```

- So sánh số nguyên không dấu:

```
sltu  rd, rs1, rs2   # rd=(rs1<rs2)?1:0
sltiu rd, rs1, imm   # rd=(rs1<imm)?1:0
```

- Ví dụ
  - s0 = 0xFFFFFFFF
  - s1 = 0x1

```
slt  t0, s0, s1 # -1 < +1 → t0=1
sltu t0, s0, s1 # 4,294,967,295 > 1 → $t0=0
```

Lệnh nhân

- Lệnh nhân hai số nguyên 32-bit và tích 32-bit:

```
mul rd, rs1, rs2   # rd = (rs1*rs2)31:0
```

- Nhân hai số nguyên 32-bit và cất 32-bit cao của tích vào thanh ghi đích rd:
  - Cả hai toán hạng là số có dấu:

```
mulh rd, rs1, rs2      # rd = (rs1*rs2)63:32
```

  - Cả hai toán hạng là số không dấu:

```
mulhu rd, rs1, rs2     # rd = (rs1*rs2)63:32
```

  - Toán hạng rs1 là số có dấu, toán hạng rs2 là số không dấu:

```
mulhsu rd, rs1, rs2    # rd = (rs1*rs2)63:32
```

Ví dụ lệnh nhân

Nhân hai số nguyên có dấu 32-bit, cất kết quả 64-bit vào cặp thanh ghi t1, t2

```
mulh   t1, s3, s4
mul    t2, s3, s4 #{t1,t2}= s3 x s4
```

Lệnh chia

- Lệnh chia số nguyên 32-bit có dấu, lấy phần nguyên

```
div rd, rs1, rs2     # rd = rs1/rs2
```

- Lệnh chia số nguyên 32-bit không dấu, lấy phần nguyên

```
divu rd, rs1, rs2    # rd = rs1/rs2
```

- Lệnh chia số nguyên 32-bit có dấu, lấy phần dư

```
rem rd, rs1, rs2     # rd = rs1%rs2
```

- Lệnh chia số nguyên 32-bit không dấu, lấy phần dư

```
remu rd, rs1, rs2    # rd = rs1%rs2
```

Các lệnh số dấu phẩy động

- Các thanh ghi số dấu phẩy động
  - 32 thanh ghi: f0, f1, … f31
  - Mỗi thanh ghi f có thể chứa số dấu phẩy động 32-bit (single precision) hoặc số dấu phẩy động 64-bit (double precision)

Tên thanh ghi	Số hiệu	Công dụng
ft0-7	f0-7	Các giá trị tạm thời
fs0-1	f8-9	Lưu các biến
fa0-1	f10-11	Tham số vào/kết quả của thủ tục
fa2-7	f12-17	Tham số vào của thủ tục
fs2-11	f18-27	Lưu các biến
ft8-11	f28-31	Các giá trị tạm thời

Các lệnh với số dấu phẩy động

- Các lệnh số học với số FP 32-bit (single-precision)
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
  - `VD:  fadd.s f0, f1, f2`
- Các lệnh số học với số FP 64-bit (double-precision)
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
  - `VD:  fadd.d f0, f1, f2`
- Các lệnh so sánh
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - Kết quả là 0 hoặc 1 trong thanh ghi đích số nguyên
  - `VD:  feq.sx5, f0, f1`
- Các lệnh load/store
  - `flw, fsw`
  - `fld, fsd`
  - `VD:  flw  f0, 4(x5)`

Các lệnh nén (Compressed Instructions)

- Các lệnh có độ dài 16-bit
- Thay thế cho các lệnh với số nguyên và số dấu phẩy động
- Sử dụng tiền tố: **c.**
- Ví dụ:
  - `c.add`
  - `c.lw`
  - `c.addi`

Lệnh giả (Pseudoinstruction)

Pseudoinstruction	RISC-V Instructions
`j label`	`jal x0, label`
`jr ra`	`jalr x0, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi x0, x0, 0`
`li s8, 0x7EF`	`addi s8, x0, 0x7EF`
`li s0, 0x12345678`	`lui s0, 0x12345` `addi s0, s0, 0x678`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L5`	`bge t2, x0, L5`

4.9. Các kiến trúc tập lệnh phổ biến

- Kiến trúc Intel x86
  - Dùng phổ biến trong Laptop, Desktop, Server
  - Từ 1978
  - Tập lệnh phức tạp (CISC)
  - Độ dài lệnh: 1-15 byte
  - 16/32/64-bit
- Kiến trúc ARM
  - Dùng phổ biến trong Smartphone, Tablet
  - Từ 1985
  - Tập lệnh thu gọn (RISC)
  - Độ dài lệnh: 4 byte (2 byte)
  - 32/64-bit

Intel x86

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

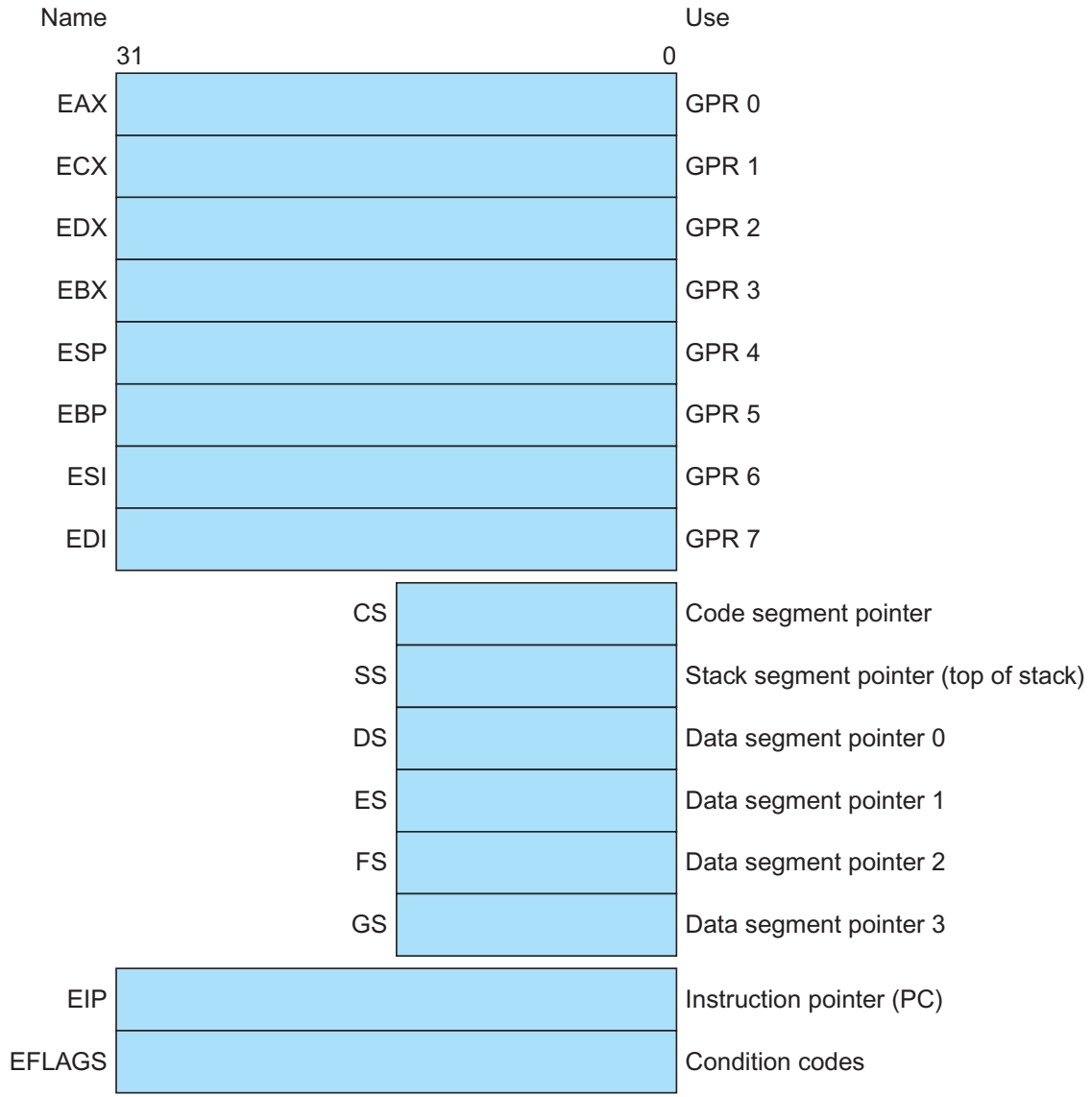
Intel x86

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

Intel x86

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



Basic x86 Addressing Modes

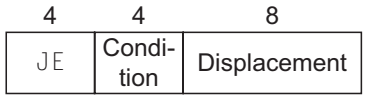
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

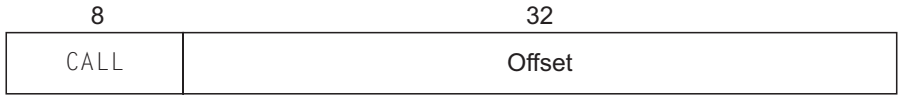
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

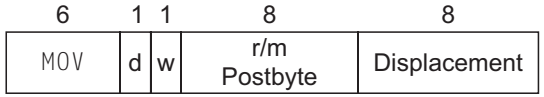
a. JE EIP + displacement



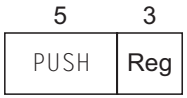
b. CALL



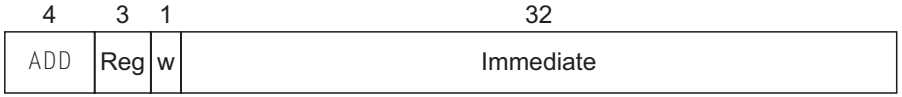
c. MOV EBX, [EDI + 45]



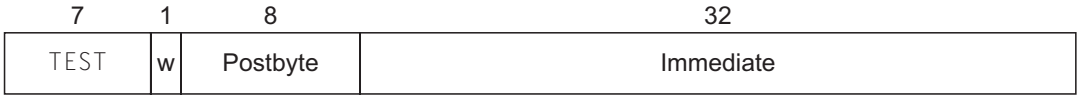
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

ARM & MIPS Similarities

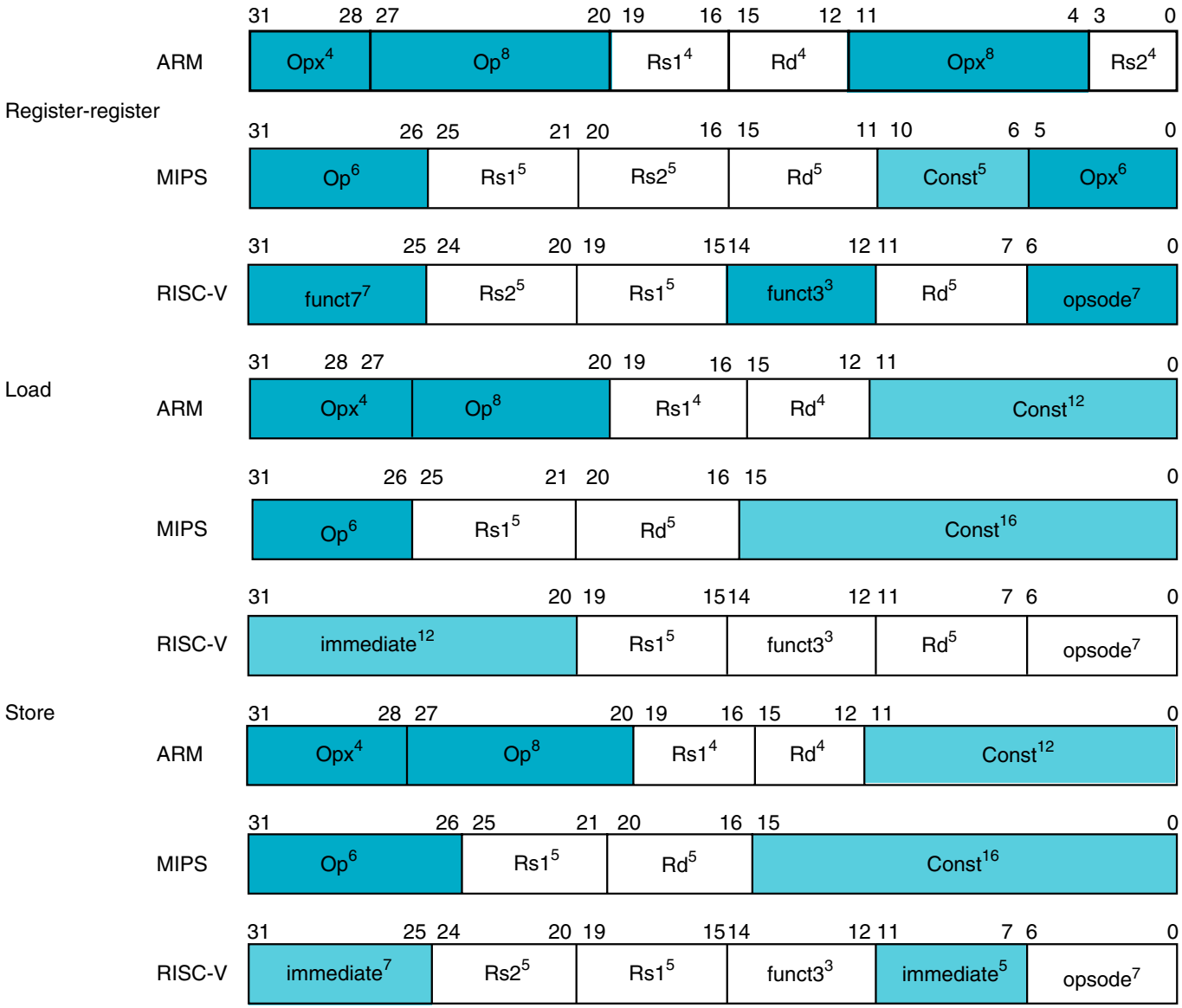
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Formats: ARM, MIPS, RISC-V



ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

Hết chương 3