



Khoa  
**CÔNG NGHỆ THÔNG TIN**  
ĐH Khoa học Tự nhiên TP HCM

# Bài 05: Kiến trúc LEGv8

**Phạm Tuấn Sơn**

**[ptson@fit.hcmus.edu.vn](mailto:ptson@fit.hcmus.edu.vn)**

# Mục tiêu

- Sau bài này, SV có khả năng:
  - Giải thích quan điểm thiết kế tập lệnh LEGv8.
  - Có khả năng dịch các cấu trúc cơ bản (lệnh tính toán, cấu trúc điều khiển, thủ tục, ngăn xếp, ...) của ngôn ngữ lập trình thành hợp ngữ, mã máy LEGv8 và ngược lại.

# Tập lệnh

- Công việc cơ bản nhất của bộ xử lý là xử lý các lệnh máy (*instruction*).
- Tập hợp các lệnh mà một bộ xử lý nào đó cài đặt gọi là tập lệnh (*Instruction Set*).
- Các bộ xử lý khác nhau có tập lệnh khác nhau. Ví dụ:
  - Core i7, i9 (Intel), Ryzen 7000, 9000 (AMD).
  - Apple A17, M3 (ARM).
  - MIPS I6400, M6250, P6600 (MIPS Technology Inc).
  - PowerPC 601 (IBM).
  - SPARC V8 (Sun).
  - ...
- Câu hỏi
  - Một chương trình thực thi (.exe) chạy trên bộ xử lý Core i7 (Intel) có thể chạy được trên bộ xử lý Core i9 (Intel) không ?
  - Một chương trình thực thi (.exe) chạy trên một bộ xử lý của Intel có thể chạy được trên bộ xử lý của AMD ?
  - Một chương trình thực thi (.exe) chạy trên một bộ xử lý của Intel có thể chạy được trên bộ xử lý ARM hay MIPS không ?

# Kiến trúc tập lệnh

- Các bộ xử lý khác nhau có cùng kiến trúc bộ lệnh (Instruction Set Architecture - ISA) có thể thực thi cùng một chương trình.
- x86 (máy tính cá nhân – PC, laptop, netbook và siêu máy tính - supercomputer).
  - x86-32 (IA-32/ i386): Intel 80386, Intel 80486, Intel Pentium, AMD Am386, AMD Am486, AMD K5, AMD K6, AMD K7, ...
  - x86-64: Intel 64 (Intel Pentium D, Intel Core 2, Intel Core i7, Intel Core i9,...), AMD64 (AMD Athlon, AMD Phenom, Ryzen 7000, Ryzen 9000,...).
- ARM (thiết bị di động – mobile, thiết bị nhúng – embedded system, máy tính cá nhân).
  - ARMv7 (32bit): ARM Cortex-M3, ARM Cortex-R8, ARM Cortex-A17, Apple A6.
  - ARMv8 (64bit): ARM Cortex-R82, ARM Cortex-A73, Apple A17, Apple M3.
- MIPS (hệ thống nhúng và siêu máy tính).
  - MIPS32: P5600, M5150,...
  - MIPS64: P6600, I6400, M6250,...
- Ngoài ra, PowerPC (máy chủ, hệ thống nhúng), SPARC (máy chủ), RISC-V (kiến trúc mở), ...

# Kiến trúc tập lệnh ARMv8

- Thuộc sở hữu của ARM Holdings ([www.arm.com](http://www.arm.com)).
- Chiếm thị phần lớn trong thị trường lõi nhúng, chip xử lý trên thiết bị di động và gần đây là chip xử lý máy tính cá nhân.
  - Ứng dụng trong điện tử tiêu dùng, thiết bị mạng/lưu trữ, máy ảnh, máy in,...
  - Điện thoại thông minh, máy tính bảng,...
  - Apple PC, laptop.
- Quan điểm thiết kế được sử dụng trong các kiến trúc tập lệnh hiện nay.
- LEGv8, một phần cơ bản của tập lệnh ARMv8, được sử dụng trong học phần này.
  - “ARMv8” sẽ được sử dụng khi đề cập tới tập lệnh đầy đủ.

## 4 nguyên tắc thiết kế tập lệnh ARMv8

- (1) Simplicity favors regularity.
  - Tập lệnh có qui tắc thì cài đặt phần cứng sẽ đơn giản.
- (2) Smaller is faster.
  - Nhỏ gọn (tập lệnh, thanh ghi, ...) sẽ nhanh hơn.
- (3) Make the common case fast.
- (4) Good design demands good compromises.
  - Những trường hợp thường xuyên được sử dụng (trong lập trình) có thể sẽ được thêm vào lệnh mới để thực thi nhanh hơn tuy nhiên cũng có thể phải phát sinh cấu trúc lệnh mới → đối lập với tính qui tắc → thỏa hiệp !

# Khảo sát các phép toán số học & luận lý CPU cần thực thi

- Các phép toán luận lý và số học như:

$$a = b + c$$

$$a = b \& c$$

$$a = b << 3$$

gồm:

- Loại phép toán
- 2 toán hạng nguồn + 1 toán hạng đích
  - Toán hạng đích là một định danh lưu trữ.
  - Toán hạng nguồn là định danh lưu trữ. Trong phép dịch, toán hạng nguồn thứ 2 có thể là một giá trị số.
- Để đơn giản và dễ dàng trong việc truy xuất bộ nhớ, tất cả các lệnh đều có chiều dài cố định là 32 bit.
  - Nguyên tắc 1: *Simplicity favors regularity.*
- Như vậy, một lệnh máy 32-bit gồm các thành phần sau:

Loại phép toán	Toán hạng nguồn 2	Số bit dịch	Toán hạng nguồn 1	Toán hạng đích
----------------	-------------------	-------------	-------------------	----------------

? bits

? bits

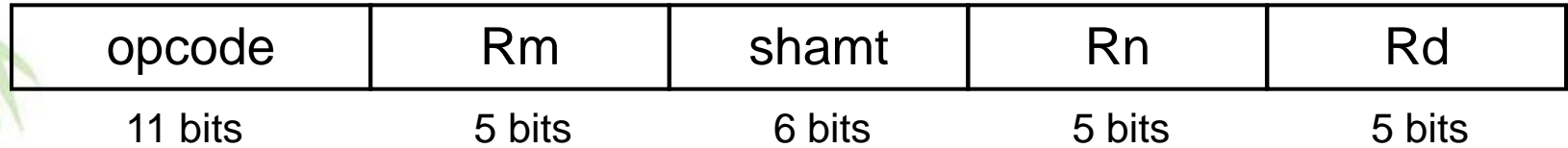
? bits

? bits

? bits



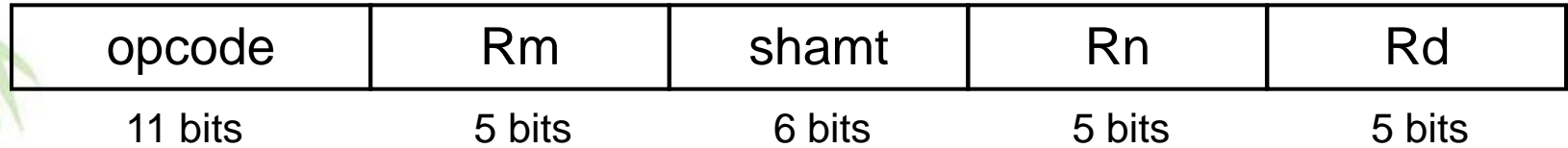
# Cấu trúc lệnh R-Format (1/2)



- opcode: mã thao tác, cho biết loại lệnh gì.
- Rn (Register source): chứa địa chỉ thanh ghi nguồn thứ 1.
- Rm (Register source): chứa địa chỉ thanh ghi nguồn thứ 2.
- Rd (Register destination): chứa địa chỉ thanh ghi đích.
  - Tại sao các toán hạng là địa chỉ thanh ghi mà không phải bộ nhớ ?
  - Tại sao mỗi trường toán hạng địa chỉ thanh ghi chỉ có 5 bit ?
- shamt: chứa số bit cần dịch trong các lệnh dịch.
  - Tại sao mỗi trường số bit dịch này có 6 bit ?
  - Nếu không phải lệnh dịch thì trường này có giá trị 0.
  - Tại sao không dùng trường Rm làm số bit dịch ?



# Cấu trúc lệnh R-Format (2/2)



- Tất cả các lệnh tính toán số học, luận lý trong tập lệnh đều có cấu trúc này hoặc ở định dạng có 3 toán hạng.
- Nguyên tắc 1: Simplicity favors regularity.
  - Cài đặt phần cứng cho các lệnh sẽ đơn giản hơn.
  - Phần cứng đơn giản sẽ dễ đạt hiệu năng cao với chi phí thấp hơn.
- LEGv8 có tập thanh ghi gồm 32 thanh ghi 64-bit.
- Định danh toán hạng trong tất cả các lệnh số học, luận lý đều là thanh ghi.
- Nguyên tắc 2: Smaller is faster.
  - Truy xuất vào tập thanh ghi có số lượng ít sẽ nhanh hơn (so với bộ nhớ có dung lượng lớn).

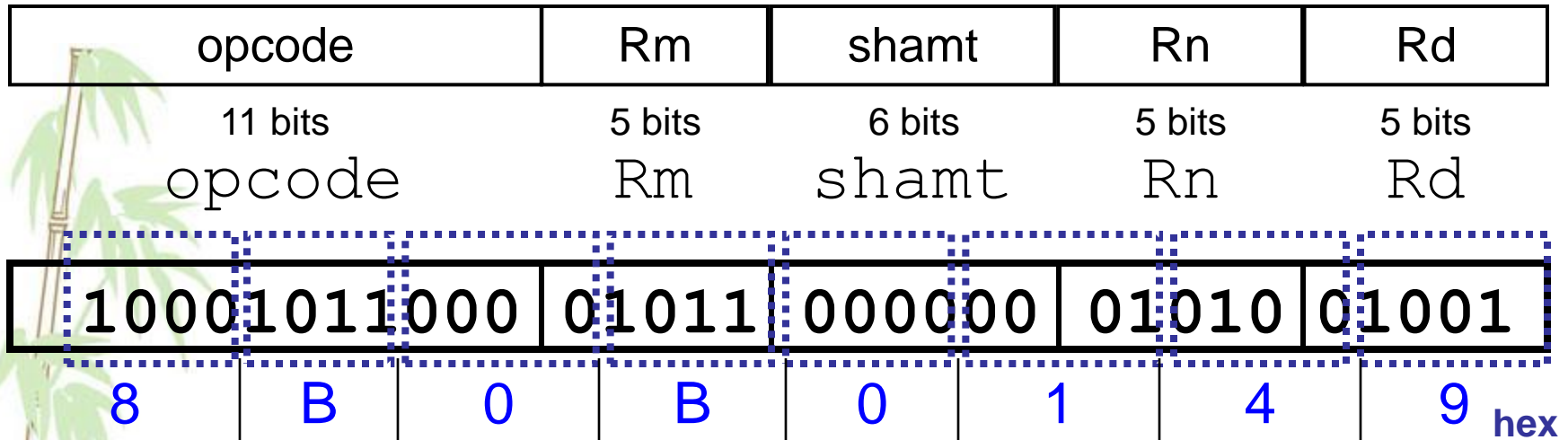
# Một số đặc điểm của toán hạng thanh ghi

- Đóng vai trò giống như biến trong các NNLT cấp cao (C, Java). Tuy nhiên, khác với biến chỉ có thể giữ giá trị theo kiểu dữ liệu được khai báo trước khi sử dụng, thanh ghi không có kiểu, thao tác trên thanh ghi sẽ xác định dữ liệu trong thanh ghi sẽ được đối xử như thế nào.
- Ưu điểm: bộ xử lý truy xuất thanh ghi nhanh nhất (hơn 1 tỉ lần trong 1 giây) vì thanh ghi là một thành phần phần cứng thường nằm chung mạch với bộ xử lý.
- Khuyết điểm: do thanh ghi là một thành phần phần cứng nên số lượng cố định và hạn chế. Do đó, sử dụng phải khéo léo.

# Tập thanh ghi

- 32 thanh ghi 64-bit (64-bit ~ “doubleword”).
  - X0 – X7: procedure arguments/results.
  - X8: indirect result location register.
  - X9 – X15: temporaries.
  - X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register.
  - X18: platform register for platform independent code; otherwise a temporary register.
  - X19 – X27: saved.
  - X28 (SP): stack pointer.
  - X29 (FP): frame pointer.
  - X30 (LR): link register (return address).
  - XZR (register 31): the constant value 0.
- Ngoài ra, ARMv8 còn có thể truy xuất vào 32-bit thấp của 1 thanh ghi 64-bit → 31 thanh ghi 32-bit W0 – W30 (32-bit ~ “word”).
  - 5-bit toán hạng thanh ghi sao có thể biểu diễn nhiều hơn 32 thanh ghi ?

# Ví dụ cấu trúc lệnh R-Format



Giá trị thập phân tương ứng của từng trường

1112	11	0	10	9
------	----	---	----	---

opcode = 1112 (mã lệnh cộng thanh ghi)

Rd = 9 (toán hạng đích là thanh ghi X9)

Rn = 10 (toán hạng nguồn thứ 1 là thanh ghi X10)

Rm = 11 (toán hạng nguồn thứ 2 là thanh ghi X11)

shamt = 0 (không phải lệnh dịch)

$$X9 = X10 + X11$$

# Lệnh hợp ngữ số học và luận lý cấu trúc R-Format

- Cú pháp: `op Rd, Rn, Rm/shamt`

– Trong đó:

`op` – Tên lệnh / thao tác (toán tử)

`Rd` – Thanh ghi (toán hạng đích) chứa kết quả

`Rn` – Thanh ghi (toán hạng nguồn thứ 1)

`Rm/shamt` – Thanh ghi hoặc hằng số (toán hạng nguồn thứ 2)

opcode	Rm	shamt	Rn	Rd
10001011000	01011	000000	01010	01001
1112	11	0	10	9

`x9 = x10 + x11`

`ADD x9, x10, x11`

# Lệnh số học cấu trúc R-Format (1/4)

- Lệnh cộng:

ADD      X19, X20, X21 (cộng giá trị 2 thanh ghi)

ADDS     X19, X20, X21 (cộng và bật các cờ điều kiện  
N (Negative), Z (Zero), V (oVerflow), C (Carry))

tương ứng với:       $a = b + c$ ; (trong C)

trong đó các thanh ghi X19, X20, X21 (trong LEGv8) tương ứng  
với các biến a, b, c (trong C)

- Lệnh trừ:

SUB      X19, X20, X21 (trừ giá trị 2 thanh ghi)

SUBS     X19, X20, X21 (trừ và bật các cờ điều kiện  
N (Negative), Z (Zero), V (oVerflow), C (Carry))

tương ứng với:       $d = e - f$ ; (trong C)

trong đó các thanh ghi X19, X20, X21 (trong LEGv8) tương ứng  
với các biến d, e, f (trong C)

- Các lệnh trên có phân biệt số không dấu hay có dấu không ? (Lưu ý là các biến trên NNLT có thể là không dấu (unsigned) hoặc có dấu (signed)).

# Lệnh số học cấu trúc R-Format (2/4)

- Làm thế nào để thực hiện câu lệnh C sau đây bằng lệnh máy ?

$$a = b + c + d - e;$$

- Các thanh ghi X19 – X27 (saved register) được quy ước sử dụng để chứa giá trị của các biến NNLT.
- Giả sử: a: X19, b: X20, c: X21, d: X22, e: X23

ADD X19, X20, X21 //  $a = b + c$

ADD X19, X19, X22 //  $a = a + d$

SUB X19, X19, X23 //  $a = a - e$

- Chú ý: một lệnh trong C có thể tương ứng nhiều lệnh máy.
- Ghi chú: ký tự “//” dùng để chú thích trong hợp ngữ cho LEGv8/ARMv8.
- Tại sao không xây dựng các lệnh máy có nhiều toán hạng nguồn hơn ?



# Lệnh số học cấu trúc R-Format (3/4)

- Làm thế nào để thực hiện dãy tính sau bằng lệnh máy ?

$$f = (g + h) - (i + j);$$

- Giả sử: f: X19, g: X20, h: X21, i: X22, j: X23
- Các thanh ghi tạm X9 – X15 được sử dụng để lưu kết quả trung gian.
- Như vậy dãy tính trên có thể được thực hiện như sau:

ADD X9, X20, X21

// temp = g + h

ADD X10, X22, X23

// temp = i + j

SUB X19, X9, X10

// f = (g+h) - (i+j)

# Lệnh số học cấu trúc R-Format (4/4)

- Lệnh nhân:

MUL X1, X2, X3 (multiply  $X1 = X2 \times X3$   
X1 chứa 64-bit thấp của tích)

SMULH X1, X2, X3 (signed multiply high  $X1 = X2 \times X3$   
X1 chứa 64-bit cao của tích nhân có dấu)

UMULH X1, X2, X3 (unsigned multiply high  $X1 = X2 \times X3$   
X1 chứa 64-bit cao của tích nhân không dấu)

- Thực hiện câu lệnh C:  $a = b * c$  đây bằng lệnh máy ?

- Lệnh chia:

SDIV X1, X2, X3 (chia có dấu (signed divide)  $X1 = X2 / X3$ )

UDIV X1, X2, X3 (chia không dấu (unsigned divide)  $X1 = X2 / X3$ )

- Làm sao để lấy phần dư của phép chia ?

- Các lệnh trên có phân biệt số không dấu hay có dấu không?  
(Lưu ý là các biến trên NNLT có thể là không dấu (unsigned) hoặc có dấu (signed)).

# Lệnh luận lý cấu trúc R-Format

- AND:

AND      X19, X20, X21 ( $X19 = X20 \& X21$ )

- OR:

ORR      X19, X20, X21 ( $X19 = X20 | X21$ )

- XOR:

EOR      X19, X20, X21 (Exclusive OR  $X19 = X20 \wedge X21$ )

- NOT:

- Không có lệnh máy cho phép NOT. Vì sao ?
- Làm sao để thực hiện phép NOT ?

- Dịch trái logic:

LSL      X19, X20, #10 (Logic shift left  $X19 = X20 \ll 10$ )

- Dịch phải logic:

LSR      X19, X20, #10 (Logic shift right  $X19 = X20 \gg 10$ )

- Dịch số học:

- ASR là lệnh dịch phải số học (arithmetic shift right) của ARMv8.18

# Ví dụ mã máy của lệnh LSR

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

LSR X19, X20, #10

opcode	Rm	shamt	Rn	Rd
11010011010	00000	001010	10100	10011
D 3 4 0 2 A 9 3 hex				

Giá trị thập phân tương ứng của từng trường

1690	0	10	20	19
------	---	----	----	----

opcode = 1690 (mã lệnh dịch phải logic)

Rd = 19 (toán hạng đích là thanh ghi X19)

Rn = 20 (toán hạng nguồn thứ 1 là thanh ghi X20)

Rm = 0 (không dùng trong lệnh dịch)

shamt = 10 (số bit dịch)

# Thanh ghi XZR

- Làm sao để thực hiện phép gán  $a = b$ ; hay  $a = 0$ ; ?
  - Thêm lệnh mới chỉ dùng 2 toán hạng ? *Nguyên tắc 1* ?
- LEGv8 định nghĩa thanh ghi XZR (thanh ghi số 31) luôn có giá trị 0 nhằm hỗ trợ thực hiện phép gán và các thao với giá trị 0.

Ví dụ:

ORR X19, XZR, X20

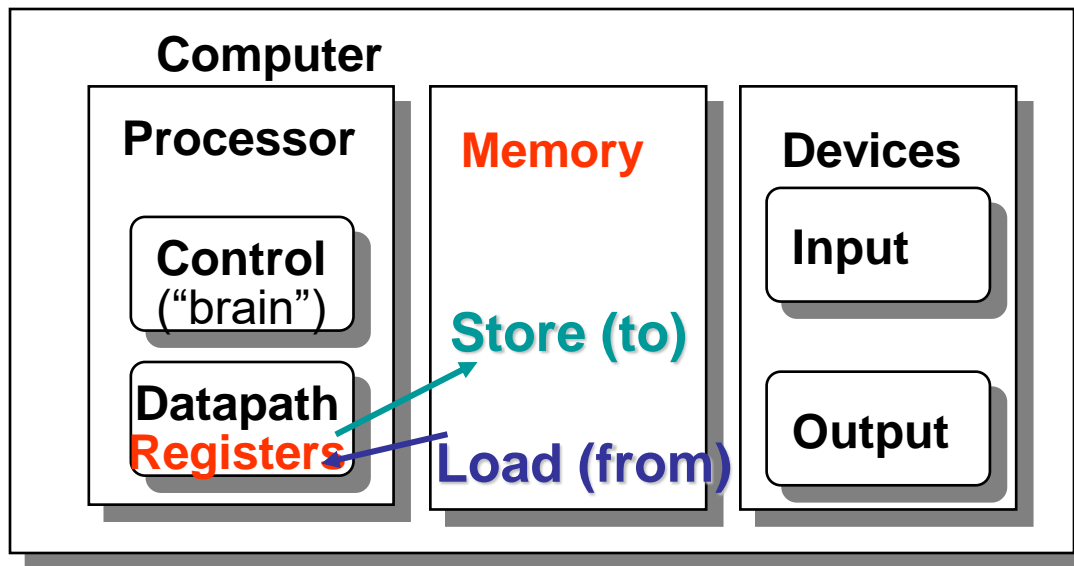
tương ứng với  $a = b$ ; (trong C)

trong đó các thanh ghi X19, X20 (trong LEGv8) tương ứng với các biến a, b (trong C)

- $a = 0$  ?
- Lệnh SUBS XZR, X20, X11 có ý nghĩa không ?

# Truy xuất bộ nhớ

- Dữ liệu từ đâu được đưa vào các thanh ghi để xử lý ?
- Cần di chuyển dữ liệu (Data transfer) giữa thanh ghi và bộ nhớ:
  - Từ bộ nhớ vào thanh ghi (nạp - load)
  - Từ thanh ghi vào vùng nhớ (lưu - store)



- Như vậy, bộ xử lý nạp các dữ liệu (và lệnh) vào các thanh ghi để xử lý rồi lưu kết quả ngược trở lại bộ nhớ

# Khảo sát cấu trúc lệnh truy xuất bộ nhớ

- Bộ nhớ là mảng 1 chiều các ô nhớ có địa chỉ

Dữ liệu	1	101	10	100	.	.	.
Địa chỉ	0	1	2	3	.	.	.

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Trong cấu trúc R-format hỗ trợ các lệnh số học và luận lý (đã tìm hiểu), các toán hạng  $Rd$ ,  $Rn$ ,  $Rm$  giữ địa chỉ các thanh ghi
- Làm sao để truy xuất dữ liệu trong bộ nhớ?
  - Cần toán hạng giữ địa chỉ ô nhớ
- Có 2 hướng giải quyết
  - Cho phép  $Rn$ ,  $Rm$  lưu địa chỉ bộ nhớ. Có khả thi ?
  - Tạo ra cấu trúc lệnh khác để thao tác với bộ nhớ



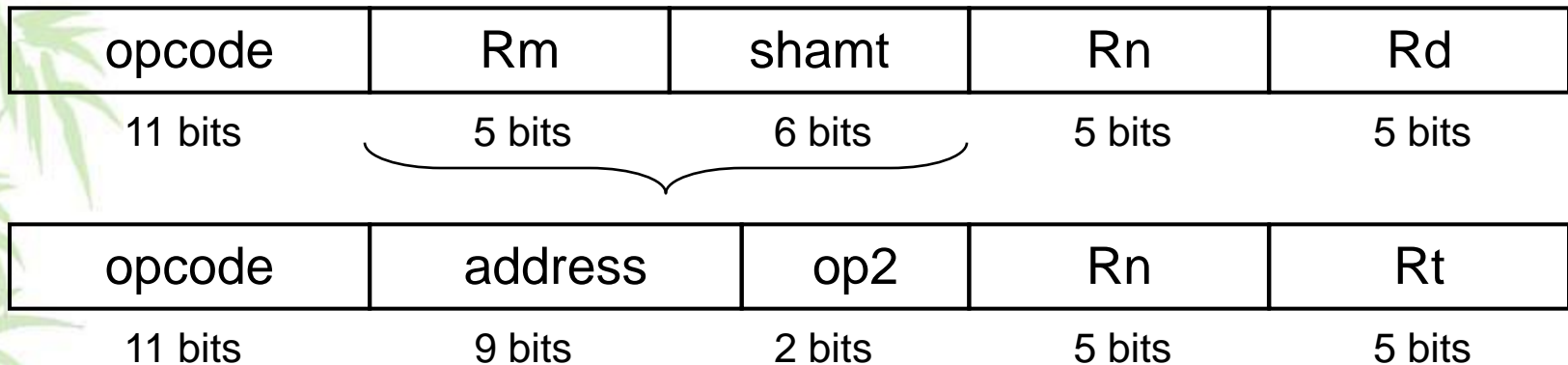
# Toán hạng bộ nhớ

- Bộ nhớ chính được sử dụng cho dữ liệu.
  - Dữ liệu đơn giản như biến đơn. Ví dụ:  $a, b, \dots$
  - Dữ liệu phức tạp hơn như: mảng, cấu trúc, dữ liệu động, ...
    - Ví dụ truy xuất phần tử mảng:  $A[8]$
- Như vậy, một cách tổng quát, để truy xuất bộ nhớ cần 2 toán hạng nguồn:
  - Một thanh ghi chứa địa chỉ bộ nhớ (thanh ghi cơ sở)
    - Đối với biến đơn, chính là địa chỉ của biến.
    - Đối với mảng, cấu trúc, ..., chứa địa chỉ bắt đầu (như là con trỏ chứa địa chỉ).
  - Một số nguyên (xem như độ dời từ địa chỉ trong thanh ghi cơ sở).
    - Đối với biến đơn, giá trị này bằng 0.
    - Đối với mảng, cấu trúc, ..., độ dời tới phần tử mảng, trường của cấu trúc.
- Địa chỉ vùng nhớ sẽ được xác định bằng tổng 2 giá trị này.

# Cấu trúc lệnh truy xuất bộ nhớ

## Cấu trúc D-Format (1/2)

- Tạo cấu trúc lệnh mới thế nào để giảm thiểu thay đổi so với cấu trúc R-Format → Cấu trúc D-Format.



→ Nguyên tắc 4: Good design demands good compromises.

- Các cấu trúc lệnh khác nhau làm phức tạp việc giải mã, nhưng vẫn đảm bảo các lệnh có kích thước 32-bit thống nhất.
- Giữ các cấu trúc giống nhau nhất có thể.

# Cấu trúc lệnh truy xuất bộ nhớ

## Cấu trúc D-Format (2/2)

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- opcode: mã thao tác, cho biết lệnh gì (tương tự opcode của R-Format).
- Rn: địa chỉ thanh ghi cơ sở chứa địa chỉ bộ nhớ cơ sở.
- address: giá trị độ dời từ giá trị trong thanh ghi cơ sở (+/- 32 doublewords). Lưu ý: bộ nhớ trong kiến trúc LEGv8 được đánh địa chỉ theo byte (8-bit).
- Rt: địa chỉ thanh ghi chứa giá trị.
  - Là “đích” đối với lệnh nạp giá trị từ bộ nhớ vào thanh ghi.
  - Là “nguồn” đối với lệnh lưu giá trị từ thanh ghi vào bộ nhớ.
- op2: được dùng trong tập lệnh đầy đủ của ARMv8.

# Cấu trúc lệnh hợp ngữ truy xuất bộ nhớ tương ứng

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Cú pháp:

op      Rt, [Rn, #address]

- Lệnh LDUR (Load register (unscaled offset)) nạp 64-bit dữ liệu (doublewords) từ bộ nhớ vào thanh ghi.

Data flow

LDUR X9, [X22, #64]

–Lệnh này nạp 64-bit dữ liệu từ bộ nhớ có địa chỉ  $(X22+64)$  vào thanh ghi X9.

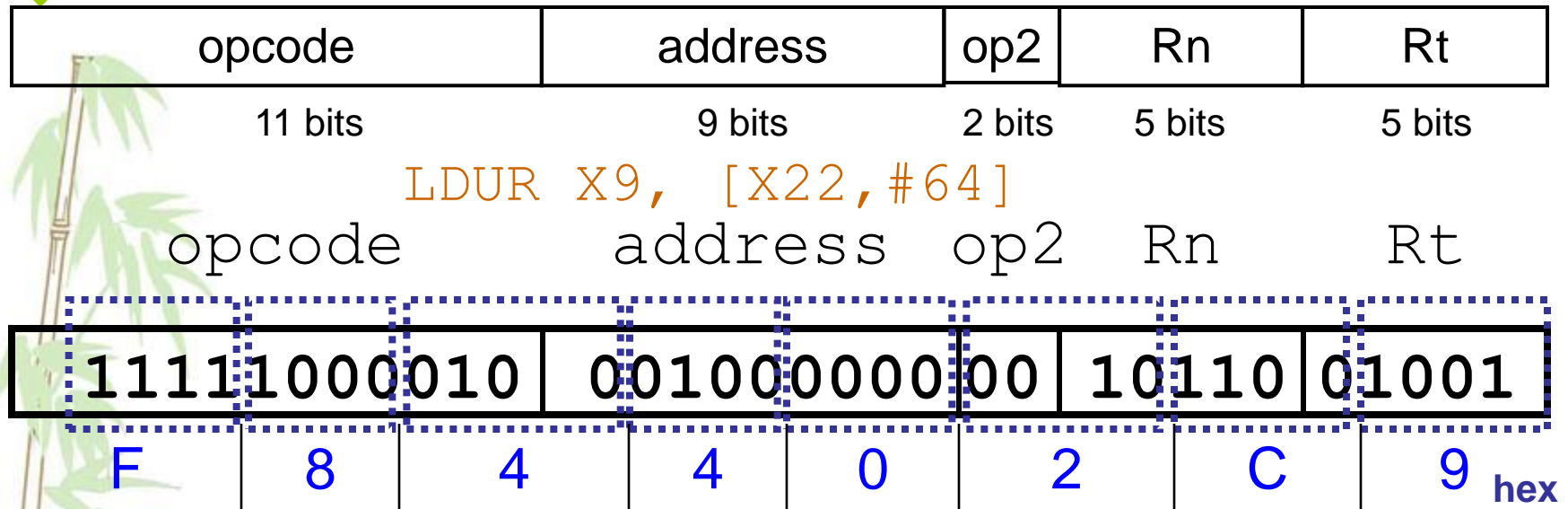
- Lệnh STUR (Store register (unscaled offset)) lưu 64-bit dữ liệu (doublewords) từ thanh ghi vào bộ nhớ.

Data flow

STUR X9, [X22, #64]

–Lệnh này lưu 64-bit dữ liệu từ thanh ghi X9 vào bộ nhớ có địa chỉ  $(X22+64)$ .

# Ví dụ mã máy của lệnh nạp LDUR

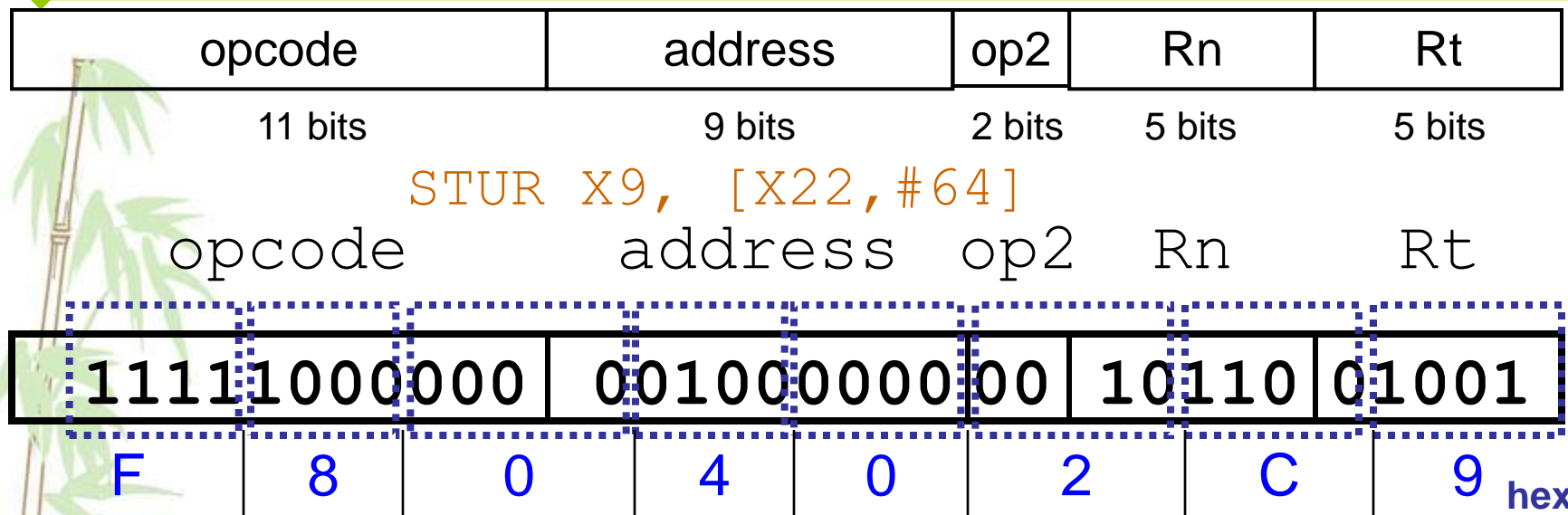


Giá trị thập phân tương ứng của từng trường

1986	64	0	22	9
------	----	---	----	---

opcode = 1986 (mã lệnh nạp dữ liệu 64-bit từ bộ nhớ)  
 Rt = 9 (toán hạng đích là thanh ghi X9)  
 Rn = 22 (toán hạng địa chỉ cơ sở là thanh ghi X20)  
 address = 64 (toán hạng giá trị độ dời)  
 op2 = 0

# Ví dụ mã máy của lệnh nạp STUR



Giá trị thập phân tương ứng của từng trường

1984	64	0	22	9
------	----	---	----	---

opcode = 1984 (mã lệnh lưu dữ liệu 64-bit vào bộ nhớ)  
 Rt = 9 (toán hạng đích là thanh ghi X9)  
 Rn = 22 (toán hạng địa chỉ cơ sở là thanh ghi X20)  
 address = 64 (toán hạng giá trị độ dời)  
 op2 = 0

# Ví dụ truy xuất mảng

- Câu lệnh C :

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

- Giả sử

- địa chỉ bắt đầu của mảng A: X22
- mảng A có kích thước 1 phần tử là 8 byte.
- h: X21

được biên dịch thành lệnh LEGv8 như sau:

```
LDUR X9, [X22, #64] // X9 = A[8]
```

```
ADD X9, X21, X9 // X9 = h + A[8]
```

```
STUR X9, [X22, #96] // A[12] = h + A[8]
```

- Chú ý:

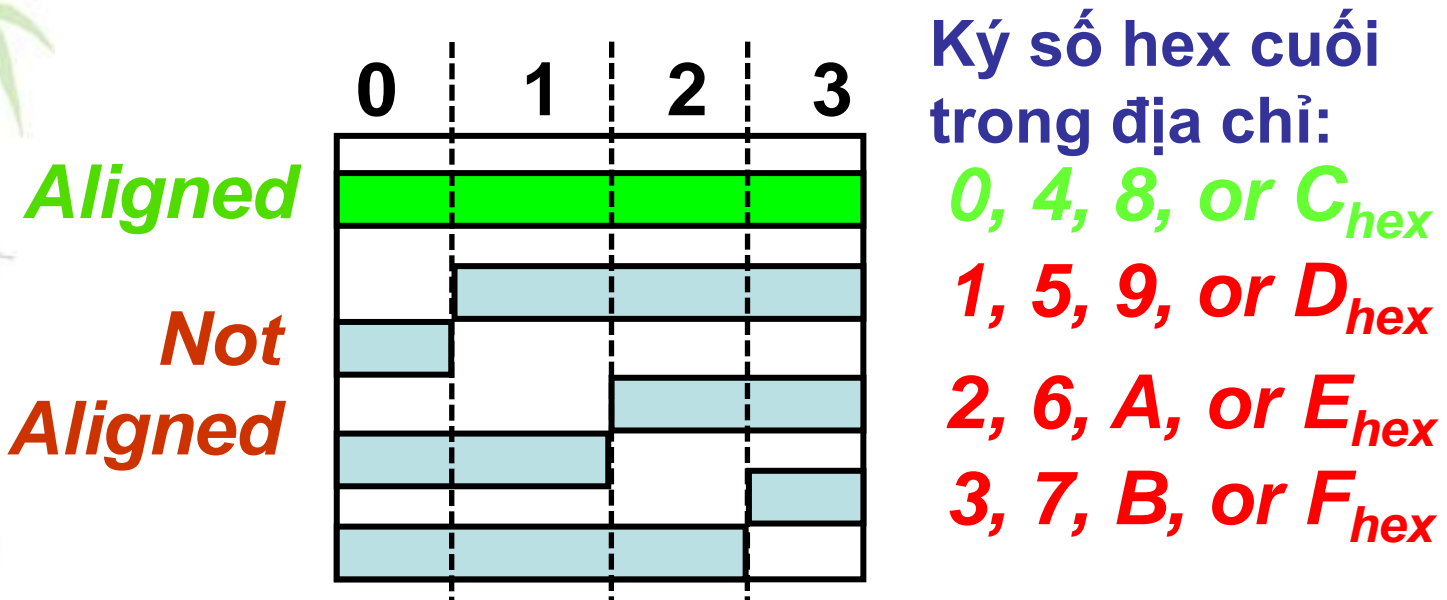
- A[8] là phần tử thứ 8 của mảng A, mỗi phần tử có kích thước 8 bytes (doublewords). Do đó, sẽ tương đương với vùng nhớ bắt đầu tại địa chỉ  $X22 + 64$



# Nguyên tắc lưu trữ và truy xuất dữ liệu trong bộ nhớ (1/2)

## • Nguyên tắc Alignment Restriction:

- Truy xuất bộ nhớ phải bắt đầu tại địa chỉ là bội số của kích thước đối tượng (word hay doublewords) muốn truy xuất.



- LEGv8/ARMv8 không yêu cầu Alignment Restriction ngoại trừ các lệnh và ngăn xếp.
  - X86 không theo yêu cầu Alignment Restriction, trong khi ARMv7 và MIPS thì có.

## • Lệnh LDUR X9, [X22, #3] có hợp lệ không ?

# Nguyên tắc lưu trữ và truy xuất dữ liệu trong bộ nhớ (2/2)

- Endianness

- Dữ liệu trong bộ nhớ có thể được lưu theo chiều địa chỉ tăng dần (leftmost) gọi là big-endian hoặc theo chiều địa chỉ giảm dần (rightmost) gọi là little-endian

- Ví dụ: lưu trữ giá trị 4 byte 12345678h trong bộ nhớ

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

- LEGv8/ARMv8 hỗ trợ cả big-endian và little-endian.

- X86 lưu trữ dữ liệu trong bộ nhớ theo nguyên tắc little-endian, trong khi MIPS theo big-endian).

# Lệnh truy xuất bộ nhớ cấu trúc D-Format (1/2)

- Truy xuất 8 bytes (doublewords)
  - LDUR Rt, [Rn, #offset]
  - STUR Rt, [Rn, #offset]
- Truy xuất 4 bytes (word)
  - LDURSW Rt, [Rn, #offset]
    - 4 byte cao của Rt được mở rộng bằng bit dấu (sign-extended).
  - STURW Rt, [Rn, #offset]
    - Lưu 4 byte thấp của Rt vào bộ nhớ.
- Truy xuất 1 bytes
  - LDURB Rt, [Rn, #offset]
    - 7 byte cao của Rt được mở rộng bằng bit 0 (zero-extended).
  - STURB Rt, [Rn, #offset]
    - Lưu 1 byte thấp của Rt vào bộ nhớ.

# Lệnh truy xuất bộ nhớ cấu trúc D-Format

- Truy xuất 2 byte (halfword)
  - LDURH Rt, [Rn, #offset]
    - 6 byte cao của Rt được mở rộng bằng bit 0 (zero-extended)
  - STURH Rt, [Rn, #offset]
    - Lưu 2 byte thấp của Rt vào bộ nhớ.
- ARMv8 có các lệnh nạp có/không dấu còn lại như LDURSB, LDURSH...
- Tại sao lại hỗ trợ các lệnh nạp/lưu 2 byte trong khi vẫn có thể sử dụng các lệnh nạp/lưu 1 byte để thực hiện thay ?
  - Nguyên tắc 3: Make the common case fast.
    - Ký tự Unicode 2 bytes .

# Con trỏ vs giá trị

- Lưu ý phân biệt 2 trường hợp sau (giả sử x: X19, y: X20)
  - Nếu ghi thì tương đương  $\text{ADD/ORR } X20, X19, XZR$   
X19 lưu giá trị  
 $y = x; \text{ (trong C)}$
  - Nếu ghi thì tương đương  $\text{LDUR } X20, [X19, \#0]$   
X19 chứa một địa chỉ (vai trò như một con trỏ)  
 $y = *x; \text{ (trong C)}$
- Hãy chuyển lệnh  $*x = *y; \text{ (trong C)}$  thành lệnh tương ứng trong LEGv8, giả sử các con trỏ x, y được lưu trong X19 và X20)

# Toán hạng thanh ghi và vùng nhớ

- Trong LEGv8/ARMv8, chỉ có các lệnh nạp, lưu mới sử dụng toán hạng vùng nhớ
  - Tại sao không sử dụng toán hạng vùng nhớ trong các lệnh khác như số học, luận lý,...?
- Một nhiệm vụ của trình biên dịch là ánh xạ các biến được sử dụng trong chương trình thành các thanh ghi
  - Điều gì xảy ra nếu biến sử dụng trong các chương trình nhiều hơn số lượng thanh ghi ?
  - Nhiệm vụ của trình biên dịch: spilling

# Thao tác với hằng số

- Làm sao để thực hiện các phép toán với một giá trị trực tiếp như:  $i = i + 1$ ; hay  $a = b - 1000$ ; ?
- Có cần thêm lệnh mới không ?
  - Có thể được thực hiện bằng cách kết hợp các lệnh nạp, lưu bộ nhớ với các thao tác trên thanh ghi ?
- Nguyên tắc 3: Make the common case fast
  - Các lệnh này được sử dụng rất thường xuyên trên NNLT.
- Tận dụng các cấu trúc lệnh R-Format hoặc D-Format ?

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- 6-bit (R-Format) hoặc 9-bit (D-Format) chỉ chứa được giá trị lớn nhất là 512 !



# Cấu trúc lệnh I-Format

- Cần càng nhiều bit cho toán hạng hằng số càng tốt.

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

- opcode: mã thao tác, cho biết loại lệnh gì.
- Rn (Register source): chứa địa chỉ thanh ghi nguồn thứ 1.
- immediate: chứa giá trị toán hạng nguồn thứ 2.
- Rd (Register destination): chứa địa chỉ thanh ghi đích.

→ Nguyên tắc 4: Good design demands good compromises.

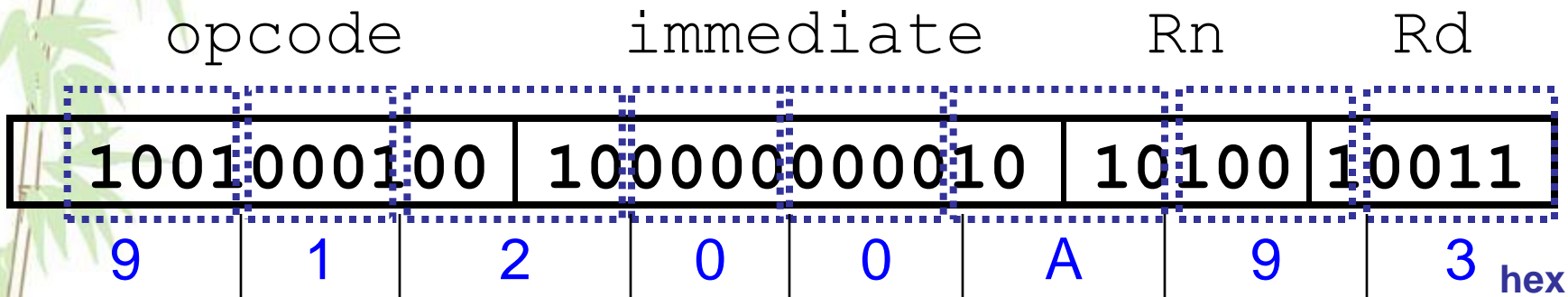
- Thêm cấu trúc lệnh mới kèm theo kích thước trường opcode thay đổi làm phức tạp phần cứng. Tuy nhiên, cấu trúc mới vẫn định dạng tương tự các cấu trúc cũ (3 toán hạng, trong đó 2 toán hạng cuối vẫn là thanh ghi đích  $R_d$  và thanh ghi nguồn thứ nhất  $R_n$ ) để giảm sự phức tạp.

# Cấu trúc lệnh hợp ngữ I-Format



• Cú pháp: `op Rd, Rn, #immediate`

• Ví dụ: `ADDI X19, X20, #2050`



Giá trị thập phân tương ứng của từng trường

580	2050	20	19
-----	------	----	----

opcode = 580 (mã lệnh cộng hằng số)

Rd = 19 (toán hạng đích là thanh ghi x19)

Rn = 20 (toán hạng nguồn thứ nhất là thanh ghi x20)

immediate = 2050 (toán hạng nguồn thứ 2 là giá trị 2050)

# Các lệnh cấu trúc I-Format

- Các lệnh số học:

ADDI     X19, X20, #1 ( $X19 = X20 + 1$ )

ADDIS    X19, X20, #1 ( $X19 = X20 + 1$ ), bật cờ N, Z, V, C

SUBI     X19, X20, #1 ( $X19 = X20 - 1$ )

SUBIS    X19, X20, #1 ( $X19 = X20 - 1$ ), bật cờ N, Z, V, C

- Các lệnh luận lý:

ANDI     X19, X20, #255 ( $X19 = X20 \& 0xFF$ )

ORRI     X19, X20, #255 ( $X19 = X20 | 0xFF$ )

EORI     X19, X20, #255 ( $X19 = X20 \wedge 0xFF$ )

- Lưu ý:

- Hợp ngữ ARMv8 không nhất thiết có lệnh ADDI mà đơn giản chỉ cần sử dụng lệnh ADD để thực hiện cộng với hằng số. Trình hợp dịch sẽ chuyển lệnh ADD thành mã (opcode) tương ứng.
- Lệnh hợp ngữ ADDI được thêm vào LEGv8 nhằm mục đích giảng dạy.

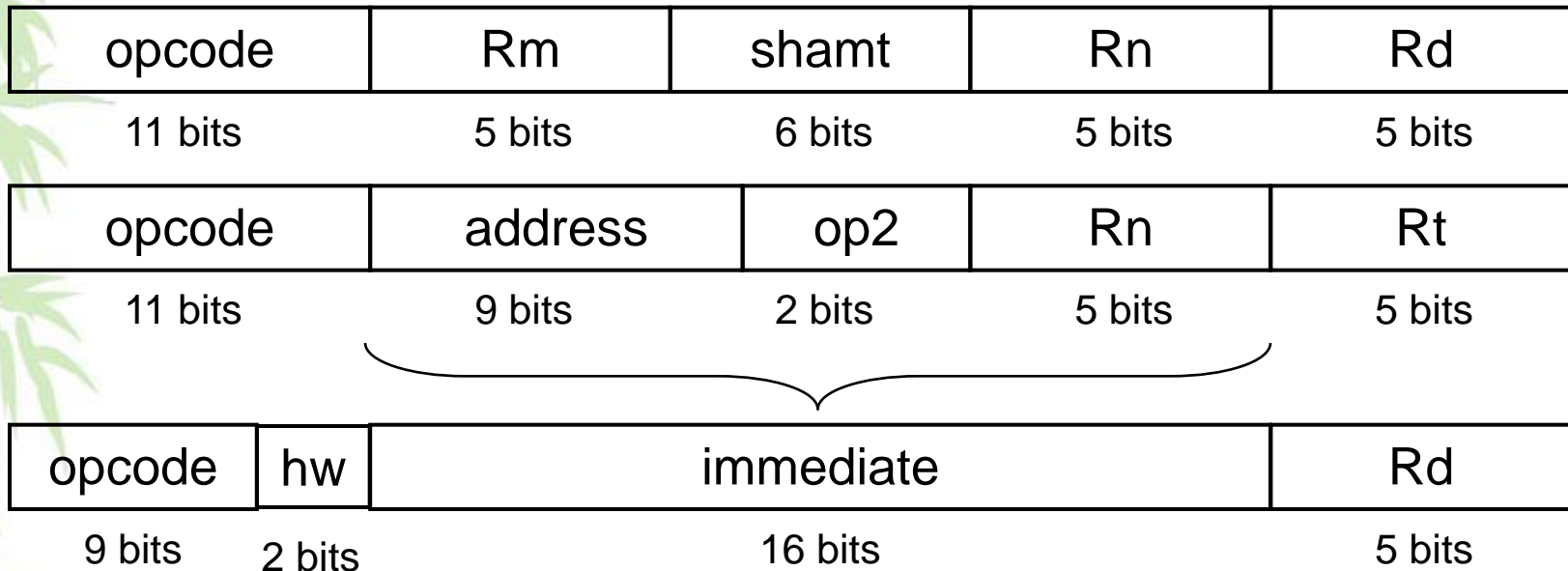
- Làm sao để thao tác với hằng số lớn hơn  $2^{12}$  ?

# Thao tác với hằng số 64-bit

## • Vấn đề:

- Mặc dù các hằng số thường nhỏ và 12-bit là đủ, nhưng nếu cần thao tác với hằng số lớn hơn, 32-bit hay 64-bit thì sao ?

- Tăng kích thước trường *immediate* lên 64-bit ?



# Cấu trúc lệnh hợp ngữ IM-Format

## Cấu trúc lệnh hợp ngữ MOVZ

opcode	hw	immediate	Rd
--------	----	-----------	----

9 bits

2 bits

16 bits

5 bits

• **Cú pháp:** `op Rd, immediate, LSL 0/16/32/48`

• **Ví dụ:** `MOVZ X19, 255, LSL 16`

opcode

hw

immediate

Rd

110100101	01	0000000011111111	10011
-----------	----	------------------	-------

C

2

A

0

1

F

F

3

hex

Giá trị thập phân tương ứng của từng trường

645	1	255	19
-----	---	-----	----

opcode = 645 (mã lệnh gán hằng số 16-bit vào thanh ghi, các bit còn lại bằng 0)

hw = 1 (logic shift left 16 bit (1 halfword)).

immediate = 255 (toán hạng nguồn thứ 2 là giá trị 255)

Rd = 19 (toán hạng đích là thanh ghi X19)

0000000000000000	0000000000000000	0000000011111111	0000000000000000
------------------	------------------	------------------	------------------

# Cấu trúc lệnh hợp ngữ IM-Format

## Cấu trúc lệnh hợp ngữ MOVK

opcode	hw	immediate	Rd
--------	----	-----------	----

9 bits

2 bits

16 bits

5 bits

• **Cú pháp:** `op Rd, immediate, LSL 0/16/32/48`

• **Ví dụ:** `MOVK X19, 15, LSL 0`

opcode

hw

immediate

Rd

111100101	00	0000000000000001111	10011
-----------	----	---------------------	-------

C

2

A

0

0

1

F

3

hex

Giá trị thập phân tương ứng của từng trường

485	0	15	19
-----	---	----	----

opcode = 485 (mã lệnh gán hằng số 16-bit vào thanh ghi, các bit còn lại không đổi).

hw = 0 (logic shift left 0 bit (0 halfword)).

immediate = 15 (toán hạng nguồn thứ 2 là giá trị 15).

Rd = 19 (toán hạng đích là thanh ghi X19).

0000000000000000	0000000000000000	0000000011111111	00000000000001111
------------------	------------------	------------------	-------------------

# Ví dụ thao tác với hằng số lớn

- Làm thế nào để thực hiện câu lệnh C sau đây bằng lệnh máy ?

$a = b + 0xABABCD CD;$

- Giả sử:  $a: X19, b: X20$

`ADDI X19, X20, #0xABABCD CD`      *//  $a = b + 0xABABCD CD$  ?*

`MOVZ X9, 0xABAB, LSL 16`      *//  $X9 = 0xABAB0000$*

`MOVK X9, 0xCDCD, LSL 0`      *//  $X9 = 0xABABCD CD$*

`ADD X19, X20, X9`      *//  $a = b + 0xABABCD CD$*



# Nhóm lệnh điều khiển

- LEGv8 đã hỗ trợ các nhóm lệnh xử lý dữ liệu:
  - Lệnh số học & luận lý.
  - Lệnh nạp lưu dữ liệu.
- Ngoài các lệnh xử lý dữ liệu tuần tự, máy tính (computer) còn phải hỗ trợ các lệnh điều khiển quá trình thực thi các lệnh, tương ứng với các cấu trúc điều khiển trên NNLT như:
  - `if ... else ...`
  - `switch ... case ...`
  - `while (...) ...`
  - `for (...) ...`
  - ...

# Lệnh `if` trong C

- 2 loại lệnh `if` trong C

```
if (condition) clause
```

```
if (condition)  
    clause1
```

```
else  
    clause2
```

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

```
if (condition) goto L1;  
    clause2;
```

```
goto L2;
```

```
L1:    clause1;
```

```
L2:
```

```
if (!condition) goto L1;  
    clause1;
```

```
goto L2;
```

```
L1:    clause2;
```

```
L2:
```

- Lệnh `if` thứ 1 có thể được diễn giải như sau:

```
if (!condition) goto L1;  
    clause;
```

```
L1:
```

# Lệnh rẽ nhánh

- Lệnh rẽ nhánh có điều kiện

CBZ register, L1

CBZ nghĩa là “compare and branch if (register is) zero”  
tương ứng với lệnh trong C: if (register == 0) goto L1

CBNZ register, L1

CBNZ nghĩa là “compare and branch if (register is) not zero”  
tương ứng với lệnh trong C: if (register != 0) goto L1

- Lệnh rẽ nhánh không điều kiện

B L1                      tương ứng với lệnh trong C: goto L1

- Ví dụ:

```
if (b == 0)
    a = a + 1;
else
    a = a + 2;
```

```
CBNZ X20, Else
ADDI X19, X19, #1
B End_If
Else:
ADDI X19, X19, #2
End_If:
```

```
CBZ X20, Else
ADDI X19, X19, #1
Else:
ADDI X19, X19, #1
// Be careful !!!
```

# Ví dụ lệnh rẽ nhánh

## • Lệnh C:

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

## • Giả sử:

f: X19, g: X20, h: X21

i: X22, j: X23

## • Chuyển thành lệnh máy LEGv8:

```
SUB    X9, X22, X23
```

```
// X9=0 if i==j
```

```
CBNZ   X9, Else
```

```
// if X9!=0 ~ i!=j
```

```
ADD    X19, X20, X21
```

```
// case i==j, f=g+h
```

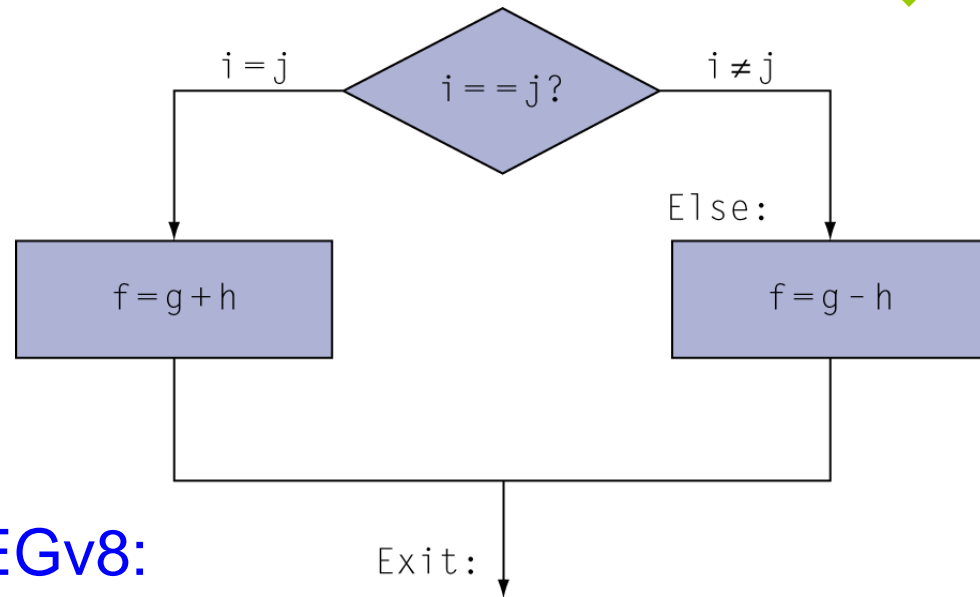
```
B      Exit
```

```
// bypass Else, jump to Exit
```

```
Else: SUB X19, X20, X21
```

```
// case i!=j, f=g-h
```

```
Exit: ...
```



# Các lệnh rẽ nhánh có điều kiện khác

- Các phép so sánh khác như:  $<$ ,  $\leq$ ,  $>$ ,  $\geq$  thì sao ?
- Dựa vào các cờ được bật bởi các lệnh số học có hậu tố S như: ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS.
  - negative (N): kết quả phép tính có MSB là 1.
  - zero (Z): kết quả phép tính bằng 0.
  - overflow (V): kết quả tràn có dấu.
  - carry (C): kết quả tràn không dấu.
- Các lệnh rẽ nhánh có điều kiện dựa vào các cờ trên B.cond
  - B.EQ (equal)
  - B.NE (not equal)
  - B.LT (less than, signed)  
B.LO (lower, unsigned)
  - B.LE (less or equal, signed)  
B.LS (lower or same, unsigned)
  - B.GT (greater than, signed)  
B.HI (higher, unsigned)
  - B.GE (greater or equal, signed)  
B.HS (higher or same, unsigned)

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	$\sim(Z=0 \ \& \ N=V)$	B.LS	$\sim(Z=0 \ \& \ C=1)$
>	B.GT	$(Z=0 \ \& \ N=V)$	B.HI	$(Z=0 \ \& \ C=1)$
≥	B.GE	N=V	B.HS	C=1

# Ví dụ lệnh rẽ nhánh có điều kiện

## B.cond

- Lệnh C: `if (a > b) a += 1;`
- Giả sử: `a:X22, b:X23`
- Chuyển thành lệnh máy LEGv8

– Nếu a và b là số có dấu

`SUBS XZR, X22, X23`

`B.LE Exit`

`ADDI X22, X22, #1`

`Exit:`

– Nếu a và b là số không dấu

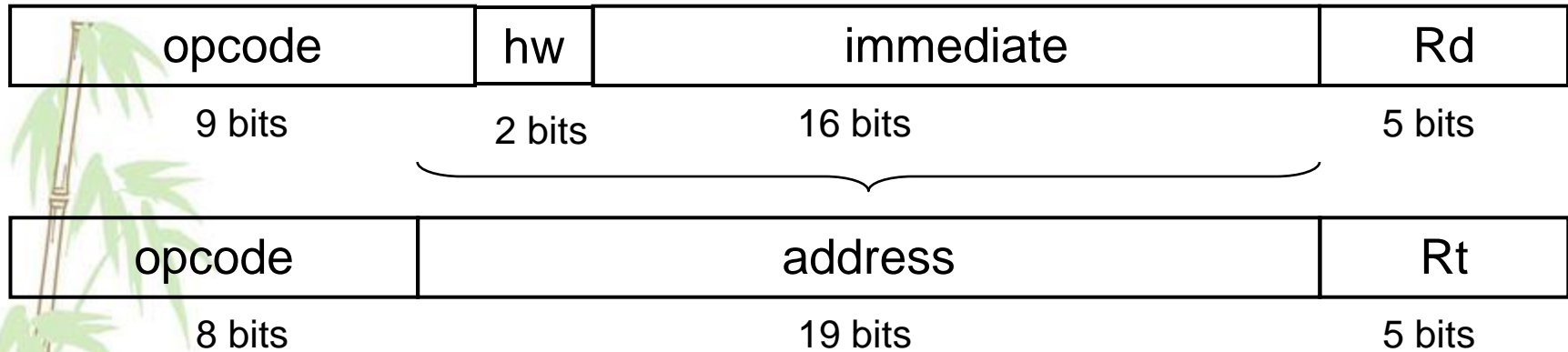
`SUBS XZR, X22, X23`

`B.LS Exit`

`ADDI X22, X22, #1`

`Exit:`

# Cấu trúc lệnh rẽ nhánh có điều kiện CB-Format



- Cú pháp lệnh hợp ngữ:

CBZ/CBNZ Rd, label

B.COND label

opcode mã lệnh nhảy có điều kiện.

address khoảng cách (số lệnh) từ địa chỉ của lệnh rẽ nhánh (giá trị thanh ghi PC hiện tại) tới nhãn label.

$$PC = PC + address * 4$$

Rt - đối với lệnh CBZ/CBNZ, là địa chỉ thanh ghi nguồn cần so sánh.  
- đối với lệnh B.COND, là mã của điều kiện gì.





# Ví dụ cấu trúc lệnh rẽ nhánh có điều kiện CB-Format

opcode	address	Rt
--------	---------	----

8 bits

19 bits

5 bits

40000 SUB X9, X22, X23

40004 CBNZ X9, Exit

40008 ADDI X22, X22, #1

40012 Exit:

opcode	address	Rt
10110101	0000000000000000010	01001
C	2	A
0	1	F
F	F	3 hex

Giá trị thập phân tương ứng của từng trường

181	2	9
-----	---	---

opcode = 181 (mã lệnh rẽ nhánh CBNZ).

address = 2 (khoảng cách tới nhãn Exit là 2 lệnh).

$PC = 40004 + 2 * 4 = 40012$

Rt = 9 (toán hạng nguồn dùng để so sánh là thanh ghi X9).

# Xác định địa chỉ của lệnh rẽ nhánh

## Định vị PC-relative addressing

- Cách tính địa chỉ rẽ nhánh:
  - Nếu không thực hiện rẽ nhánh:
$$PC = PC + 4$$
lệnh thực thi tiếp theo là lệnh kế tiếp trong bộ nhớ.
  - Nếu thực hiện rẽ nhánh:
$$PC = PC + (\text{address} * 4)$$
lệnh thực thi tiếp theo không phải là lệnh kế tiếp trong bộ nhớ.
- Trường `address` là 1 số có dấu. Như vậy, có thể nhảy tới/lui 1 khoảng  $\pm 2^{18}$  words ( $\pm 1$  MB,  $\sim 256.000$  lệnh) từ lệnh đang được thực hiện, đủ đáp ứng hầu hết các yêu cầu nhảy lặp của chương trình.
- Giá trị các trường của lệnh rẽ nhánh có thay đổi không nếu di chuyển mã nguồn ?
- Tuy nhiên, nếu cần nhảy một khoảng lớn hơn thì sao ?

# Cấu trúc lệnh rẽ nhánh không điều kiện B-Format

- Cú pháp lệnh hợp ngữ:

op label



opcode mã lệnh nhảy không điều kiện.

address khoảng cách (số lệnh) từ địa chỉ của lệnh rẽ nhánh (giá trị thanh ghi PC hiện tại) tới nhãn label.

$$PC = PC + address * 4$$

có thể nhảy tới/lui 1 khoảng  $\pm 2^{25}$  words ( $\pm 128$  MB,  $\sim 32.000.000$  lệnh)

Exit: ...

## hex

Giá trị thập phân tương ứng của từng trường

2

$$PC = 40012 + 2 \cdot 4 = 40020$$

# Ví dụ vòng lặp

- Lệnh C:

```
for (i=0; i≤N, i++) { s += i; }
```

- Giả sử:

```
i: X19, N: X20, s: X21
```

- Chuyển thành lệnh máy LEGv8:

```
        ORR    X19, XZR, XZR        // i=0
Loop:   SUBS    XZR, X19, X20        // so sánh i với N
        B.HI    Exit                // thoát nếu i > N
        ADD     X21, X21, X19        // s = s + i
        ADDI    X19, X19, #1         // i++
        B       Loop
```

```
Exit:  ...
```

- Cho biết giá trị trường `address` của các lệnh rẽ nhánh.

- Nếu kiểm tra điều kiện lặp ở trên, ta thực hiện so sánh N với i bằng lệnh `SUBS XZR, X20, X19` thì sao ?

# Ví dụ vòng lặp duyệt mảng

- Lệnh C:

```
while (i < N) { s += A[i]; i += 1; }
```

- Giả sử:

i: X19, N: X20, s: X21, X22: địa chỉ mảng A

- Chuyển thành lệnh máy LEGv8:

```
Loop: SUBS    XZR, X19, X20           // so sánh i với N
      B.HS    Exit                   // thoát nếu i ≥ N
      LSL     X9, X19, #3             // phân tử thứ i tại địa
      ADD     X10, X22, X9            // chỉ thứ i*8 cộng với
      LDUR    X11, [X10, #0]         // địa chỉ bắt đầu mảng A
      ADD     X21, X21, X11           // s = s + A[i]
      ADDI    X19, X19, #1           // i += 1
      B       Loop
```

Exit:

- Cho biết giá trị trường address của các lệnh rẽ nhánh.

# Ví dụ lệnh switch trong C

- Lệnh switch ... case ... trong C

```
switch (k) {  
    case 0: f=i+j; break;    /* k=0 */  
    case 1: f=g+h; break;    /* k=1 */  
    case 2: f=g-h; break;    /* k=2 */  
    case 3: f=i-j; break;    /* k=3 */  
}
```

- Viết lại dưới dạng các lệnh if như sau:

```
if (k==0) f=i+j;  
else if (k==1) f=g+h;  
    else if (k==2) f=g-h;  
        else if (k==3) f=i-j;
```

- Giả sử ánh xạ biến vào thanh ghi như sau:

f: X19, g: X20, h: X21,  
i: X22, j: X23, k: X24

- Hãy biên dịch thành lệnh LEGv8.



# Ví dụ so sánh điều kiện phức tạp

- Điều kiện and:

```
do {  
    i--;  
}  
while (j < 2 && j ≥ i);
```

- Điều kiện or:

```
do {  
    i--;  
}  
while (j < 2 || j ≥ i);
```

- Hãy biên dịch 2 đoạn mã trên thành lệnh LEGv8, giả sử  
i: X19, j: X20.

# Thủ tục trong C

```
main() {  
    int a,b,c;  
    ...  
    c = sum(a,b);  
    ...  
}  
/* khai báo hàm sum */  
int sum (int x, int y) {  
    return x+y;  
}
```

- Khai báo thủ tục, gọi thủ tục và quay về được chuyển thành lệnh máy như thế nào ?
- Tham số của thủ tục được truyền vào như thế nào ?
- Kết quả trả về của thủ tục được truyền ra ngoài như thế nào ?



# Khai báo thủ tục, gọi thủ tục và quay về được chuyển thành lệnh máy như thế nào ?

- Khi gọi thủ tục thì lệnh tiếp theo được thực hiện là lệnh đầu tiên của thủ tục.  
→ Có thể xem tên thủ tục là một nhãn và lời gọi thủ tục là một lệnh nhảy tới nhãn này.
- Sau khi thực hiện xong thủ tục phải quay về thực hiện tiếp lệnh ngay sau lời gọi thủ tục.

**C**

**LEGv8**

```
main() {  
    c=sum(...);  
    ...  
}  
  
int sum (...) {  
    return ...  
}
```

```
B sum    // gọi thủ tục sum  
return:  // địa chỉ quay về  
...  
  
sum:     // khai báo thủ tục sum  
B return // quay về sau khi  
          // thực thi xong thủ tục
```

- Thủ tục `sum` được gọi ở chỗ khác thì quay về thế nào ? <sup>60</sup>

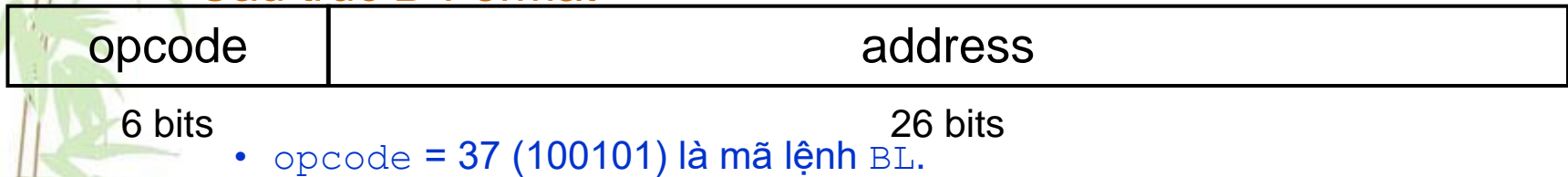
# Các lệnh thủ tục

- **Lệnh gọi thủ tục:**

- **Branch and Link:** BL label

- (1) Link: Lưu địa chỉ của lệnh kế tiếp vào thanh ghi X30 (LR – Link Register).
    - (2) Branch: nhảy tới nhãn label

- **Cấu trúc B-Format**

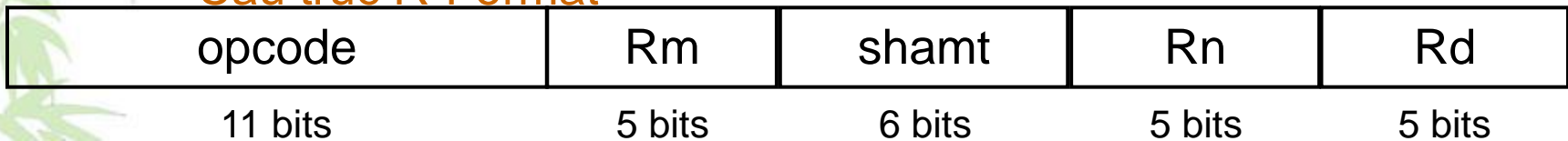


- **Lệnh quay về sau khi thực thi xong thủ tục:**

- **Branch to Register:** BR Register

- Nhảy tới địa chỉ nằm trong thanh ghi Register.
    - Lệnh BR LR nhảy tới địa chỉ trong thanh ghi LR, được lưu trước đó bởi lệnh BL.

- **Cấu trúc R-Format**



- opcode = 1712 (11010110000) là mã của lệnh BR.

- Rd : địa chỉ thanh ghi đích (chứa địa chỉ cần nhảy tới).

# Ví dụ các lệnh thủ tục

```
int main() {  
    ...  
    c=sum(a,b);  
    ...  
}  
int sum(int x, int y) {  
    return x+y;  
}
```

địa chỉ

...		
...		
1008	BL	sum // lưu địa chỉ quay về vào thanh
		// ghi LR=1012 và nhảy tới sum
1012	...	
...		
2000	sum:	// khai báo thủ tục sum
...		
2004	BR	LR // nhảy tới địa chỉ 1012 trong LR

C

LE

G

V

8

# Thanh số của thủ tục và kết quả trả về

```
int main() {
    ...
    c=sum(a,b);
    // a:X19,b:X20,c:X21
}
int sum(int x, int y) {
    return x+y;
}
```

- Sử dụng bộ nhớ hay thanh ghi làm tham số và kết quả trả về ?
  - Thanh ghi nhanh hơn.
  - Các thanh ghi X0 – X7 được sử dụng làm tham số và kết quả trả về.
- Nếu nhiều hơn 8 tham số thì sao ?
  - Tăng số lượng thanh ghi tham số bao nhiêu cho đủ ?

địa chỉ

1000	ORR X0, XZR, X19	// truyền tham số x=a ~ X0=X19
1004	ORR X1, XZR, X20	// truyền tham số y=b ~ X1=X20
1008	BL sum	// lưu địa chỉ quay về vào thanh ghi LR=1012 và nhảy tới sum
1012	ORR X21, XZR, X0	// gán kết quả vào c ~ X21=X0
...		
2000	sum:	// khai báo thủ tục sum
	ADD X0, X0, X1	
2004	BR LR	// nhảy tới địa chỉ 1012 trong LR

C

L

E

G

V

8

# Bài tập

```
main() {  
    int i,j,k,m; // i:X19, j:X20, k:X21, m:X22  
    ...  
    i = mult(j,k); ...  
    m = mult(i,i); ...  
}  
/* khai báo hàm mult */  
int mult (int mcand, int mlier){  
    int product;  
    product = 0;  
    while (mlier > 0) {  
        product = product + mcand;  
        mlier = mlier -1; }  
    return product;  
}
```



# Thủ tục lồng nhau

```
main() {  
    ...  
    ...  
    c = sumSquare(a,b);  
    ...  
}  
  
int sumSquare(int x, int y)  
{  
    return mult(x,x)+ y;  
}  
  
int mult (int x, int y) {  
    ...  
    return ...;  
}
```

```
main:  
    ORR    X0, XZR, X19  
    ORR    X1, XZR, X20  
    BL     sumSquare  
    ORR    X21, XZR, X0  
    ...  
  
sumSquare:  
    ORR    X1, XZR, X0  
    BL     mult  
    ADD    X9, X0, X1  
    ORR    X0, XZR, X9  
    BR     LR  
  
mult:  
    ...  
    BR     LR
```

# Vấn đề thủ tục lồng nhau

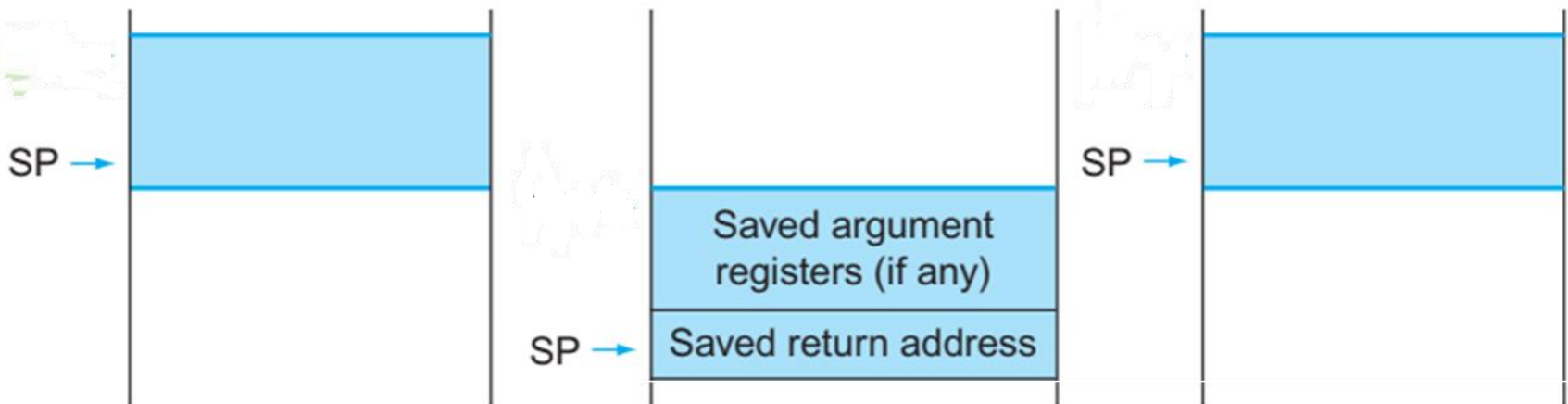
```
main:
    ORR    X0, XZR, X19
    ORR    X1, XZR, X20
    BL     sumSquare
    ADD    X21, XZR, X0
    ...
sumSquare:
    ORR    X1, XZR, X0
    BL     mult
    ADD    X9, X0, X1
    ORR    X0, XZR, X9
    BR     LR
mult:
    ...
    BR     LR
```

- Thủ tục `sumSquare` gọi thủ tục `mult`.
  - Vấn đề
    - Thanh ghi tham số `X1` của thủ tục `sumSquare` bị ghi đè khi truyền tham số `X1` cho thủ tục `mult`.
    - Địa chỉ quay về của thủ tục `sumSquare` trong thanh ghi `LR` bị ghi đè bởi địa chỉ trả về của thủ tục `mult` khi thủ tục này được gọi
    - Như vậy cần phải lưu lại giá trị thanh ghi `X1` và địa chỉ quay về của thủ tục `sumSquare` (trong thanh ghi `LR`) trước khi bị thay đổi để phục hồi sau đó.
- Sử dụng thanh ghi ?
- Bao nhiêu cho đủ ?
- Sử dụng bộ nhớ
- Ngăn xếp (stack).

# Nguyên tắc sử dụng ngăn xếp của LEGv8

- Vùng nhớ ngăn xếp hoạt động theo cơ chế LIFO (Last In First Out)
- Con trỏ ngăn xếp, thanh ghi  $SP$  (X28), được sử dụng để định vị vùng ngăn xếp (lưu địa chỉ bộ nhớ tại đỉnh ngăn xếp).
- Sử dụng ngăn xếp trong thủ tục:
  - Ở đầu thủ tục, khai báo kích thước vùng ngăn xếp cần dùng bằng cách **giảm** giá trị thanh ghi ngăn xếp.
  - Ở cuối thủ tục, sau khi sử dụng ngăn xếp xong, trả lại vị trí ban đầu của vùng ngăn xếp bằng cách **tăng** giá trị thanh ghi ngăn xếp.

High address



Low address

(a) Before procedure call

(b) During procedure call

(c) After procedure call

# Ví dụ thủ tục lồng nhau sử dụng ngăn xếp

```

sumSquare:    // x,y : X0,X1
               SUBI    SP, SP, #16    // khai báo 16 bytes (2 doublewords)
               // ngăn xếp cần dùng cất địa chỉ
               // quay về và thanh ghi tham số X1.
"push"        STUR     LR, [SP, #8]   // cất địa chỉ quay về của thủ tục
               // sumSquare vào ngăn xếp.
"push"        STUR     X1, [SP, #0]   // cất tham số X1(y) vào ngăn xếp
               ORR      X1, XZR, X0    // gán tham số x của mult vào X1
               BL       mult          // gọi thủ tục mult, LR bị ghi đè
"pop"         LDUR     X1, [SP, #0]   // sau khi thực thi xong thủ tục
               // mult, khôi phục X1(y) của thủ
               // tục sumSquare từ ngăn xếp.
               ADD     X9, X0, X1     // mult() + y.
               ORR      X0, XZR, X9    // lưu vào thanh ghi kết quả X0.
"pop"         LDUR     LR, [SP, #8]   // khôi phục địa chỉ quay về của
               // thủ tục sumSquare từ ngăn xếp.
               ADDI    SP, SP, #16    // trả lại vị trí thanh ghi ngăn xếp
               BR       LR

mult:         ...
               BR       LR
    
```

# Nguyên tắc về việc bảo toàn giá trị thanh ghi trong thủ tục

- Các thanh ghi cần được bảo toàn giá trị trong thủ tục:
  - Các thanh ghi biến  $X19 - X25$ .
  - Thanh ghi ngăn xếp  $SP$ .
  - Thanh ghi chứa địa chỉ trả về  $LR$ .
- Các thanh ghi không cần bảo toàn giá trị trong thủ tục:
  - Các thanh ghi tạm  $X9 - X15$ .
  - Thanh ghi tham số, kết quả trả về  $X0 - X7$ .
  - Nếu các thanh ghi này nếu bị thay đổi và cần sử dụng lại trong thủ tục thì lưu lại và phục hồi.

# Trắc nghiệm

caller:     // đọc ghi X19,X0,X1,X9,SP,LR,mem  
          ...     // **cất các thanh ghi vào ngăn xếp?**  
          BL e     // gọi thủ tục e  
          ...     // đọc ghi X19,X0,X1,X9,SP,LR,mem  
          BR LR    // quay về thủ tục gọi r

callee:     // đọc ghi X19,X0,X1,X9,SP,LR,mem  
          ...  
          BR LR    // quay về thủ tục r


**Thủ tục r cần cất các thanh ghi nào vào ngăn xếp trước khi gọi “BL e” ?**

0:   0 of (X19,SP,X0,X1,X9,LR)  
1:   1 of (X19,SP,X0,X1,X9,LR)  
2:   2 of (X19,SP,X0,X1,X9,LR)  
3:   3 of (X19,SP,X0,X1,X9,LR)  
4:   4 of (X19,SP,X0,X1,X9,LR)  
5:   5 of (X19,SP,X0,X1,X9,LR)  
6:   6 of (X19,SP,X0,X1,X9,LR)

# Đáp án

```
caller:    // đọc ghi X19,X0,X1,X9,SP,LR,mem  
...        // cất các thanh ghi vào ngăn xếp?  
BL e      // gọi thủ tục e  
...        // đọc ghi X19,X0,X1,X9,SP,LR,mem  
BR LR     // quay về thủ tục gọi r  
  
callee:   // đọc ghi X19,X0,X1,X9,SP,LR,mem  
...  
BR LR     // quay về thủ tục r
```

Thủ tục r cần cất các thanh ghi nào vào ngăn xếp trước khi gọi “BL e” ?



0:	0 of	(X19, SP, X0, X1, X9, LR)
1:	1 of	(X19, SP, X0, X1, X9, LR)
2:	2 of	(X19, SP, X0, X1, X9, LR)
3:	3 of	(X19, SP, X0, X1, X9, LR)
4:	4 of	(X19, SP, X0, X1, X9, LR)
5:	5 of	(X19, SP, X0, X1, X9, LR)
6:	6 of	(X19, SP, X0, X1, X9, LR)

Không cần  
cất vào  
ngăn xếp

Cần cất vào  
ngăn xếp



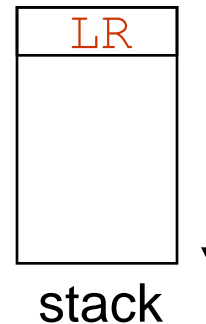
# Cấu trúc cơ bản của thủ tục

## Đầu thủ tục

```
entry_label:  
SUBI SP, SP, framesize  
STUR LR, [SP, #(framesize-8)] // cất địa chỉ trả về  
                                // nếu có gọi thủ tục khác  
// Lưu tạm các thanh ghi khác nếu cần
```

## Thân thủ tục ...

(có thể gọi các thủ tục khác...)



## Cuối thủ tục

```
// Phục hồi các thanh ghi đã lưu tạm nếu cần  
LDUR LR, [SP, #(framesize-8)] // khôi phục địa chỉ  
                                // trả về được lưu ở đầu thủ tục  
ADDI SP, SP, framesize  
BR LR
```

# Ví dụ hàm String Copy

- Ngôn ngữ C


- Chuỗi ký tự kết thúc bằng ký tự null.

```
void strcpy (char x[], char y[])  
{  
    size_t i;  
    i = 0;  
    while ((x[i]=y[i]) != '\0')  
        i += 1;  
}
```

- Hãy biên dịch thành lệnh LEGv8.

# Biên dịch hàm String Copy

strcpy:



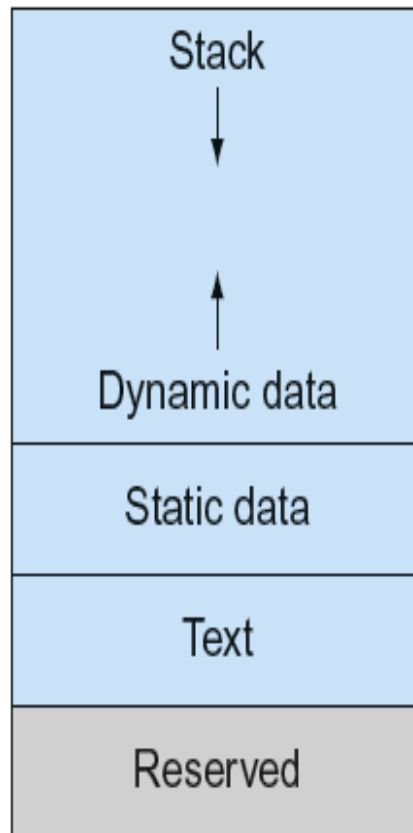
```
        SUBI    SP, SP, #8                // reserve 1 item in stack
        STUR    X19, [SP, #0]            // push X19
        ORR     X19, XZR, XZR            // i=0
L1:     ADD     X10, X19, X1              // X10 = addr of y[i]
        LDURB   X11, [X10, #0]           // X11 = y[i]
        ADD     X12, X19, X0             // X12 = addr of x[i]
        STURB   X11, [X12, #0]           // x[i] = y[i]
        CBZ     X11, L2                  // if y[i] == 0 then exit
        ADDI    X19, X19, #1             // i = i + 1
        B       L1                      // next iteration of loop
L2:     LDUR    X19, [SP, #0]            // restore saved X19
        ADDI    SP, SP, #8              // pop 1 item from stack
        BR     LR                       // and return
```

# Mô hình cấp phát bộ nhớ của C

- Một chương trình C thực thi sẽ được cấp phát các vùng nhớ sau:

Địa chỉ

FP  
SP → 0000 007f ffff fffc<sub>hex</sub>



Vùng nhớ được sử dụng trong quá trình thực thi thủ tục như lưu các biến cục bộ, lưu địa chỉ trả về,...

Vùng nhớ chứa các biến cấp phát động. Ví dụ: con trỏ C được cấp phát động bởi hàm malloc()

Vùng nhớ chứa các toàn cục / biến tĩnh của mỗi chương trình

Mã nguồn chương trình

GP → 0000 0000 1000 0000<sub>hex</sub>

PC → 0000 0000 0040 0000<sub>hex</sub>

# Ví dụ hàm bubble sort

```
void swap(long long int v[], size_t k) {
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (long long int v[], size_t n) {
    size_t i, j;
    for (i = 0; i < n; i += 1)
    {
        for (j=i-1; j>=0 && v[j] > v[j + 1]; j -= 1)
        {
            swap(v, j);
        }
    }
}
```

# Biên dịch hàm swap

```
void swap(long long int v[], size_t k) {  
    long long int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

•Giả sử temp: X9

swap: LSL	X10, X1, #3	// $X10 = k * 8$
ADD	X10, X0, X10	// $X10 = \text{address of } v[k]$
LDUR	X9, [X10, #0]	// $X9 = v[k]$
LDUR	X11, [X10, #8]	// $X11 = v[k+1]$
STUR	X11, [X10, #0]	// $v[k] = X11$ ( $v[k+1]$ )
STUR	X9, [X10, #8]	// $v[k+1] = X9$ ( $v[k]$ )
BR	LR	// return to calling // routine

# Biên dịch hàm sort (1/3)

```
void sort (  
    long long int v[],  
    size_t n)  
{  
    size_t i, j;  
    for (  
        i = 0;  
        i < n;  
        i += 1)  
    {  
        for (  
            j=i-1;  
            j>=0 &&  
            v[j] > v[j + 1];  
            j -= 1)  
        {  
            swap(v, j);  
        }  
    }  
}
```

```
sort:  
SUBI SP,SP,#40 // make room on stack for 5 reg  
STUR X30,[SP,#32] // save LR on stack  
STUR X22,[SP,#24] // save X22 on stack  
STUR X21,[SP,#16] // save X21 on stack  
STUR X20,[SP,#8] // save X20 on stack  
STUR X19,[SP,#0] // save X19 on stack  
ORR X21,XZR,X0 // copy parameter X0 into X21  
ORR X22,XZR,X1 // copy parameter X1 into X22  
ORR X19,XZR,XZR // i = 0  
for1tst:  
SUBS XZR,X19,X1 // compare X19 to X1 (i to n)  
B.GE exit1 // go to exit1 if X19 ≥ X1 (i≥n)  
SUBI X20, X19, #1 // j = i - 1  
for2tst:  
SUBS XZR,X20,XZR // compare X20 to 0 (j to 0)  
B.LT exit2 // go to exit2 if X20 < 0 (j < 0)
```

•Giả sử: i: X19, j: X20



# Biên dịch hàm sort (2/3)

```
void sort (  
    long long int v[],  
    size_t n)  
{  
    size_t i, j;  
    for (  
        i = 0;  
        i < n;  
        i += 1)  
    {  
        for (  
            j = i - 1;  
            j >= 0 &&  
            v[j] > v[j + 1];  
            j -= 1)  
        {  
            swap(v, j);  
        }  
    }  
}
```

```
LSL X10,X20,#3 // reg X10 = j * 8  
ADD X11,X0,X10 // reg X11 = v + (j * 8)  
LDUR X12,[X11,#0] // reg X12 = v[j]  
LDUR X13,[X11,#8] // reg X13 = v[j + 1]  
SUBS XZR,X12,X13 // compare X12 to X13  
B.LE exit2 // go to exit2 if X12 ≤ X13  
ORR X0,XZR,X21 // first swap parameter is v  
ORR X1,XZR,X20 // second swap parameter is j  
BL swap  
SUBI X20,X20,#1 // j -= 1  
B for2tst // branch to test of inner loop  
exit2:  
ADDI X19,X19,#1 // i += 1  
B for1tst // branch to test of outer loop  
exit1:
```

•Giả sử: i: X19, j: X20

# Biên dịch hàm sort (3/3)

```
void sort (  
    long long int v[],  
    size_t n)  
{  
    size_t i, j;  
    for (  
        i = 0;  
        i < n;  
        i += 1)  
    {  
        for (  
            j=i-1;  
            j>=0 &&  
            v[j] > v[j + 1];  
            j -= 1)  
        {  
            swap(v, j);  
        }  
    }  
}
```

```
STUR X19,[SP,#0] // restore X19 from stack  
STUR X20,[SP,#8] // restore X20 from stack  
STUR X21,[SP,#16] // restore X21 from stack  
STUR X22,[SP,#24] // restore X22 from stack  
STUR X30,[SP,#32] // restore LR from stack  
SUBI SP,SP,#40 // restore stack pointer  
BR LR // return to calling routine
```

•Giả sử: i: X19, j: X20



# Ví dụ xóa mảng bằng cách truy xuất mảng vs truy xuất con trỏ

- Mọi người đã từng được dạy rằng sử dụng con trỏ trong C để đạt được hiệu quả cao hơn so với mảng !

<pre>clear1(int array[], int size) {     int i;     for (i = 0; i &lt; size; i += 1)         array[i] = 0; }</pre>	<pre>clear2(int *array, int size) {     int *p;     for (p = &amp;array[0]; p &lt; &amp;array[size];         p = p + 1)         *p = 0; }</pre>
<pre>ORR X9,XZR,XZR // i = 0 loop1: LSL X10,X9,#3 // X10 = i * 8 ADD X11,X0,X10 // X11 = address                 // of array[i]  STUR XZR,[X11,#0] // array[i] = 0 ADDI X9,X9,#1 // i = i + 1 SUBS XZR,X9,X1 // compare i to                 // size B.LT loop1 // if (i &lt; size)             // go to loop1</pre>	<pre>ORR X9,XZR,X0 // p = address of                 // array[0] LSL X10,X1,#3 // X10 = size * 8 ADD X11,X0,X10 // X11 = address                 // of array[size]  loop2: STUR XZR,0[X9,#0] // Memory[p] = 0 ADDI X9,X9,#8 // p = p + 8 SUBS XZR,X9,X11 // compare p to                 // &amp;array[size] B.LT loop2 // if (p &lt; &amp;array[size])             // go to loop2</pre>

# Tóm tắt các lệnh LEGv8 theo nhóm chức năng (1/2)

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD X1, X2, X3	$X1 = X2 + X3$	Three register operands
	subtract	SUB X1, X2, X3	$X1 = X2 - X3$	Three register operands
	add immediate	ADDI X1, X2, 20	$X1 = X2 + 20$	Used to add constants
	subtract immediate	SUBI X1, X2, 20	$X1 = X2 - 20$	Used to subtract constants
	add and set flags	ADDS X1, X2, X3	$X1 = X2 + X3$	Add, set condition codes
	subtract and set flags	SUBS X1, X2, X3	$X1 = X2 - X3$	Subtract, set condition codes
	add immediate and set flags	ADDIS X1, X2, 20	$X1 = X2 + 20$	Add constant, set condition codes
	subtract immediate and set flags	SUBIS X1, X2, 20	$X1 = X2 - 20$	Subtract constant, set condition codes
Logical	and	AND X1, X2, X3	$X1 = X2 \& X3$	Three reg. operands; bit-by-bit AND
	inclusive or	ORR X1, X2, X3	$X1 = X2   X3$	Three reg. operands; bit-by-bit OR
	exclusive or	EOR X1, X2, X3	$X1 = X2 \wedge X3$	Three reg. operands; bit-by-bit XOR
	and immediate	ANDI X1, X2, 20	$X1 = X2 \& 20$	Bit-by-bit AND reg. with constant
	inclusive or immediate	ORRI X1, X2, 20	$X1 = X2   20$	Bit-by-bit OR reg. with constant
	exclusive or immediate	EORI X1, X2, 20	$X1 = X2 \wedge 20$	Bit-by-bit XOR reg. with constant
	logical shift left	LSL X1, X2, 10	$X1 = X2 \ll 10$	Shift left by constant
	logical shift right	LSR X1, X2, 10	$X1 = X2 \gg 10$	Shift right by constant

# Tóm tắt các lệnh LEGv8 theo nhóm chức năng (2/2)

Category	Instruction	Example	Meaning	Comments
Data transfer	load register	LDUR X1, [X2, 40]	$X1 = \text{Memory}[X2 + 40]$	Doubleword from memory to register
	store register	STUR X1, [X2, 40]	$\text{Memory}[X2 + 40] = X1$	Doubleword from register to memory
	load signed word	LDURSW X1, [X2, 40]	$X1 = \text{Memory}[X2 + 40]$	Word from memory to register
	store word	STURW X1, [X2, 40]	$\text{Memory}[X2 + 40] = X1$	Word from register to memory
	load half	LDURH X1, [X2, 40]	$X1 = \text{Memory}[X2 + 40]$	Halfword memory to register
	store half	STURH X1, [X2, 40]	$\text{Memory}[X2 + 40] = X1$	Halfword register to memory
	load byte	LDURB X1, [X2, 40]	$X1 = \text{Memory}[X2 + 40]$	Byte from memory to register
	store byte	STURB X1, [X2, 40]	$\text{Memory}[X2 + 40] = X1$	Byte from register to memory
	load exclusive register	LDXR X1, [X2, 0]	$X1 = \text{Memory}[X2]$	Load; 1st half of atomic swap
	store exclusive register	STXR X1, X3 [X2]	$\text{Memory}[X2] = X1; X3 = 0 \text{ or } 1$	Store; 2nd half of atomic swap
Conditional branch	move wide with zero	MOVZ X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest zeros
	move wide with keep	MOVK X1, 20, LSL 0	$X1 = 20 \text{ or } 20 * 2^{16} \text{ or } 20 * 2^{32} \text{ or } 20 * 2^{48}$	Loads 16-bit constant, rest unchanged
	compare and branch on equal 0	CBZ X1, 25	if $(X1 == 0)$ go to PC + 100	Equal 0 test; PC-relative branch
Unconditional branch	compare and branch on not equal 0	CBNZ X1, 25	if $(X1 != 0)$ go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch
Unconditional branch	branch	B 2500	go to PC + 10000	Branch to target address; PC-relative
	branch to register	BR X30	go to X30	For switch, procedure return
	branch with link	BL 2500	$X30 = \text{PC} + 4; \text{PC} + 10000$	For procedure call PC-relative



# Tóm tắt các loại toán hạng của LEGv8



Name	Example	Comments
32 registers	X0-X30, XZR	Fast locations for data. In LEGv8, data must be in registers to perform arithmetic, register XZR always equals 0.
$2^{62}$ memory words	Memory[0], Memory[4], . . . , Memory[4,611,686,018,427,387,904]	Accessed only by data transfer instructions. LEGv8 uses byte addresses, so sequential doubleword addresses differ by 8. Memory holds data structures, arrays, and spilled registers.





# Tóm tắt các cấu trúc lệnh và các loại định vị của LEGv8

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

## 1. Immediate addressing



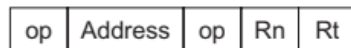
## 2. Register addressing



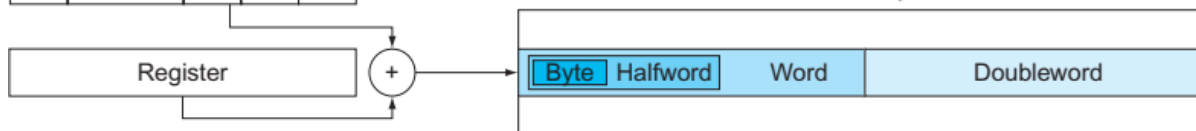
Registers

Register

## 3. Base addressing



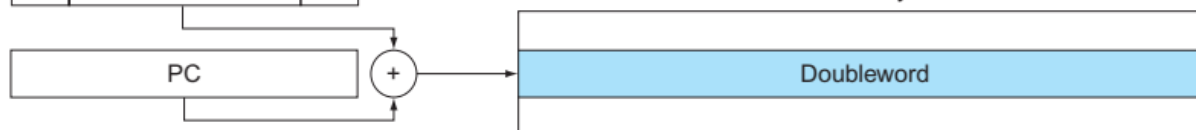
Memory



## 4. PC-relative addressing



Memory





# Giải mã lệnh LEGv8

Instruction	Opcode	Opcode Size	11-bit opcode range		Instruction Format
			Start	End	
B	000101	6	160	191	B - format
STURB	00111000000	11	448		D - format
LDURB	00111000010	11	450		D - format
B.cond	01010100	8	672	679	CB - format
ORRI	1011001000	10	712	713	I - format
EORI	1101001000	10	840	841	I - format
STURH	01111000000	11	960		D - format
LDURH	01111000010	11	962		D - format
AND	10001010000	11	1104		R - format
ADD	10001011000	11	1112		R - format
ADDI	1001000100	10	1160	1161	I - format
ANDI	1001001000	10	1168	1169	I - format
BL	100101	6	1184	1215	B - format
ORR	10101010000	11	1360		R - format
ADDSD	10101011000	11	1368		R - format
ADDIS	1011000100	10	1416	1417	I - format
CBZ	10110100	8	1440	1447	CB - format
CBNZ	10110101	8	1448	1455	CB - format
STURW	10111000000	11	1472		D - format
LDURSW	10111000100	11	1476		D - format
STXR	11001000000	11	1600		D - format
LDXR	11001000010	11	1602		D - format
EOR	11101010000	11	1616		R - format
SUB	11001011000	11	1624		R - format
SUBI	1101000100	10	1672	1673	I - format
MOVZ	110100101	9	1684	1687	IM - format
LSR	11010011010	11	1690		R - format
LSL	11010011011	11	1691		R - format
BR	11010110000	11	1712		R - format
ANDSD	11101010000	11	1872		R - format
SUBSD	11101011000	11	1880		R - format
SUBIS	1111000100	10	1928	1929	I - format
ANDIS	1111001000	10	1936	1937	I - format
MOVK	111100101	9	1940	1943	IM - format
STUR	11111000000	11	1984		D - format
LDUR	11111000010	11	1986		D - format



# Tóm tắt các lệnh LEGv8 với cấu trúc tương ứng



LEGv8 instructions	Name	Format
add	ADD	R
subtract	SUB	R
add immediate	ADDI	I
subtract immediate	SUBI	I
add and set flags	ADDS	R
subtract and set flags	SUBS	R
add immediate and set flags	ADDIS	I
subtract immediate and set flags	SUBIS	I
load register	LDUR	D
store register	STUR	D
load signed word	LDURSW	D
store word	STURW	D
load half	LDURH	D
store half	STURH	D
load byte	LDURB	D
store byte	STURB	D
load exclusive register	LDXR	D
store exclusive register	STXR	D
move wide with zero	MOVZ	IM
move wide with keep	MOVK	IM
and	AND	R
inclusive or	ORR	R
exclusive or	EOR	R
and immediate	ANDI	I
inclusive or immediate	ORRI	I
exclusive or immediate	EORI	I
logical shift left	LSL	R
logical shift right	LSR	R
compare and branch on equal 0	CBZ	CB
compare and branch on not equal 0	CBNZ	CB
branch conditionally	B, cond	CB
branch	B	B
branch to register	BR	R
branch with link	BL	B

## Lệnh giả LEGv8

- Lệnh giả (Pseudo Instruction) là các lệnh hợp ngữ không có cài đặt lệnh máy tương ứng, nhằm mục đích giúp cho việc lập trình hợp ngữ dễ dàng hơn

Pseudo LEGv8	Name	Format
move	MOV	R
compare	CMP	R
compare immediate	CMPI	I
load address	LDA	M

# Tập lệnh ARMv8 đầy đủ

## Các lệnh tính toán số nguyên

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Arithmetic Register	<b>ADD</b>	Add	Logical Immediate	<b>ANDI</b>	Bitwise AND Immediate
	<b>ADDS</b>	Add and set flags		<b>ANDIS</b>	Bitwise AND and set flags Immediate
	<b>SUB</b>	Subtract		<b>ORRI</b>	Bitwise inclusive OR Immediate
	<b>SUBS</b>	Subtract and set flags		<b>EORI</b>	Bitwise exclusive OR Immediate
	<b>CMP</b>	Compare		<b>TSTI</b>	Test bits Immediate
	<b>CMN</b>	Compare negative			
	<b>NEG</b>	Negate	Shift Register Shift Immed	<b>LSL</b>	Logical shift left Immediate
Arithmetic Immediate	<b>NEGS</b>	Negate and set flags		<b>LSR</b>	Logical shift right Immediate
	<b>ADDI</b>	Add Immediate		<b>ASR</b>	Arithmetic shift right Immediate
	<b>ADDS</b>	Add and set flags Immediate		<b>ROR</b>	Rotate right Immediate
	<b>SUBI</b>	Subtract Immediate		<b>LSRV</b>	Logical shift right register
	<b>SUBIS</b>	Subtract and set flags Immediate		<b>LSLV</b>	Logical shift left register
	<b>CMPI</b>	Compare Immediate		<b>ASRV</b>	Arithmetic shift right register
	<b>CMNI</b>	Compare negative Immediate		<b>RORV</b>	Rotate right register
Arithmetic Extended	<b>ADD</b>	Add Extended Register	Move Wide Immed late	<b>MOVZ</b>	Move wide with zero
	<b>ADDS</b>	Add and set flags Extended		<b>MOVK</b>	Move wide with keep
	<b>SUB</b>	Subtract Extended Register		<b>MOVN</b>	Move wide with NOT
	<b>SUBS</b>	Subtract and set flags Extended		<b>MOV</b>	Move register
	<b>CMP</b>	Compare Extended Register	Bit Field Insert & Extract	<b>BFM</b>	Bitfield move
	<b>CMN</b>	Compare negative Extended		<b>SBFM</b>	Signed bitfield move
Arithmetic with Carry	<b>ADC</b>	Add with carry		<b>UBFM</b>	Unsigned bitfield move (32-bit)
	<b>ADCS</b>	Add with carry and set flags		<b>BF I</b>	Bitfield insert
	<b>SBC</b>	Subtract with carry		<b>BFXIL</b>	Bitfield extract and insert low
	<b>SBCS</b>	Subtract with carry and set flags		<b>SBFIZ</b>	Signed bitfield insert in zero
	<b>NGC</b>	Negate with carry		<b>SBFX</b>	Signed bitfield extract
	<b>NGCS</b>	Negate with carry and set flags		<b>UBFIZ</b>	Unsigned bitfield insert in zero
				<b>UBFX</b>	Unsigned bitfield extract
Logical Register	<b>AND</b>	Bitwise AND		<b>EXTR</b>	Extract register from pair
	<b>ANDS</b>	Bitwise AND and set flags	Sign Extend	<b>SXTB</b>	Sign-extend byte
	<b>ORR</b>	Bitwise inclusive OR		<b>SXTH</b>	Sign-extend halfword
	<b>EOR</b>	Bitwise exclusive OR		<b>SXTW</b>	Sign-extend word
	<b>BIC</b>	Bitwise bit clear		<b>UXTB</b>	Unsigned extend byte
	<b>BICS</b>	Bitwise bit clear and set flags		<b>UXTH</b>	Unsigned extend halfword
	<b>ORN</b>	Bitwise inclusive OR NOT	Bit Operation	<b>CLS</b>	Count leading sign bits
	<b>EON</b>	Bitwise exclusive OR NOT		<b>CLZ</b>	Count leading zero bits
	<b>MVN</b>	Bitwise NOT		<b>RBIT</b>	Reverse bit order
	<b>TST</b>	Test bits		<b>REV</b>	Reverse bytes in register
				<b>REV16</b>	Reverse bytes in halfwords
				<b>REV32</b>	Reverses bytes in words



# Tập lệnh ARMv8 đầy đủ

## Các lệnh nạp/lưu dữ liệu

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Unscaled	<b>LDUR</b>	Load register (unscaled offset)	Exclusive	<b>LDXR</b>	Load Exclusive register
	<b>LDURB</b>	Load byte (unscaled offset)		LDXRB	Load Exclusive byte
	<b>LDURSB</b>	Load signed byte (unscaled offset)		LDXRH	Load Exclusive halfword
	<b>LDURH</b>	Load halfword (unscaled offset)		LDXP	Load Exclusive Pair
	<b>LDURSH</b>	Load signed halfword (unscaled offset)		<b>STXR</b>	Store Exclusive register
	<b>LDURSW</b>	Load signed word (unscaled offset)		STXRB	Store Exclusive byte
	<b>STUR</b>	Store register (unscaled offset)		STXRH	Store Exclusive halfword
	<b>STURB</b>	Store byte (unscaled offset)		STXP	Store Exclusive Pair
	<b>STURH</b>	Store halfword (unscaled offset)		LDAXR	Load-acquire Exclusive register
	<b>STURW</b>	Store word (unscaled offset)		LDAXRB	Load-acquire Exclusive byte
	<b>LDA</b>	Load address		LDAXRH	Load-acquire Exclusive halfword
Scaled, Extended, Pre-& Post-Indexed	<b>LDR</b>	Load register	Exclusive Acquire/Release	LDAXP	Load-acquire Exclusive Pair
	<b>LDRB</b>	Load byte		STLXR	Store-release Exclusive register
	<b>LDRSB</b>	Load signed byte		STLXRB	Store-release Exclusive byte
	<b>LDRH</b>	Load halfword		STLXRH	Store-release Exclusive halfword
	<b>LDRSH</b>	Load signed halfword		STLXP	Store-release Exclusive Pair
	<b>LDRSW</b>	Load signed word		LDP	Load Pair
	<b>STR</b>	Store register		LDPSW	Load Pair signed words
	<b>STRB</b>	Store byte		STP	Store Pair
	<b>STRH</b>	Store halfword		ADRP	Compute address of 4KB page at a PC-relative offset
			PC	ADR	Compute address of label at a PC-relative offset

# Tập lệnh ARMv8 đầy đủ

## Các lệnh rẽ nhánh, thủ tục

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Conditional Branch	<b>B.cond</b>	Branch conditionally	Conditional Select	CSEL	Conditional select
	<b>CBNZ</b>	Compare and branch if nonzero		CSINC	Conditional select increment
	<b>CBZ</b>	Compare and branch if zero		CSINV	Conditional select inversion
	<b>TBNZ</b>	Test bit and branch if nonzero		CSNEG	Conditional select negation
	<b>TBZ</b>	Test bit and branch if zero		<i>CSET</i>	Conditional set
Unconditional Branch	<b>B</b>	Branch unconditionally		<i>CSETM</i>	Conditional set mask
	<b>BL</b>	Branch with link		<i>CINC</i>	Conditional increment
	<b>BLR</b>	Branch with link to register		<i>CINV</i>	Conditional invert
	<b>BR</b>	Branch to register		<i>CNEG</i>	Conditional negate
	<b>RET</b>	Return from subroutine	Conditional Compare	CCMP	Conditional compare register
				CCMPI	Conditional compare immediate
				CCMN	Conditional compare negative register
				CCMNI	Conditional compare negative immediate

## • Chương 2

- David A. Patterson, John L. Hennessy. (2016). ***Computer Organization and Design ARM Edition: The Hardware Software Interface*** (1st ed.). Oxford: Morgan Kaufmann.