HỆ THỐNG MÁY TÍNH

Ths. Lê Viết Long

Bài 05 - Bộ lệnh LEGv8

Giới thiệu

LEGv8 là một tập hợp con của kiến trúc tập lệnh ARMv8, được tạo ra nhằm phục vụ giáo dục

Kiến trúc ARMv8: Là một kiến trúc tập lệnh 64-bit, hỗ trợ các thiết bị hiện đại như điện thoại thông minh và máy tính bảng.

Đơn giản hóa: LEGv8 loại bỏ các chi tiết phức tạp, chỉ giữ lại các phần quan trọng để dễ học và hiểu.

Mã nguồn mở: LEGv8 là dự án mã nguồn mở, cho phép cộng đồng tham gia đóng góp và phát triển tiếp theo.

Đặc điểm của LEGv8

- **RISC (Reduced Instruction Set Computing):** LEGv8 dựa trên kiến trúc RISC, nghĩa là nó có một tập hợp lệnh nhỏ gọn và dễ hiểu.
- **64-bit architecture:** LEGv8 là một phần của kiến trúc ARMv8, hỗ trợ cả chế độ 32-bit và 64-bit, nhưng thường tập trung vào chế độ 64-bit.
- **Bộ lệnh LEGv8** bao gồm các nhóm lệnh như:
  - Tính toán số học
  - Lệnh Logic
  - Lệnh di chuyển dữ liệu
  - Lệnh điều khiển rẽ nhánh

Tập thanh ghi

- **Thanh ghi tổng quát:** LEGv8 cung cấp 32 thanh ghi tổng quát kích thức 64 bit (DoubleWord) (X0-X31)
- Ngoài ra còn có 32 thanh ghi con kích thước 32 bit (Word) (W0-W31)
- → Tính toán nhanh hơn, hỗ trợ format lệnh 32 bit

- **Thanh ghi cờ:** Flag Register (NZCV)
  - Bộ flags gồm Negative (N), Zero (Z), Carry (C) và Overflow (V).
- **Thanh ghi số thực:**
  - 32 bit: S0-S31
  - 64 bit: D0-D31

Tập thanh ghi

- **X0-X7:** Làm đối số / kết quả trả về của hàm
- **X8:** Chứa (vị trí) địa chỉ kết quả trả về
- **X9 – X15**: Thanh ghi tạm
- **X16 – X17** (IP0 – IP1): Thanh ghi tạm của linker hoặc cũng dung làm thanh ghi tạm cho các trường hợp khác.
- **X18**: platform register cho code không phụ thuộc flatform hoặc làm thanh ghi tạm.
- **X19 – X27**: Thanh ghi lưu trữ
- **X28 (SP):** Trỏ đến đỉnh ngăn xếp (stack pointer).
- **X29 (FP):** Trỏ đến khung trang (frame pointer).
- **X30 (LR):** Link register (địa chỉ quay về)
- **XZR (X31):** Chứa hằng số 0

Lệnh tính toán số học

Cú pháp:

opt opr, opr1, opr2

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

Phép cộng và Trừ

- **Phép cộng:**
  - ADD X1,X2,X3 *// X1 = X2 + X3*
- **Phép trừ:**
  - SUB X1,X2,X3 *// X1 = X2 - X3*
- **Phép cộng hằng số:**
  - ADDI X1,X2,#20 *// X1 = X2 + 20*
- **Phép trừ hằng số:**
  - SUBI X1,X2,#20 *// X1 = X2 - 20*

Phép cộng và trừ

- **Các phép tính thay đổi cờ**
  - ADDS X1,X2,X3 *// X1 = X2 + X3*
  - SUBS X1,X2,X3 *// X1 = X2 - X3*
  - ADDIS X1,X2,#20 *// X1 = X2 + 20*
  - SUBIS X1,X2,#20 *// X1 = X2 - 20*

Negative(N)	
Zero (Z)	
Overflow (V)	
Carry (C)	

Ví dụ 1

- Chuyển thành lệnh LEGv8 từ lệnh C:

a = b + c + d – e

- Chia nhỏ thành nhiều lệnh LEGv8:

```
ADD    X0, X1, X2          //  a = b + c
ADD    X0, X0, X3          //  a = a + d
SUB    X0,X0, X4           //  a = a – e
```

- Tại sao dùng nhiều lệnh hơn C?

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

Ví dụ 2

- Chuyển thành lệnh LEGv8 từ lệnh C:

f = (g + h) – (i + j)

- Chia nhỏ thành nhiều lệnh LEGv8:

```
ADD    X9, X1, X2          //  temp1 = g + h
ADD    X10, X3, X4         //  temp2 = i + j
SUB    X0, X9, X10         //  f = temp1 – temp2
```

Phép nhân và chia số nguyên

- **Phép nhân:**
  - MUL X1,X2,X3 *// X1 = X2 * X3 (64 bit thấp của 128 bit KQ)*
  - SMULH X1,X2,X3 *// X1 = X2 * X3 (64 bit cao của 128 bit KQ có dấu)*
  - SMULL X1,X2,X3 *// X1 = X2 * X3 (64 bit thấp của 128 bit KQ có dấu)*
- **Phép chia**
  - SDIV X1,X2,X3 *// X1 = X2 / X3 (Phép chia 2 số có dấu)*
  - UDIV X1,X2,X3 *// X1 = X2 / X3 (Phép chia 2 số không dấu)*

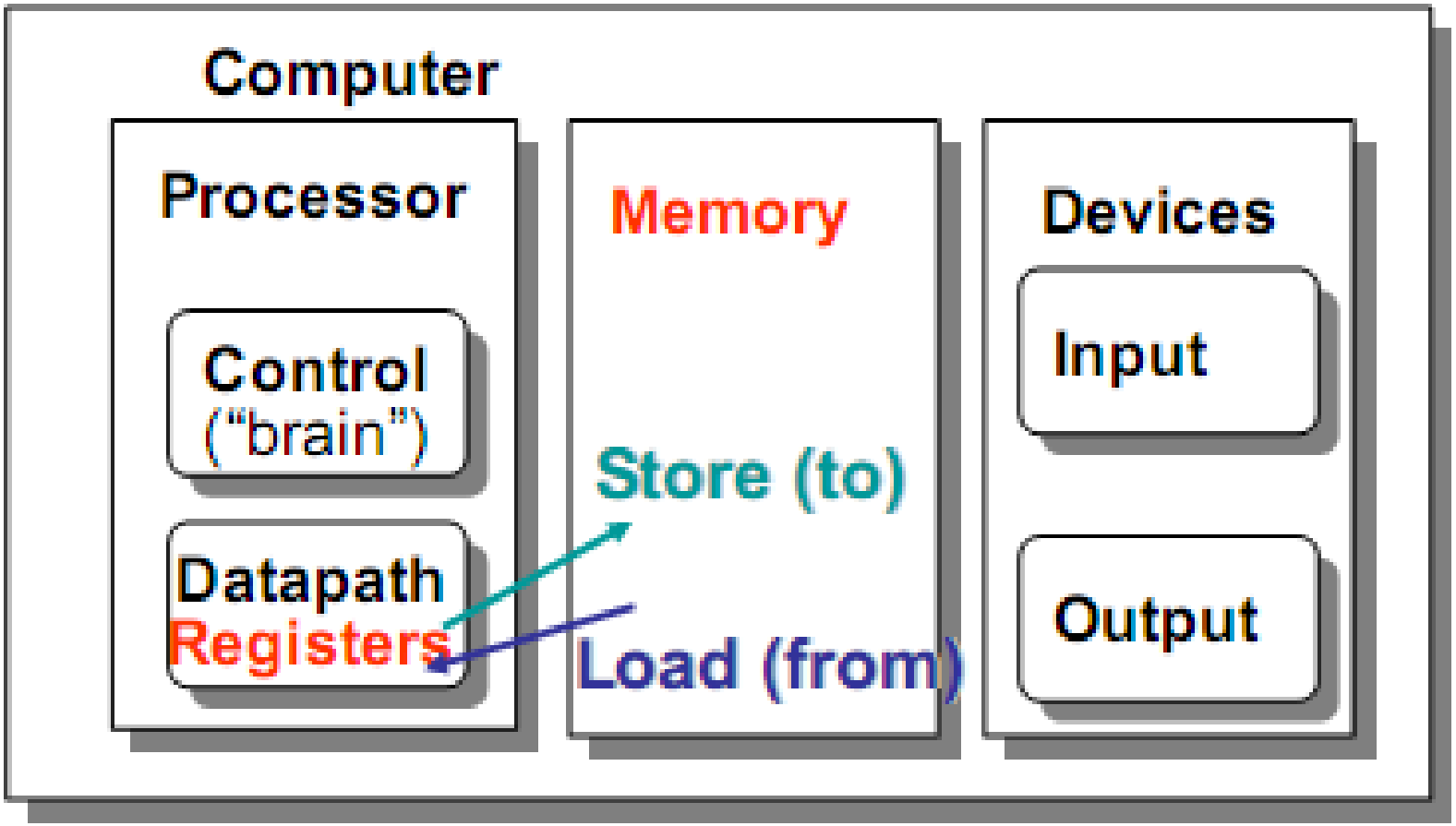
Lệnh di chuyển dữ liệu

- Một số nhận xét:
  - Ngoài các biến đơn, còn có các biến phức tạp thể hiện nhiều kiểu cấu trúc dữ liệu khác nhau, ví dụ như array
  - Các cấu trúc dữ liệu phức tạp có số phần tử dữ liệu nhiều hơn số thanh ghi của CPU → làm sao lưu ??

  → Lưu phần nhiều data trong RAM, chỉ load 1 ít vào thanh ghi của CPU khi cần xử lý

  - Vấn đề lưu chuyển dữ liệu giữa thanh ghi và bộ nhớ ?

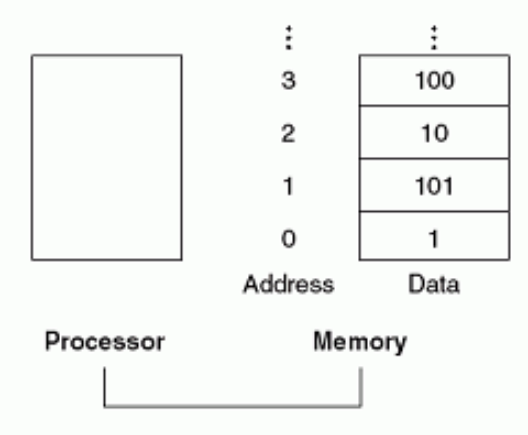
  → Nhóm lệnh lưu chuyển dữ liệu (data transfer)


Computer
Processor
Control
("brain")
Datapath
Registers
Memory
Store (to)
Load (from)
Devices
Input
Output

Bộ nhớ chính

- Có thể được xem như là array 1 chiều rất lớn, mỗi phần tử là 1 ô nhớ có kích thước bằng nhau
- Các ô nhớ được đánh số thứ tự từ 0 trở đi

  → Gọi là địa chỉ (address) ô nhớ
- Để truy xuất dữ liệu trong ô nhớ cần phải cung cấp địa chỉ ô nhớ đó



Cấu trúc lệnh

- Cú pháp:

  opt opr, [opr1, #opr2]

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

Hai thao tác chính

- Load register: Nạp 1 doubleword dữ liệu, từ bộ nhớ vào 1 thanh ghi trên CPU

LDUR X1, [X2, #40]

Gán X1 bằng giá trị tại ô nhớ có địa chỉ (X2 + 40)

- Store register: Lưu 1 doubleword dữ liệu, từ thanh ghi trên CPU, ra bộ nhớ

STUR X1, [X2, #40]

Lưu giá trị trong thanh ghi X1 vào ô nhớ có địa chỉ (X2 + 40)

Lưu ý

- X2 trong ví dụ trên được gọi là thanh ghi cơ sở (base register) thường dùng để lưu địa chỉ bắt đầu của mảng / cấu trúc
- 40 gọi là độ dời (offset) thường dùng để truy cập các phần tử mảng hay cấu trúc

Ví dụ 1

- Giả sử A là 1 array gồm 100 dobuleword với địa chỉ bắt đầu (địa chỉ nền – base address) chứa trong thanh ghi X1. Giá trị các biến g, h lần lượt chứa trong các thanh ghi X2 và X3
- Hãy chuyển thành mã hợp ngữ LEGv8:

g = h + A[8]

Phần tử A[8] có giá trị offset là 64 (8 x 8)

- Trả lời:

```
LDUR    X9, [X1, #64]     // X9 = A[8]
ADD     X2, X3, X9        // X2 = X3 + X9
```

Ví dụ 2

- Hãy chuyển thành mã hợp ngữ LEGv8:

A[12] = h - A[8]

  - A[8] có giá trị offset là 64 (8 x 8)
  - A[12] có độ dời offset là 96 (12 x 8)

- **Trả lời:**

```
LDUR    X9, [X1, #64]      // Chứa A[8] vào X9
SUB     X9, X3, X9         // X9 = X3 – X9
STUR    X9, [X1, #96]      // Kết quả X9 vào A[12]
```

Mở rộng: Load, Store 1 Word

- Ngoài việc hỗ trợ Load, Store 1 DoubleWord (LDUR, STUR), LEGv8 còn hỗ trợ load, store 1 Word
  - Load word: LDURSW
  - Store word: STURW
  - Cú pháp lệnh tương tự LDUR, STUR
- Ví dụ:

```
LDURSW   X2, [X1,#40]
```

Lệnh này nạp giá trị 1 word tại ô nhớ có địa chỉ (X1 + 40) vào 4 byte thấp của thanh ghi X2

Mở rộng: Load, Save 2 byte (1/2 Word)

- LEGv8 còn hỗ trợ load, store 1/2 word (2 byte)
  - Load half: LDURH
  - Store half: STURH
  - Cú pháp lệnh tương tự LDUR, STUR
- Ví dụ:

```
LDURH   X2, [X1,#40]
```

Lệnh này nạp giá trị 2 byte tại ô nhớ có địa chỉ (X1 + 40) vào 2 byte thấp của thanh ghi X2

Mở rộng: Load, Store 1 byte

- LEGv8 còn hỗ trợ load, store 1byte
  - Load half: LDURB
  - Store half: STURB
  - Cú pháp lệnh tương tự LDUR, STUR
- Ví dụ:

```
LDURB   X2, [X1,#40]
```

Lệnh này nạp giá trị 1 byte tại ô nhớ có địa chỉ (X1 + 40) vào 2 byte thấp của thanh ghi X2

Load, Store độc quyền

- LEGv8 còn hỗ trợ load, store độc quyền trong trường có nhiều luồng, hoặc tiến trình cùng truy cập vùng nhớ.
  - Load exclusive register: LDXR
  - Store exclusive register : STXR
  - Cú pháp lệnh tương tự LDUR, STUR
- Ví dụ:

```
LDXR   X2, [X1,#40]
```

Lệnh này nạp giá trị 8 byte tại ô nhớ có địa chỉ (X1 + 40) vào thanh ghi X2

Gán giá trị vào thanh ghi

- **Move wide with Zero**: sau khi gán giá trị vào thanh ghi, các bit còn lại của thanh ghi sẽ được gán bằng 0

```
MOVZ X1, #20, LSL #0
```

Lệnh này có nghĩa là gán x1 = 20 * 2^0 (LSL: Left shift Logical với giá trị dịch là 0, 16, 32 hoặc 48)

- **Move wide with keep**: sau khi gán giá trị vào thanh ghi, các bit còn lại của thanh ghi sẽ được giữ nguyên

```
MOVK X1, #20, LSL #0
```

- **Gán giá trị giữa 2 thanh ghi**:

```
MOV X1, X2                    // X1 = X2
```

Lệnh này có nghĩa là gán x1 = 20 * 2^0 (LSL: Left shift Logical với giá trị dịch là 0, 16, 32 hoặc 48)

- **Gán địa chỉ biến vào thanh ghi**:

```
ADRP x0, num1                 // Đưa địa chỉ cơ sở của biến num1 vào thanh ghi x0
ADD x0, x0, :lo12:num1        // Cộng offset của biến num1 vào thanh ghi x0
```

Phép toán luận lý

- Cú pháp:

  opt opr, opr1, opr2

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

Phép toán luận lý

- LEGv8 hỗ trợ 3 nhóm lệnh cho các phép toán luận lý trên bit:
  - AND, ORR, EOR (xor): Toán hạng nguồn thứ 2 (opr2) phải là thanh ghi
  - ANDI, ORRi, EORI: Toán hạng nguồn thứ 2 (opr2) là hằng số
  - Các lệnh tác động Flags (NZCV): ANDS, ANDIS
- Lưu ý: LEGv8 không hỗ trợ trực tiếp phép luận lý NOT
- Để thực hiện phép not ta có thể thực hiện xor thanh ghi đó với giá trị toàn bit 1.
- Ví dụ:

```
MOV X2, 0xFFFFFFFFFFFFFFFF
EOR X1, X1, X2
```

Phép dịch luận lý

- Cú pháp:

  opt opr, opr1, opr2

  - opt (operator): Tên thao tác
  - opr (operand): Thanh ghi (toán hạng đích) chứa kết quả
  - opr1 (operand 1): Thanh ghi (toán hạng nguồn 1)
  - opr2 (operand 2): Hằng số < 64 (Số bit dịch)

Phép dịch luận lý

- LEGv8 hỗ trợ 2 nhóm lệnh cho các phép dịch luận lý trên bit:
  - Dịch luận lý
    - Dịch trái (LSL –left shift logical): Thêm vào các bit 0 bên phải
    - Dịch phải (RSL – right shift logical): Thêm vào các bit 0 bên trái

Ví dụ:

```
LSL X1,X1,#10  // Ý nghĩa: X1 = X1 << 10
```

  - Dịch số học
    - Không có dịch trái số học
    - Dịch phải (RSA – right shift arithmetic): Thêm các bit = giá trị bit dấu bên trái

Phần 4: Rẽ nhánh

- Tương tự lệnh if trong C: Có 2 loại
  - if (condition) clause
  - if (condition)

    clause1

    else

    clause2
- Lệnh if thứ 2 có thể diễn giải như sau:

```
if (condition)   goto L1   // if → Làm clause1
clause2                    // else → Làm clause2
goto L2                    // Làm tiếp các lệnh khác
L1:     clause1
L2:   …
```

Rẽ nhánh trong LEGv8

- Rẽ nhánh có điều kiện
  - CBZ X1, label // if (X1 == 0) goto label
  - CBNZ X1, label // if (X1 != 0) goto label
- So sánh 2 giá trị với nhau ?
  - CMP opr1, opr2 // opr1: thanh ghi, opr2: thanh ghi / hằng số
  - Kiểm tra kết quả so sánh bằng lệnh nhảy có điều kiện:

    B.Cond label //If(Cond) goto label

    Ví dụ:

    CMP X1, X2

    B.EQ label
- Rẽ nhánh không điều kiện

  B label // goto label

  BR X1 //goto address in X1

  BL label // X30 = PC + 4, goto label

Cond	Ý nghĩa
EQ	X1 == X2
NE	X1 != X2
GT (unsigned: HI)	X1 > X2
LT (unsigned: LO)	X1 < X2
GE (unsigned: HS)	X1 >= X2
LE (unsigned: LS)	X1 <= X2

Ví dụ

- Biên dịch câu lệnh sau trong C thành lệnh hợp ngữ LEGv8:

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

  else f = g – h;

- Ánh xạ biến f, g, h, i, j tương ứng vào các thanh ghi: X0, X1, X2, X3, X4
- Lệnh LEGv8:

```
                CMP X3, X4
                B.EQ  TrueCase      // if(i == j) goto TrueCase
                SUB  X0, X1, X2     // f = g – h (false)
                B     Fin           // goto "Fin" label
TrueCase:       ADD  X0, X1, X2     // f = g + h (true)
Fin:            …
```

Xử lý vòng lặp

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

```
do {
        g = g + A[i];
        i  = i + j;
} while (i != h);
```

- Ta có thể viết lại:

```
Loop:       g = g + A[i];
            i  = i + j;
            if (i != h) goto Loop;
```

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

Xử lý vòng lặp

- Ánh xạ biến vào các thanh ghi như sau:

g	h	i	j	base address of A
X1	X2	X3	X4	X5

- Trong ví dụ trên có thể viết lại thành lệnh LEGv8 như sau:

```
Loop:   LSL    X9, X3, 3        # X9 = i * 2^3
        ADD    X9, X9, X5       # X9 = addr A[i]
        LDUR   X9, [X9,#0]      # X9 = A[i]
        ADD    X1, X1, X9       # g = g + A[i]
        ADD    X3, X3, X4       # i = i + j
        CMP X3,X2
        B.NE   Loop             # if (i != j) goto Loop
```

Xử lý vòng lặp

- Tương tự cho các vòng lặp phổ biến khác trong C:
  - while
  - for
  - do…while
- Nguyên tắc chung:
  - Viết lại vòng lặp dưới dạng goto
  - Sử dụng các lệnh LEGv8 rẽ nhánh có điều kiện

Trình con (Thủ tục)

- LEGv8 hỗ trợ 1 số thanh ghi để lưu trữ dữ liệu cho thủ tục:
  - Đối số input (argument input): X0 - X7
  - Kết quả trả về (return …): X0 - X1
  - Các thanh ghi không bảo toàn: X0-X18
  - Các thanh ghi bảo toàn: X19-X28
  - Địa chỉ quay về LR (link return): X30
- Gọi hàm trong LEGv8: BL Ten_Ham
- Nếu có nhu cầu lưu nhiều dữ liệu (đối số, kết quả trả về, biến cục bộ) hơn số lượng thanh ghi kể trên?

→ Bao nhiêu thanh ghi là đủ ?

→ Sử dụng ngăn xếp (stack)

Thủ tục lồng nhau

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

```
int sumSquare (int x, int y)
{
        return mult (x, x) + y;
}
```

- Thủ tục sumSquare sẽ gọi thủ tục mult trong thân hàm của nó
- Vấn đề:
  - Địa chỉ quay về của thủ tục sumSquare lưu trong thanh ghi X30 sẽ bị ghi đè bởi địa chỉ quay về của thủ tục mult khi thủ tục này được gọi!
  - Như vậy cần phải lưu lại (backup) trong bộ nhớ chính địa chỉ quay về của thủ tục sumSquare (trong thanh ghi X30) trước khi gọi thủ tục mult

→ Sử dụng ngăn xếp (Stack)

Ngăn xếp (Stack)

- Ngăn xếp gồm nhiều ô nhớ kết hợp (vùng nhớ) nằm trong bộ nhớ chính
- Cấu trúc dữ liệu lý tưởng để chứa tạm các giá trị trong thanh ghi
  - Thường chứa địa chỉ trả về, các biến cục bộ của trình con, nhất là các biến có cấu trúc (array, list…) không chứa vừa trong các thanh ghi trong CPU
- Được định vị và quản lý bởi **stack pointer**
  - Đưa dữ liệu từ thanh ghi vào stack:

```
SUB SP, SP, #8
STR X0, [SP]                    // SP = SP – 8, Lưu X0 vào vị trí SP,
```

  - Lấy dữ liệu từ stack chép vào thanh ghi

```
LTR X0, [SP]                    // Gán X0 = giá trị tại SP, SP = SP + 8
ADD SP, SP, #8
```

- Trong LEGv8 dành sẵn 1 thanh ghi X28 (hoặc SP) để lưu trữ stack pointer

Cấu trúc của thủ tục

- **Đầu thủ tục:**

```
Procedure_Label:
        SUBI SP, SP, #8
        STR   LR, [SP]                  // cất LR (kích thước 8 byte) vào stack (push)
        // Lưu tạm các thanh ghi khác (nếu cần)
```

- **Thân thủ tục:**

```
        // Xử lý thủ tục
        BL  other_procedure   // Gọi các thủ tục khác (nếu cần)
```

- **Cuối thủ tục:**

```
        # khôi phục các thanh ghi khác (nếu cần)
        LTR   LR, [SP]                  //khôi phục LR
        ADDI SP, SP, #8
        RET                             //Quay về
```

Cấu trúc chương trình LEGv8


```
.section .data
        var1:   .word   5   // Biến toàn cục khởi tạo
.section .bss
        var2:   .space  100  // Biến toàn cục không khởi tạo
.section .text
.global _start      // Bắt đầu chương trình
_start:
        // Mã chương trình
```

Kiểu dữ liệu

- **Kiểu tĩnh (khai báo trong .section .data)**
  - Số nguyên: byte (1byte), hword (2bye), word (4byte), dword (8 byte)
  - Số thực: float (single floating point), double (double floating point)
  - Chuỗi: ascii, asciz
  - **Kiểu động (khai báo trong .section .bss):** .skip, .space
- **Ví dụ**

```
.section .data
        var1: .word 42              // Khai báo biến var1 có giá trị 42
        var2: .dword 100            // Khai báo biến var2 có giá trị 100
        array: .byte 1, 2, 3, 4     // Khai báo mảng array gồm 4 phần tử
        array2: .half 1000, 2000, 3000, 4000        // Khai báo mảng array2 gồm 4 phần tử
        float1: .float 3.14         // Khai báo biến float1 có giá trị 3.14
        string: .ascii "Hello, LEGv8!"    // Khai báo biến string có giá trị là chuỗi "Hello, LEGv8!"
.section .bss
        str: .space 100             // khai báo chuỗi 100 byte
        arr: .skip 100              // Khai báo arrary 100 byte
```

Hello.asm using library C


```
.extern printf
.section .data
        tb1:  .asciz "Hello, world!\n"
.section .text
.global main
main:
        // Đưa con trỏ đến chuỗi vào x0
        adrp x0, tb1          // Đưa địa chỉ cơ sở của biến tb1 vào thanh ghi x0
        add x0, x0, :lo12:tb1          // Cộng offset của biến tb vào thanh ghi x0
        bl printf                       // Gọi hàm printf

        // Thoát chương trình
        mov x0, #0            // Giá trị trả về của hàm là 0
        mov x8, #93           // khởi tạo x8 = syscall exit
        svc #0                // gọi syscall
```

Biên dịch và chạy chương trình

- Môi trường: Linux (64 bit)
- Hiện tại chưa có simulator hỗ trợ chạy LEGv8

➔ Cài đặt và giả lập bộ lệnh ARM64 (QEMU)

Lưu ý: Một số lệnh trong LEGv8 không hỗ trợ trong ARM64

sudo apt install gcc-9-aarch64-gnu

sudo apt install qemu

- Biên dịch chương trình

aarch64-linux-gnu-as hello.asm -o hello.o

aarch64-linux-gnu-gcc-9 hello.o -o hello.elf -lc -static

- Chạy chương trình

qemu-aarch64 ./hello.elf

Phụ lục 1 – Tóm tắt

Name	Special Name	Register Number	Usage	Preserved on call?
X0-X7		0-7	Arguments/Results	No
X8		8	Indirect result location register	No
X9-X15		9-15	Temporary registers	No
X16	IP0	16	First intra-procedure-call scratch register; other times used as temporary register	No
X17	IP1	17	Second intra-procedure-call scratch register; other times used as temporary register	No
X18		18	Platform register for platform independent code; otherwise a temporary register	No
X19-X27		19-27	Callee-saved registers	Yes
X28	SP	28	Stack pointer	Yes
X29	FP	29	Frame pointer	Yes
X30	LR	30	Link Register (return address)	Yes
XZR		31	The constant value 0	N.A.

Arithmetics instructions

Instruction	Format	Example		Meaning
Add	R	ADD	X1, X2, X3	X1 = X2 + X3
Subtract	R	SUB	X1, X2, X3	X1 = X2 – X3
Add immediate	I	ADDI	X1, X2, #20	X1 = X2 + 20
Subtract immediate	I	SUBI	X1, X2, #20	X1 = X2 – 20
Add and set flags*	R	ADDS	X1, X2, X3	X1 = X2 + X3
Subtract and set flags*	R	SUBS	X1, X2, X3	X1 = X2 – X3
Add immediate and set flags*	I	ADDIS	X1, X2, #20	X1 = X2 + 20
Subtract immediate and set flags*	I	SUBIS	X1, X2, #20	X1 = X2 – 20

Data Transfer

Instruction	Format	Example		Meaning
Load register*	D	LDUR	X1, [X2, #40]	X1 = Memory[X2 + 40]
Store register*	D	STUR	X1, [X2, #40]	Memory[X2 + 40] = X1
Load signed word	D	LDURSW	X1, [X2, #40]	X1 = Memory[X2 + 40]
Store word	D	STURW	X1, [X2, #40]	Memory[X2 + 40] = X1
Load halfword	D	LDURH	X1, [X2, #40]	X1 = Memory[X2 + 40]
Store halfword	D	STURH	X1, [X2, #40]	Memory[X2 + 40] = X1
Load byte	D	LDURB	X1, [X2, #40]	X1 = Memory[X2 + 40]
Store byte	D	STURB	X1, [X2, #40]	Memory[X2 + 40] = X1
Load exclusive register	D	LDXR	X1, [X2, #0]	X1 = Memory[X2]
Store exclusive register	D	STXR	X1, X3, [X2]	Memory[X2] = X1; X3=0 or 1
Move wide with zeros	IM	MOVZ	X1, #20, LSL #0	X1 = 20 (LSL shift can be 0, 16, 32 or 48)
Move wide with keep	IM	MOVK	X1, #20, LSL #0	X1 = 20 (LSL shift can be 0, 16, 32 or 48)

Logical

Instruction	Format	Example		Meaning
Bitwise AND	R	AND	X1, X2, X3	X1 = X2 & X3
Bitwise AND and set flags	R	ANDS	X1, X2, X3	X1 = X2 & X3
Bitwise inclusive OR	R	ORR	X1, X2, X3	X1 = X2 | X3
Bitwise exclusive OR	R	EOR	X1, X2, X3	X1 = X2 ^ X3
Bitwise AND Immediate	I	ANDI	X1, X2, #20	X1 = X2 & 20
Bitwise AND Immediate and set flags	I	ANDIS	X1, X2, #20	X1 = X2 & 20
Bitwise inclusive OR Immediate	I	ORRI	X1, X2, #20	X1 = X2 | 20
Bitwise exclusive OR Immediate	I	EORI	X1, X2, #20	X1 = X2 ^ 20
Logical shift left Immediate	R	LSL	X1, X2, #10	X1 = X2 << 10
Logical shift right Immediate	R	LSR	X1, X2, #10	X1 = X2 >> 10

Instruction	Format	Example	Meaning
Branch	B	B Label	Go to Label
Branch to register	R	BR X1	Go to address in X1
Branch with link	B	BL Label	X30 = PC + 4; go to Label

Conditional Branch

Instruction	Format	Example	Meaning
Compare and branch if zero	CB	CBZ X1, Label	If (X1 == 0) go to Label
Compare and branch if nonzero	CB	CBNZ X1, Label	If (X1 != 0) go to Label
Branch conditionally	CB	B.cond Label	If (condition true) go to Label

Unconditional Branch

Instruction	Format	Example	Meaning
Branch	B	B Label	Go to Label
Branch to register	R	BR X1	Go to address in X1
Branch with link	B	BL Label	X30 = PC + 4; go to Label

Condition codes and Flags

LEGv8 Condition Code	Meaning		Flag Test
EQ	Equal	X2 == X3	Z==1
NE	Not equal	X2 != X3	Z==0
HS	Unsigned higher or same	X2 >= X3	C==1
LO	Unsigned lower	X2 < X3	C==0
HI	Unsigned higher	X2 > X3	Z==0 && C==1
LS	Unsigned lower or same	X2 <= X3	! (Z==0 && C==1)
GE	Signed greater than or equal	X2 >= X3	N==V
LT	Signed less than	X2 < X3	N != V
GT	Signed greater than	X2 > X3	Z==0 && N == V
LE	Signed less than or equal	X2 <= X3	! (Z ==0 && N==V)

Phụ lục 2: Core Instruction Formats

All LEGv8 instructions have a fixed size of 32 bits and the representation is little endian.

The fields in LEGv8 instructions are named as follows:

- *opcode*: Basic operation of the instruction.
- *Rm*: The second register source operand.
- *shamt*: Shift amount.
- *Rn*: The first register source operand.
- *Rd*: The register destination operand. It gets the result of the operation. Designated *Rt* when the result is an address.
- *ALU_immediate*: Immediate value operand used in arithmetic immediate instructions.
- *DT_address*: Data transfer memory address.
- *BR_address*: Target address for unconditional branch.
- *COND_BR_address*: Target address for conditional branch.
- *MOV_immediate*: Immediate value operand to be shifted into register quadrant.

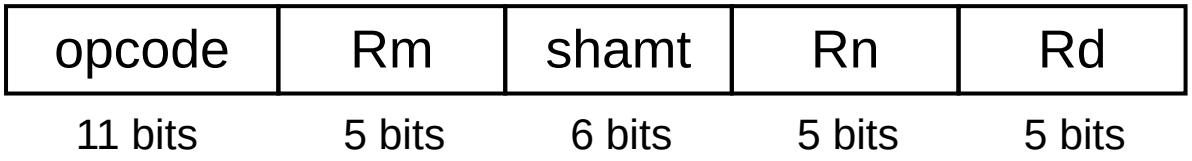
Core Instruction Formats

All LEGv8 instructions have a fixed size of 32 bits and the representation is little endian.

The fields in LEGv8 instructions are named as follows:

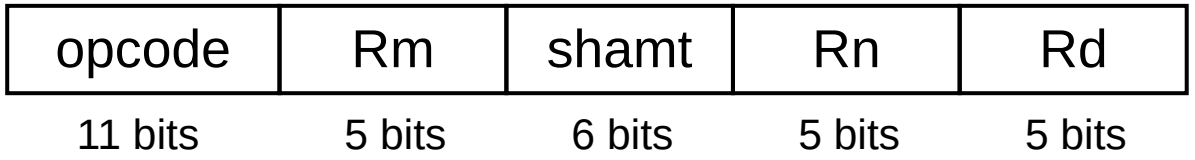
- *opcode*: Basic operation of the instruction.
- *Rm*: The second register source operand.
- *shamt*: Shift amount.
- *Rn*: The first register source operand.
- *Rd*: The register destination operand. It gets the result of the operation. Designated *Rt* when the result is an address.
- *ALU_immediate*: Immediate value operand used in arithmetic immediate instructions.
- *DT_address*: Data transfer memory address.
- *BR_address*: Target address for unconditional branch.
- *COND_BR_address*: Target address for conditional branch.
- *MOV_immediate*: Immediate value operand to be shifted into register quadrant.

LEGv8 R-format Instructions



- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

R-format Example

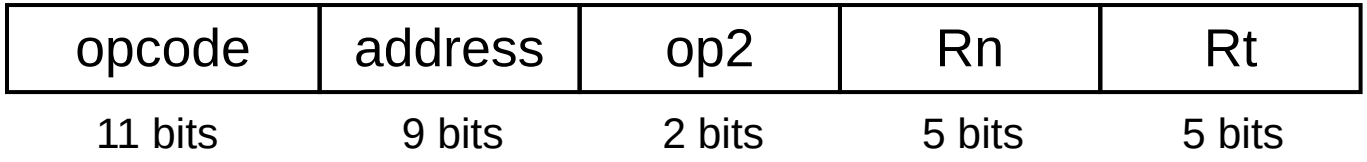


ADD X9,X20,X21

1112_{dec}	21_{dec}	0_{dec}	20_{ten}	9_{dec}
10001011000_{bin}	10101_{bin}	000000_{bin}	10100_{bin}	01001_{bin}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{bin} = 8B150289_{hex}$

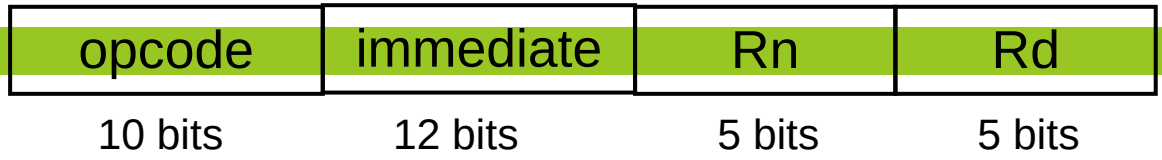
LEGv8 D-format Instructions



- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number

- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

LEGv8 I-format Instructions



- Immediate instructions, e.g. ADDI, SUBI
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended
- Reduce the complexity by keeping the formats similar

Instruction	Format	opcode	Rm	shamt	address	op2	Rn	Rd
ADD (add)	R	1112_{ten}	reg	0	n.a.	n.a.	reg	reg
SUB (subtract)	R	1624_{ten}	reg	0	n.a.	n.a.	reg	reg
ADDI (add immediate)	I	580_{ten}	reg	n.a.	constant	n.a.	reg	n.a.
SUBI (sub immediate)	I	836_{ten}	reg	n.a.	constant	n.a.	reg	n.a.
LDUR (load word)	D	1986_{ten}	reg	n.a.	address	0	reg	n.a.
STUR (store word)	D	1984_{ten}	reg	n.a.	address	0	reg	n.a.

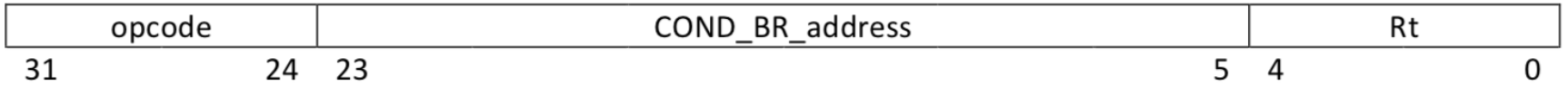
Figure 2.5 LEGv8 instruction encoding.
In the table above, "reg" means a register number between 0 and 31, "address" means a 9-bit address or 12-bit constant, and "n.a." (not applicable) means this field does not appear in this format. The op2 field expands the opcode field.

B-Format, CB-Format, IW-Format

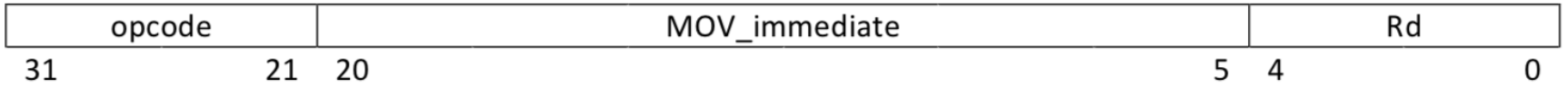
The op2 field is an extension of the opcode field.

3.3.4 B-FORMAT

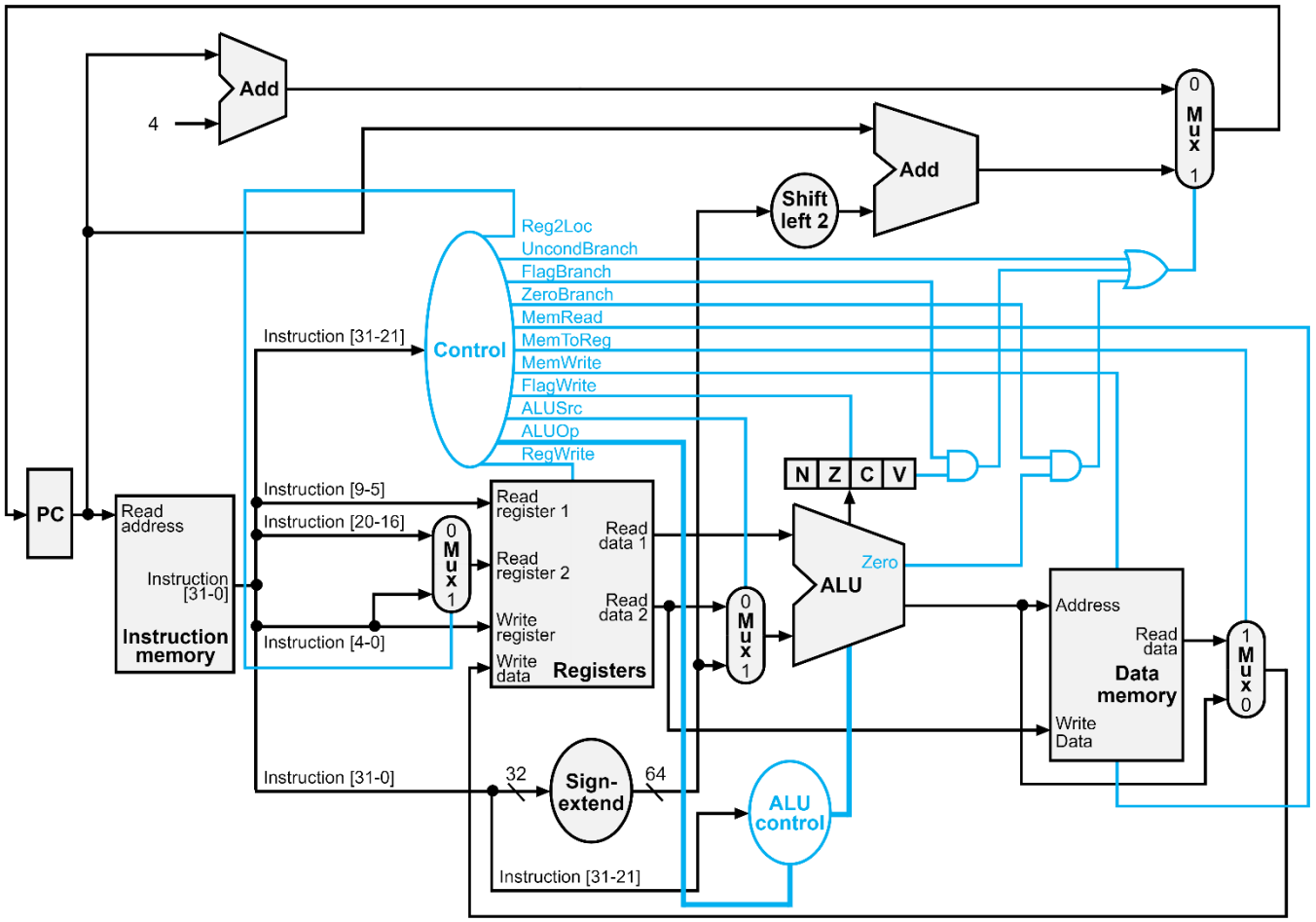
3.3.5 CB-FORMAT



3.3.6 IW-FORMAT



Mô phỏng mạch xử lý LEGv8



Tín hiệu điều khiển Control Unit

- **Reg2Loc:** Tín hiệu điều khiển để xác định nguồn địa chỉ của thanh ghi thứ hai (R[n]) trong các lệnh R-format. Nếu Reg2Loc là 1, địa chỉ thanh ghi thứ hai được lấy từ trường Rt của lệnh; nếu 0, nó được lấy từ trường Rm.
- **UncondBranch**: Tín hiệu điều khiển cho các lệnh nhảy không điều kiện (unconditional branch). Nếu tín hiệu này được bật, bộ xử lý sẽ nhảy tới địa chỉ mục tiêu mà không cần kiểm tra điều kiện nào.
- **FlagBranch**: Tín hiệu điều khiển cho các lệnh rẽ nhánh dựa trên các cờ trạng thái (flag). Nếu tín hiệu này được bật, bộ xử lý sẽ kiểm tra các cờ trạng thái (như Zero flag hoặc Negative flag) để quyết định có nhảy tới địa chỉ mục tiêu hay không.
- **ZeroBranch**: Tín hiệu điều khiển cho các lệnh nhánh khi kết quả của một phép toán là 0. Nếu tín hiệu này được bật và kết quả của phép toán trong ALU là 0, bộ xử lý sẽ nhảy tới địa chỉ mục tiêu.
- **MemRead**: Tín hiệu điều khiển để đọc dữ liệu từ bộ nhớ. Nếu tín hiệu này được bật, dữ liệu sẽ được đọc từ bộ nhớ và đưa vào một thanh ghi.

Tín hiệu điều khiển Control Unit

- **MemtoReg**: Tín hiệu điều khiển để xác định dữ liệu từ bộ nhớ có được ghi vào thanh ghi hay không. Nếu tín hiệu này được bật, dữ liệu từ bộ nhớ sẽ được ghi vào thanh ghi đích (destination register).
- **MemWrite**: Tín hiệu điều khiển để ghi dữ liệu vào bộ nhớ. Nếu tín hiệu này được bật, dữ liệu từ thanh ghi sẽ được ghi vào bộ nhớ.
- **FlagWrite**: Tín hiệu điều khiển để ghi cập nhật cờ trạng thái (flag). Nếu tín hiệu này được bật, các cờ trạng thái (như Zero flag, Negative flag) sẽ được cập nhật dựa trên kết quả của ALU.
- **ALUSrc**: Tín hiệu điều khiển để chọn nguồn của toán hạng thứ hai (second operand) cho ALU. Nếu tín hiệu này được bật, toán hạng thứ hai sẽ được lấy từ một hằng số (immediate); nếu không, toán hạng thứ hai sẽ được lấy từ thanh ghi thứ hai.
- **ALUOp**: Tín hiệu điều khiển để xác định phép toán cụ thể mà ALU sẽ thực hiện (như cộng, trừ, AND, OR). Giá trị của tín hiệu này quyết định hoạt động cụ thể của ALU.
- **RegWrite**: Tín hiệu điều khiển để ghi dữ liệu vào thanh ghi đích. Nếu tín hiệu này được bật, dữ liệu sẽ được ghi vào thanh ghi đích xác định bởi lệnh.

Một số giá trị ALUOp

- Giá trị ALUOp
  - 00: phép toán Load/Store
  - 01: phép toán rẻ nhánh
  - 10: Phép toán số học hoặc logic
- Tín hiệu ALU Control
  - 0000: AND
  - 0001: OR
  - 0010: ADD
  - 0110: SUB
  - …

Phụ lục 3: x86 vs LEGv8

	X86	LEGv8
Đặc điểm	• Là một kiến trúc CISC (Complex Instruction Set Computer). • Hỗ trợ một tập lệnh phức tạp và đa dạng. • Có nhiều biến thể và chế độ hoạt động khác nhau.	• là một kiến trúc RISC (Reduced Instruction Set Computer). • Thiết kế đơn giản với một số lệnh cơ bản và ít biến thể. • Hỗ trợ các chế độ xử lý khác nhau như user mode, supervisor mode, và hypervisor mode.
Ưu điểm	• Hỗ trợ các tính năng phong phú như SIMD (Single Instruction, Multiple Data) và các lệnh phức tạp. • Được sử dụng rộng rãi trong nhiều ứng dụng máy tính và server.	• Hiệu suất cao với các lệnh đơn giản, dễ dàng thực hiện trên phần cứng. • Tiết kiệm năng lượng và không gian bộ nhớ.
Nhược điểm	• Phức tạp trong việc thiết kế và triển khai trên phần cứng.Tốn nhiều nguồn tài nguyên hơn so với các kiến trúc RISC.	• Yêu cầu nhiều lệnh hơn để thực hiện các tác vụ phức tạp so với CISC. • Cần phải có sự hỗ trợ phần cứng tốt để tối ưu hiệu suất.

Homework

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