PHÁT HIỆN LỖI VÀ LỖ HỔNG PHẦN MỀM

Bài 03. Một số kiến thức nền tảng

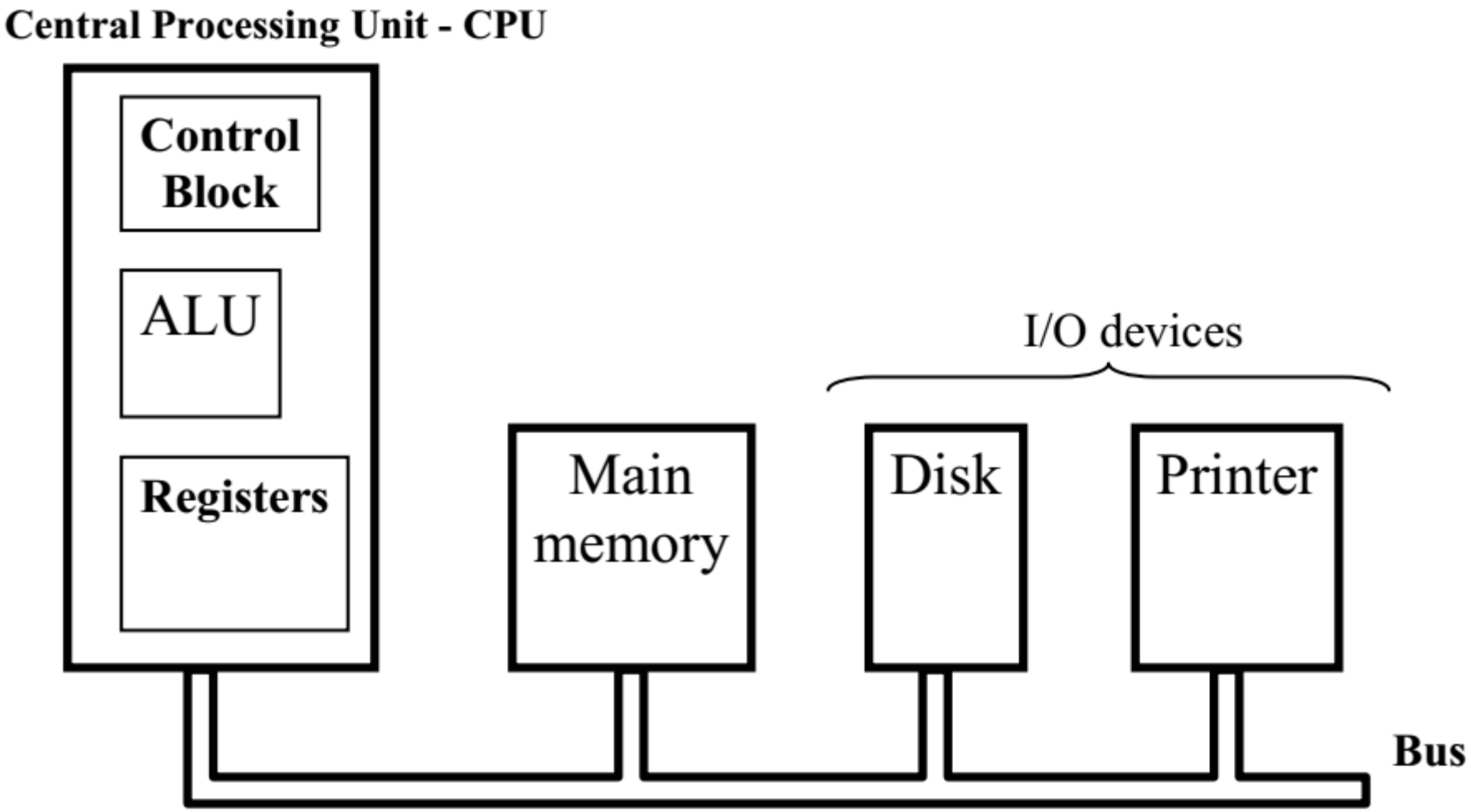
1. Kiến trúc CPU
2. Stack
3. Hàm và gọi hàm

Tài liệu tham khảo

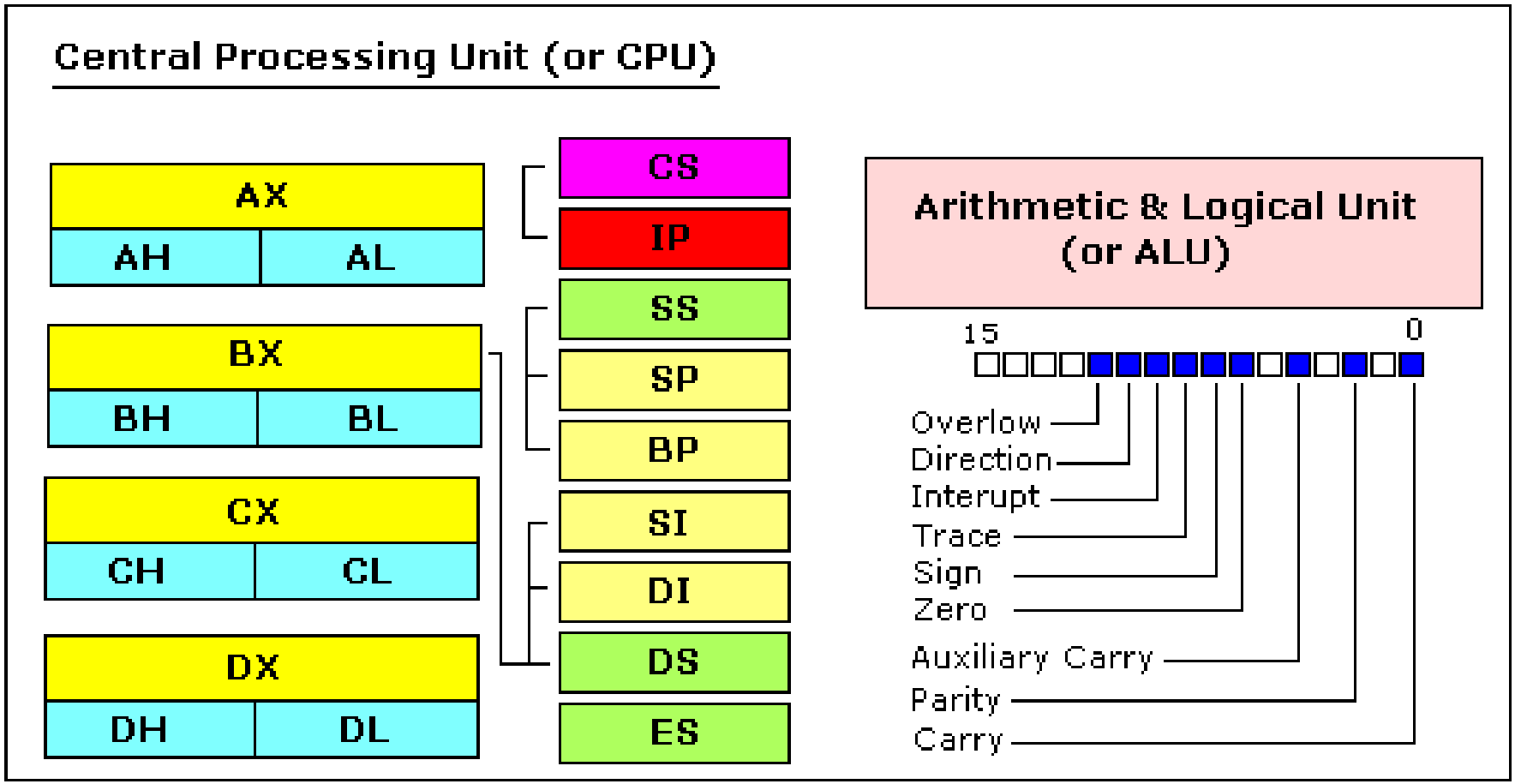
1. Nguyễn Thành Nam, **Chương 2// Nghệ thuật tận dụng lỗi phần mềm**, NXB Khoa học & Kỹ thuật, 2009
2. **NASM Assembly Language Tutorials** https://asmtutor.com

1. Kiến trúc CPU
2. Stack
3. Hàm và gọi hàm

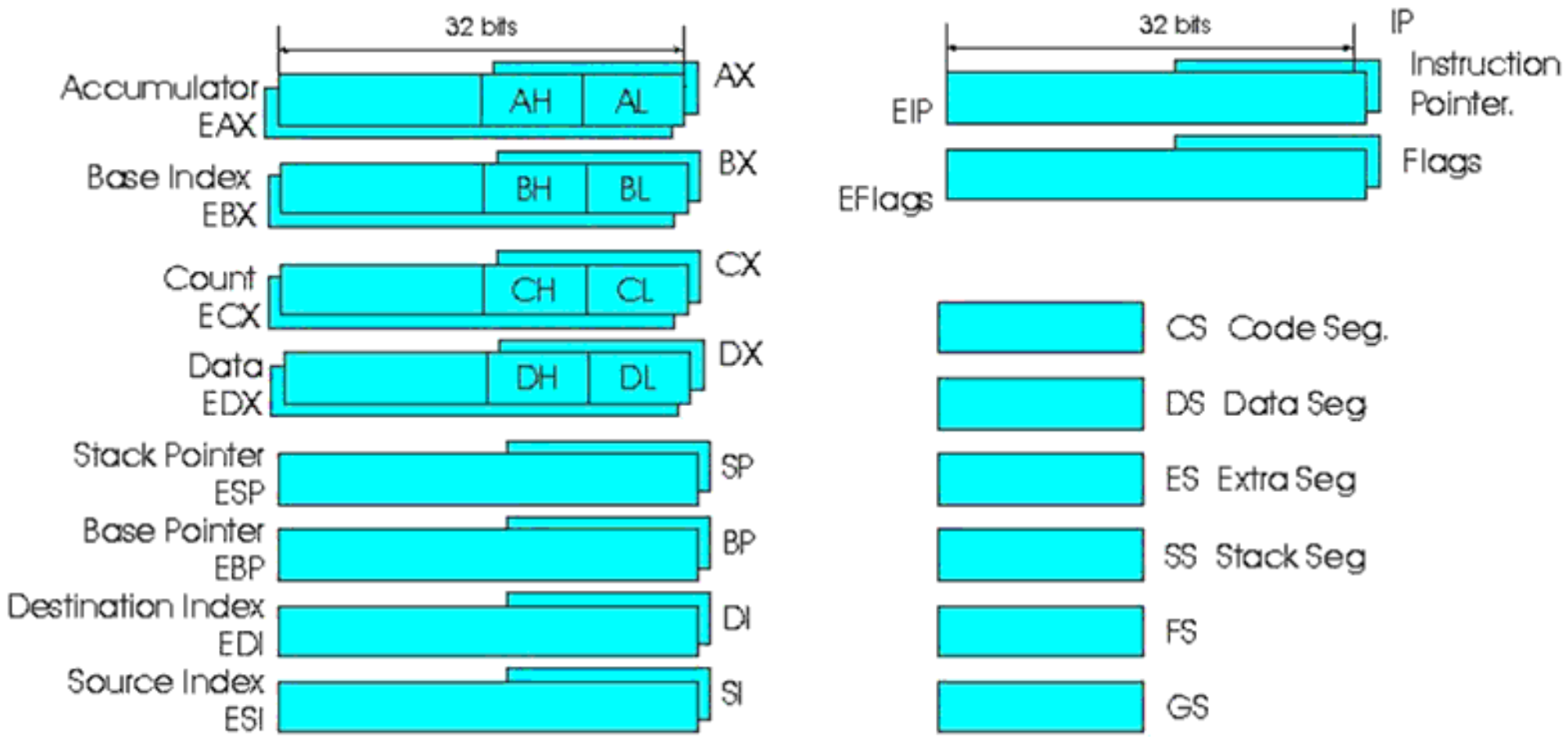
Kiến trúc máy tính



Các thanh ghi của CPU Intel 8086



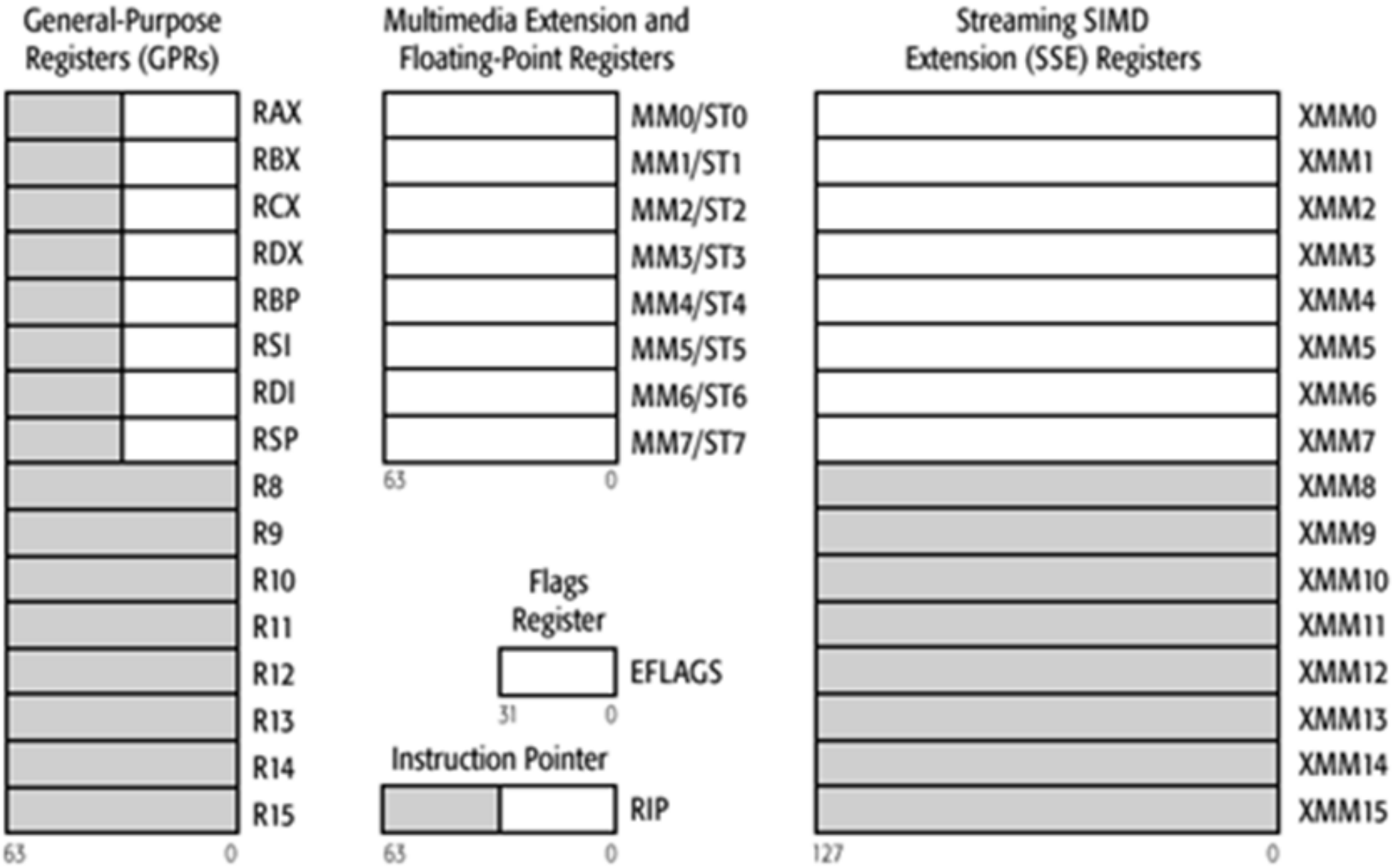
Các thanh ghi 80x86 (32 bít)



Các thanh ghi 80x86 (32 bít)

- Thanh ghi đa dụng: EAX, EBX, ECX, EDX
- Thanh ghi xử lý chuỗi: EDI, ESI
- Thanh ghi ngăn xếp: EBP, ESP
- Thanh ghi con trỏ lệnh: EIP
- Thanh ghi cờ: EFLAGS
- Thanh ghi phân vùng: không còn được sử dụng ở kiến trúc 32 bít

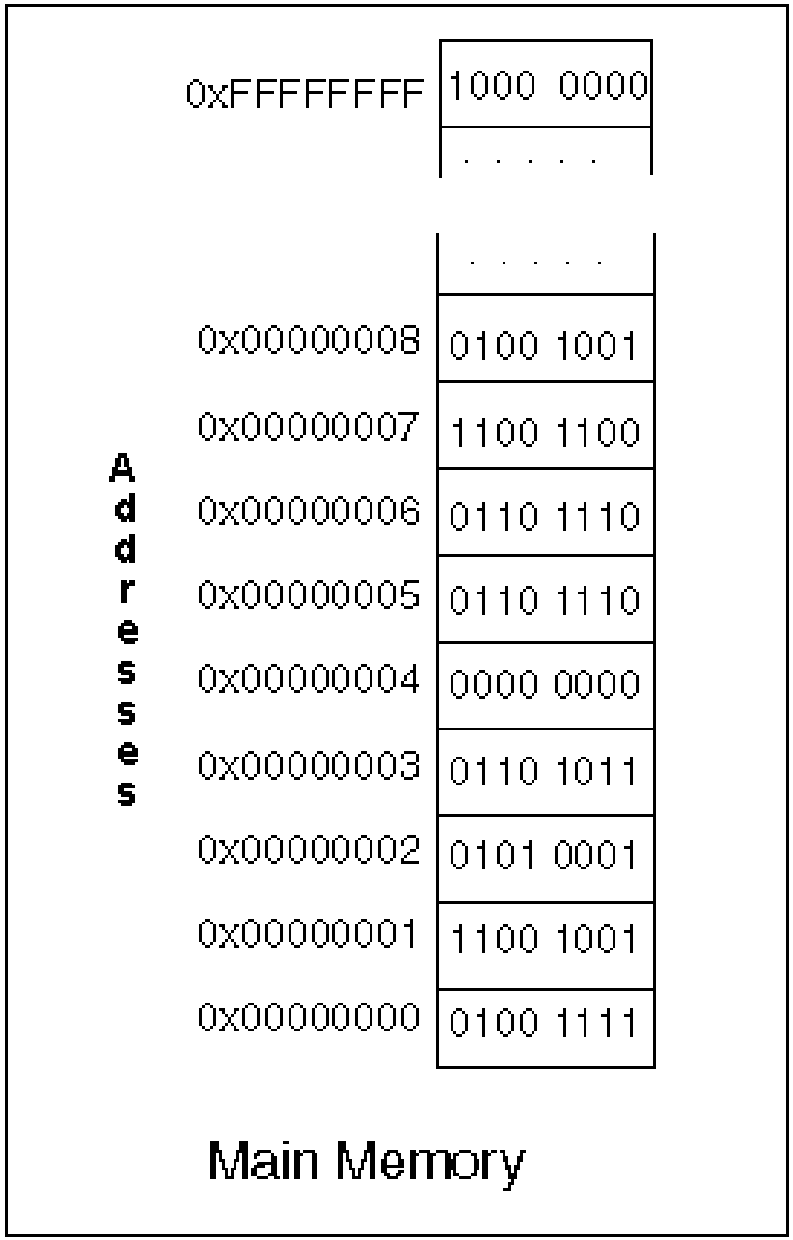
Các thanh ghi x86-64 (64,128 bít)



Bộ nhớ

- Bộ nhớ chính: RAM
- RAM chứa rất nhiều ô nhớ, mỗi ô 1 byte.
- RAM dùng để chứa một phần hệ điều hành, các lệnh chương trình, các dữ liệu…
- Mỗi ô nhớ có địa chỉ duy nhất và địa chỉ này được đánh số từ 0 trở đi.

Địa chỉ bộ nhớ



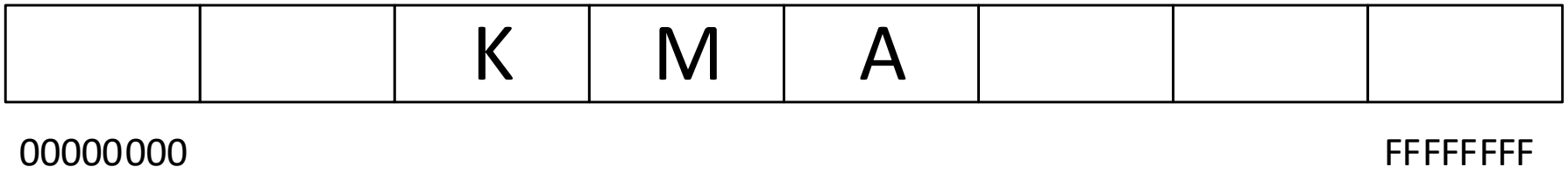
Mô hình bộ nhớ tuyến tính

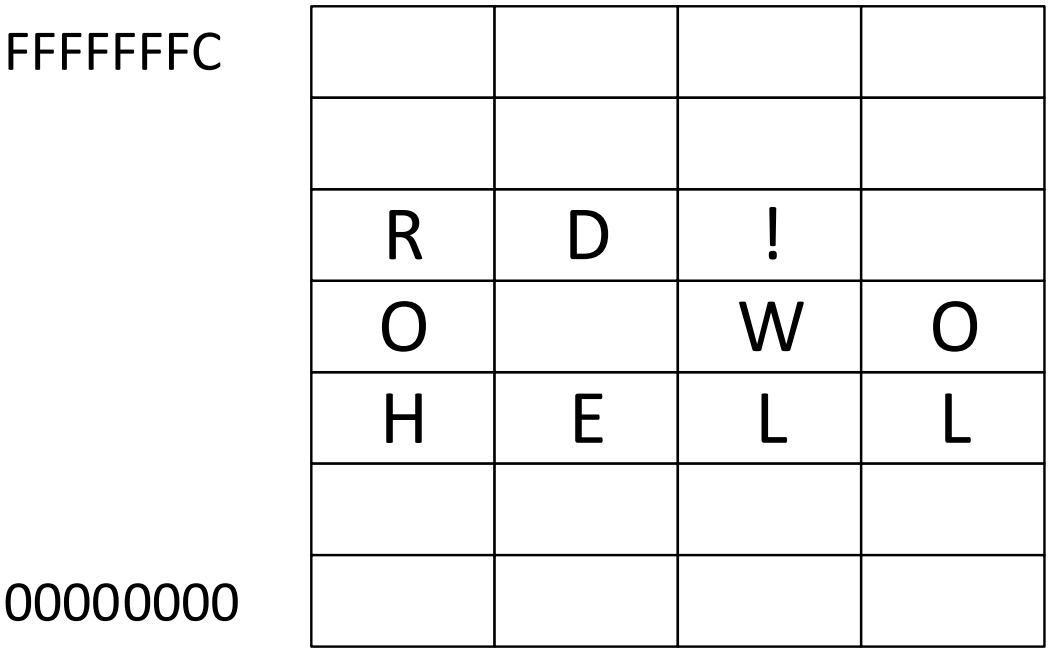
- Flat memory model
- Là mô hình bộ nhớ (dưới một cách nhìn nào đó) mà các ô nhớ được đánh địa chỉ liên tiếp từ 0 đến MAXBYTE-1
- Các mô hình khác:
  - phân đoạn (segmented)
  - phân trang (paged)

Mô hình bộ nhớ tuyến tính

- Các chương trình 32 bít ở Protected Mode luôn sử dụng mô hình Flat.
- Mỗi chương trình có thể coi là nó có riêng 4 GB RAM.
- Mã lệnh và dữ liệu cùng nằm trong một không gian địa chỉ.

2 kiểu biểu diễn bộ nhớ



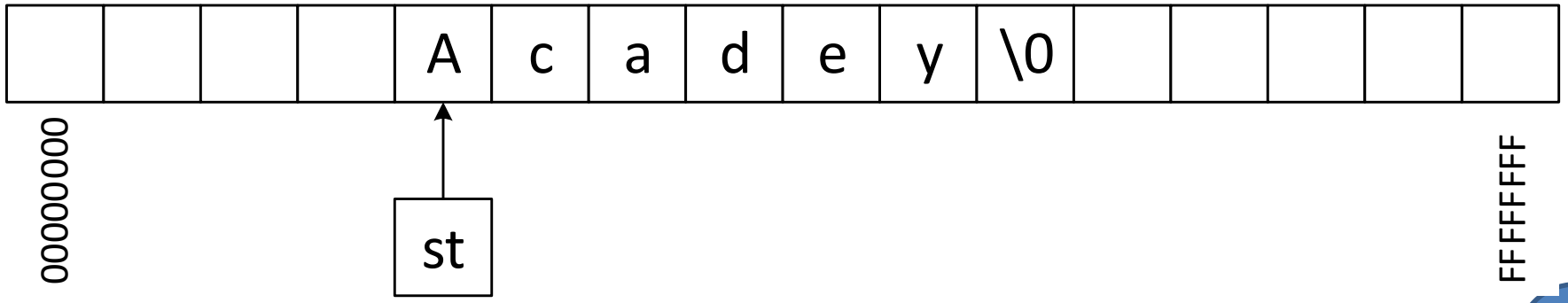


Địa chỉ thường là số hexa

Hướng ghi dữ liệu

- Các hàm nhập dữ liệu trong các ngôn ngữ lập trình luôn ghi dữ liệu vào RAM theo chiều tăng dần của địa chỉ

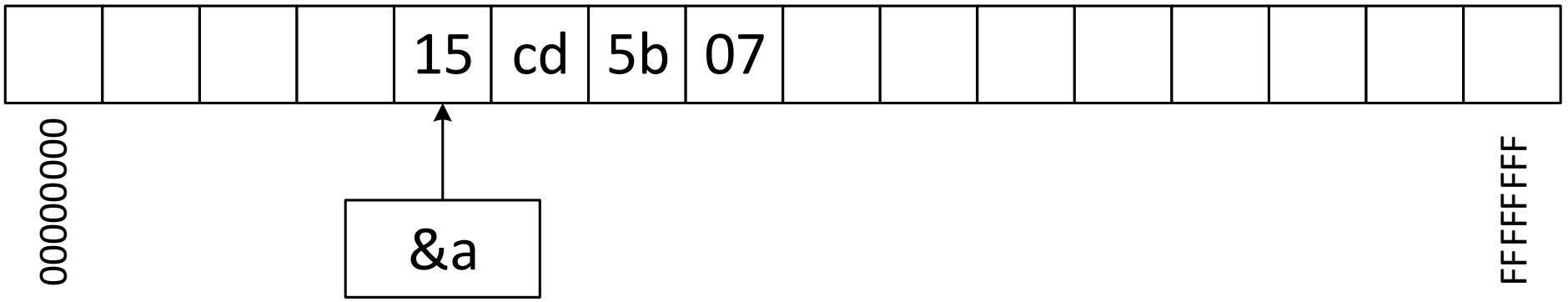
```
char st[100];
gets(st);
```



Trật tự byte: little-endian

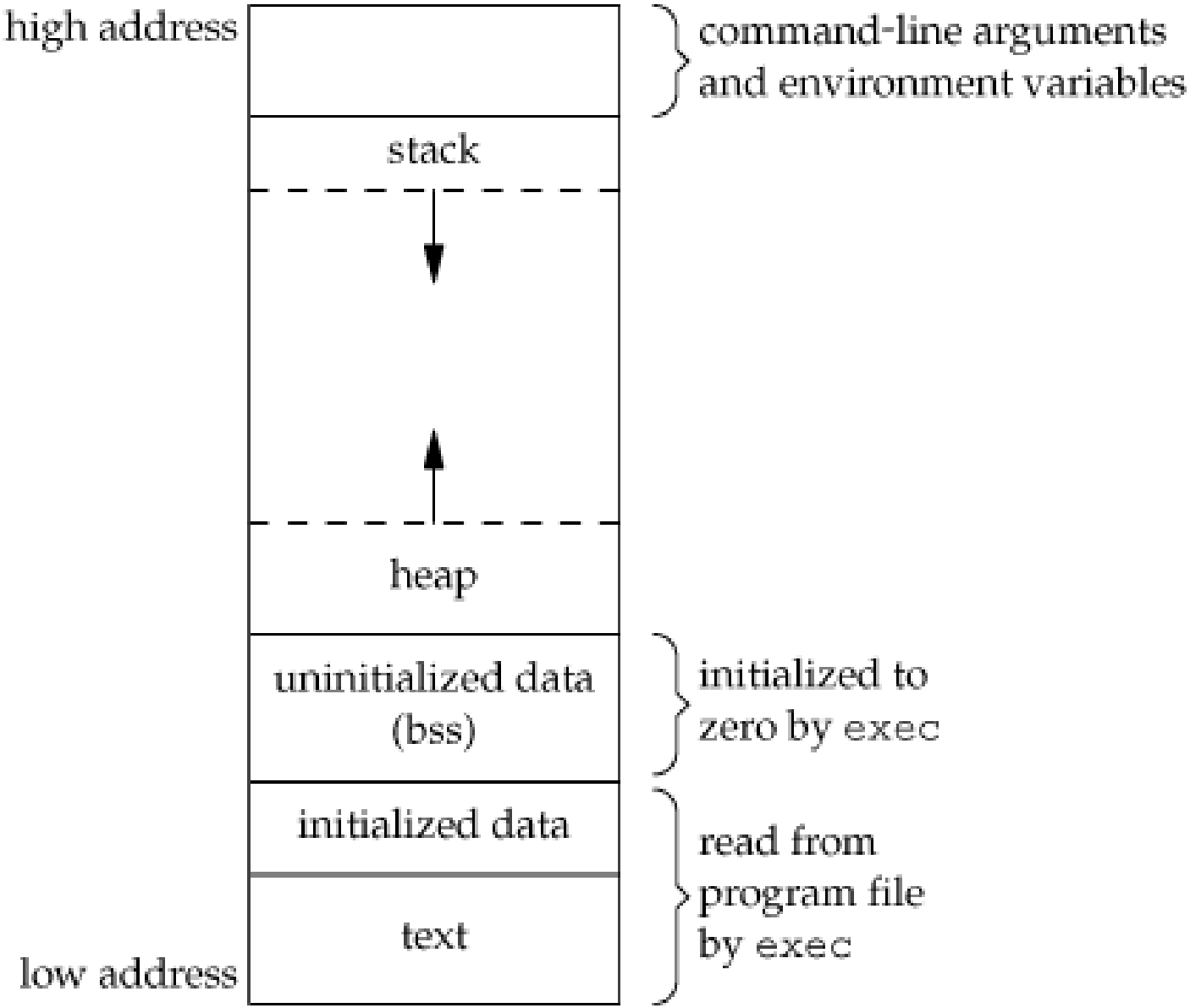
- Các máy tính hiện đại sử dụng little-endian trong biểu diễn số

```
unsigned int a = 123456789;     //0x075BCD15
```



1. Kiến trúc CPU
2. Stack
3. Hàm và gọi hàm

Process's memory layout



Stack

- **Ngăn xếp (stack)** là một vùng nhớ được hệ điều hành cấp phát cho chương trình khi nạp
- Kích thước stack được xác định khi biên dịch chương trình
  - Có thể chỉ định kích thước stack qua tham số cho trình biên dịch
  - Mặc định khoảng 1 MB

Chức năng của stack

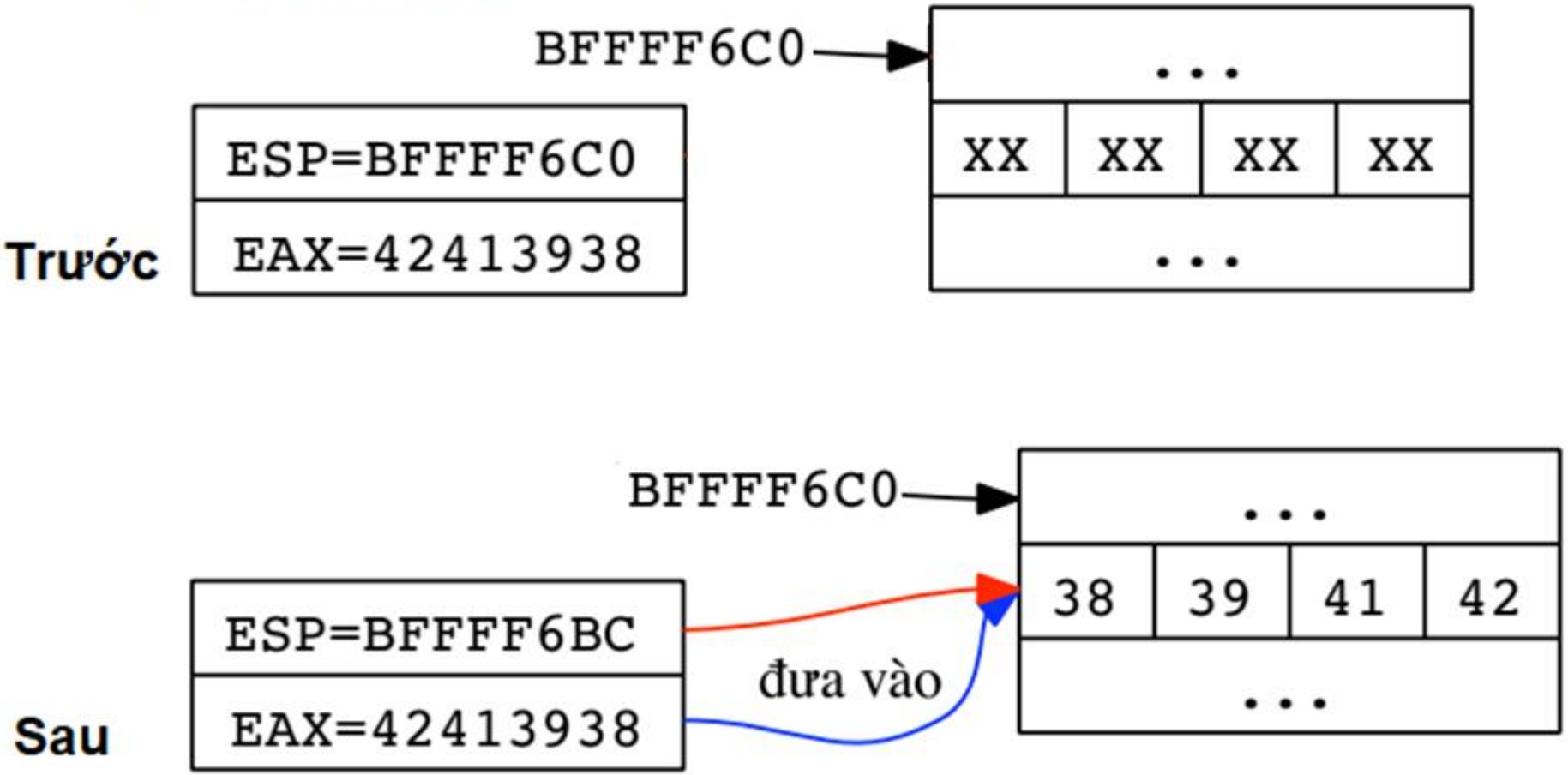
- Chứa các biến cục bộ
- Lưu địa chỉ trả về khi gọi hàm
- Truyền tham số khi gọi hàm
- Lưu giữ con trỏ "this" trong lập trình hướng đối tượng

Thao tác trên ngăn xếp

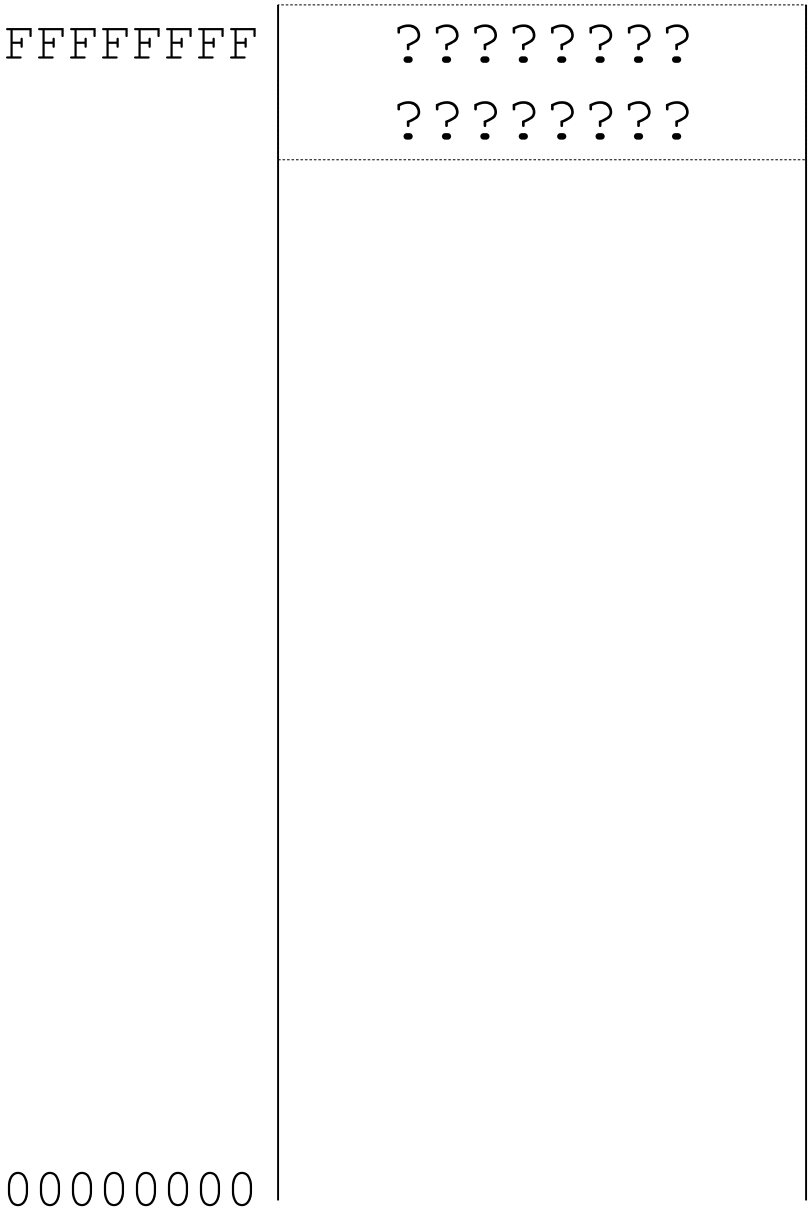
- Trong x86 mỗi phần tử stack là 4 byte
- Stack được quản lý qua ESP
- Hai thao tác cơ bản: PUSH và POP
- PUSH
  - Giảm giá trị của ESP: ESP = ESP – 4
  - Ghi đối số (4 byte) vào [ESP]
- POP
  - Đọc 4 byte tại [ESP] vào đối số
  - Tăng giá trị của ESP: ESP = ESP + 4

Thao tác trên ngăn xếp

Câu lệnh PUSH EAX



Stack Frame

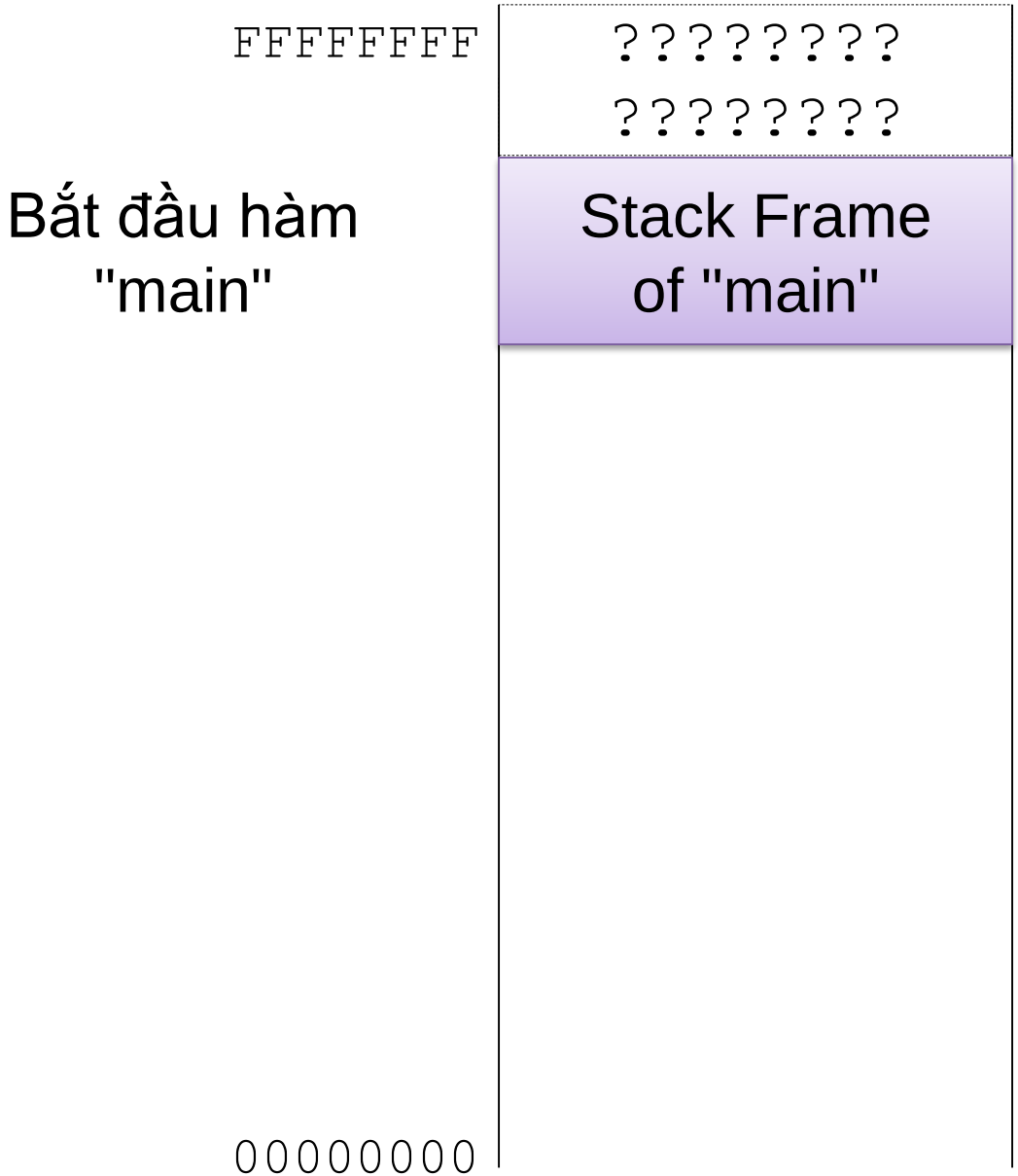



```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame

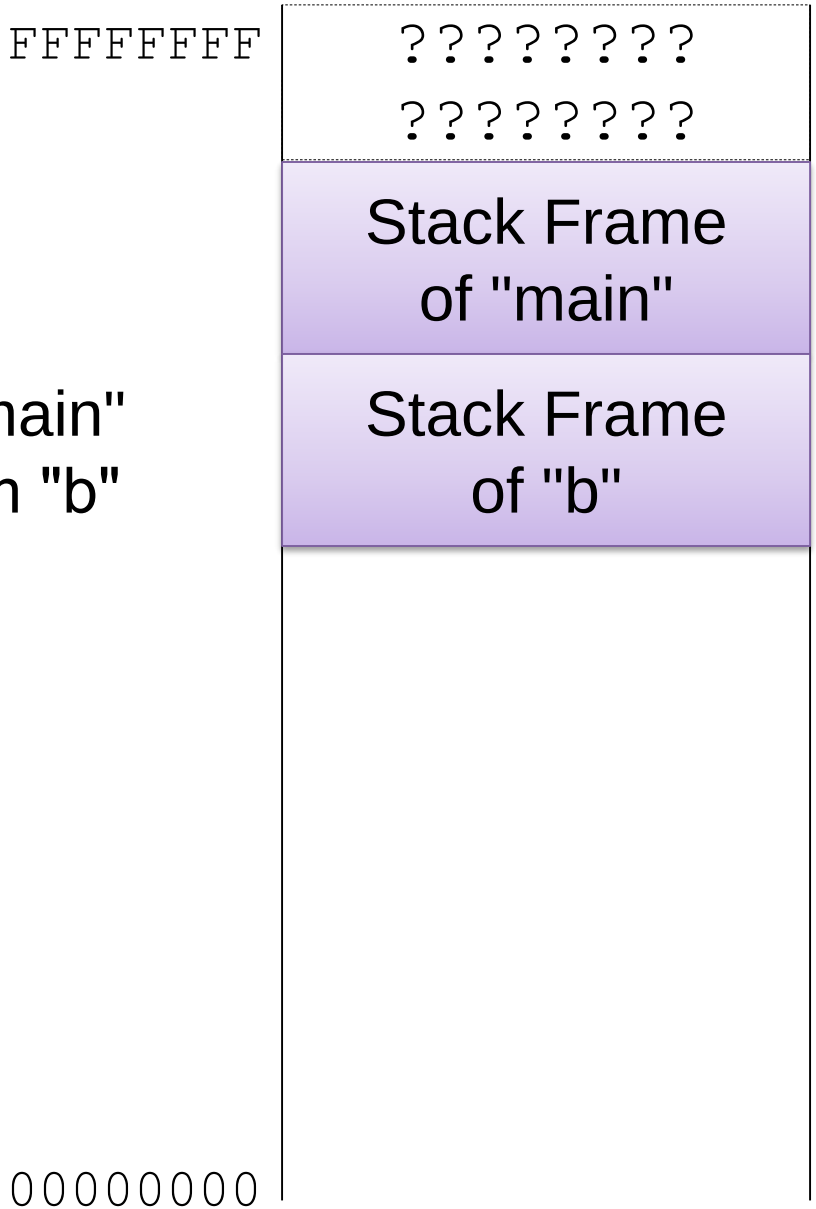



```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame



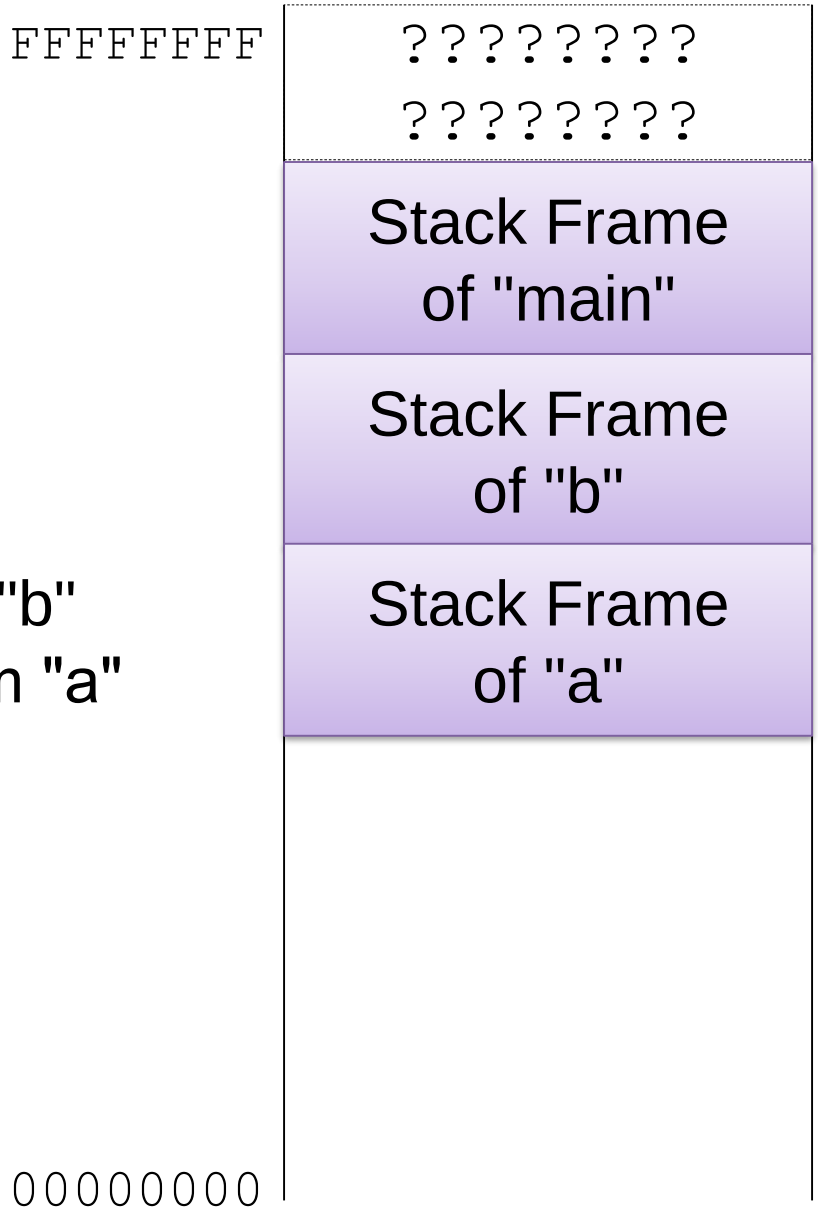
Hàm "main" gọi hàm "b"

```
void main()
{
    b();
}

void b()
{
    a();
}

void a()
{
}
```

Stack Frame



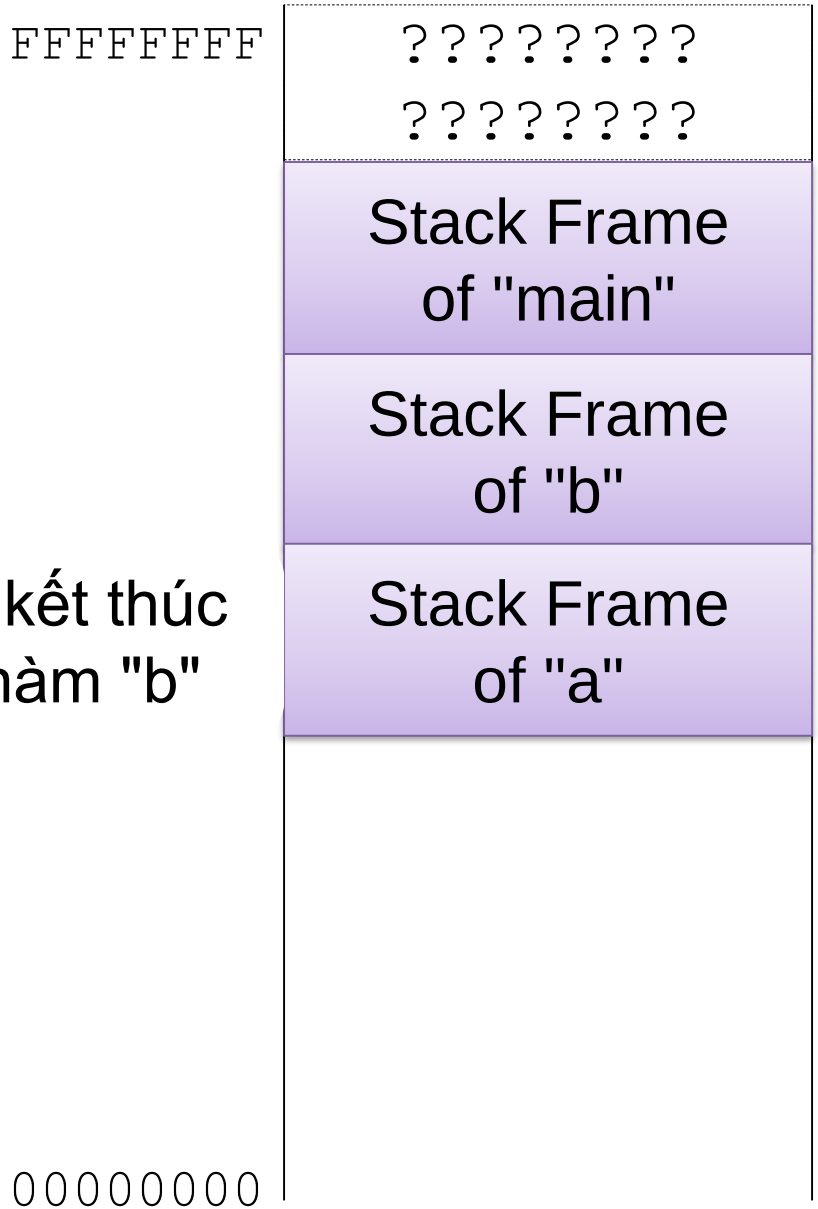
Hàm "b" gọi hàm "a"

```
void main()
{
    b();
}

void b()
{
    a();
}

void a()
{
}
```

Stack Frame



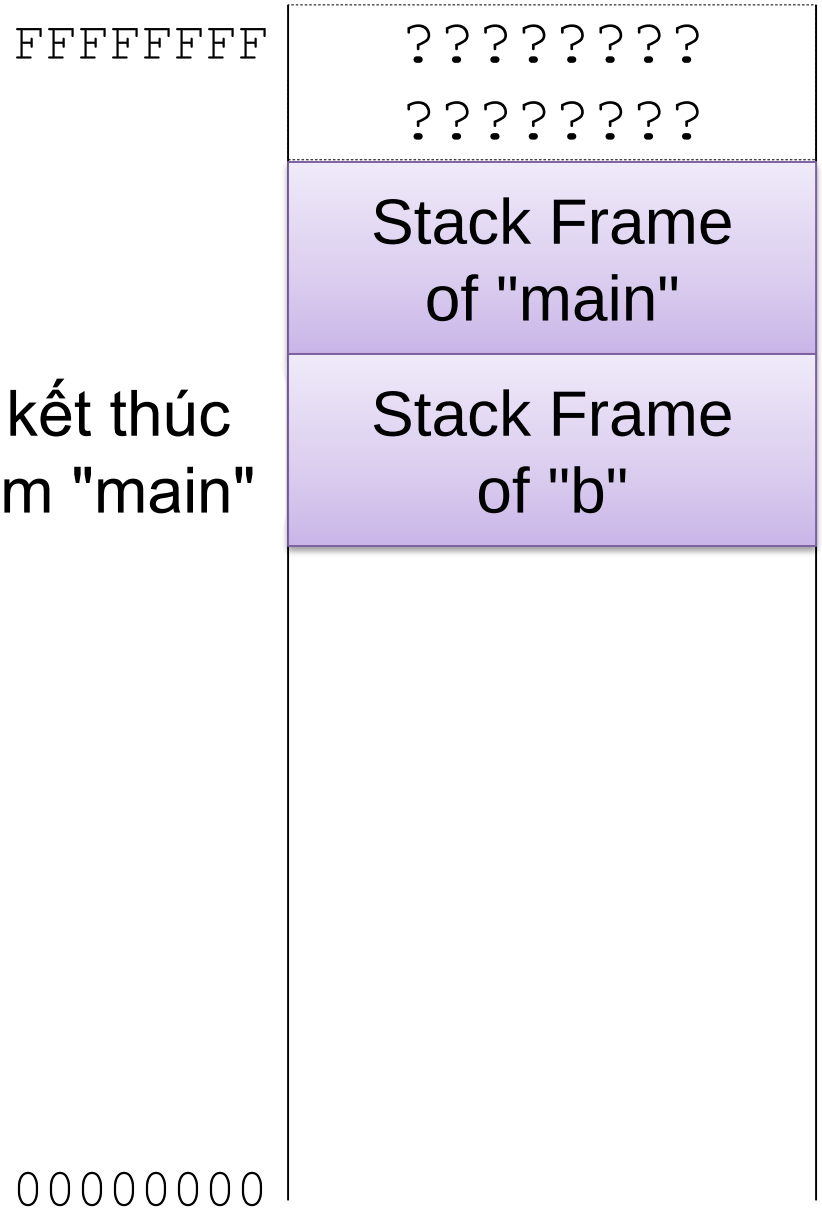
Hàm "a" kết thúc
Trở lại hàm "b"

```
void main()
{
    b();
}

void b()
{
    a();
}

void a()
{
}
```

Stack Frame



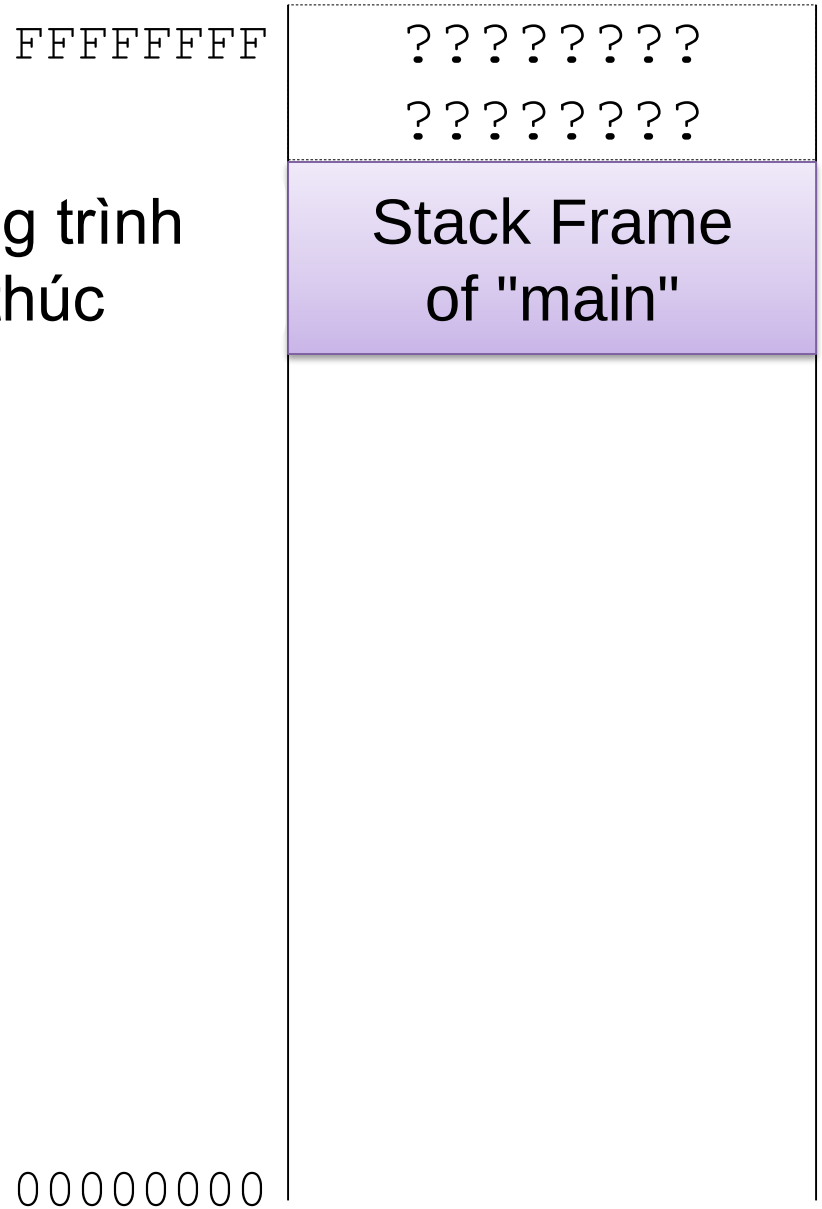
Hàm "b" kết thúc
Trở lại hàm "main"

```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame




```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

1. Kiến trúc CPU
2. Stack
3. Hàm và gọi hàm

Hàm

- **Hàm (Procedure)** là một đoạn chương trình con mà có thể được gọi bởi một chương trình khác để thực thi một nhiệm vụ nhất định

```
procedure_name
    ;some instructions
    RET
```

Hàm

- Thông thường, nếu hàm có trả về một kết quả thì kết quả đó được đặt trong EAX trước khi hàm kết thúc.
- Ví dụ:

```
MySimpleProc
    add     eax, ebx
    sub     eax, edx
    ret
```

Gọi hàm

```
MySimpleProc:
    add     eax, ebx
    sub     eax, edx
    ret
```

Việc gọi một hàm bao gồm:
- Nạp các tham số cần thiết
- Thực hiện lệnh CALL

```
_start:
    mov     eax, 11
    mov     ebx, 22
    mov     edx, 33
    call    MySimpleProc    ;EAX = 11+22-33 = 0
    ret
```

Gọi hàm

- Tham số có thể được nạp vào thanh ghi
  - Ưu: nhanh
  - Nhược: có thể không đủ thanh ghi
- Cần kết hợp nạp tham số vào stack
- Người xây dựng hàm có toàn quyền lựa chọn cách thức nạp tham số. Nhưng cần có quy ước chung:
  - Mọi người hiểu mã của nhau
  - Mọi người có thể sử dụng hàm của nhau

Calling Convention

- Phổ biến: **stdcall** (Windows API) và **cdecl** (standard C library)
  - Giống
    - Truyền tham số qua stack
    - caller phải bảo quản EAX, ECX và EDX nếu cần (callee phải bảo quản các thanh ghi khác)
  - Khác
    - cdecl: caller phải cân bằng stack
    - stdcall: callee phải cân bằng stack
- Có thể gặp: **fastcall**

cdecl

```
; int SubSquare(int x, int y)
; Return: x^2 – y^2
SubSquare:
        push    ebp                 ; Bảo quản giá trị của EBP
        mov     ebp, esp            ; ebp là cơ sở (base) để đọc tham số
        push    ebx                 ; Bảo quản giá trị của EBX trước khi dùng nó
        mov     eax, [ebp+08h]      ; x
        mov     edx, [ebp+0ch]      ; y
        mov     ebx, eax            ; ebx = x
        sub     eax, edx            ; eax = x – y
        add     ebx, edx            ; ebx = x + y
        mul     ebx                 ; eax *= ebx
        pop     ebx                 ; Hoàn lại ebx ban đầu
        pop     ebp                 ; Hoàn lại ebp ban đầu
        ret                         ; Kết quả lưu trong EAX

_start:
        mov     eax, 10
        push    eax             ; Truyền tham số thứ 2
        mov     eax, 20
        push    eax             ; Truyền tham số thứ 1
        call    SubSquare       ; ESP được bảo toàn
        add     esp, 8          ; Cân bằng stack, tổng cộng 8 byte tham số
        ret
```

cdecl

Trong hàm SubSquare ta sử dụng (làm thay đổi giá trị) 4 thanh ghi là EAX, EBX, EDX và EBP. Tuy nhiên, theo quy ước C thì chương trình gọi hàm (caller) phải chịu trách nhiệm bảo quản các thanh ghi EAX, ECX và EDX nên trong hàm ta phải bảo quản giá trị hai thanh ghi là EBP và EBX (dòng 14 và 16), và hoàn lại giá trị cho hai thanh ghi này (dòng 23, 24) trước khi hàm kết thúc.

Ngoài ra, ta cũng làm thay đổi giá trị của ESP với các chỉ thị PUSH và POP, nhưng số lượng lệnh PUSH luôn được đảm bảo bằng số lượng lệnh POP, do đó giá trị của ESP ở đầu và cuối hàm là như nhau.
Trong hàm main, khi gọi hàm SubSquare, có hai tham số với tổng kích thước 8 byte được truyền vào stack làm giá trị của ESP giảm đi 8 (dòng 30, 32) nên để thực hiện nhiệm vụ cân bằng stack thì cần tăng ESP lên 8 (dòng 34) sau khi gọi hàm.

stdcall

```
; int SubSquare(int x, int y)
; Return: x^2 – y^2
SubSquare:
        push    ebp                 ; Bảo quản giá trị của EBP
        mov     ebp, esp            ; ebp là cơ sở (base) để đọc tham số
        push    ebx                 ; Bảo quản giá trị của EBX trước khi dùng nó
        mov     eax, [ebp+08h]      ; x
        mov     edx, [ebp+0ch]      ; y
        mov     ebx, eax            ; ebx = x
        sub     eax, edx            ; eax = x – y
        add     ebx, edx            ; ebx = x + y
        mul     ebx                 ; eax *= ebx
        pop     ebx                 ; Hoàn lại ebx ban đầu
        pop     ebp                 ; Hoàn lại ebp ban đầu
        ret     8                   ; Cân bằng stack với 8 byte. Kết quả lưu trong EAX

_start:
        mov     eax, 10
        push    eax             ; Truyền tham số thứ 2
        mov     eax, 20
        push    eax             ; Truyền tham số thứ 1
        call    SubSquare       ; ESP được tăng 8 trước khi trở về
        ret
```

stdcall

Cách thức mà một hàm lấy tham số được truyền qua stack:

- Thanh ghi EBP được sử dụng để lưu giá trị của thanh ghi ESP ở thời điểm bắt đầu hàm. Do vậy, cặp dòng lệnh đầu tiên trong một hàm là việc lưu lại giá trị của EBP vào trong stack (dòng 14) và đưa giá trị của ESP vào EBP (dòng 15), và dòng lệnh cuối cùng trước lệnh RET là khôi phục lại giá trị ban đầu của EPB (dòng 24).
- Tham số đầu tiên được truyền vào stack sẽ nằm ở địa chỉ [EBP+08h]. Điều này được giải thích như sau:
  - Tham số đầu tiên được đẩy vào stack sau cùng
  - Khi gọi hàm, địa chỉ của lệnh kế tiếp được tự động đẩy vào stack → ESP bị giảm đi 4 (32 bít).
  - Khi đẩy giá trị EBP vào stack, thanh ghi ESP lại giảm thêm 4.
  - Như thế, tổng cộng ESP đã giảm 8 byte so với vị trí của tham số đầu tiên.

Cấu trúc hàm

```
Hàm
{
    Phần dẫn nhập;
    Phần thân hàm;
    Phần kết thúc;
}
```

Stack frame of function call

Calling a function B from function A on a typical "generic" system might involve the following steps:

function A:

- push space for the return value
- push parameters
- push the return address

jump to the function B

function B:

- push the address of the previous stack frame
- push values of registers that this function uses (so they can be restored)
- push space for local variables
- do the necessary computation
- restore the registers
- restore the previous stack frame
- store the function result
- jump to the return address

function A:

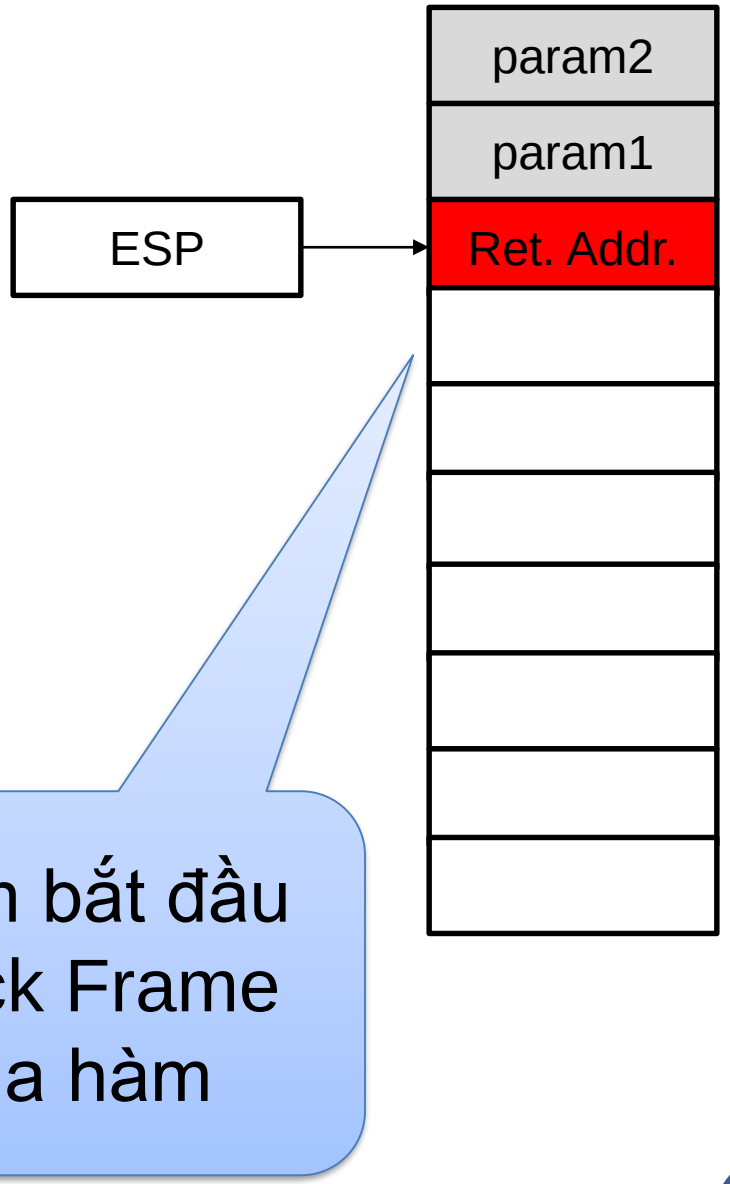
- pop the parameters
- pop the return value

Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```

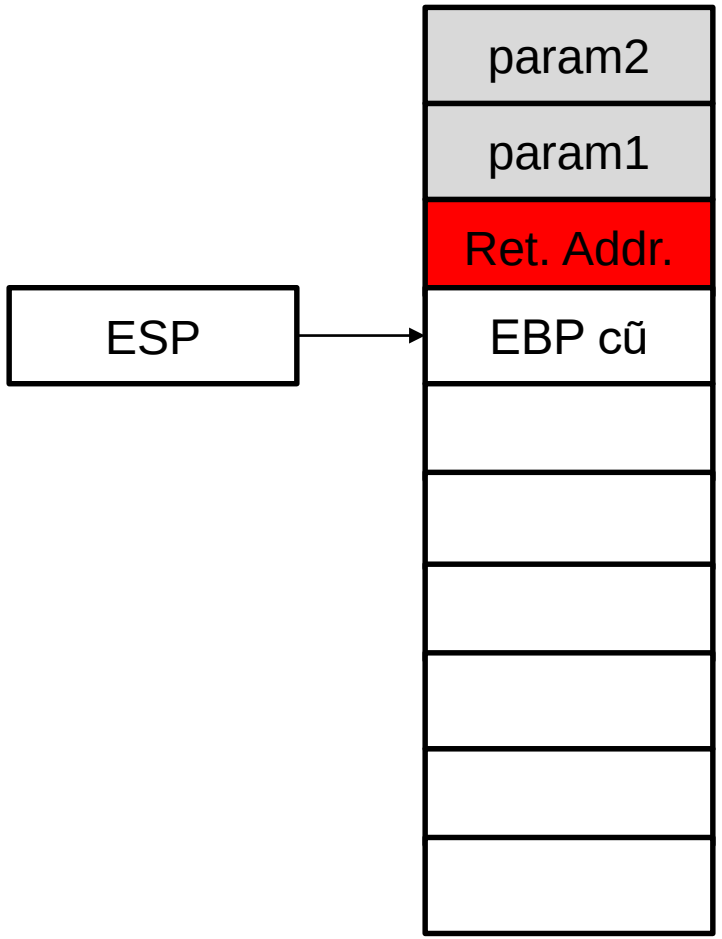


Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```

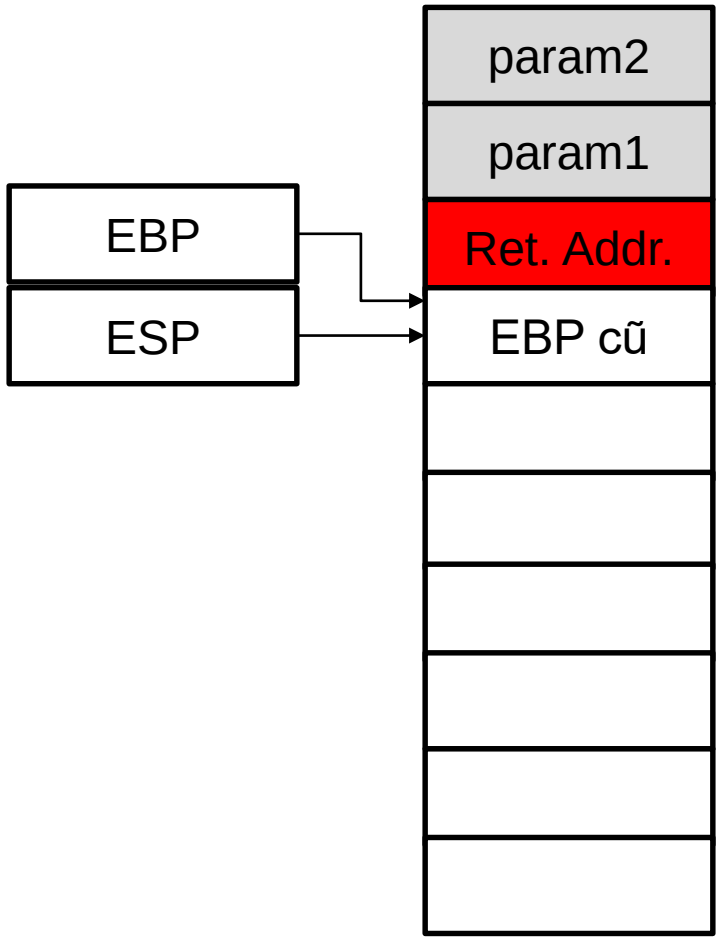


Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```



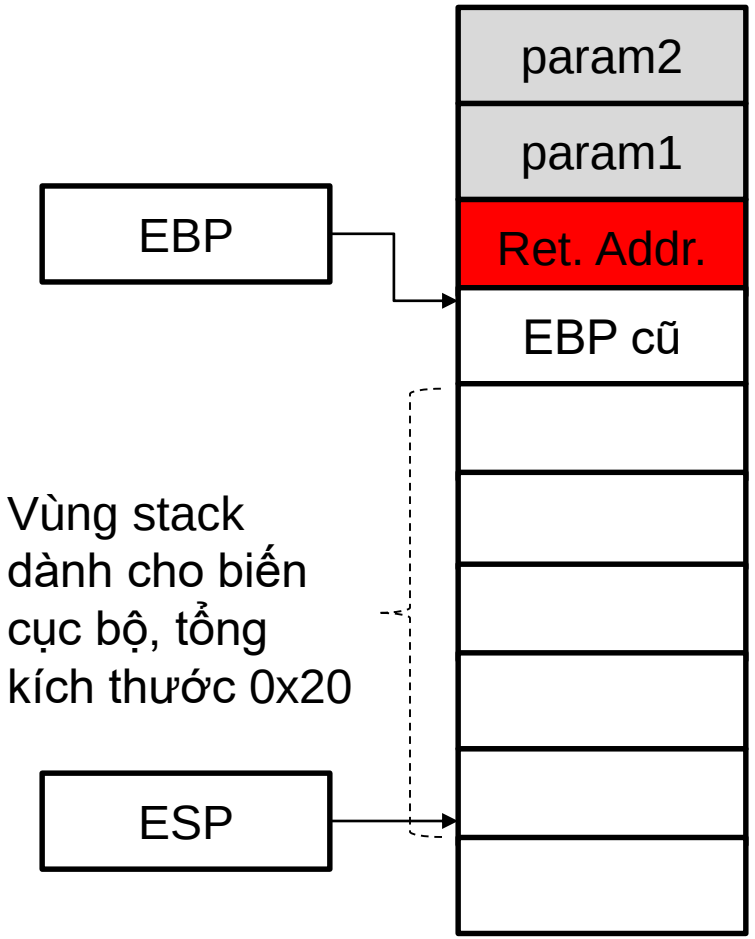
Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần th
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```

Câu lệnh phụ (nếu hàm có sử dụng biến cục bộ)

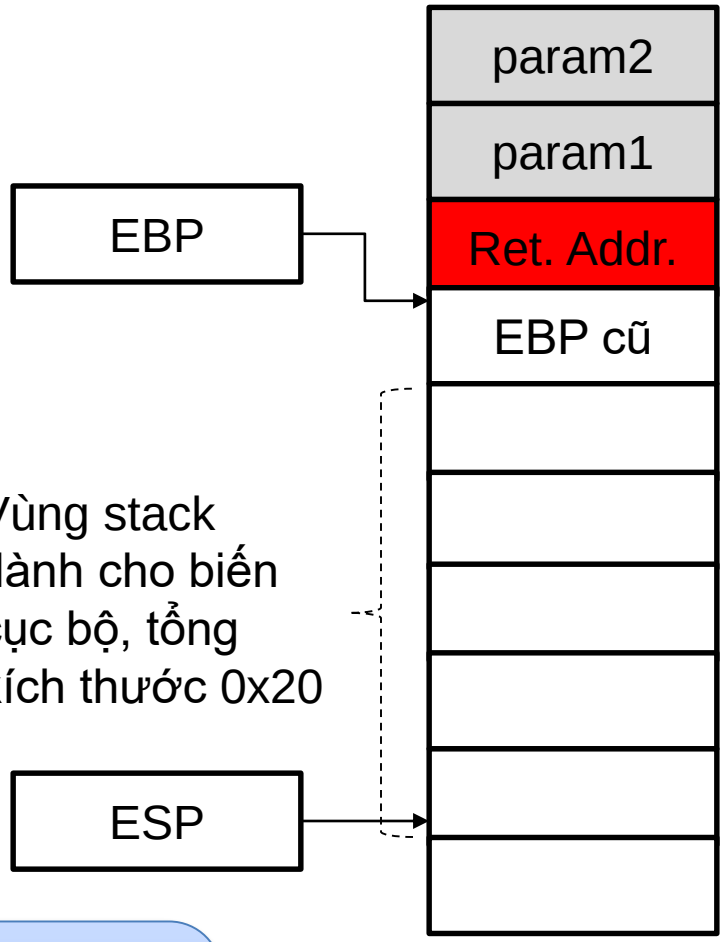


Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```



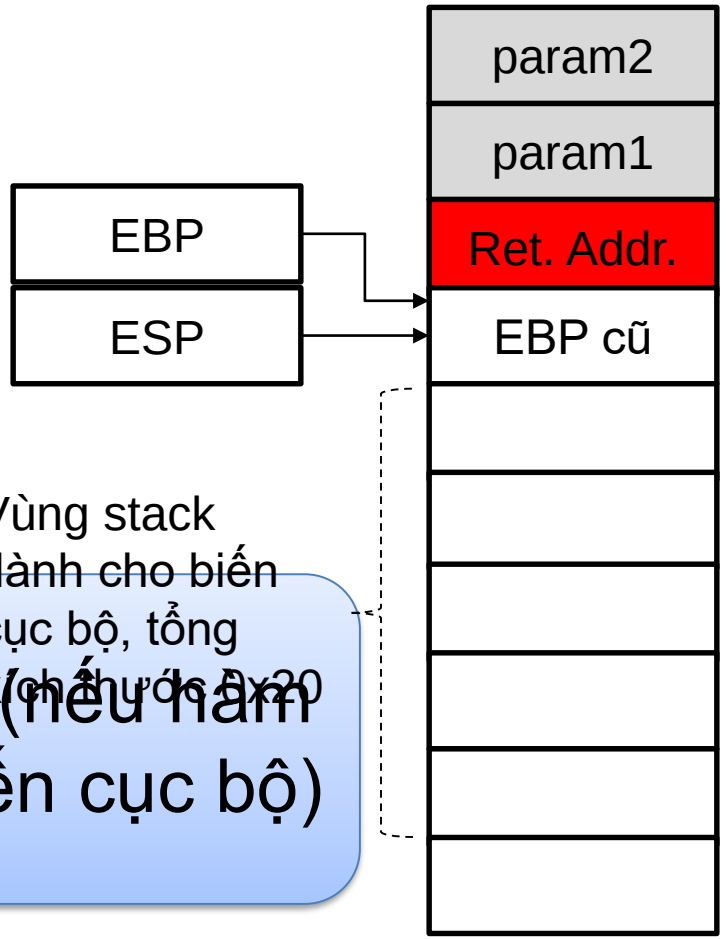
EBP không thay đổi

Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```



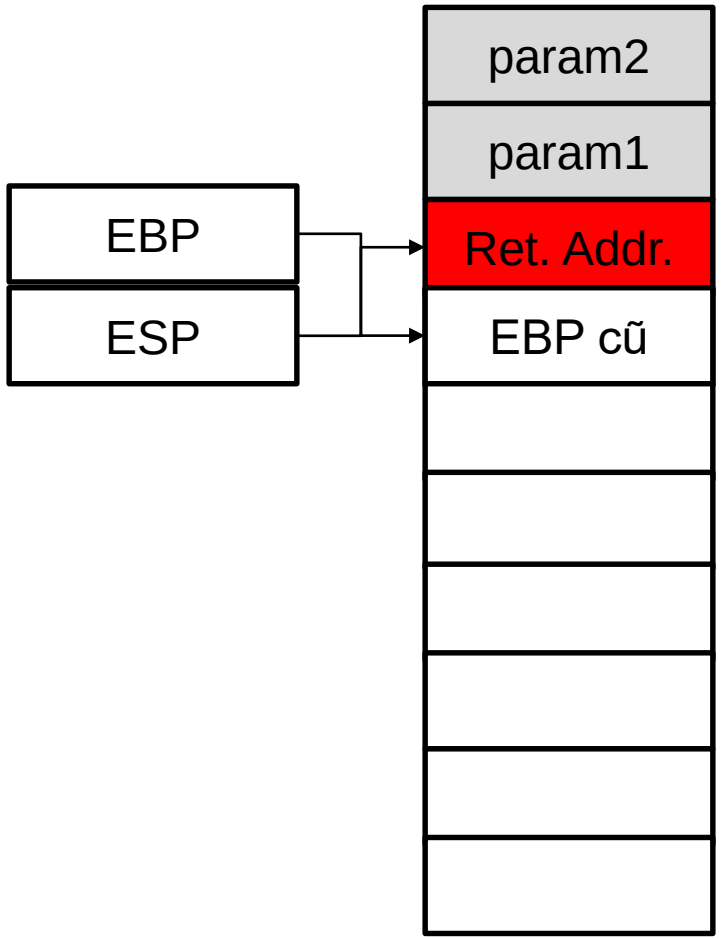
Câu lệnh phụ (nếu hàm có sử dụng biến cục bộ)

Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```

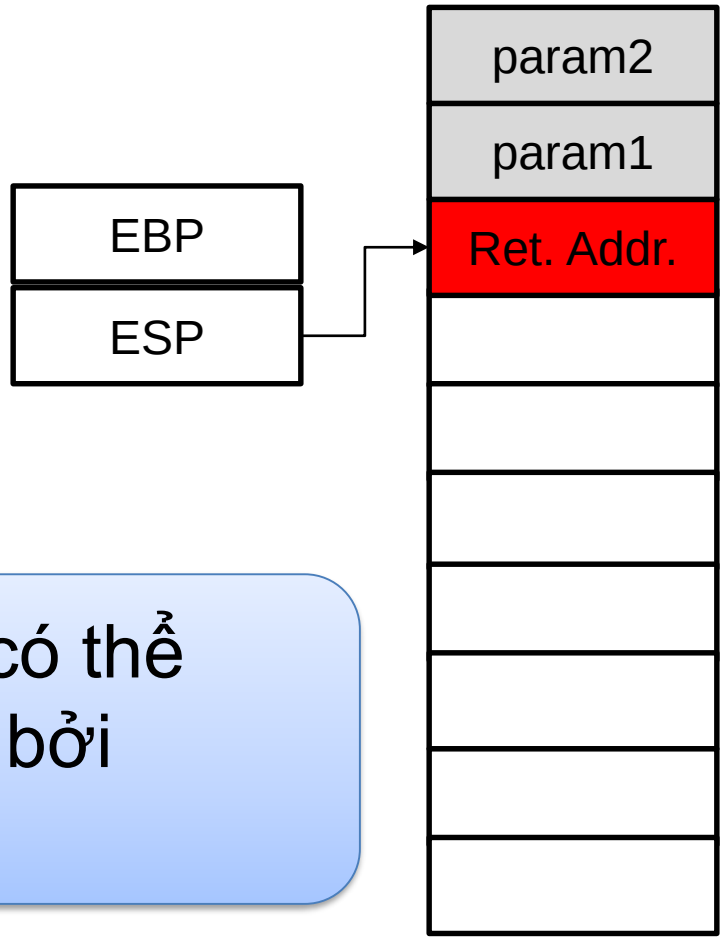


Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```



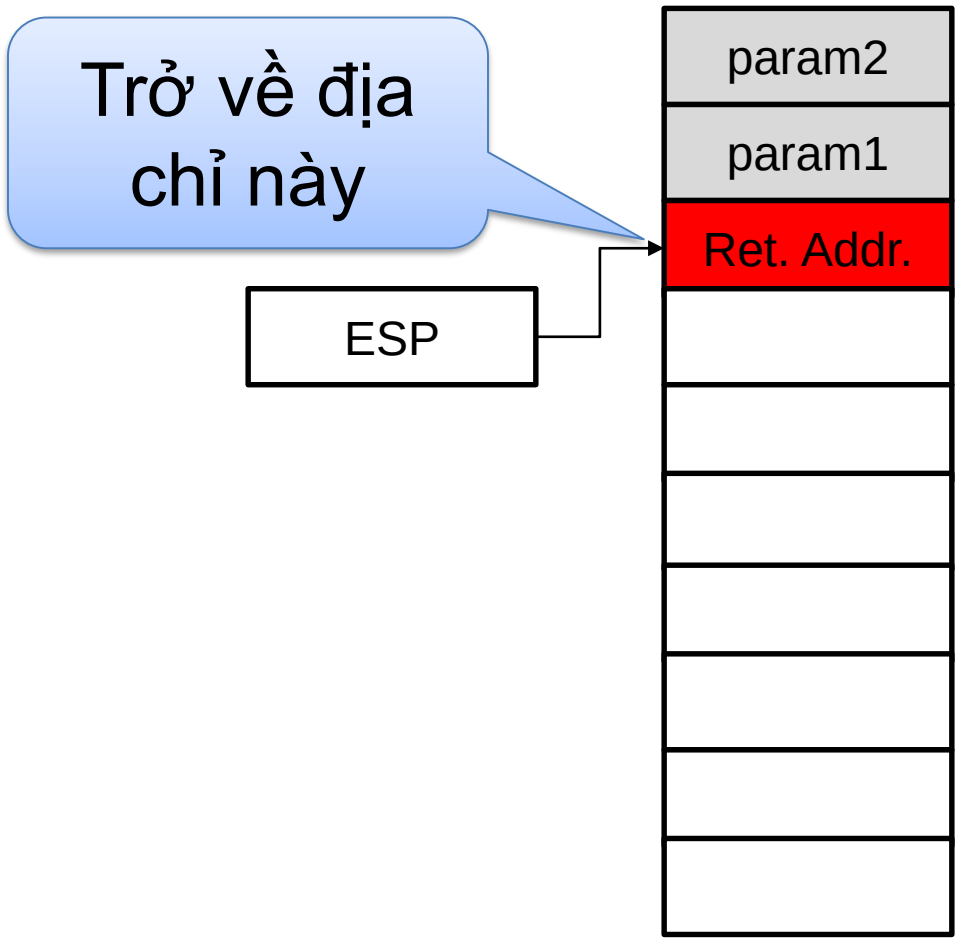
Cặp lệnh này có thể được thay thế bởi LEAVE

Cấu trúc hàm

```
;Phần dẫn nhập
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, 0x20

;Phần thân hàm
;....

;Phần kết thúc
MOV     ESP, EBP
POP     EBP
RET
```



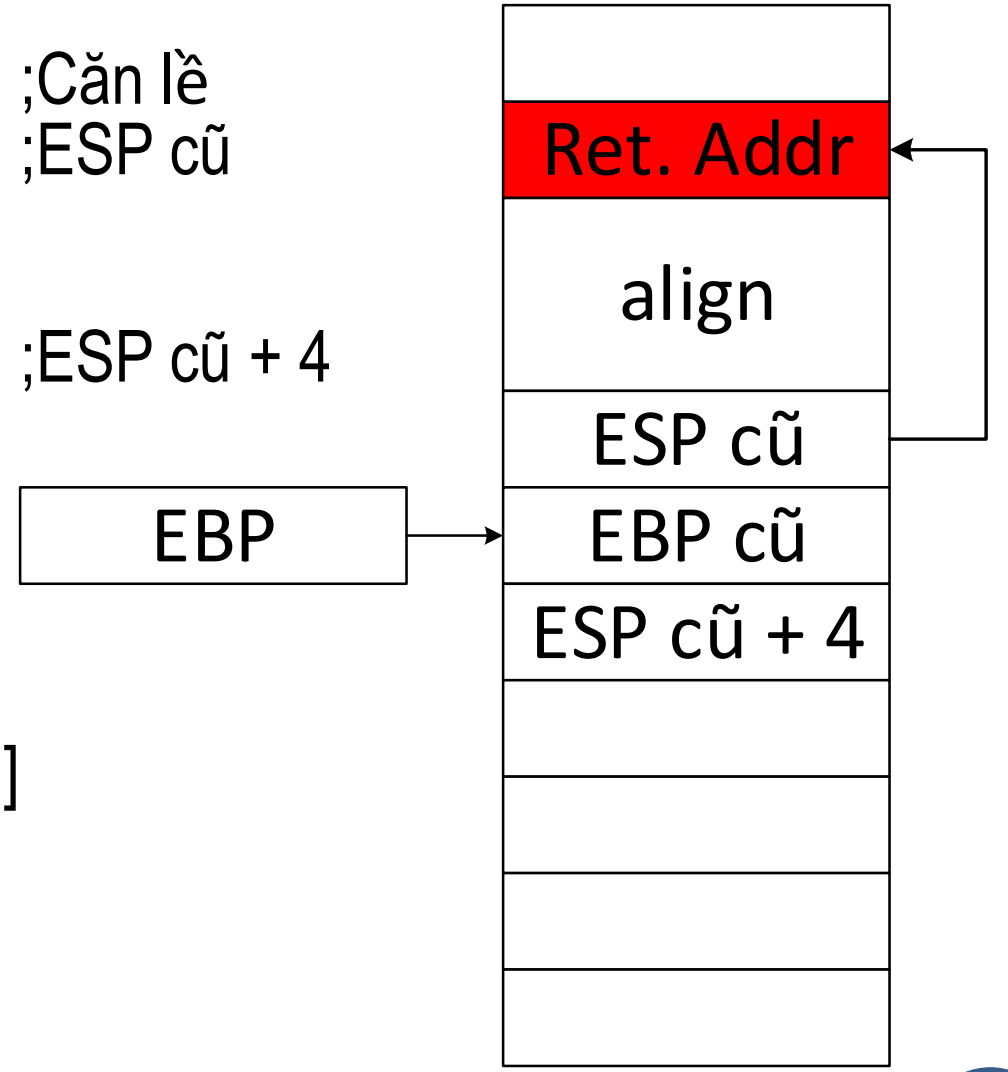
Cấu trúc mới cho hàm main()
(gcc 5.4 trở về sau)

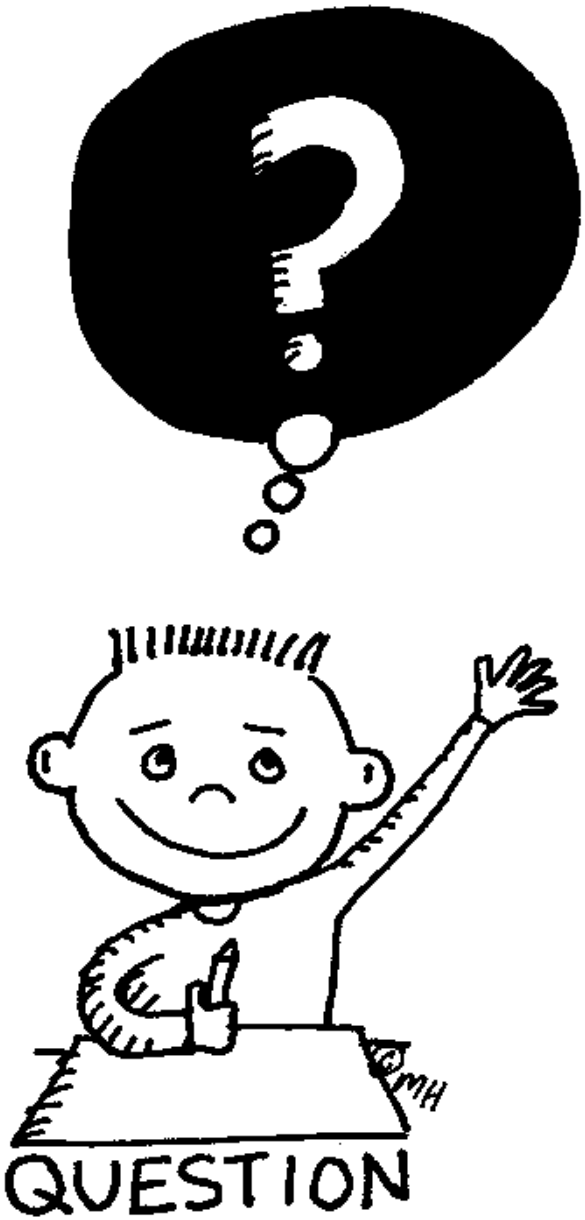
Cấu trúc hàm main() sinh bởi gcc 5.4

```
;Phần dẫn nhập
lea     ecx, [esp+4]
and     esp, fffffff0h              ;Căn lề
push    DWORD PTR [ecx-4]           ;ESP cũ
push    ebp
mov     ebp, esp
push    ecx                         ;ESP cũ + 4

;Phần thân hàm
;....

;Phần kết thúc
mov     ecx,DWORD PTR [ebp-4]
mov     esp, ebp
pop     ebp
lea     esp, [ecx-4]
ret
```




QUESTION

Một số công cụ cần thiết

- IDA Pro with Hex-Rays
- GDB
- GCC
- NASM (có thể dùng qua SASM)

GDB
The GNU Project Debugger

Tự học

- Làm quen với các công cụ
- Ôn lại kiến thức về hợp ngữ (có thể sử dụng tài liệu [2])
- https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/
- https://reverseengineering.stackexchange.com/questions/15173/what-is-the-purpose-of-these-instructions-before-the-main-preamble

Tự học

- https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://stackoverflow.com/questions/38781118/why-is-gcc-generating-an-extra-return-address/38783034
- http://unixwiz.net/techtips/win32-callconv-asm.html
- https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/