KHAI THÁC
LỖ HỔNG PHẦN MỀM

Bài 1. Kiến thức nền tảng

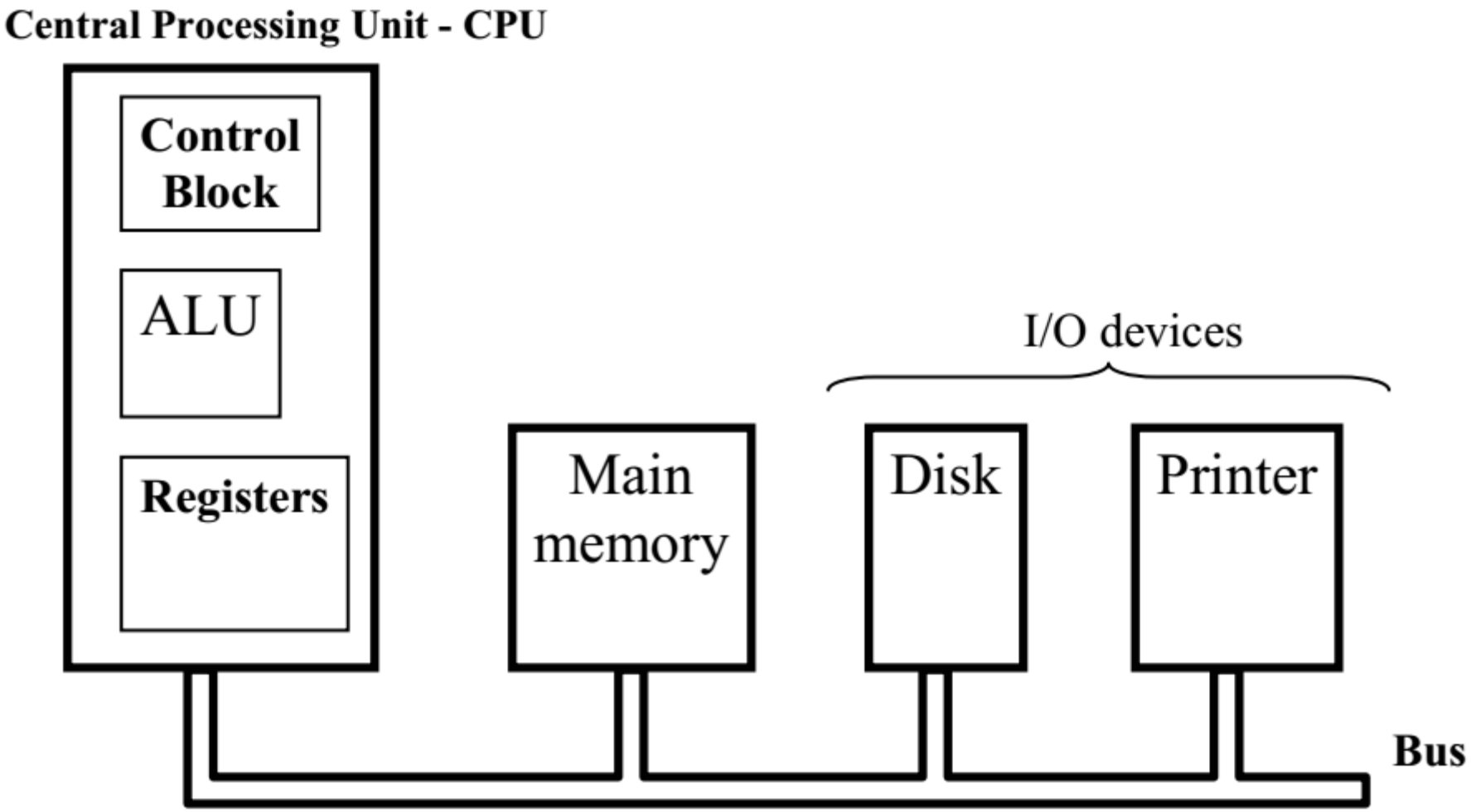
1. Kiến trúc máy tính
2. Stack
3. Hàm và gọi hàm
4. Lỗ hổng phần mềm

Tài liệu tham khảo

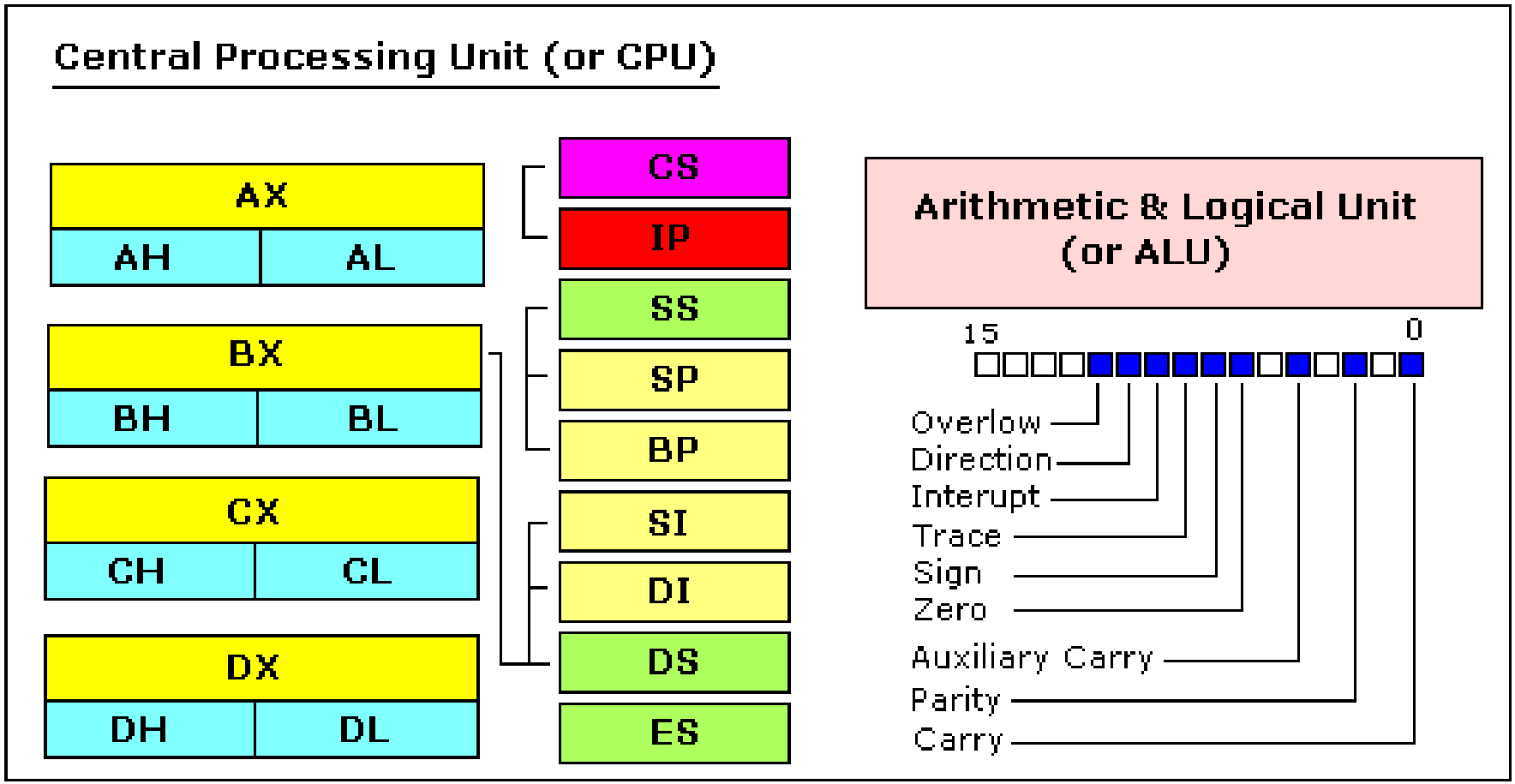
1. Nguyễn Thành Nam, **Chương 2// Nghệ thuật tận dụng lỗi phần mềm**, NXB Khoa học & Kỹ thuật, 2009
2. **NASM Assembly Language Tutorials** https://asmtutor.com

1. Kiến trúc máy tính
2. Stack
3. Hàm và gọi hàm
4. Lỗ hổng phần mềm

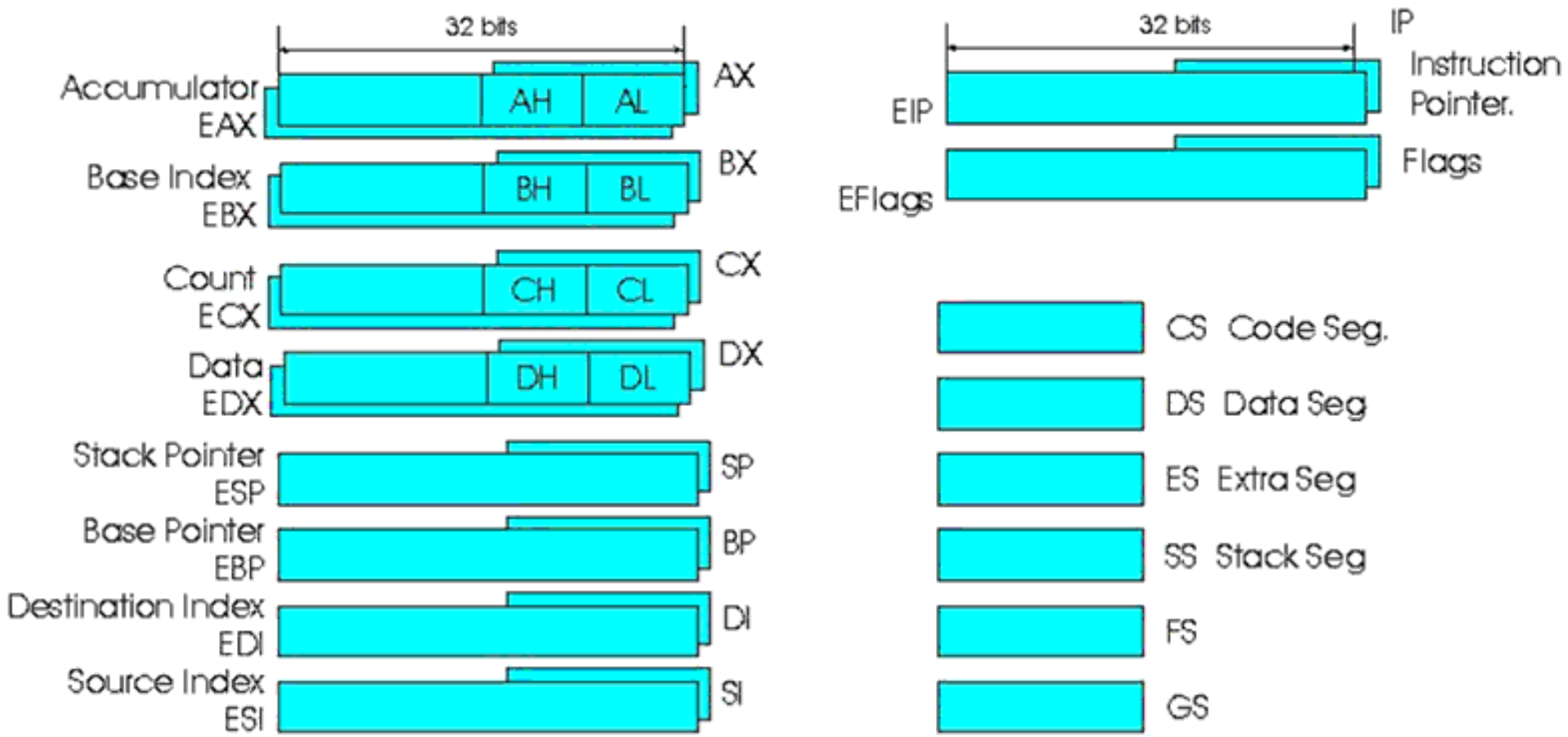
Kiến trúc máy tính



Các thanh ghi của CPU Intel 8086



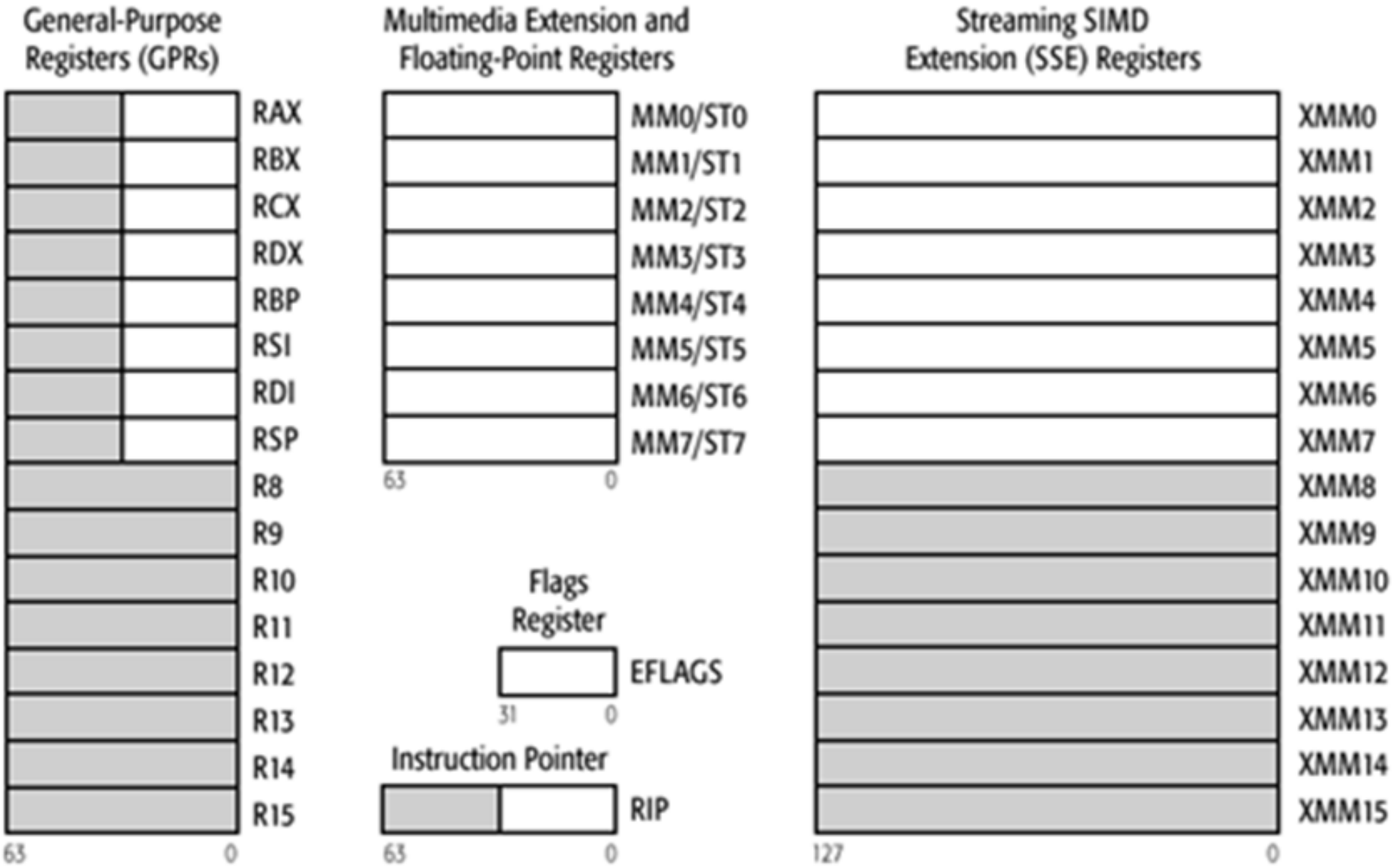
Các thanh ghi 80x86 (32 bít)



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

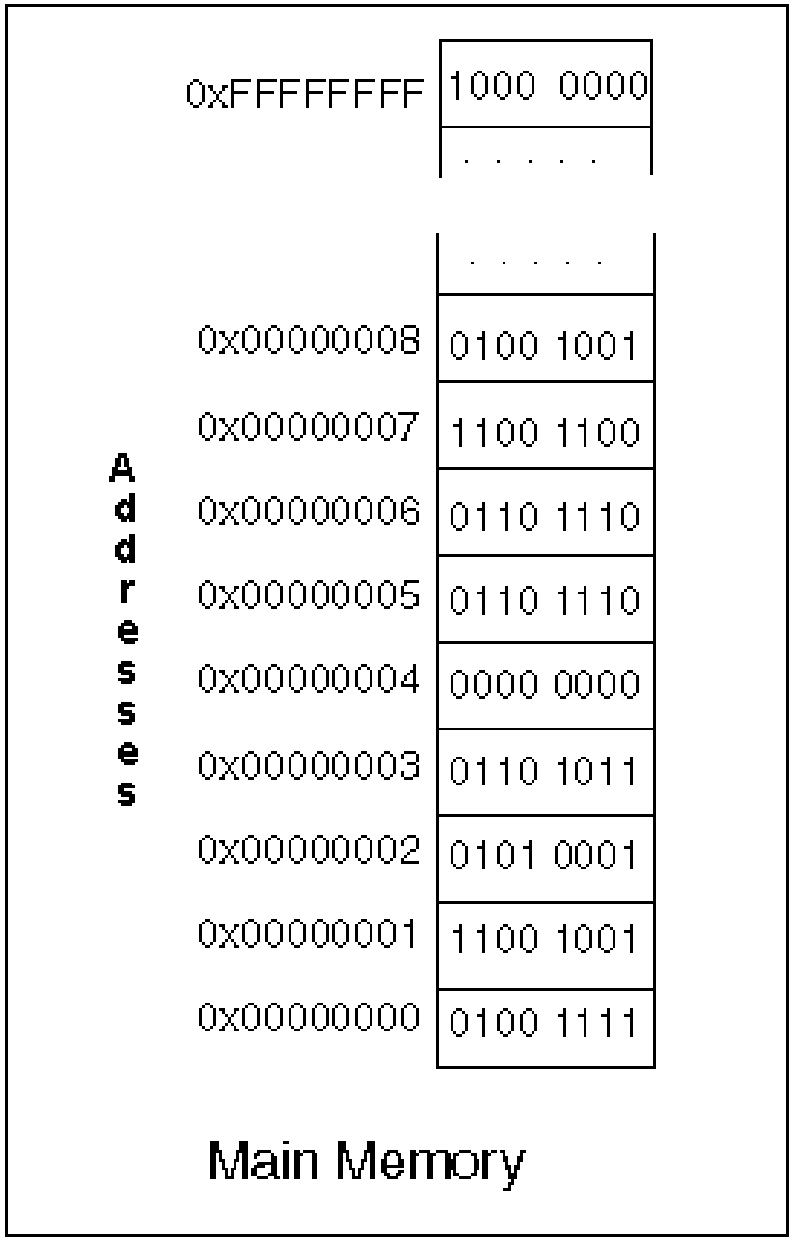
Các thanh ghi x86-64 (64,128 bít)



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



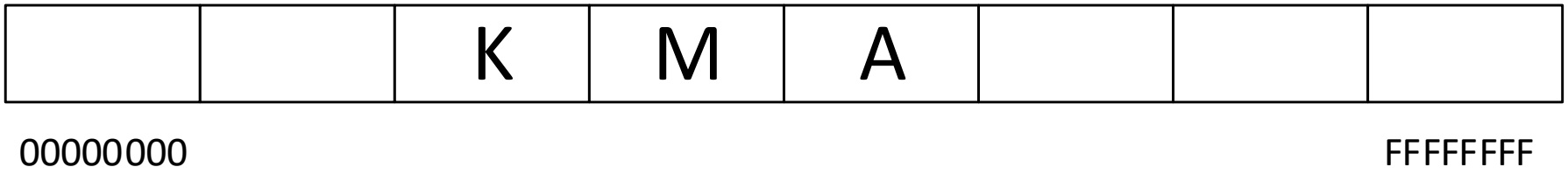
Mô hình bộ nhớ tuyến tính

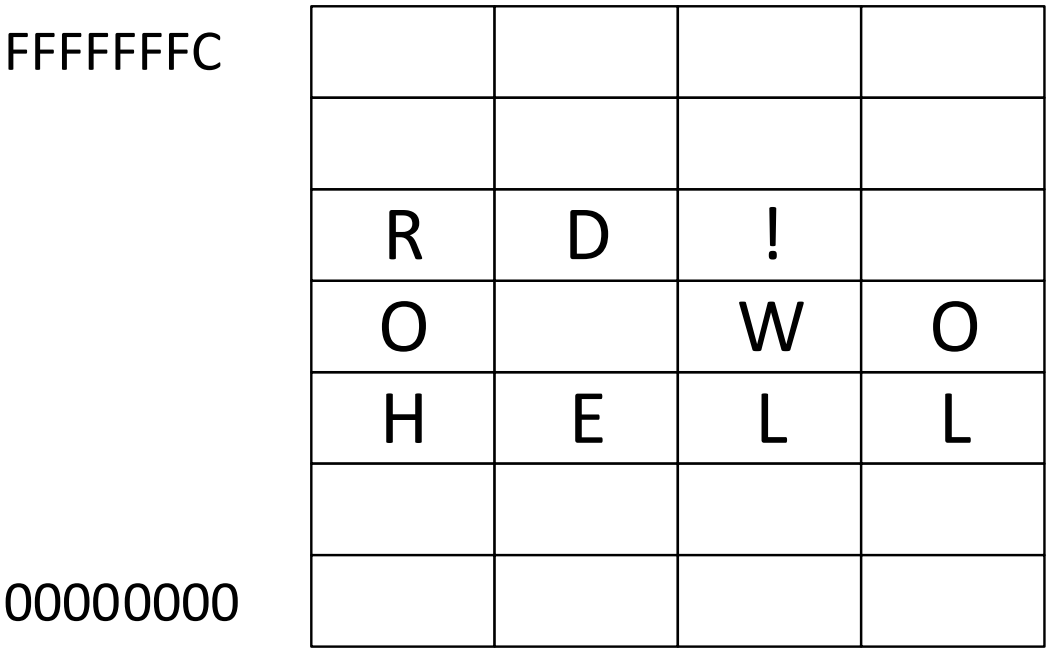
- Flat memory model
- Là mô hình bộ nhớ (dưới một cách nhìn nào đó) mà các ô nhớ được đánh địa chỉ liên tiếp từ 0 đến MAXBYTE-1
- Các mô hình khác:
  - phân đoạn (segmented)
  - phân trang (paged)

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



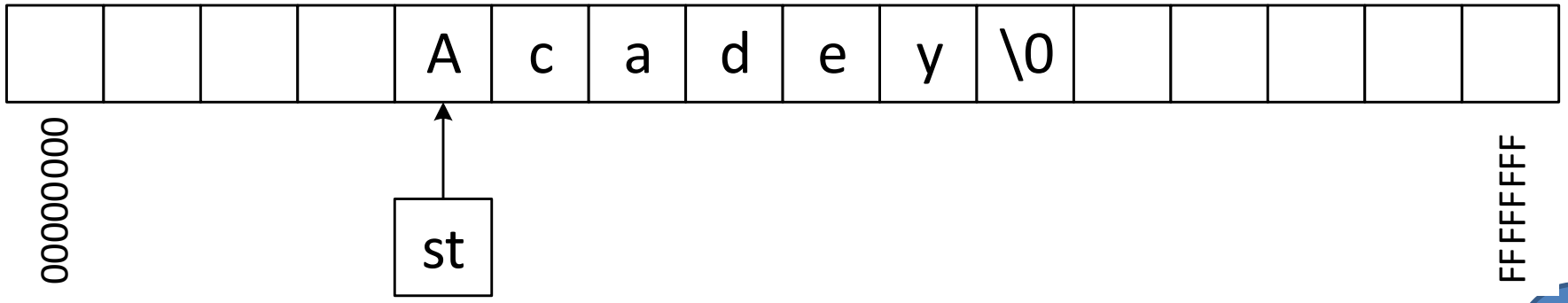


Địa chỉ thường là số hexa

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

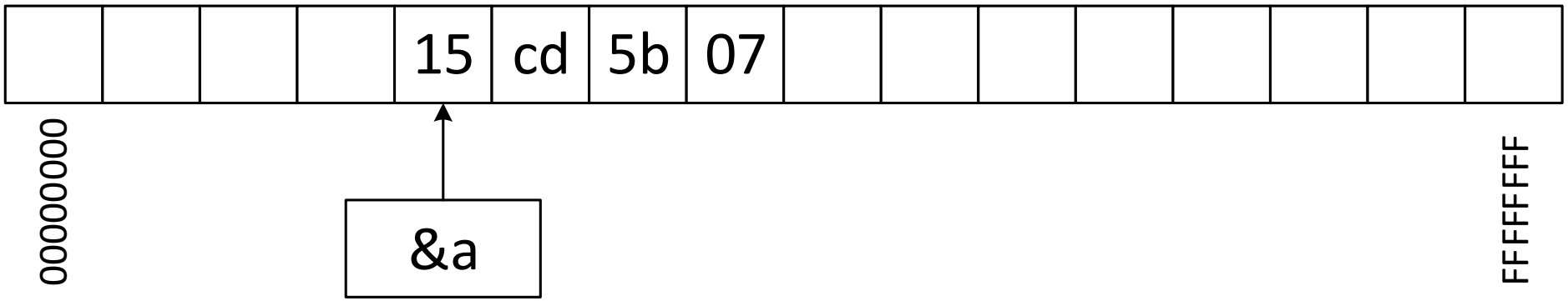
```
char st[100];
gets(st);
```



Trật tự byte: little-endian

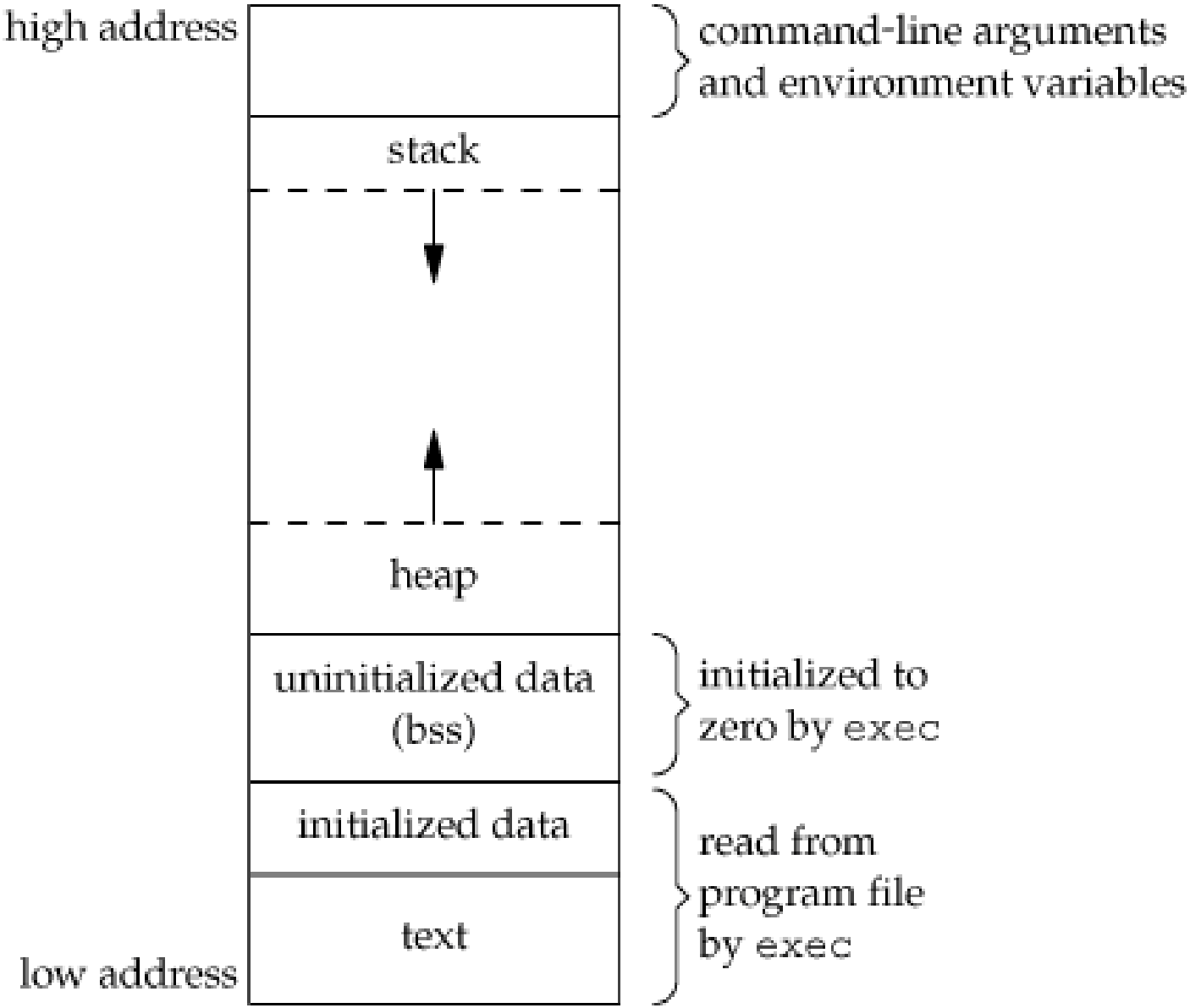
- Các máy tính hiện đại sử dụng little-endian trong biểu diễn số

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



1. Kiến trúc máy tính
2. Stack
3. Hàm và gọi hàm
4. Lỗ hổng phần mềm

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

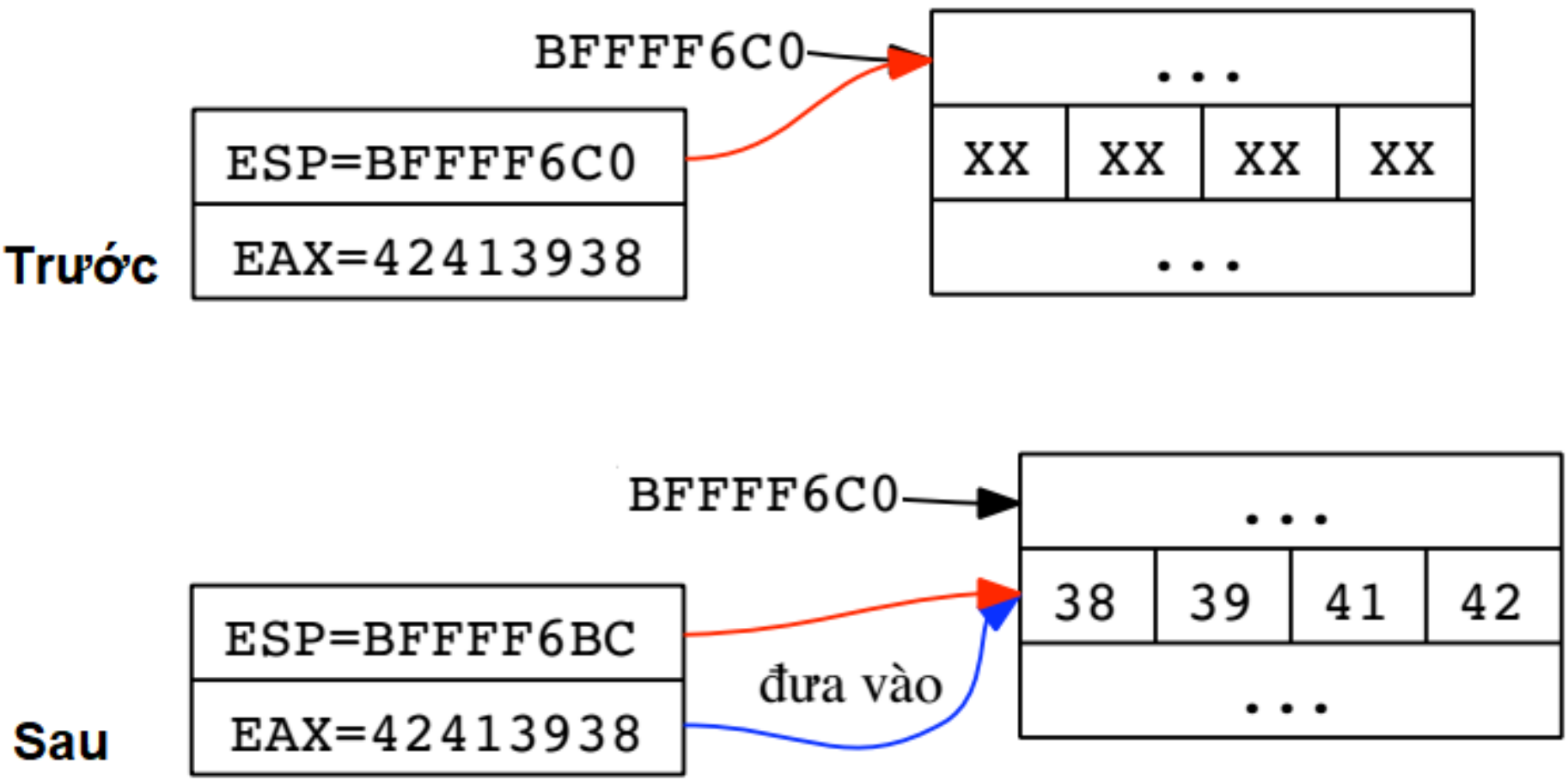
- Chứa các biến cục bộ
- Lưu địa chỉ trả về khi gọi hàm
- Truyền tham số khi gọi hàm
- Lưu giữ con trỏ "this" trong lập trình hướng đối tượng

Thao tác trên ngăn xếp

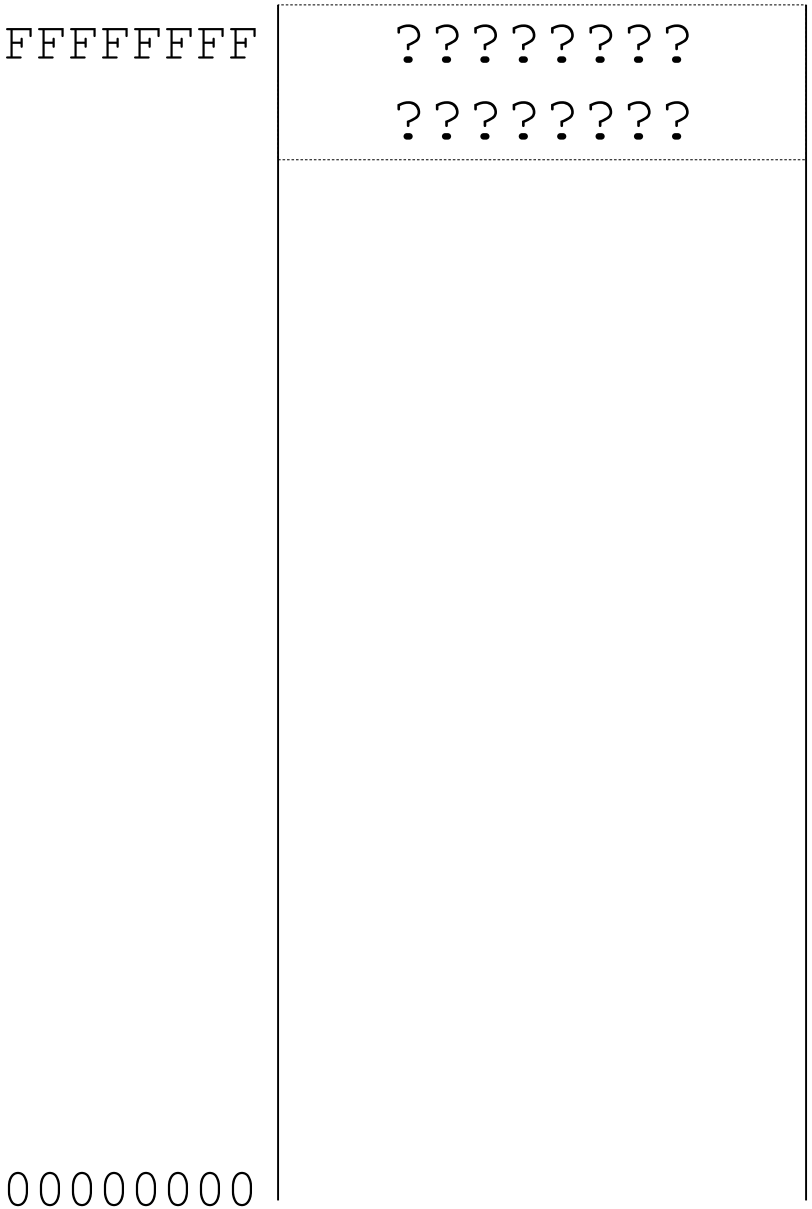
- Trong x86 mỗi phần tử stack là 4 byte
- Stack được quản lý qua ESP
- Hai thao tác cơ bản: PUSH và POP
- PUSH
  - Giảm giá trị của ESP: ESP = ESP – 4
  - Ghi đối số (4 byte) vào [ESP]
- POP
  - Đọc 4 byte tại [ESP] vào đối số
  - Tăng giá trị của ESP: ESP = ESP + 4

Thao tác trên ngăn xếp

Câu lệnh PUSH EAX



Stack Frame

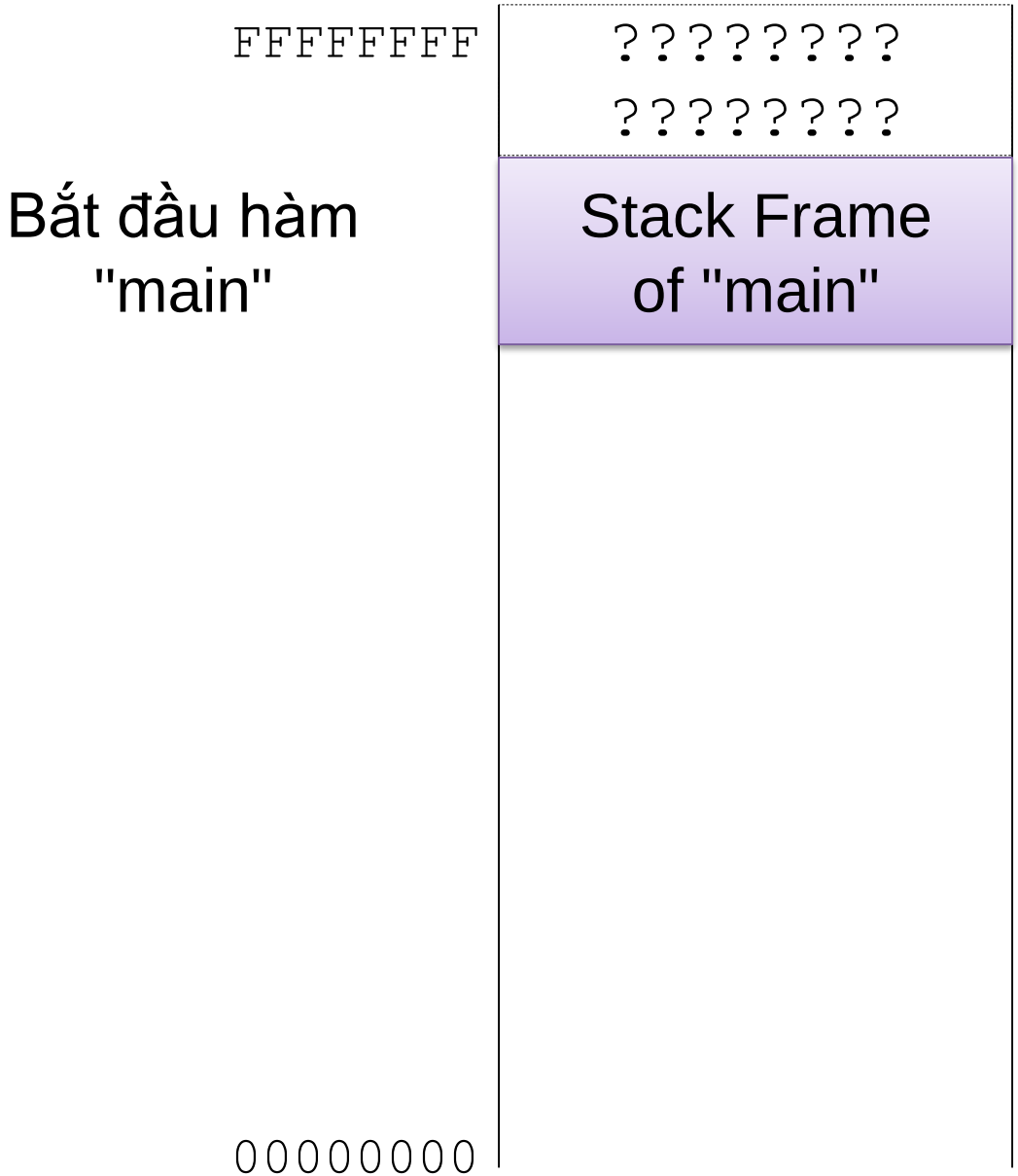



```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame

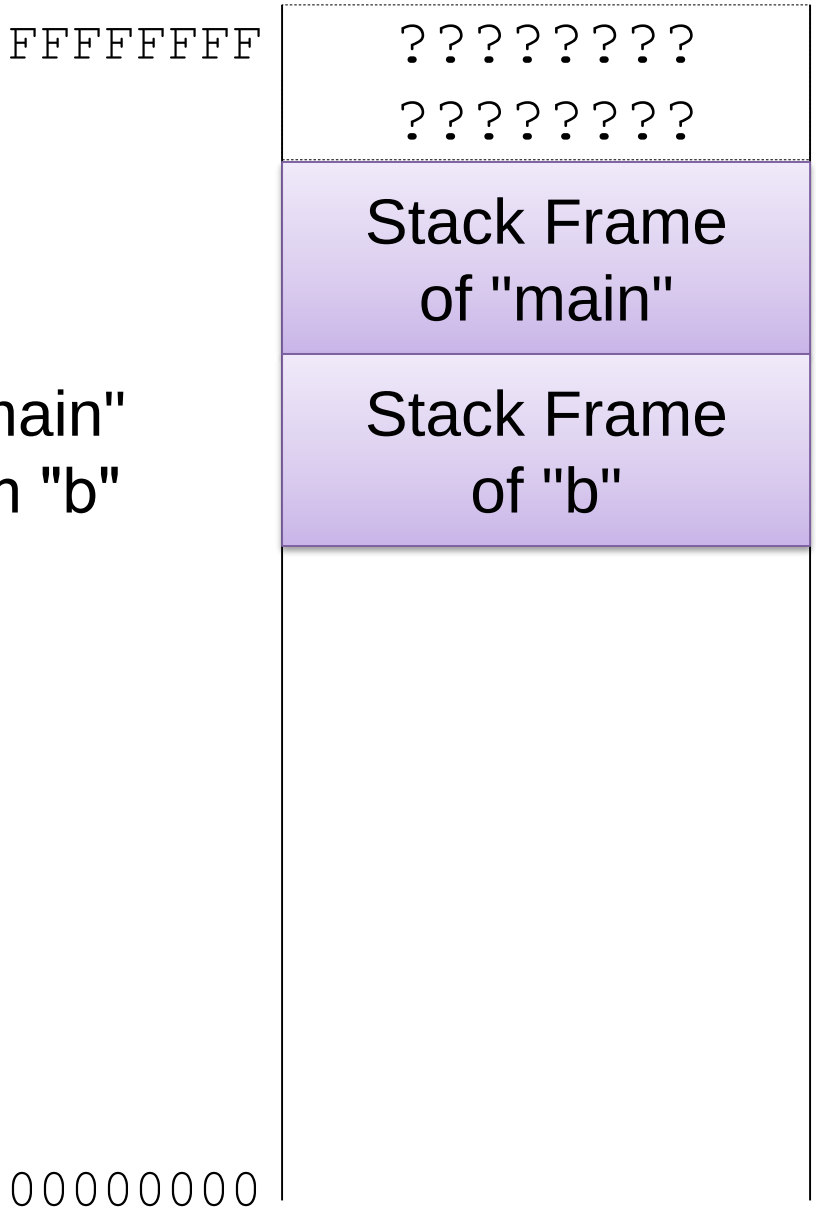



```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame



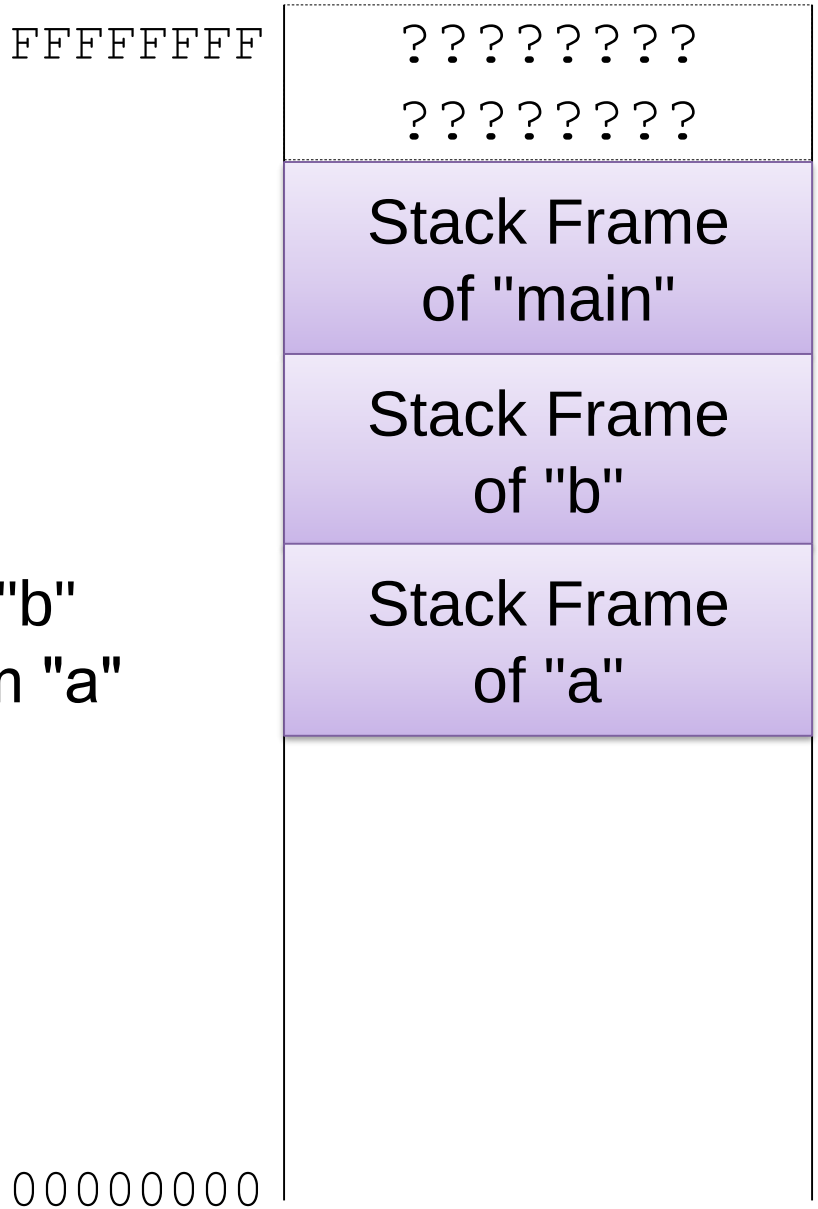
Hàm "main" gọi hàm "b"

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

void b()
{
    a();
}

void a()
{
}
```

Stack Frame



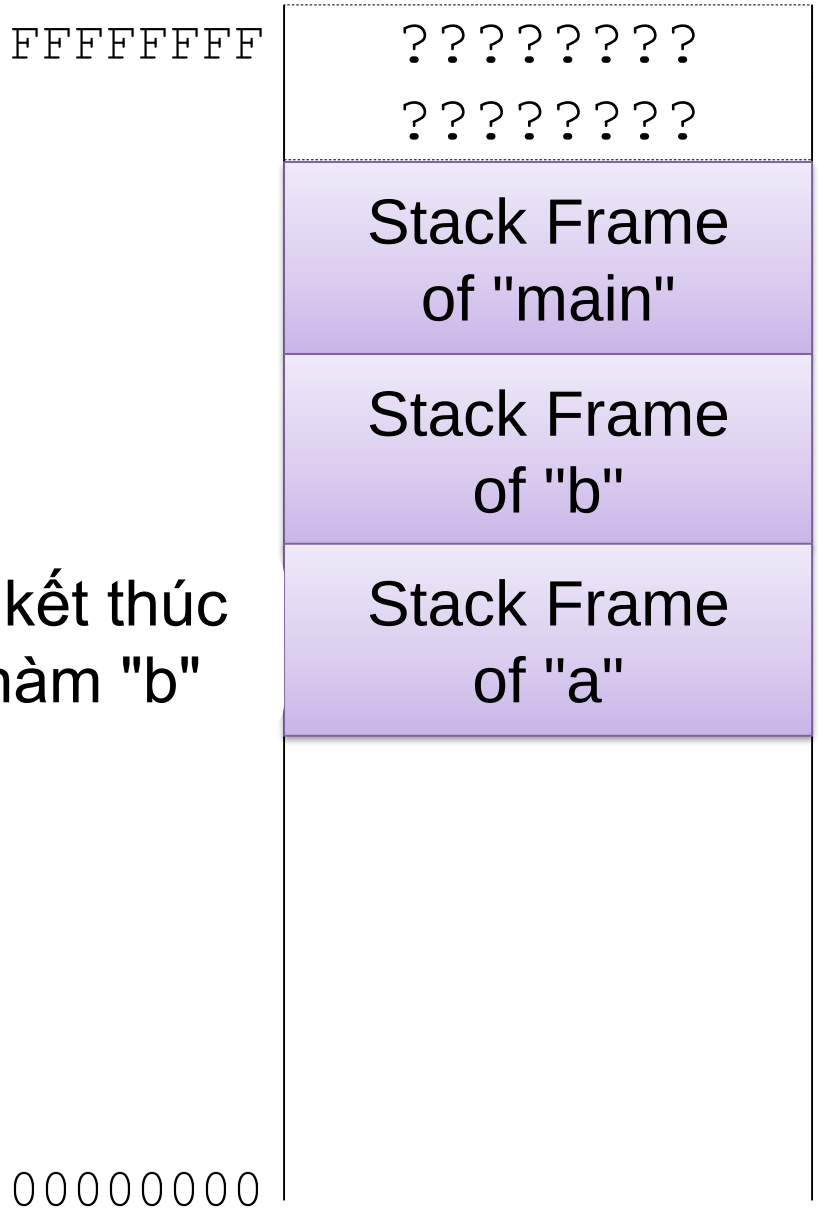
Hàm "b" gọi hàm "a"

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

void b()
{
    a();
}

void a()
{
}
```

Stack Frame



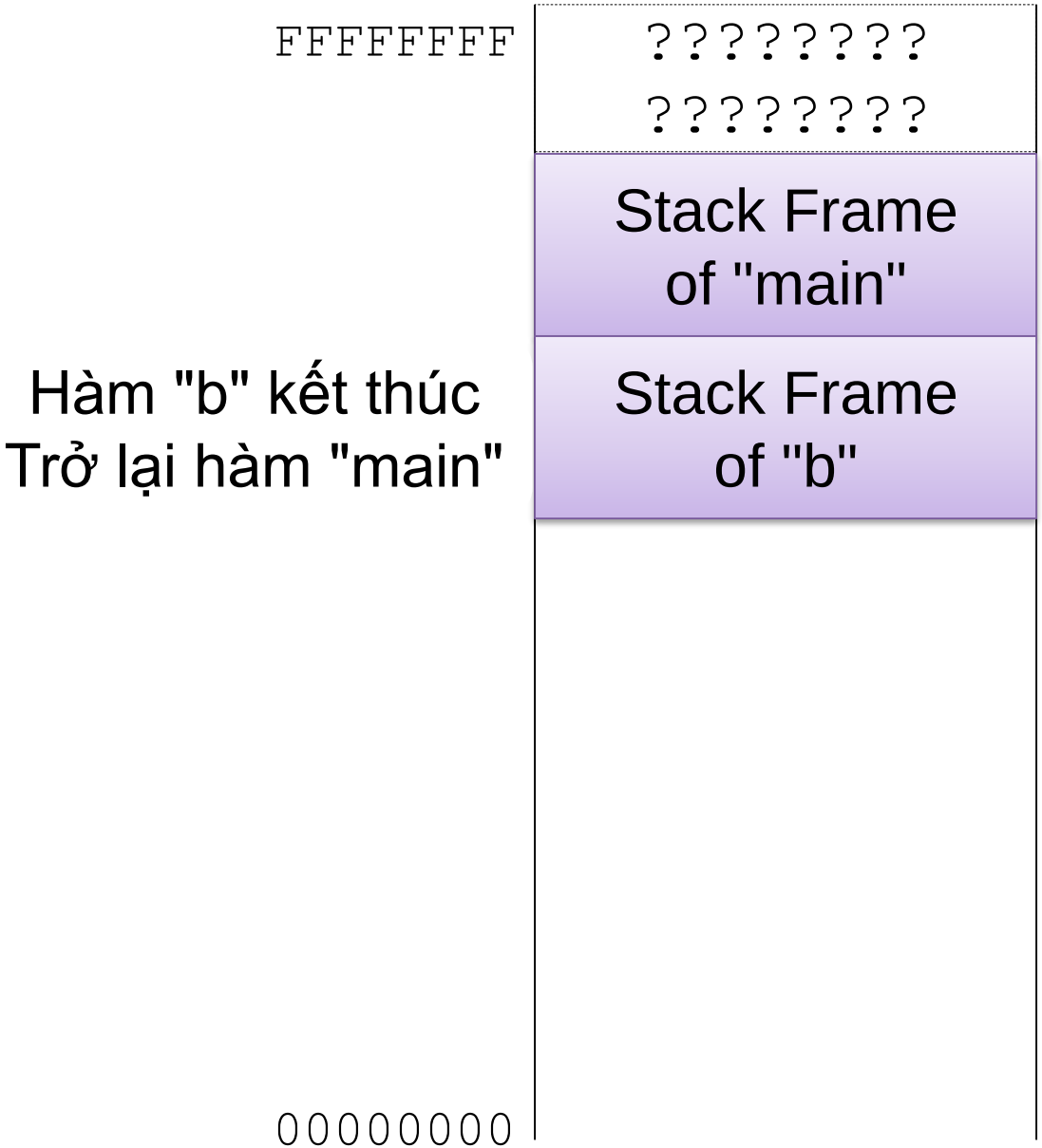
Hàm "a" kết thúc
Trở lại hàm "b"

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

void b()
{
      a();
}

void a()
{
}
```

Stack Frame

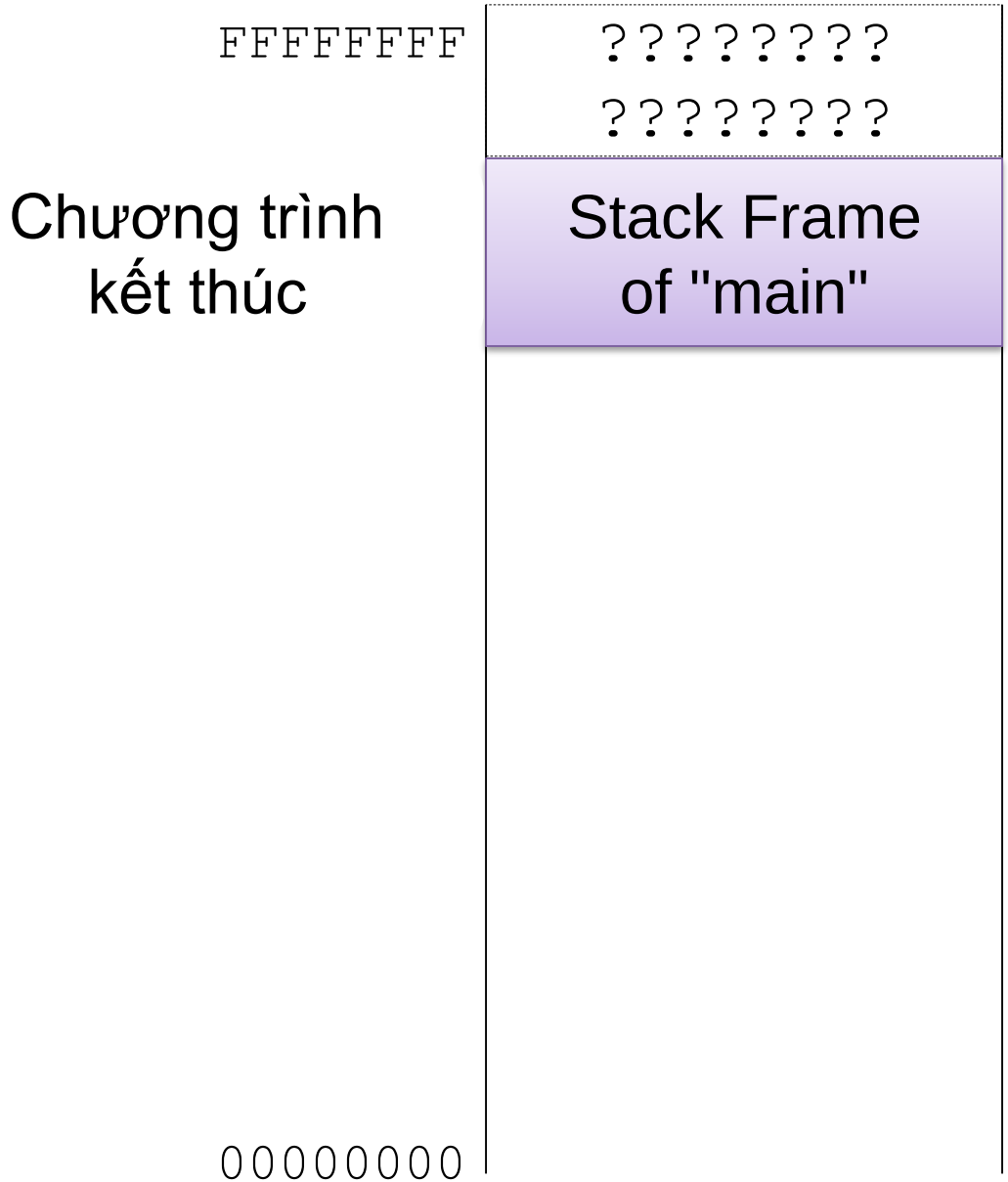



```
void main()
{
      b();
}

void b()
{
      a();
}

void a()
{
}
```

Stack Frame




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

void b()
{
      a();
}

void a()
{
}
```

1. Kiến trúc máy tính
2. Stack
3. Hàm và gọi hàm
4. Lỗ hổng phần mềm

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
_start:
    mov     eax, 11
    mov     ebx, 22
    mov     edx, 33
    call    MySimpleProc    ;EAX = 11+22-33 = 0
    mov     ebx, 0
    mov     eax, 1
    int     80h
```

Việc gọi một hàm bao gồm:

- Nạp các tham số cần thiết
- Thực hiện lệnh CALL

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; tham số được truyền từ phải sang trái
    - 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:
        push    10              ; Truyền tham số thứ 2
        push    20              ; Truyền tham số thứ 1
        call    SubSquare       ; ESP được bảo toàn
        add     esp, 8          ; Cân bằng stack, tổng cộng 8 byte tham số
        ret
```

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

Sự trở về từ hàm được gọi

```
08048060 <MyProc>:
 8048060:   01 d8              add   eax,ebx
 8048062:   29 d0              sub   eax,edx
 8048064:   c3                 ret

08048065 <_start>:
 8048065:   b8 0b 00 00 00     mov   eax,0xb
 804806a:   bb 16 00 00 00     mov   ebx,0x16
 804806f:   ba 21 00 00 00     mov   edx,0x21
 8048074:   e8 e7 ff ff ff     call  8048060 <MyProc>
 8048079:   bb 00 00 00 00     mov   ebx,0x0
 804807e:                      mov   eax,0x1
 8048083:                      int   0x80
```

Mã hợp ngữ

Sự trở về từ hàm được gọi

```
08048060 <MyProc>:
 8048060:   01 d8                add    eax,ebx
 8048062:   29 d0                sub    eax,edx
 8048064:   c3                   ret

08048065 <_start>:
 8048065:   b8 0b 00 00 00       mov    eax,0xb
 804806a:   bb 16 00 00 00       mov    ebx,0x16
 804806f:   ba 21 00 00 00       mov    edx,0x21
 8048074:   e8 e7 ff ff ff       call   8048060 <MyProc>
 8048079:   bb 00 00 00 00       mov    ebx,0x0
 804807e:   b8 01 00 00 00       mov    eax,0x1
 8048083:   cd 80                int    0x80
```

- Mã máy
- Độ dài lệnh mã máy là không cố định

Sự trở về từ hàm được gọi

```
08048060 <MyProc>:
 8048060:   01 d8
 8048062:   29 d0
 8048064:   c3

08048065 <_sta
 8048065:   b
 804806a:   bb 16 00 00 00      mov   ebx,0x16
 804806f:   ba 21 00 00 00      mov   edx,0x21
 8048074:   e8 e7 ff ff ff      call  8048060 <MyProc>
 8048079:   bb 00 00 00 00      mov   ebx,0x0
 804807e:   b8 01 00 00 00      mov   eax,0x1
 8048083:   cd 80               int   0x80
```

- Địa chỉ của lệnh mã máy khi thực thi chương trình
- Tên hàm là địa chỉ của lệnh đầu tiên trong hàm
- Các lệnh được thực hiện tuần tự từ trên xuống, trừ khi có lệnh nhảy.

Sự trở về từ hàm được gọi

```
08048060 <MyProc>:
 8048060:   01 d8                   add   eax,ebx
 8048062:   29 d0                   sub   eax,edx
 8048064:   c3                      ret

08048065 <_start>:
 8048065:   b8 0b 00 00 00          mov   eax,0xb
 804806a:   bb 16 00 00 00          mov   ebx,0x16
 804806f:   ba 21 00 00 00          mov   edx,0x21
 8048074:   e8 e7 ff ff ff          call  8048060 <MyProc>
 8048079:   bb 00 00 00 00          mov     0x0
 804807e:
 8048083:
```

Lệnh "call" sẽ khiến chương trình nhảy đến lệnh ở 08048060

Sự trở về từ hàm được gọi

```
08048060 <MyProc>:
 8048060:   01 d8                   add    eax,ebx
 8048062:   29 d0                   sub    eax,edx
 8048064:   c3                      ret

08048065 <_start>
 8048065:   b8 0b
 804806a:   bb 16
 804806f:   ba 21
 8048074:   e8 e7 ff ff ff          call   8048060 <MyProc>
 8048079:   bb 00 00 00 00          mov    ebx,0x0
 804807e:   b8 01 00 00 00          mov    eax,0x1
 8048083:   cd 80                   int    0x80
```

- Tiếp theo lệnh "ret" sẽ là lệnh nào?
- Rõ ràng là lệnh ở 08048079
- Nhưng bằng cách nào????????

Sự trở về từ hàm được gọi

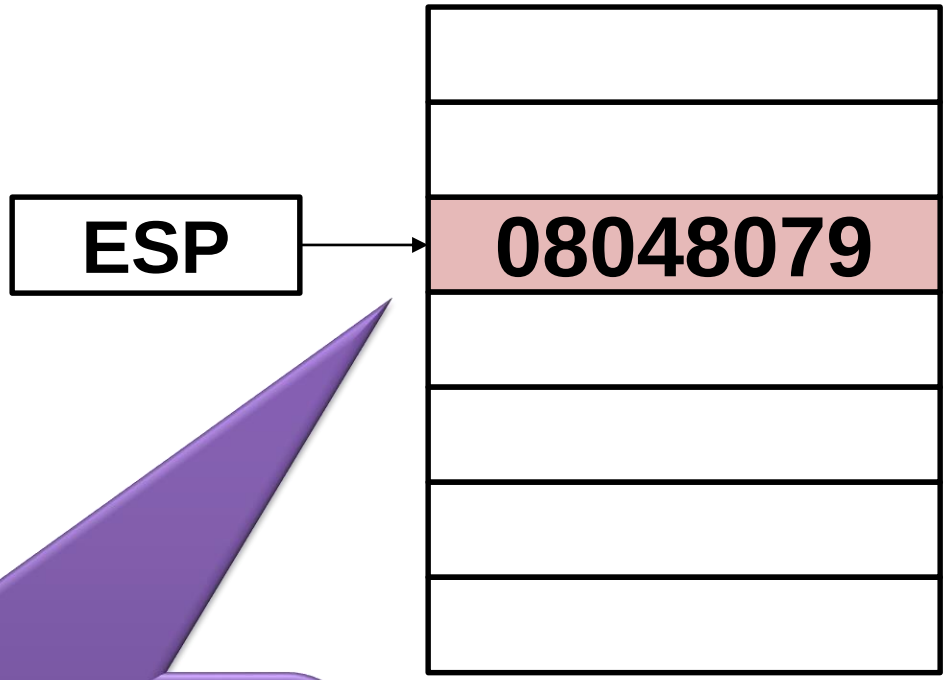
```
08048060 <MyProc>:
 8048060:   01 d8                   add    eax,ebx
 8048062:   29 d0                   sub    eax,edx
 8048064:   c3                      ret

08048065 <_start>:
 8048065:   b8 0b 0
 804806a:   bb 16 0
 804806f:   ba 21 00 00 00          mov    edx,0x21
 8048074:   e8 e7 ff ff ff          call   8048060 <MyProc>
 8048079:   bb 00 00 00 00          mov    ebx,0x0
 804807e:   b8 01 00 00 00          mov    eax,0x1
 8048083:   cd 80                   int    0x80
```

- Trước khi nhảy đến hàm được gọi, địa chỉ của lệnh kế tiếp sau lệnh "call" sẽ được đưa vào stack
- Đó gọi là "địa chỉ trở về" (return address). Một cách gần đúng:

```
push    08048079h
jmp     08088060h
```

Sự trở về từ hàm được gọi



- Khi hàm kết thúc, lệnh RET sẽ đưa EIP trở về 08048079
- Một cách gần đúng
  ret = pop eip

Cấu trúc hàm

Hàm:

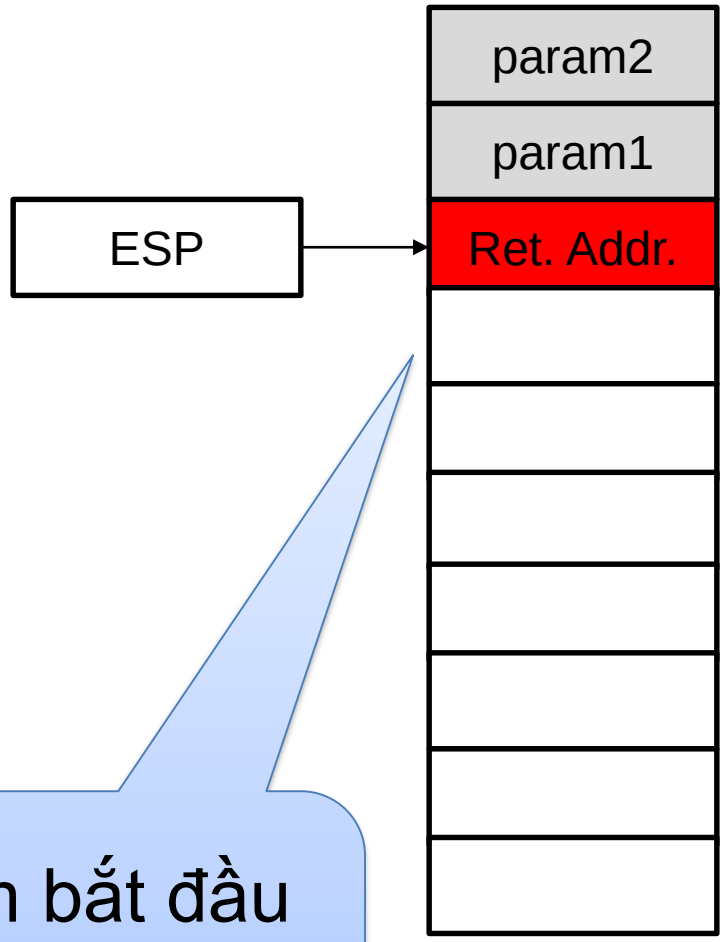
- Phần dẫn nhập;
- Phần thân hàm;
- Phần kết thúc;

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
```

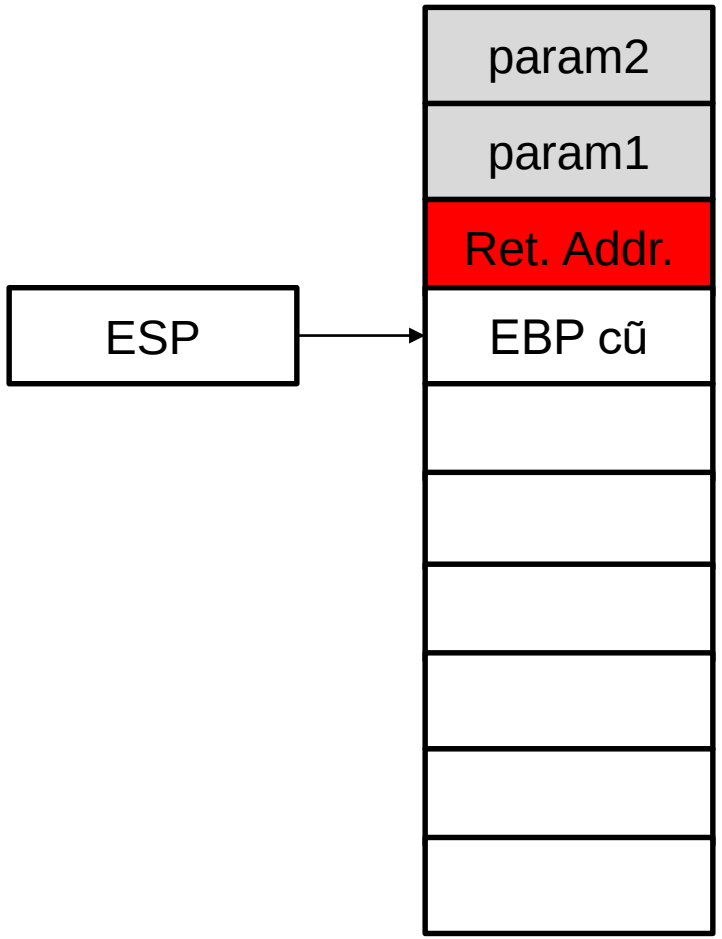


Cấu trúc hàm

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

;Phần thân hàm
;....

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

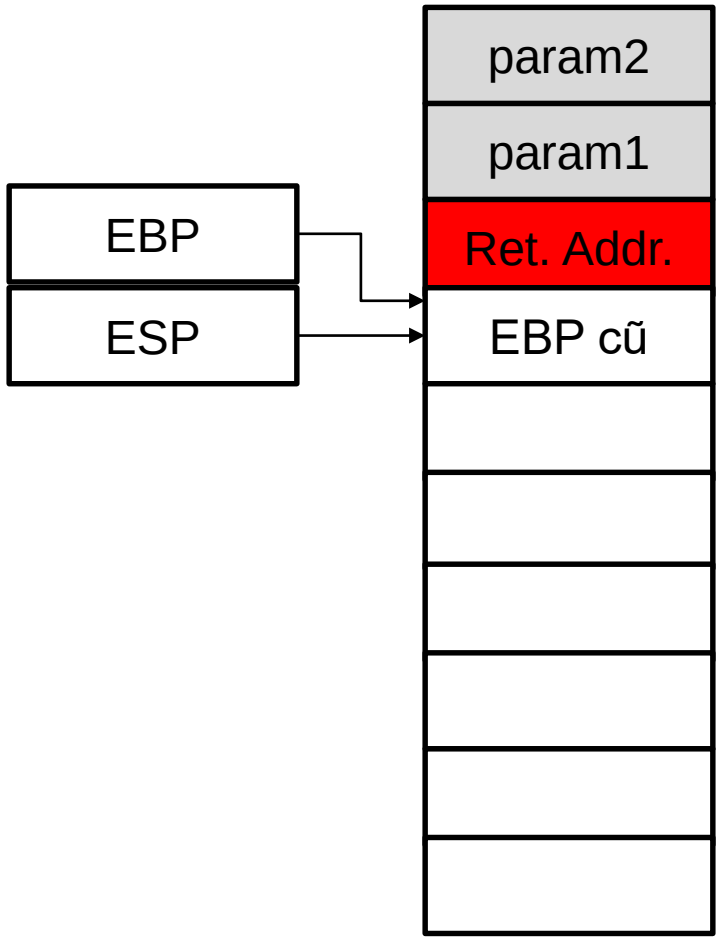


Cấu trúc hàm

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

;Phần thân hàm
;....

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

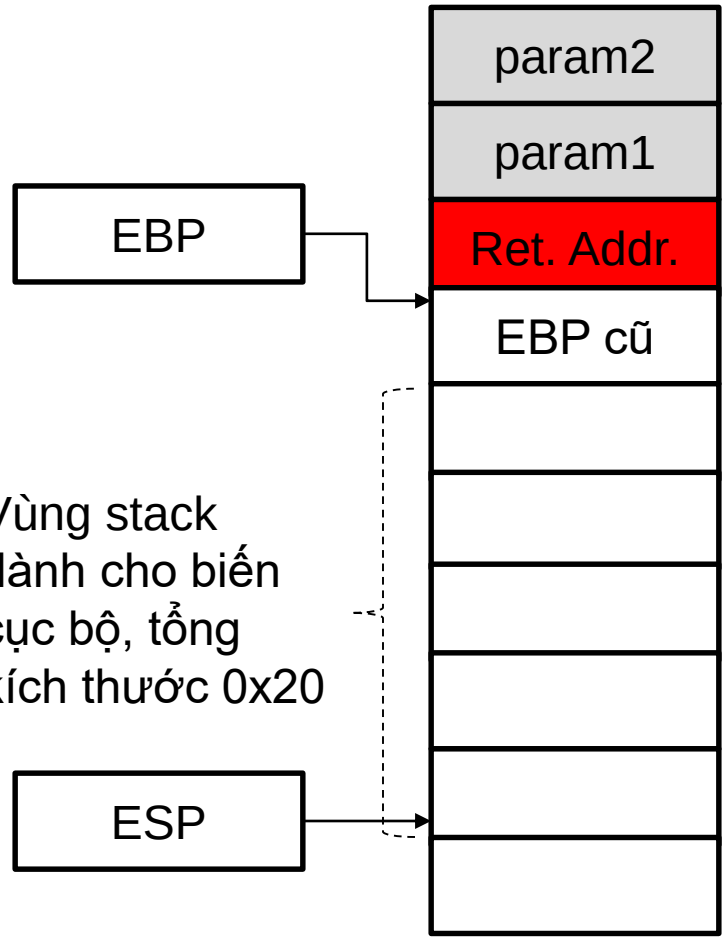


Cấu trúc hàm

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

;Phần thân hàm
;....

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

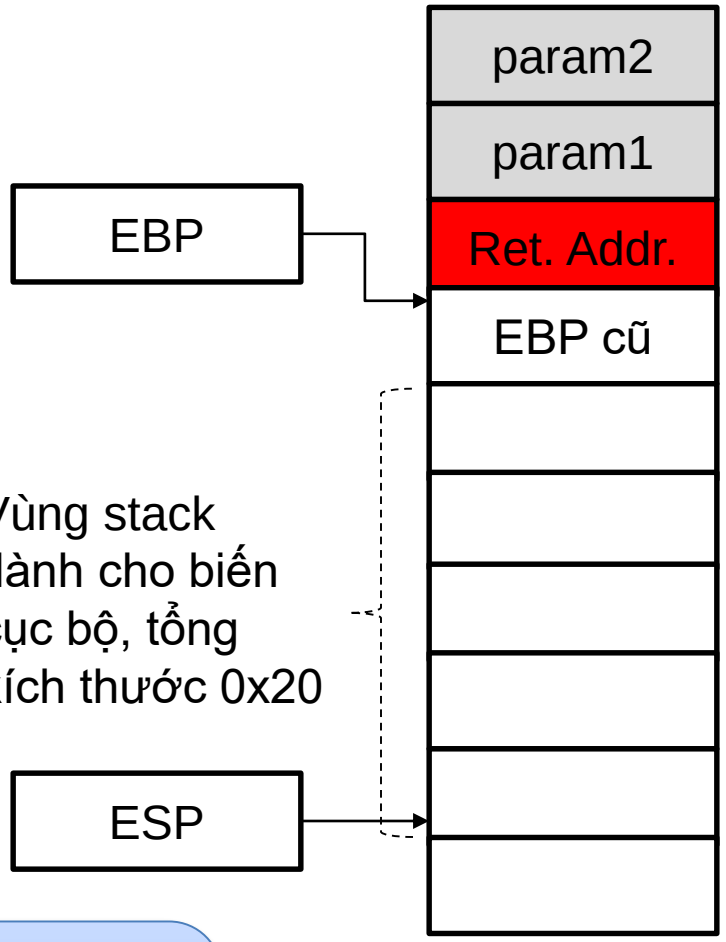


Cấu trúc hàm

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

;Phần thân hàm
;....

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



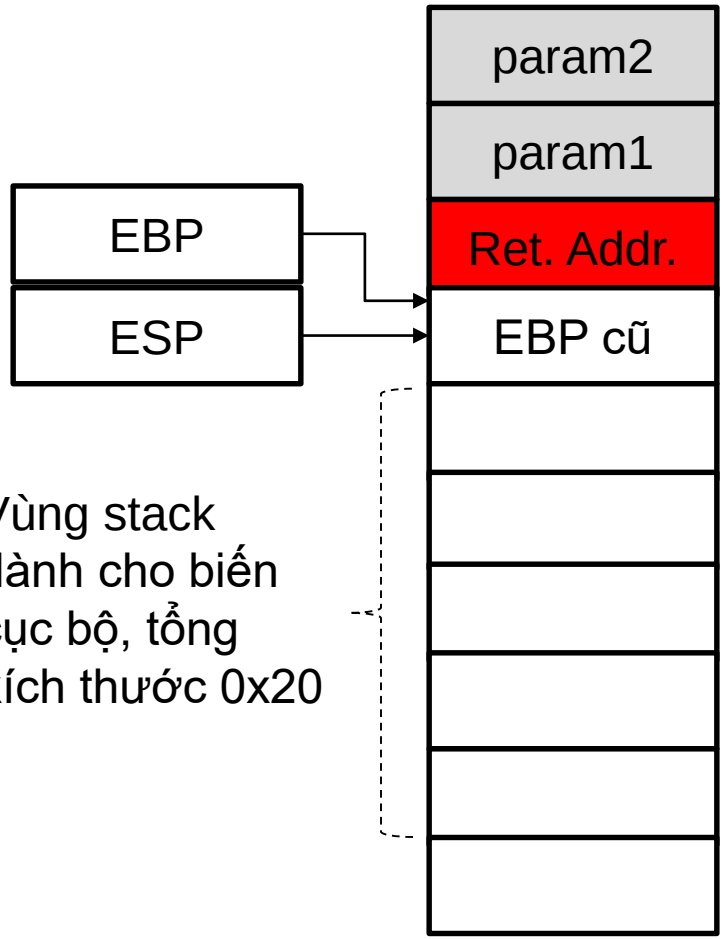
EBP sẽ không thay đổi trong thân hàm

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
```

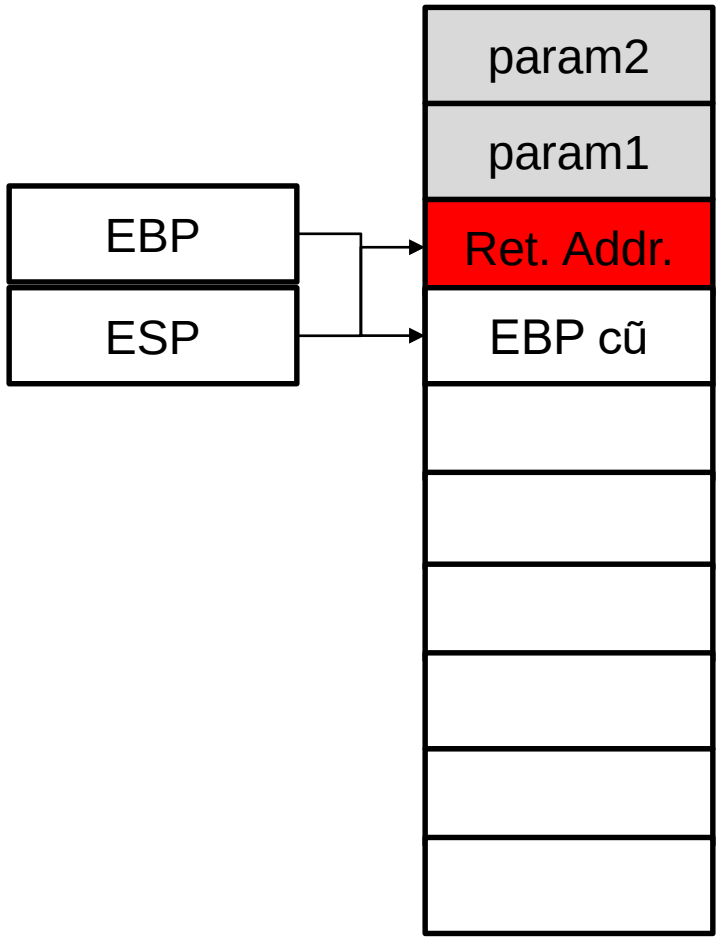


Cấu trúc hàm

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

;Phần thân hàm
;....

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



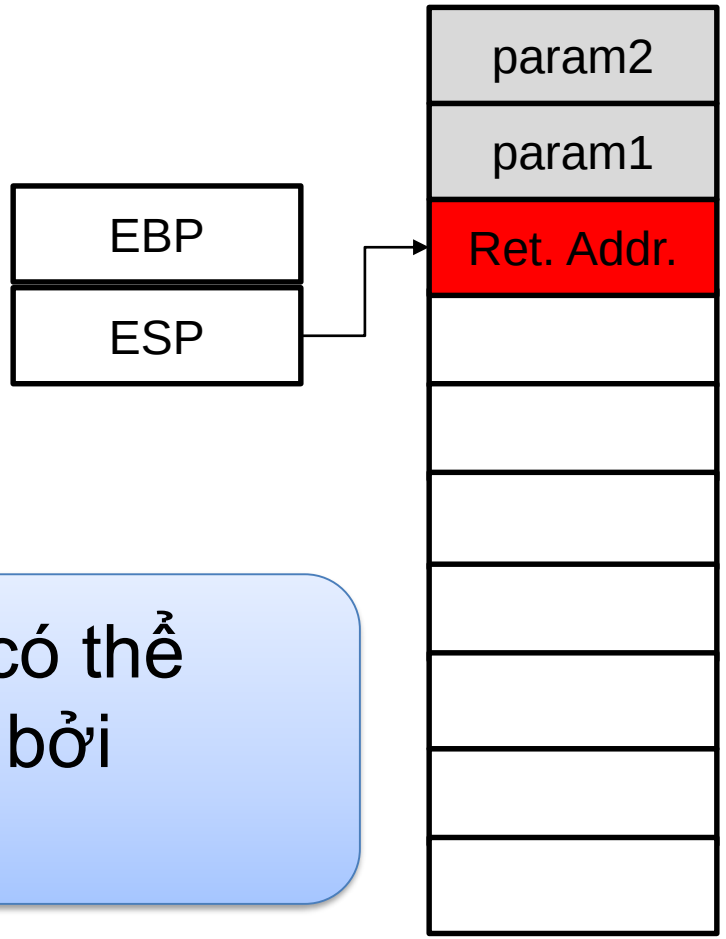
Cấu trúc hàm

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

;Phần thân hàm
;....

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

Cặp lệnh này có thể được thay thế bởi LEAVE

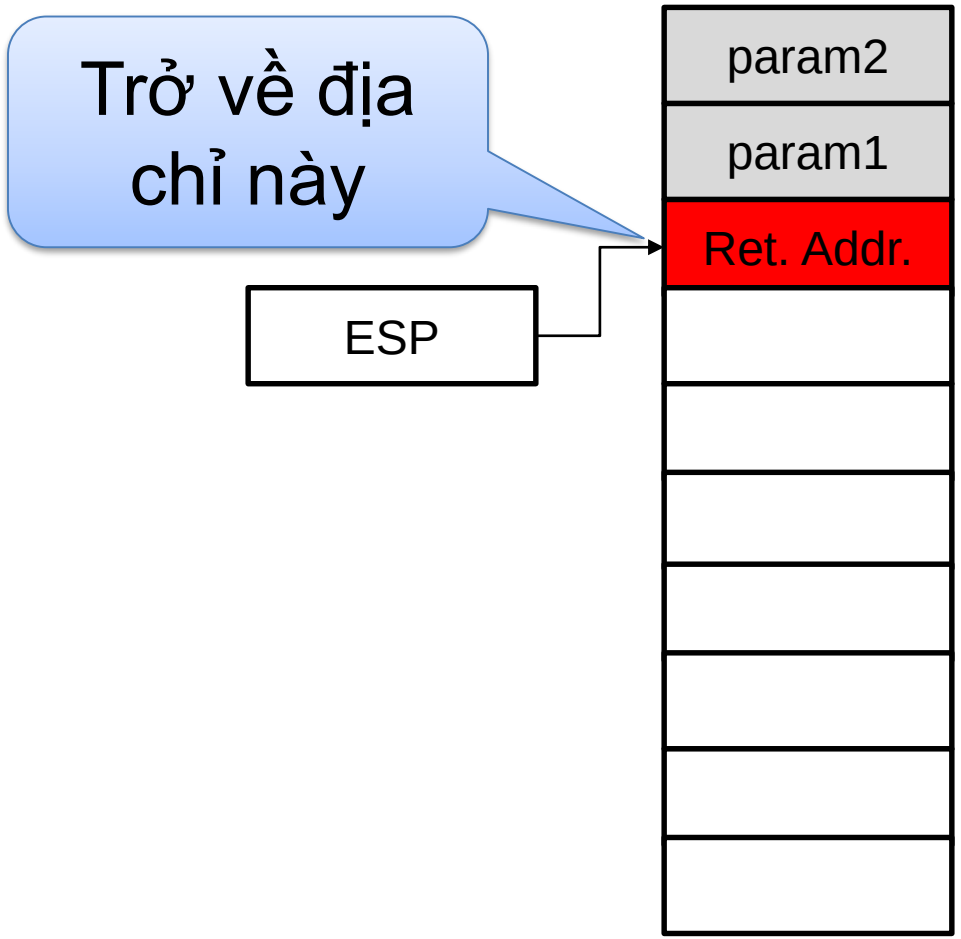


Cấu trúc hàm

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

;Phần thân hàm
;....

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



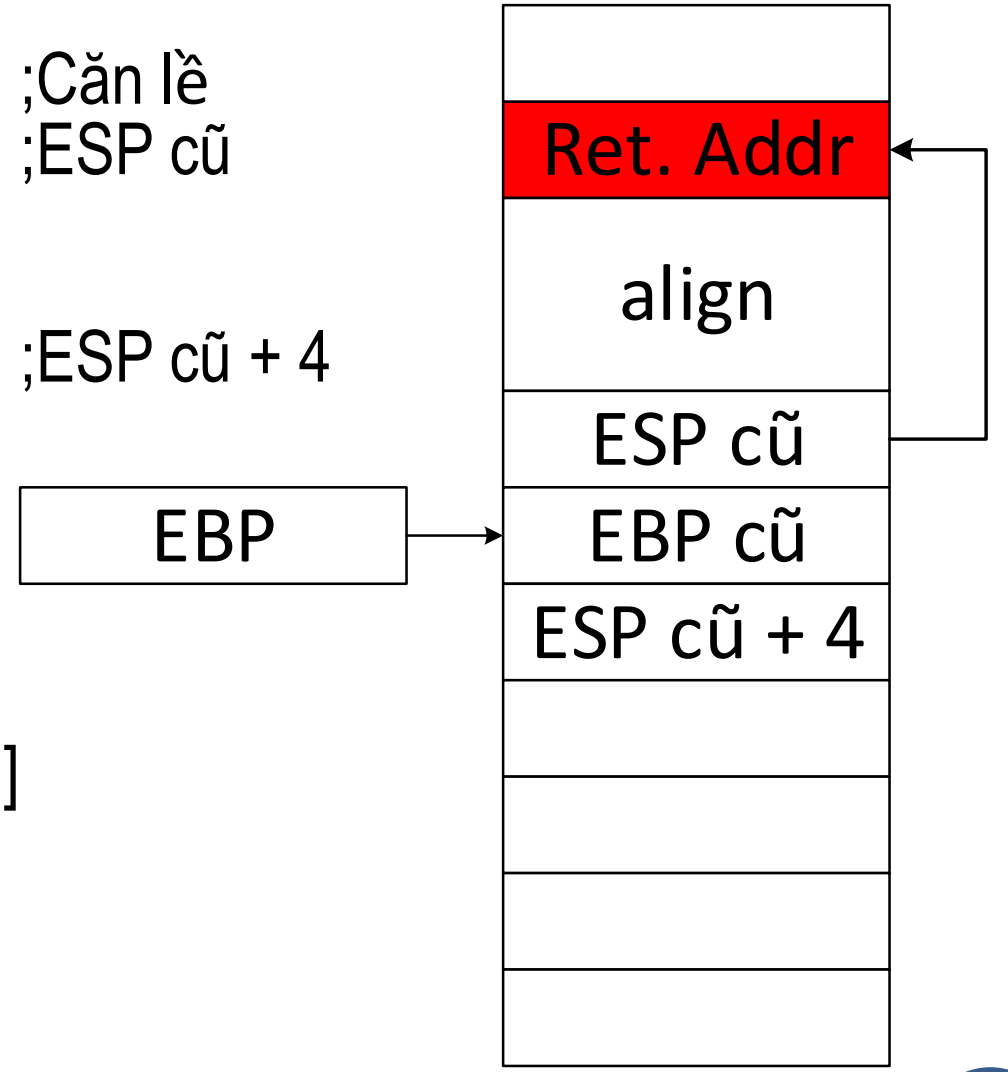
Cấu trúc mới cho hàm main()
(gcc 5.4 trở về sau)

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
```



1. Kiến trúc máy tính
2. Stack
3. Hàm và gọi hàm
4. Lỗ hổng phần mềm

Lỗ hổng phần mềm

- **Lỗ hổng phần mềm (software vulnerability)** là khiếm khuyết trong thiết kế, lập trình phần mềm mà kẻ tấn công có thể lợi dụng để làm thay đổi hoạt động bình thường của phần mềm

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng khiến dữ liệu có kích thước lớn có thể tràn ra khỏi vùng đệm để chứa nó

```
char st[10];
gets(st);
```

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng khiến dữ liệu chuỗi bị diễn giải như một chuỗi định dạng

```
char st[10];
gets(st);
printf(st);
```

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng khiến kết quả phép toán trên số nguyên bị diễn giải sai khi vượt quá phạm vi giá trị

```
int a, b;
scanf("%d", &b);
if(a+b < a)
        printf("b < 0");
```

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng khiến ký tự kết thúc chuỗi bị ghi đè

```
char *st1, *st2;
...
for(int i=0; i<=strlen(st1); i++)
        st1[i] = st2[i];
```

Lỗ hổng do lập trình

Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

Lỗ hổng trong vấn đề đồng bộ dữ liệu khiến một tiến trình vẫn xử lý dữ liệu cũ, trong khi dữ liệu đã được cập nhật bởi một tiến trình khác

Lỗ hổng do lập trình

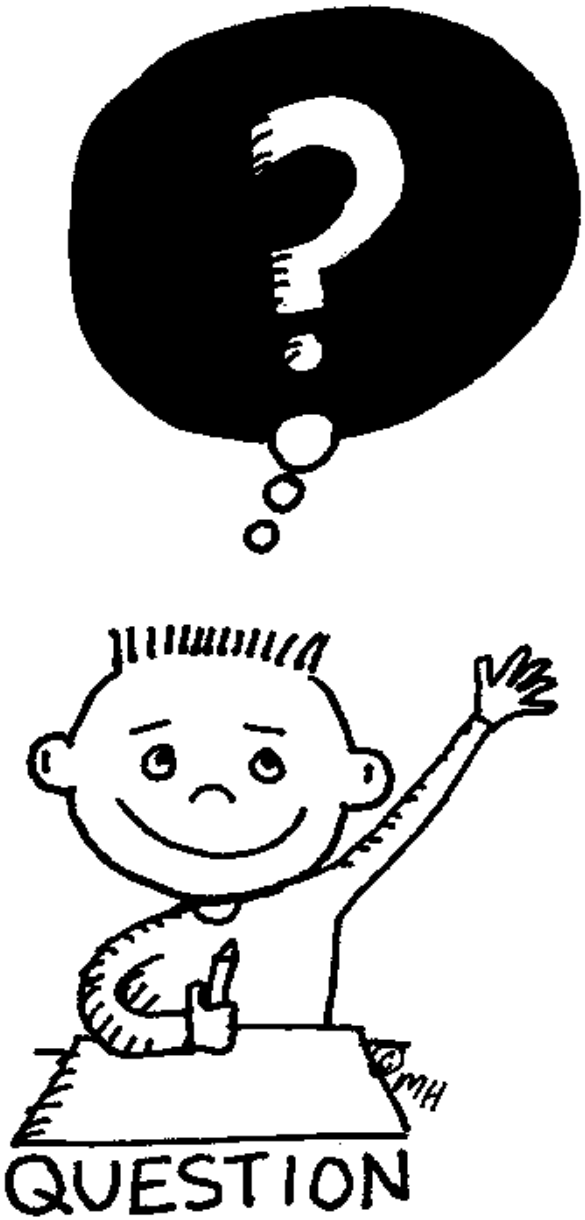
Buffer Overflow	Format String	Integer Overflow
Off-by-One	Race Condition	●●●

- Lỗ hổng web: SQL Injection, XSS, CSRF...
- Sử dụng các thành tố mật mã không tốt
- Giải phóng bộ nhớ 2 lần
- Sử dụng bộ nhớ sau khi đã giải phóng
- ...

Một số công cụ cần thiết

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

GDB
The GNU Project Debugger


QUESTION

Tự học

- Làm quen với các công cụ
- Ôn lại kiến thức về hợp ngữ (có thể sử dụng tài liệu [2])