NTUST 資安研究社

Reverse Engineering

Inndy / inndy.tw@gmail.com

在開始之前...

- 我假設你已經會...
  - 二進位、十進位、十六進位之間互相轉換
  - 寫程式（知道 if, loop, function, recursion, variable, array）
  - C 語言、基本 Pointer 操作
- 如果有聽不太明白的地方可以舉手發問
- 如果以上有超過一半以上看不懂或不知道，你可能走錯教室了

Outline

- 原始碼、編譯器與執行檔
- 記憶體模型
- 程式區段
- 記憶體區段與佈局
- 初探組合語言
  - CPU 暫存器
- Stack

逆向工程介紹

- 你沒有程式的原始碼
  - 你想要知道這個程式是怎麼實作的
  - 你想要寫遊戲外掛
  - 你想要破解功能驗證
  - 你想要寫軟體註冊機
  - 你想要找漏洞
- 你需要逆向工程！

不同的編譯/執行方式

- 編譯成原生機器碼（Native）
  - C, C++, Obj-C, Swift, golang, haskell, Android (ART) ...
- 編譯成中間碼（Intermediate Language (IL) or byte code）
  - .NET (C#, VB, F#, ...), Java, Python (precompiled to pyc), ...
- 直譯（Interpreting）
  - Python, Ruby, JavaScript, VBScript, LISP

原始碼、編譯器與可執行檔

原始碼、編譯器與可執行檔

不會完全一樣，但足以編譯出一樣功能的程式或是瞭解程式行為

編譯器種類

- AOT (Ahead-of-time)
  - 就是你知道的 C, C++ 編譯器，從原始碼編譯成可執行檔案
- JIT (Just-in-time)
  - Chrome 的 V8 JavaScript engine屬於這種，跑起來的時候編譯成機器碼
- Transcompilation
  - 從一種表達方式編譯成另外一種，source code to source code (Cython), byte code to native (Android Run Time, ART)

我們討論的範圍...

- 只有 AOT Compiler 編譯出來的 Native Code
  - x86 and x86_64 兩種架構
- 其他的呢？
  - 工具都給你了，自己學吧 XD
  - Java：JD-GUI, jadx, cfr
  - Android (Java)：jadx, dex2jar + JD-GUI
  - .NET：ILSpy, .NET Reflector, dotPeek, JustDecompile
  - Python (pyc)：uncompyle2

記憶體模型

- 大家對以下名詞有多瞭解？
  - bit / byte
  - 變數
  - 記憶體指標（Pointer）
  - 陣列
  - 結構體（Structure）
  - Union

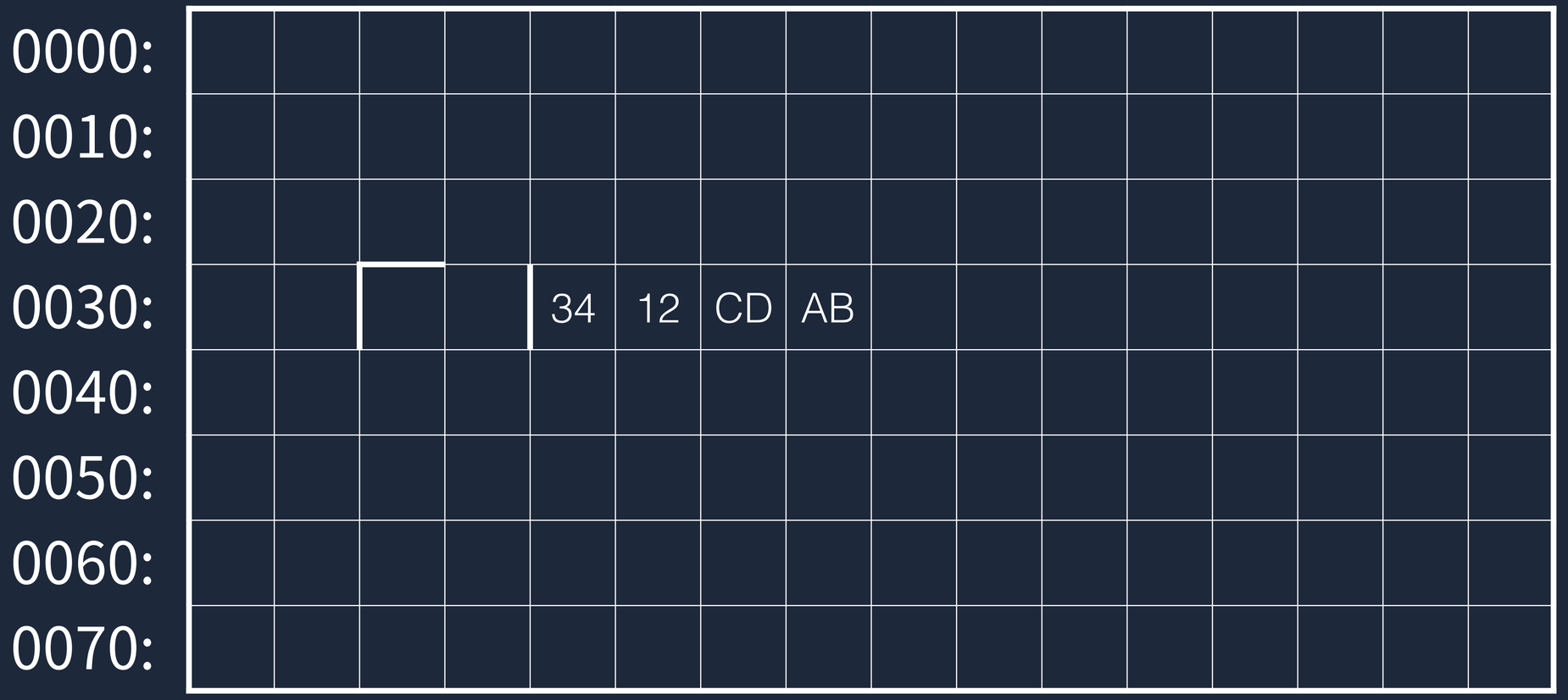
記憶體模型

- 每個格子是 1 byte
- 每個格子有自己的編號（記憶體地址，Memory address），依序遞增
- 一個變數可能會用好幾個 byte
  - 變數的 memory address 是第一個 byte 的 address

記憶體模型

```
int a = 0xabcd1234;
assert(&a == (int*)0x0034);
assert(sizeof(int) == 4);
```

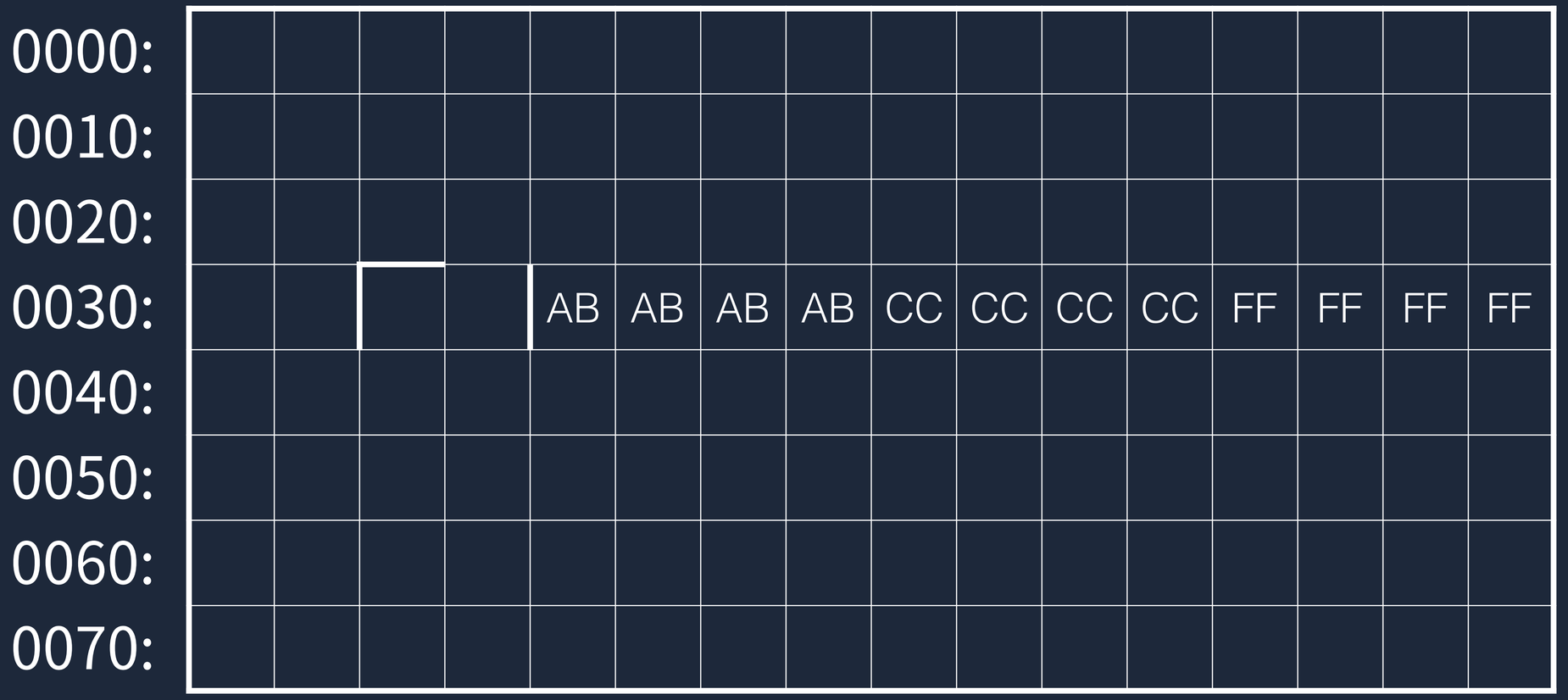
記憶體模型



記憶體模型

```
int arr[3] = {
  0xabababab,
  0xcccccccc,
  0xffffffff
};
assert(a == (int*)0x0034);
assert(sizeof(int) == 4);
```

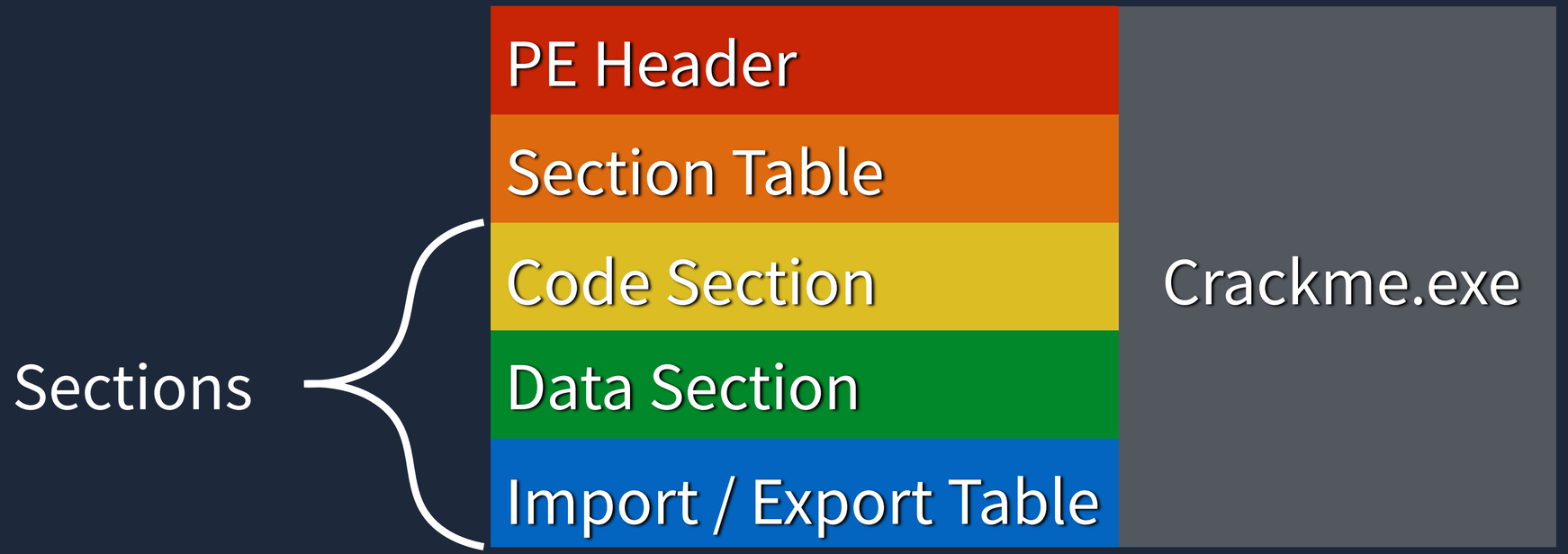
記憶體模型


0000:
0010:
0020:
0030:
0040:
0050:
0060:
0070:
AB AB AB AB CC CC CC CC FF FF FF FF

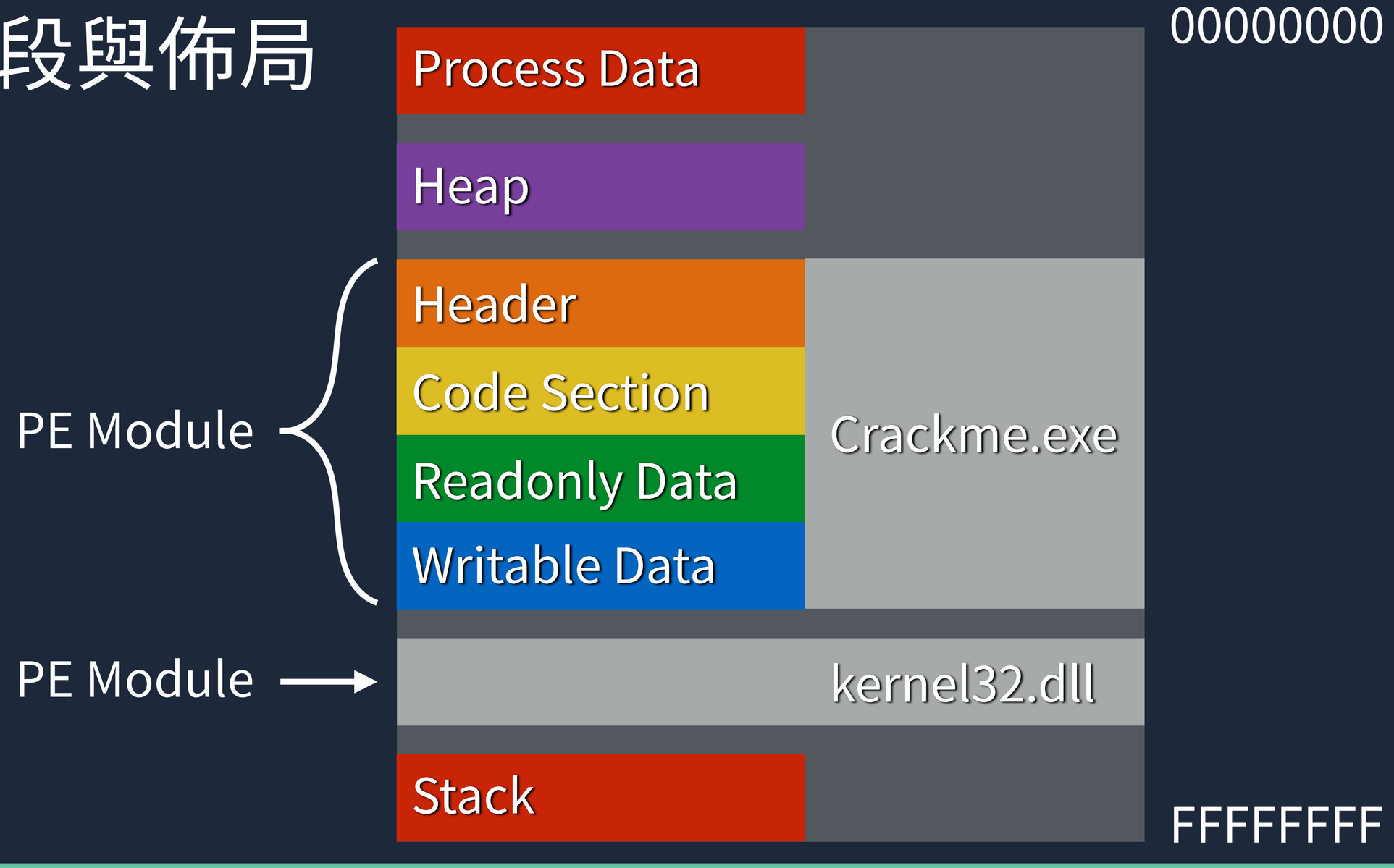
整數儲存方式

- 兩種方式：big endian, little endian
- x86 使用 little endian
  - int a = 0x12345678 → 0x78, 0x56, 0x34, 0x12 (in memory)
  - long long b = 0x1234567890abcdef
    - → 0xef, 0xcd, 0xab, 0x90, 0x78, 0x56, 0x34, 0x12 (in memory)
  - big endian
    - int a = 0x12345678 → 0x12, 0x34, 0x56, 0x78 (in memory)

程式區段


PE Header
Section Table
Code Section
Data Section
Import / Export Table
Crackme.exe
Sections

記憶體區段與佈局


00000000
Process Data
Heap
PE Module
Header
Code Section
Readonly Data
Writable Data
Crackme.exe
PE Module
kernel32.dll
Stack
FFFFFFFF

初探組合語言

這些資料是什麼？

```
00000000: 56 53 83 ec 04 8b 5c 24 10 83 fb 02 7e 24 31 f6  VS....\$....~$1.
00000010: 8d 43 ff 83 ec 0c 83 eb 02 50 e8 e1 ff ff ff 83  .C.......P......
00000020: c4 10 01 c6 83 fb 02 7f e7 8d 46 01 83 c4 04 5b  ..........F....[
00000030: 5e c3
```

初探組合語言

反組譯之後...

```
address   opcode      assembly code
00000000  56          push esi
00000001  53          push ebx
00000002  83EC04      sub esp,byte +0x4
00000005  8B5C2410    mov ebx,[esp+0x10]
00000009  83FB02      cmp ebx,byte +0x2
0000000C  7E24        jng 0x32
0000000E  31F6        xor esi,esi
00000010  8D43FF      lea eax,[ebx-0x1]
00000013  83EC0C      sub esp,byte +0xc
00000016  83EB02      sub ebx,byte +0x2
00000019  50          push eax
0000001A  E8E1FFFFFF  call dword 0x0
0000001F  83C410      add esp,byte +0x10
00000022  01C6        add esi,eax
00000024  83FB02      cmp ebx,byte +0x2
00000027  7FE7        jg 0x10
00000029  8D4601      lea eax,[esi+0x1]
0000002C  83C404      add esp,byte +0x4
0000002F  5B          pop ebx
00000030  5E          pop esi
00000031  C3          ret
```

初探組合語言

原始碼長這樣！

```
int fib(int n)
{
    if(n <= 2) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

初探組合語言 - 構成

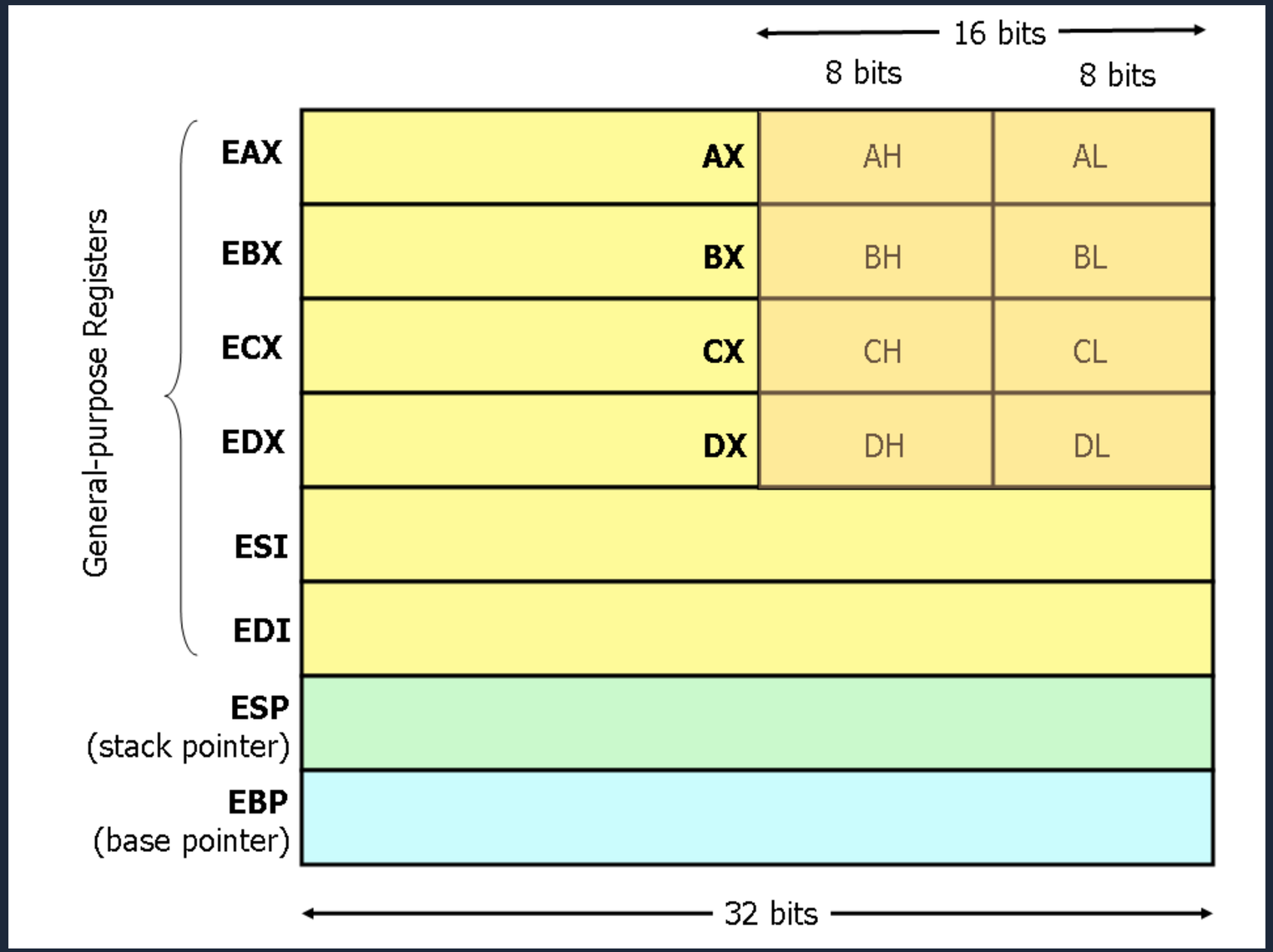
- 由很多指令組成程式
  - 加法、乘法、Xor、Not、比較、跳轉...
  - Operation [[[Arg1], Arg2], Arg3]
    - add eax, ebx // eax += ebx
- 暫存器
  - CPU 內固定的幾個變數

初探組合語言

```
EAX = 0x12345678
 AX = 0x5678
 AL = 0x78
 AH = 0x56
```

補充：SI, SH, SL,
DI, DH, DL

EIP：指向程式現在執行的指令



初探組合語言 - 64bits 差異

- x86_64 多了 64bits register 和額外的八個暫存器
  - EAX (32bits) → RAX (64bits)
  - R8, R9, R10, R11, R12, R13, R14, R15
  - R8 (64bits), R8D (32bits), R8W (16bits), R8B (8bits)
- 運算效率更好
  - 以前 32bits 暫存器要做兩次加法才能完成 long long (64bits integer) 相加，現在只要一次
- Calling convention 不同，參數會先放暫存器

初探組合語言

- 資料種類
  - 暫存器 - reg
    - EAX, EBX, ECX, EDX, EDI, ESI, ESP...
  - 數值 - imm
    - 0, 1, 9487, 0xc8763, 0xffffffff
  - 記憶體參考 - mem
    - byte ptr [0x12345678]
    - dword ptr [ebx + eax*8 + 9]

初探組合語言

- add dst, src
  - dst += src
  - dst：mem, reg
  - src：mem, reg, imm
  - dst, src 不可同時為 mem
- add eax, 7 // eax += 7
- add dword ptr [0x123456+eax*4], ebx // arr[eax] += ebx
- add eax, ebx // eax += ebx

初探組合語言

- `mov dst, src - dst  = src`
- `add dst, src - dst += src`
- `sub dst, src - dst -= src`
- `and dst, src - dst &= src`
- `or  dst, src - dst |= src`
- `xor dst, src - dst ^= src`
- `not dst      - dst  = ~dst`
- `inc dst      - dst++`

初探組合語言

```
// 算數練習
mov eax, 8
mov ebx, 7
xor ecx, ecx
inc ecx
add eax, ebx
add ebx, ebx
sub eax, ebx
sub ecx, eax
```

初探組合語言 - Stack

- `push val`
  - `val: imm, reg, mem`
  - 把數值 `push` 進堆疊
- `pop target`
  - `target: reg, mem`
  - 把數值從堆疊 `pop` 到指定的地方

初探組合語言

- jmp target
  - target: reg, imm, mem
  - 無條件跳轉到 target 的指令繼續執行
- call target
  - 函式呼叫，跟 jmp 有點像，但是把返回位址 push 進堆疊
- ret
  - 從 function 跳回返回位址，等於 pop eip

初探組合語言

- `cmp a, b`
  - `a: reg, mem`
  - `b: imm, reg`
- 比較完之後 CPU 會把比較結果記錄在 EFLAGS 裡面
- `je addr, jz addr` - 如果比較相等就跳轉
- `jg addr` - 如果 `a > b` 就跳轉
- `jl addr` - 如果 `a < b` 就跳轉
- `jge addr` - 如果 `a >= b` 就跳轉
- `jle addr` - 如果 `a <= b` 就跳轉

初探組合語言 - EFLAGS

	OF		SF	ZF		CF
...	11	...	7	6	...	0

- OF — Overflow
- SF — Signed
- ZF — Zero Flag
- CF — Carry Flag
- Reference: https://courses.engr.illinois.edu/ece390/books/labmanual/assembly.html