

Basic reverse engineering on x86

This is for those who want to learn
about basic reverse engineering on x86
(Feel free to use this, email me if you need a keynote version.)
v0.1

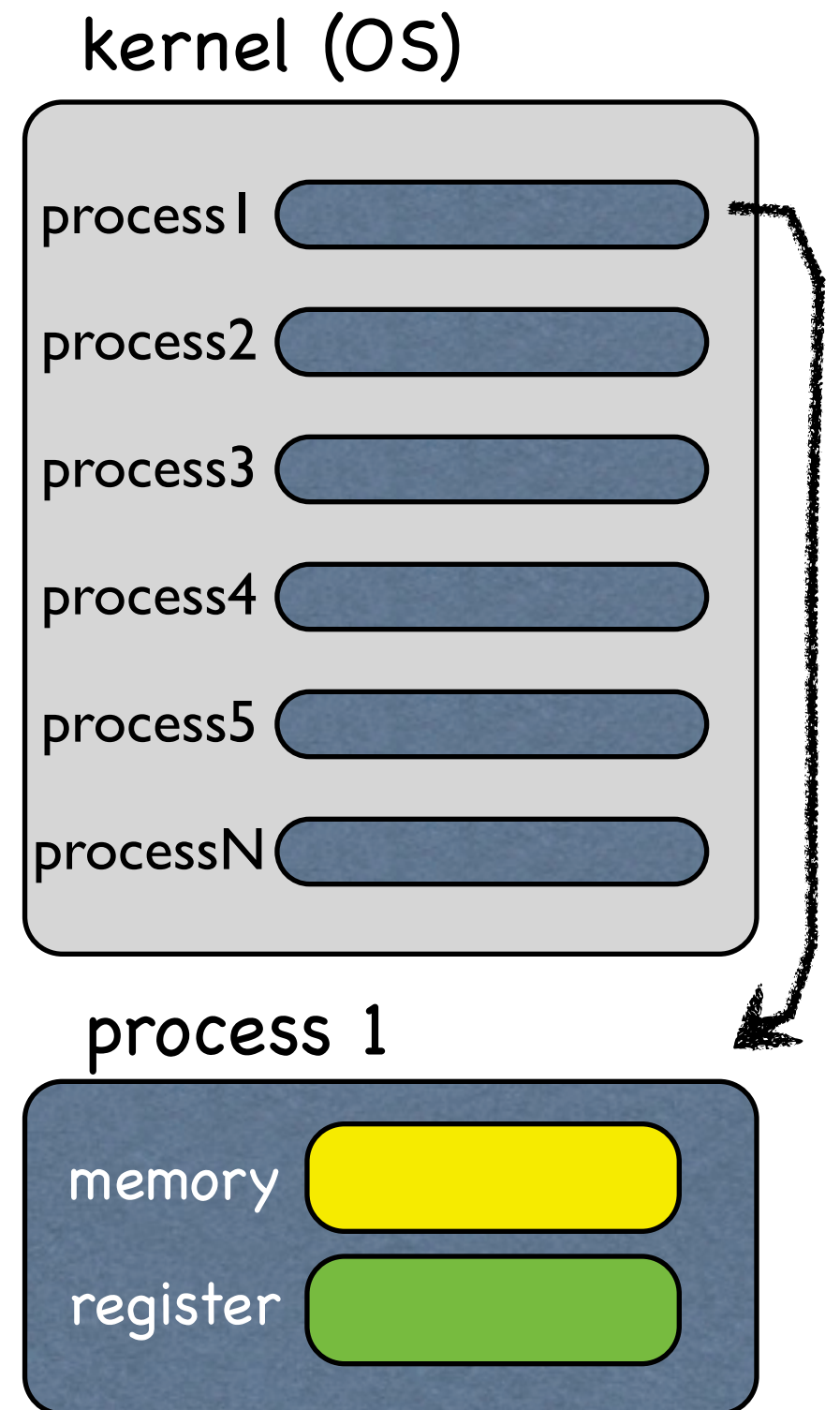
Seungjin Beist Lee
beist@grayhash.com
<http://grayhash.com>

INTRO

- Basic architecture of modern computers
- Basic and most used assembly instructions on x86
- Installing an assembly compiler and RE tools
- Practice code in assembly

Remind

- CPU, registers and memory



For beginners

- You need to think that only CPU, registers, memory and external drives like HDD or SSD are used in your computer
- Ignore software/hardware interrupts at the moment
- The 3 items are enough to get the concept in this lecture
 - CPU, registers, memory

Assembly instructions

- CPU vendors make new assembly instructions for every brand new CPU
- But you don't have to learn about all the instructions
- At the first, around 20~30 instructions are enough

Popular instructions

- Most of instructions are arithmetic operations, branches, data move and so on in most programs
- And system calls
- They usually cover over 80% in many programs

About the grammar

- Assembly grammar itself is easy (both x86 and arm)
- But the side effect is complicated in x86
- And x86 is CISC (Complex Instruction Set Computing)

About the grammar

- Instruction can be
 - Opcode
 - Opcode + operand
 - Opcode + operands
- Opcode
 - Operation code
- Operand
 - Argument for opcode

Size

- Instruction size
 - The x86 architecture is a variable instruction length
 - From 1 byte to 17 bytes for 80386 (including operands)
- The default operand size
 - 8, 16 and 32 bits

Opcode

- Opcode is like when you want to say
 - “I want to add a value to a value.” (*ADD*)
 - “I want to subtract a value from a value.” (*SUB*)

Operand

- Operands can be
 - Memory
 - Registers
 - Immediate values (Only for source operands)
- In a way that
 - “I want to add *a value* to *a value*.” (*add register, 2*)
 - “I want to subtract *a value* from *a value*.” (*sub register, 2*)

Instruction samples

- `add eax, 2`
- `add ebx, 4`
- `add eax, ebx`
- `sub eax, 2`
- `sub ebx, 4`
- `sub eax, ebx`
- `Easy!`

Registers

- There are 4 types
 - General registers - EAX, EBX, ECX, EDX
 - Segment registers - CS, DS, ES, FS, GS, SS
 - Index and pointers - ESI, EDI, EBP, EIP, ESP
 - Indicator - EFLAGS

Registers

- But, when you do reversing on most of user level programs in x86, you could ignore Segment registers since most of times you don't have to deal with them
- EFLAGS is important to understand the side effect
- You can't control EIP directly
 - EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP are ok

Registers

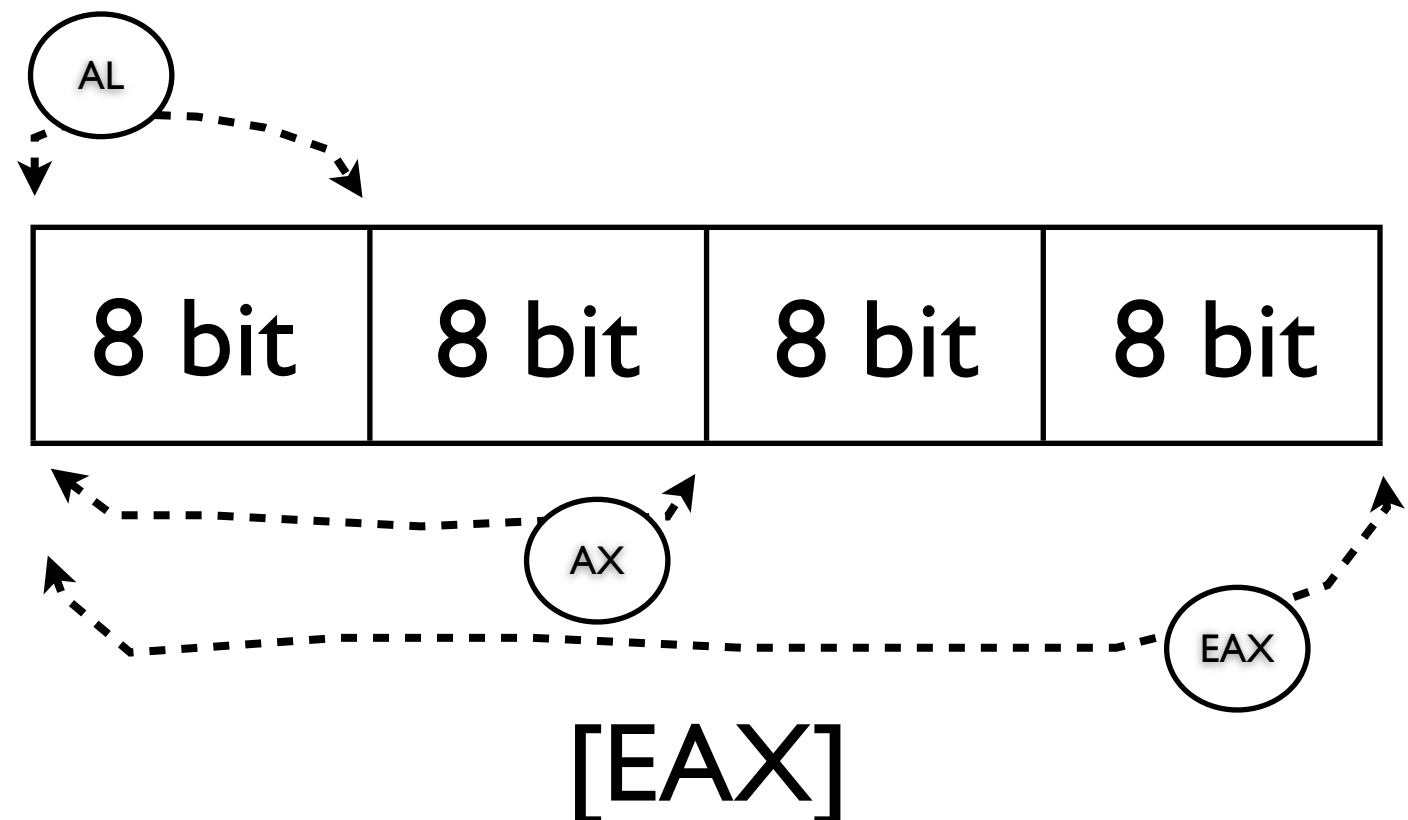
- For examples
 - (O) - MOV EAX, 0x2
 - (O) - MOV ESP, 0x2
 - (X) - MOV EIP, 0x2

Registers

- Even though you can control the all registers directly except EIP, there are something
 - ESP - pointing to current address of stack
 - EBP - frame pointer of function
 - ESI - source when you use copy opcode
 - EDI - destination when you use copy opcode
 - EAX - a value for return or multiply opcode or something
 - ECX - a number how many times when you use copy op
- Not that complicated, you will see

Split off registers

- A register can be broken into
- And each has a different size
 - AL - 8 bit (or AH)
 - AX - 16 bit
 - EAX - 32 bit



Operands

- Remember that operands can be 8, 16 and 32 bits
- Memory and immediate value are as well
 - Example
 - `mov ax, word ptr[0x401000]`
 - `mov ax, 0x4141`
- Memory
 - BYTE (8bit), WORD (16bit), DWORD (32bit)

Opcode with any operand

- There are some opcode that don't need any operand
- Example:
 - `nop` (no operation)

2 ways to write in ASM

- There is a bit different between INTEL and AT&T
- Example:
 - INTEL: `mov eax, 0x4`
 - AT&T: `mov $0x4, eax`
- There are more differences but very slight
- It's mostly about opposite of direction
 - source, destination or destination, source
- We'll take INTEL style

mov instruction

- mov instruction is for assigning
- Example:
 - `mov eax, 0x4`
 - `mov dword ptr[0x401000], eax`
 - `mov dword ptr[0x401000], 0x4141`
 - `mov eax, ebx`
 - `mov eax, dword ptr[0x401000]`

sub instruction

- sub instruction is to subtract a value from a value
- Example:
 - `sub eax, 0x4`
 - `sub dword ptr[0x401000], eax`
 - `sub dword ptr[0x401000], 0x4141`
 - `sub eax, ebx`
 - `sub eax, dword ptr[0x401000]`

add instruction

- add instruction is to add a value to a value
- Example:
 - `add eax, 0x4`
 - `add dword ptr[0x401000], eax`
 - `add dword ptr[0x401000], 0x4141`
 - `add eax, ebx`
 - `add eax, dword ptr[0x401000]`

cmp instruction

- cmp instruction is to compare a value to a value
- Example:
 - `cmp eax, 0x4`
 - `cmp dword ptr[0x401000], eax`
 - `cmp dword ptr[0x401000], 0x4141`
 - `cmp eax, ebx`
 - `cmp eax, dword ptr[0x401000]`

Destination must be writable

- It is very obvious that destinations must be writable
 - Memory and registers
- Immediates are just immediates, they can't be writable
 - So, immediates are never for destination operands

test instruction

- test instruction is usually to know if a value is 0
- Example:
 - `test eax, eax`
- It does actually *and* operation for eax and itself
- So, if eax is not 0, it'll be always not 0
- If it's 0, it's always 0
- You see this case many times - “if (a == 0) { }” in C code

EFLAGS time


- EFLAGS is updated after instructions got executed
- So that you know the result of these instructions
 - `cmp, test`
 - And others make EFLAGS updated
 - almost all instruction, even `add opcode`
- But, again, for beginners, you don't worry about EFLAGS now

je instruction

- je instruction is to jump to at an address if the result is equal
- Example:
 - 0x401096: MOV EAX,1
 - 0x40109B: CMP EAX,1
 - 0x40109E: JE SHORT 004010A2
 - 0x4010A0: MOV ECX,EAX
 - 0x4010A2: MOV EAX,EBX
- As EAX is 1, the instruction at 0x4010A0 will be not executed



jne instruction

- jne instruction is to jump to at an address if the result is not equal
- Example:
 - 0x401096: MOV EAX,1
 - 0x40109B: CMP EAX,2
 - 0x40109E: JNE SHORT 004010A2
 - 0x4010A0: MOV ECX,EAX
 - 0x4010A2: MOV EAX,EBX
- As it's not equal, the instruction of 0x4010A0 will be not executed

jmp instruction

- jmp instruction is to jump to at an address
- Example:
 - 0x40108A: MOV EAX,4
 - 0x40108F: JMP SHORT 00401093
 - 0x401091: MOV EAX,EBX
 - 0x401093: MOV ECX,EBX
- The instruction at 0x401091 will be not executed



Branches are important

- Catching up branches is one of most important things when you do reverse engineering
- “if, jump, else” is everywhere in modern programs
- There are many more than “jmp/je/jne”
 - js/jns/jo/jno/jc/jnc/jb/jbe/jae/ja/jl/jle/jge/jg
- But it sounds very logic, for examples
 - je - jump equal
 - jne - jump not equal
- [http://en.wikipedia.org/wiki/Branch_\(computer_science\)](http://en.wikipedia.org/wiki/Branch_(computer_science))

xor instruction

- xor instruction is very simple, it's to xor a value with a value
- Example:
 - `xor eax, eax`
- The result will be 0

push instruction

- push instruction is to push a value onto stack memory
- Example:
 - `push 0x4`
 - `push eax`
 - `push dword ptr[0x401000]`
- After a push operation, ESP value is decreased
 - Remember, ESP points to a current address of stack

pop instruction

- pop instruction is to pop a value from stack memory
- Example:
 - `pop eax`
 - `pop dword ptr[0x401000]`
- After a pop operation, ESP value is increased

call

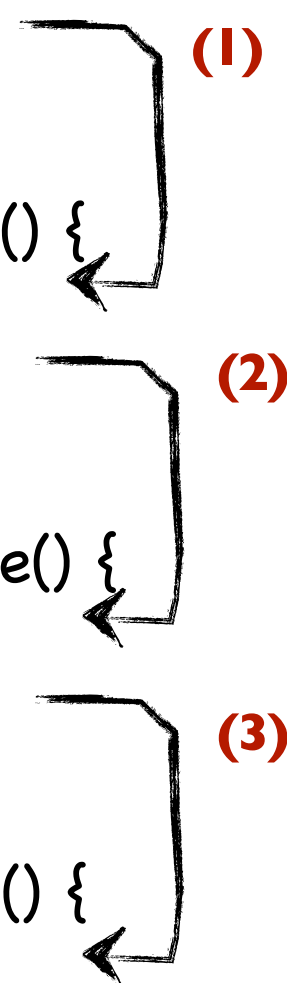
- call instruction is to call a function
- jmp instruction is to just jump to an address
- But, call instruction pushes the next instruction address onto stack memory
 - So that the callee can know where to go back
- Example:
 - call eax
 - call dword ptr[0x401000]
 - call 0x401000

ret

- ret instruction to return to a caller
- It pops a return address from stack
 - This is how a callee can go back to a caller
- Example:
 - ret
- ret opcode can have an argument, but we'll ignore it for now

How to go back to callers

```
main() {  
    my_first_code();  
}  
  
void my_first_code() {  
    my_dumb_code();  
}  
  
void my_dumb_code() {  
    my_l33t_code();  
}  
  
void my_l33t_code() {  
    printf("meh");  
}
```

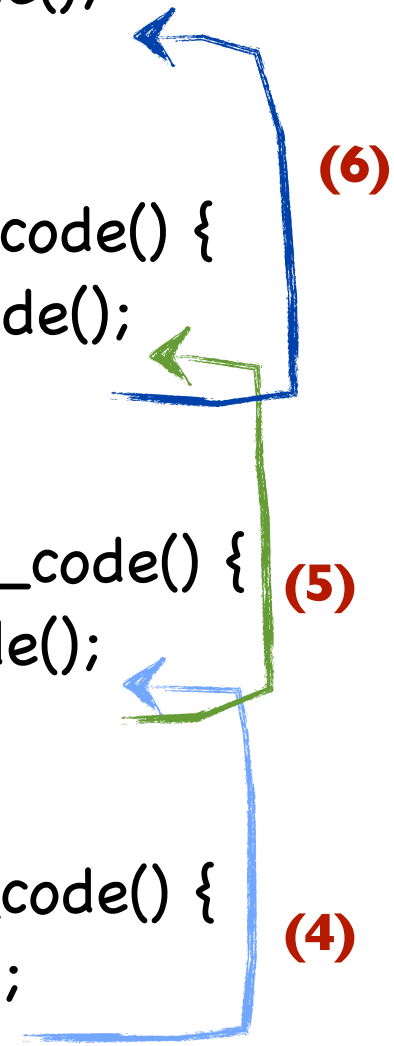


(1)

(2)

(3)

```
main() {  
    my_first_code();  
}  
  
void my_first_code() {  
    my_dumb_code();  
}  
  
void my_dumb_code() {  
    my_l33t_code();  
}  
  
void my_l33t_code() {  
    printf("meh");  
}
```



(6)

(5)

(4)

How to go back to callers

```
0x401015: call 0x401064
0x40101A: mov eax, ebx
...
...
...
...
...
0x401064: nop
0x401065: ret
...
...
```

```
push 0x40101A
jmp  0x401064
```

call instruction pushes the next instruction on stack

ret instruction gets the value from stack and

```
mov eip, [esp]
```

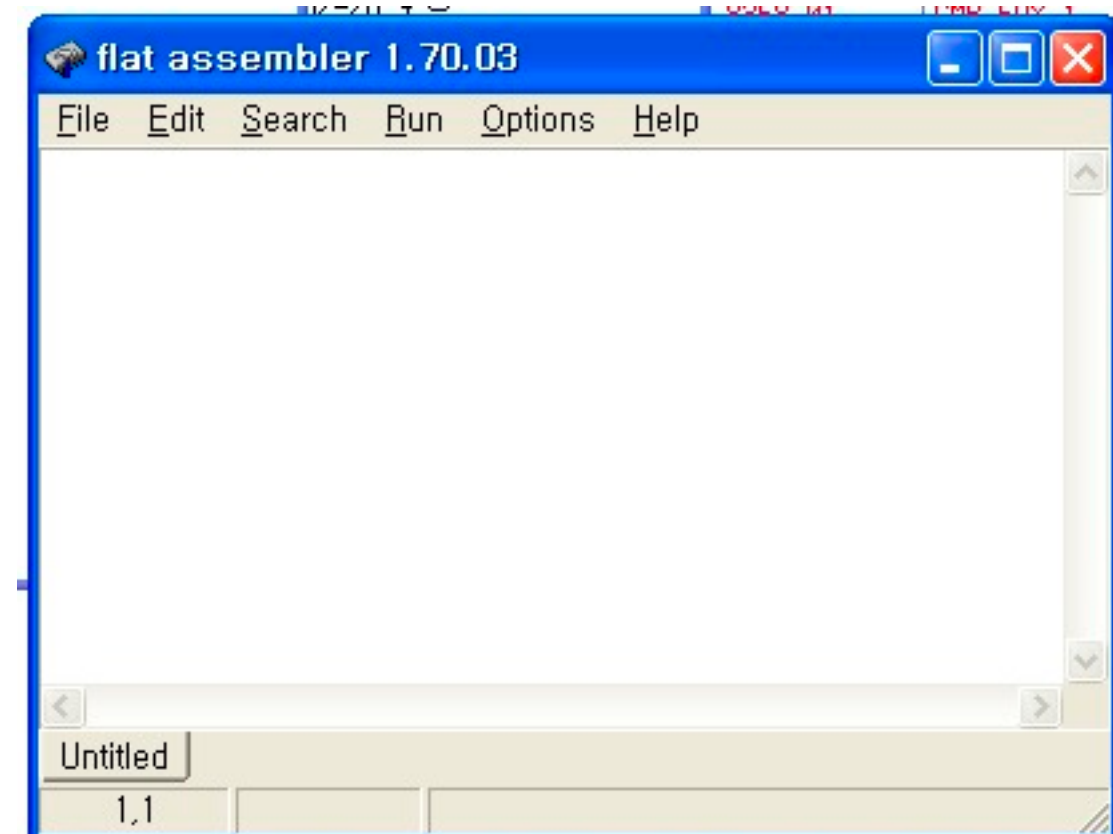
These are pseudo-code, it's different in real world

Addressing modes

- We've mentioned only register, immediate, direct memory, and register indirect addressing modes
- But there are more
 - Base-index
 - Base-index with displacement
 - Direct offset addressing (by the compiler)
- However, we'll not cover those 3 addressing modes

Installing before practice

- Flat assembler
 - A neat assembly compiler (<http://flatassembler.net>)
 - <http://115.68.24.145/fasmw17003.zip>
 - Run “FASMW.EXE”



Your first assembly

- Type this code in Flat Assembler

```
include 'win32ax.inc'

.code
start:
    mov eax, 2
    mov ecx, 3
    nop
    mov eax, 4
    mov ebx, dword [0x401000] ; without 'ptr'
.end start
```

- 1. [File] - [Save as] - [test.asm]
- 2. [Run] - [Compile]
- Then, check out if test.exe is generated

To use label in flat assembler

- To jump, you can specify a label

```
include 'win32ax.inc'

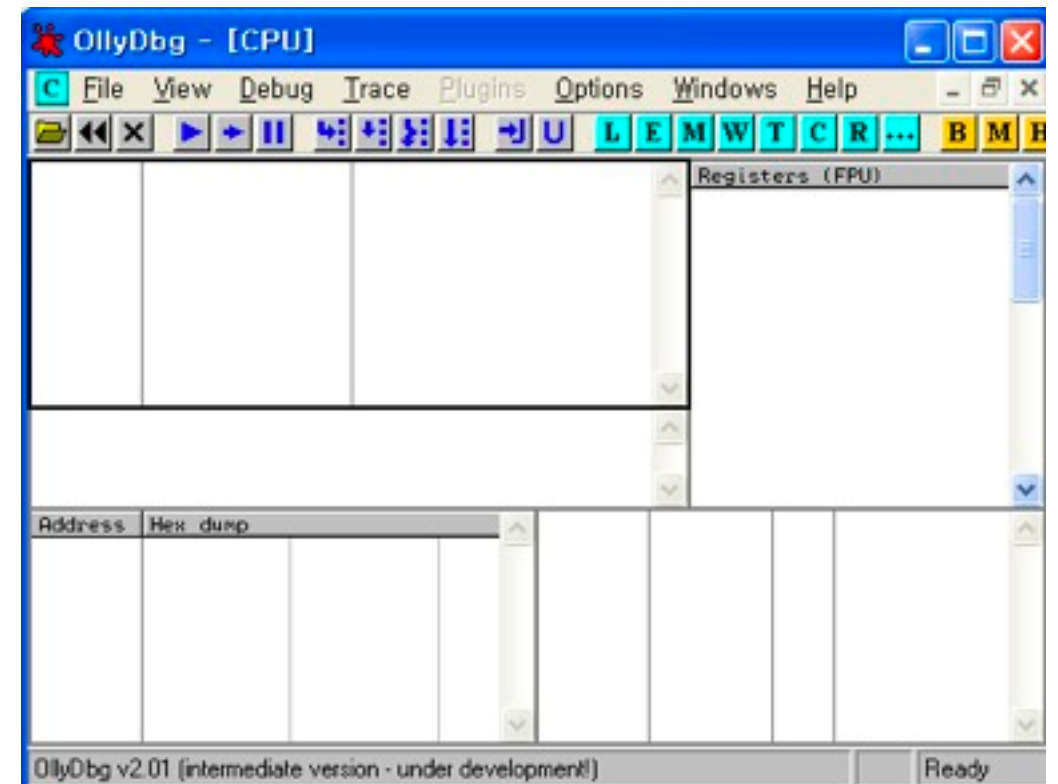
.code
start:
    mov eax, 2
    mov ecx, 3
    jmp test_label

test_label:
    nop
    xor ebx, ebx
.end start
```

- You use labels for implementing branches
 - if - else, for, while, etc

Installing before practice

- Olly Debugger
 - A popular debugger for Windows (<http://www.ollydbg.de>)
 - <http://115.68.24.145/odbg110.zip>
 - Run “ollydbg.exe”



Olly Debugger

- [File] - [Open] - Select the test.exe
- You'll see your program being debugged
- Basic commands
 - F7 - Step into
 - F8 - Step out
 - F9 - Run

Practice time

1. 주어진 설명에 맞게 어셈블리로 코드 구현
2. Flat Assembler를 이용하여 컴파일
3. 컴파일한 바이너리 파일을 OllyDBG에서 Open
4. 구현한 코드를 Step-by-step으로 따라가보며 값 확인
5. 정확하게 구현하였는지 점검

Practice I

1. 레지스터 값 추적하기

- eax 값에 0x100 값을 할당
- eax를 ebx로 이동
- ebx 값에 0x10 값을 빼기
- ebx를 ecx로 이동
- ecx를 edx로 이동
- edx에 ecx 값을 플러스하기
- edx 값 확인하기

Practice 2

2. 스택 변수 값 추적하기

- esp를 0x4 만큼 빼기
- 현재의 esp가 가르키고 있는 메모리에 0x100 값을 할당하기
- esp를 0x4 만큼 빼기
- 현재의 esp가 가르키고 있는 메모리에 0x90 값을 할당하기
- esp를 0x4 만큼 빼기
- 현재의 esp가 가르키고 있는 메모리에 0x80 값을 할당하기
- pop 명령어를 사용하여 eax로 가져오기
- pop 명령어를 사용하여 ebx로 가져오기
- pop 명령어를 사용하여 ecx로 가져오기
- eax, ebx, ecx 값 확인하기
- esp가 가르키고 있는 데이터의 값들을 확인하기 (현재, -4, -8)

Practice 3

3. if 문 구현하기

- esp를 0x100 만큼 빼기
- pop 명령어를 이용하여 eax로 가져오기
- 만약 eax가 0xffff 값보다 크다면 ebx에 1를 할당하기
- 만약 eax가 0xffff 값보다 작다면 ebx에 0를 할당하기
- 구현 완료 후 eax 값 및 ebx 값이 제대로 됐는지 확인하기

Practice 4

4. for 문 구현하기

- 다음 C 코드의 내용을 어셈블리로 구현

```
ebx = 0;
for(ecx=0; ecx<8; ecx++) {
    ebx = ebx + ecx + 1;
}
edx = ebx;
```

- edx의 값 확인

Practice 5

5. 함수 호출하고 리턴하기

- func_1, func_2, func_3 label 선언
- func_1: eax에 0x10 값을 할당
- func_2: ebx에 0x30 값을 할당
- func_3: ecx에 eax와 ebx를 더한 값을 할당
- 메인 함수에서 func_1, func_2, func_3 함수를 차례로 호출
- 위 3개 함수의 실행이 끝난 후 본체 함수에서 ecx 값 확인하기

Practice 6

6. 문자열 가져오기 trick

- start 함수 외에 get_string 함수를 추가 선언
- start 함수에서 get_string 함수를 호출
- get_string 함수에서의 return address 가져와서 eax에 저장
- call eax를 하여 start 함수로 복귀
- 이때, call eax 함수 뒤에 db "test_go" 선언
- start 함수로 복귀한 후, ebx 레지스터가 "test_go" 메모리의 시작 부분을 가르키도록 설정
- ebx 값 확인

[TIP]

```
call func_address  
db "this_is_test"  
db 0x0
```

Practice 7

7. 암호화 연산 구현하기 (simple xor)

[암호화 연산]

- eax가 "reversing" 문자열을 가르키게 하기
- for 문 구현
- for 문 안에서 key 값과 "reversing" 값을 xor 연산하기
- 연산 후의 값 확인

[복호화 연산]

- 암호화 연산에서 생성된 암호문을 반대로 다시 xor하기
 - for 문으로 구현
 - xor할 key는 동일함
 - 연산 후의 값 확인
-
- key: nothing

REFERENCES

- To be added later