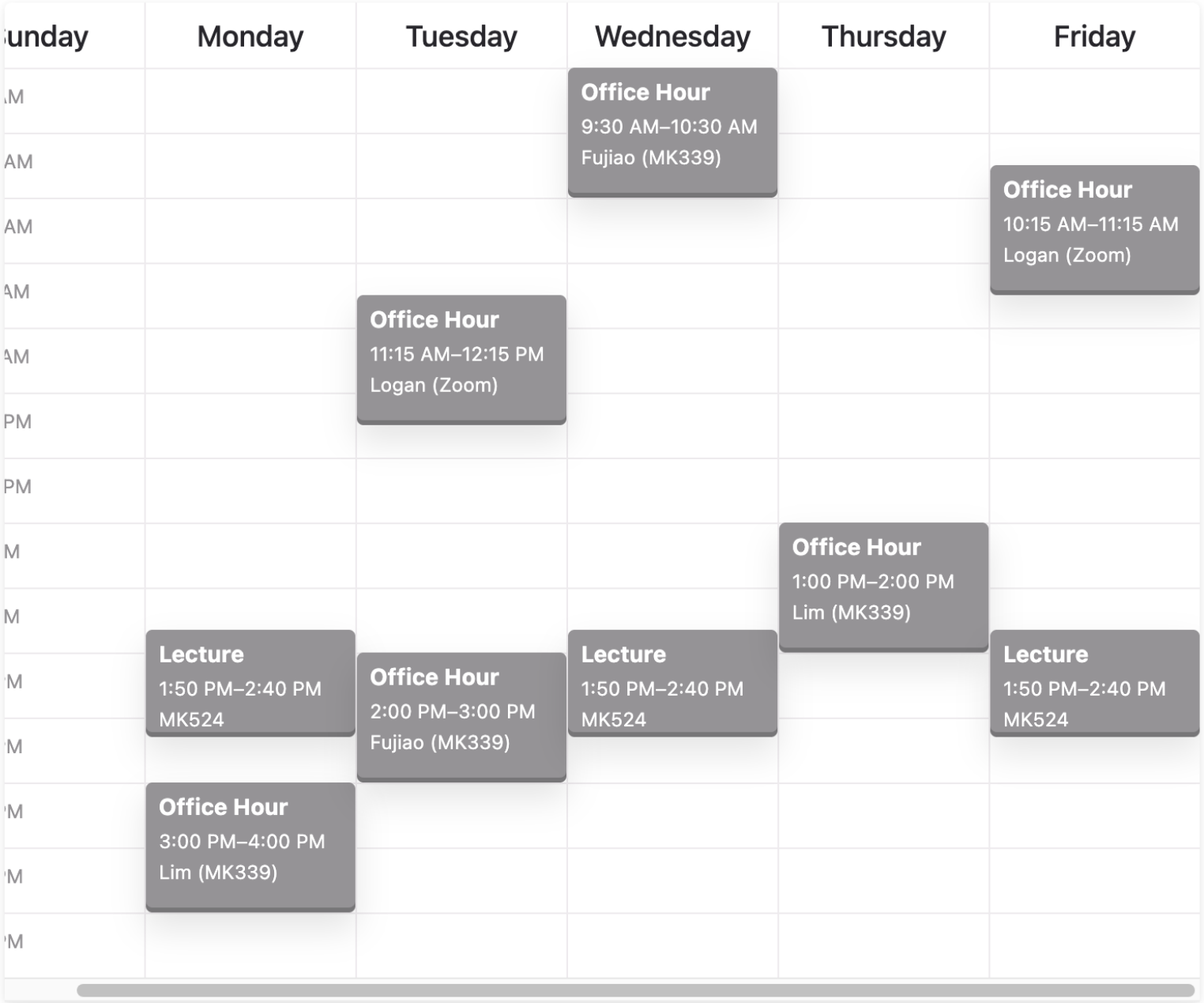
Software Security
COSC 466/566
Spring 2023

Dr. Doowon Kim

THE UNIVERSITY OF
TENNESSEE

Weekly Schedule



Today’s class

- Compilation
- x86-64 Assembly

Compilation

Overview

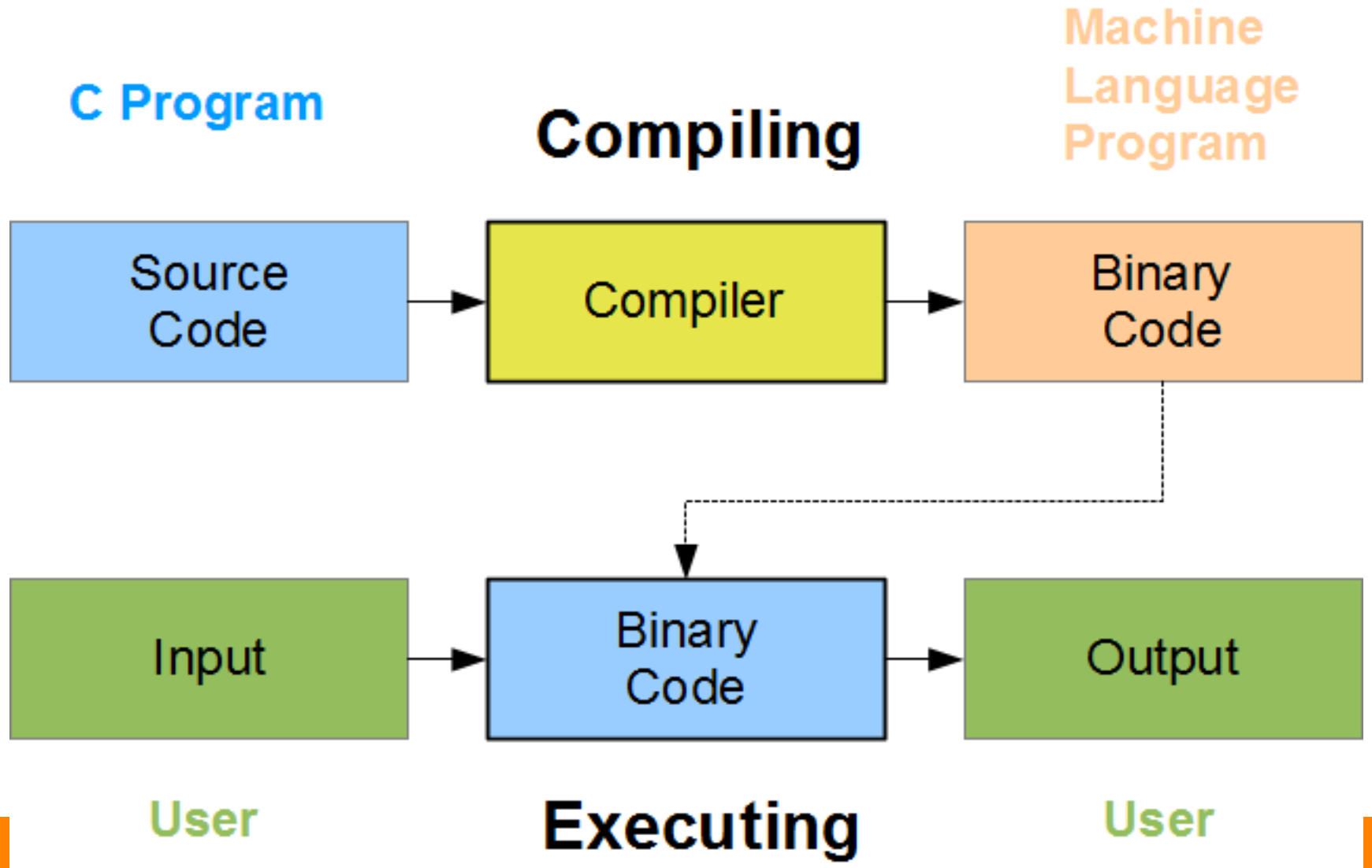
Code, Machine language, and Assembly language

- You write a code in a programming language (say in C)

- Compiler (gcc or clang) translate program to the *object files*
  - *Object* file is a collection of *machine codes translation + header information*
  - Basically, 0s and 1s

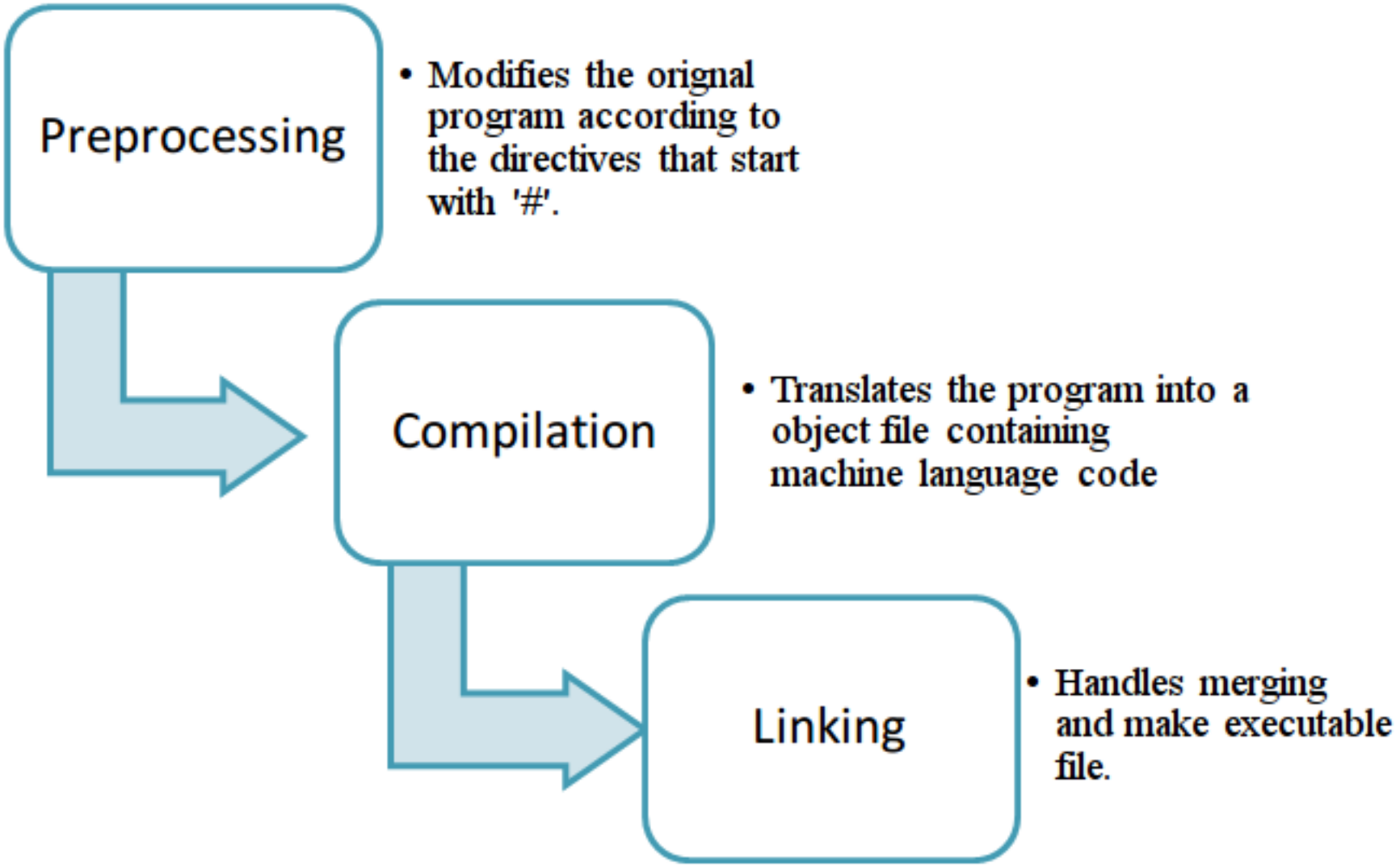
- Assembly (language): decorated version of machine language
  - With instruction name, variable names
  - Human readable translation of machine language
  - *Assembler* translates Assembly into object files (or machine codes)

Code, Machine language, and Assembly language



https://medium.com/@wendymayorgasegura/what-happens-when-you-type-gcc-main-c-13db088e94fd

Code, Machine language, and Assembly language

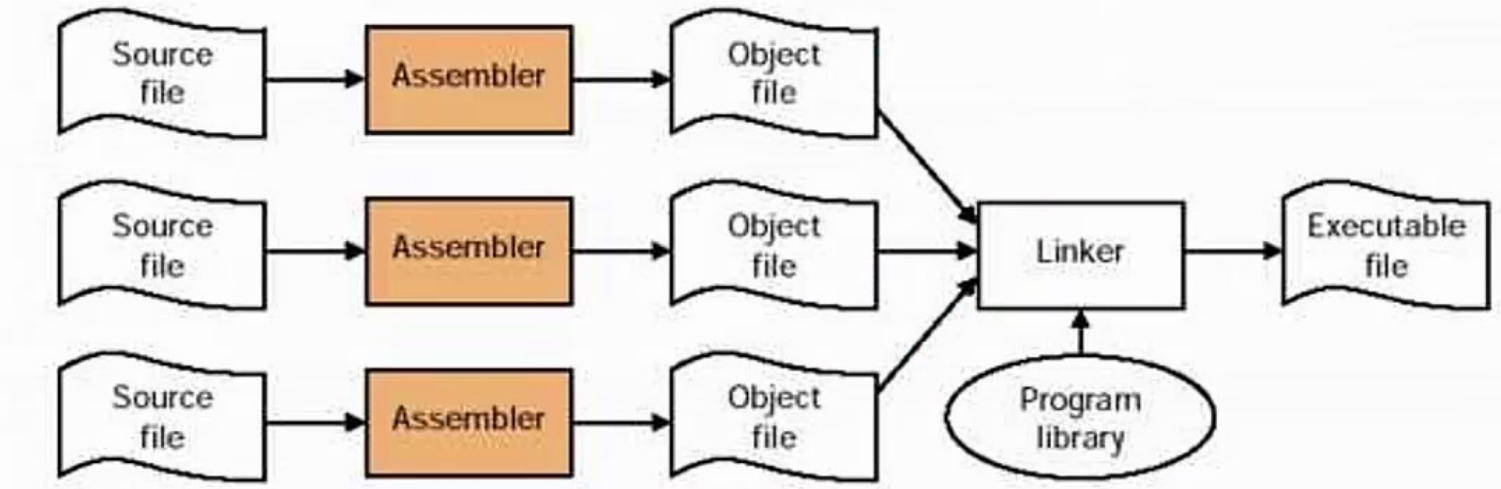


https://isocpp.org/blog/2013/11/compiling-and-linking

Code, Machine language, and Assembly language

Compile, assemble, and link to executable

`gcc test.c` produces `test.exe`



https://www.youtube.com/watch?v=N2y6csonII4

Lab: GCC

- Please access a hydra server (if you want, you can try it in your personal computer)
- Copy **hello.c**

- “gcc –o hello hello.c”
- execute “./hello”

- Let’s take a look at each step in compilation

Lab: Preprocessing

- Lines starting with a # character are interpreted by the preprocessor as preprocessor commands.

- Open hello.c file

- Use –E option: this option causes gcc to run the preprocessor, display the expanded output, and then exit without compiling the resulting source code
  - The value of the macro TEST is substituted directly into the output

Lab: Compilation

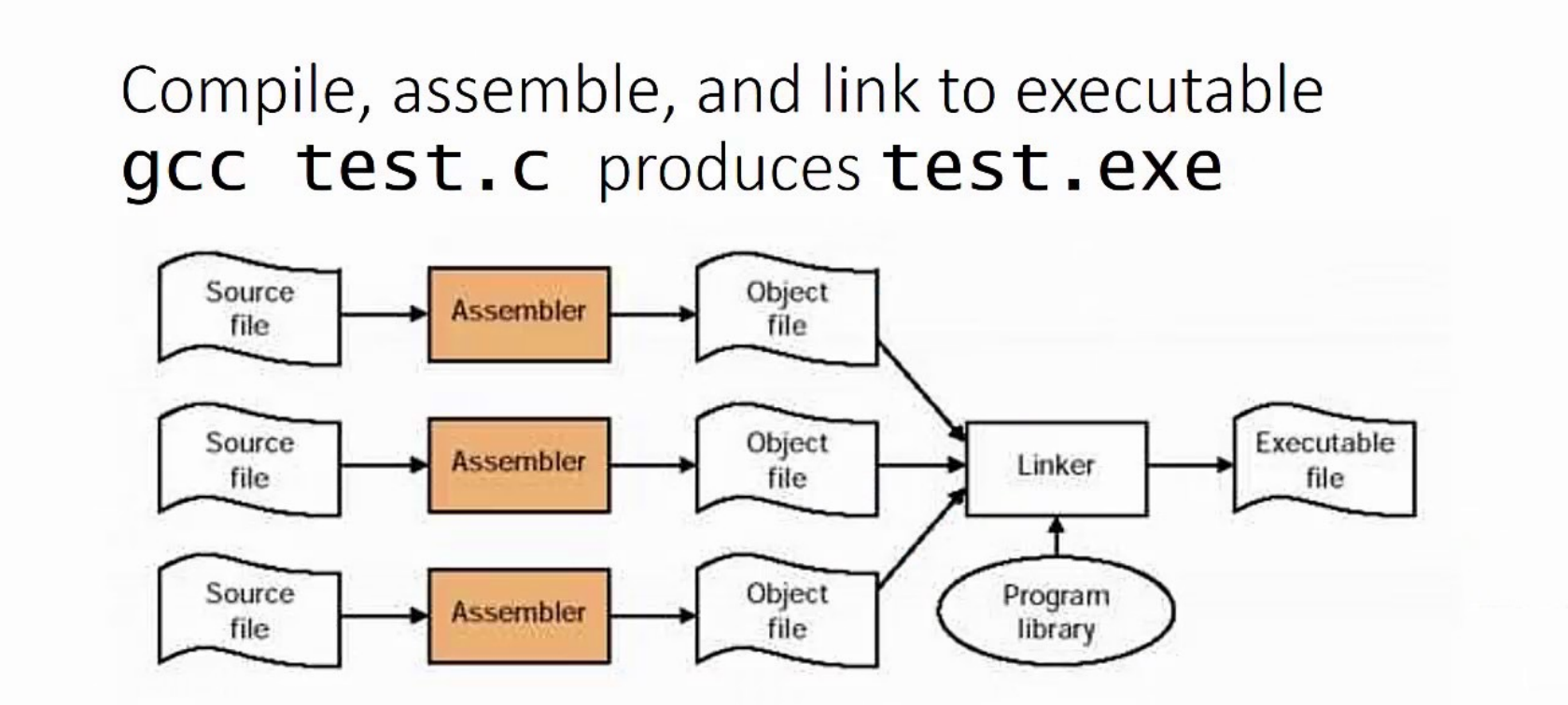
- Use –c option in gcc

```
-c  Compile or assemble the source files, but do not link.  The
    linking stage simply is not done.  The ultimate output is in
    the form of an object file for each source file.

    By default, the object file name for a source file is made by
    replacing the suffix .c, .i, .s, etc., with .o.

    Unrecognized input files, not requiring compilation or
    assembly, are ignored.
```

- Can you execute it?
  - Nope. This is called an object file, and is the machine code translation produced by the compiler

Lab: Assembly

- The preprocessed code is translated to assembly instructions specific to the target processor architecture.



Lab: Assembly

- The preprocessed code is translated to assembly instructions specific to the target processor architecture.

- gcc -S hello.c
  - The S option is …

```
-S  Stop after the stage of compilation proper; do not assemble.
    The output is in the form of an assembler code file for each
    non-assembler input file specified.

    By default, the assembler file name for a source file is made
    by replacing the suffix .c, .i, etc., with .s.

    Input files that don't require compilation are ignored.
```

Lab: Assembly

- The preprocessed code is translated to assembly instructions specific to the target processor architecture.

- gcc -c hello.s
  - What do you have now?

Lab: Linking

- The next step is to link the object file to produce such an executable.

- gcc -o hello hello.o

- “Hello” is the executable.

CTF: Assignment

- Due date: before Monday class (1:50PM, Feb 6th, 2023)

Provide abstraction: a program

- (Computer) Program
  - Definition: a set of instructions for an OS to execute
  - An example program for Linux computer

Provide abstraction: a program

- (Computer) Program
  - **Definition**: a set of instructions for an OS to execute
  - An example program for Linux computer

You

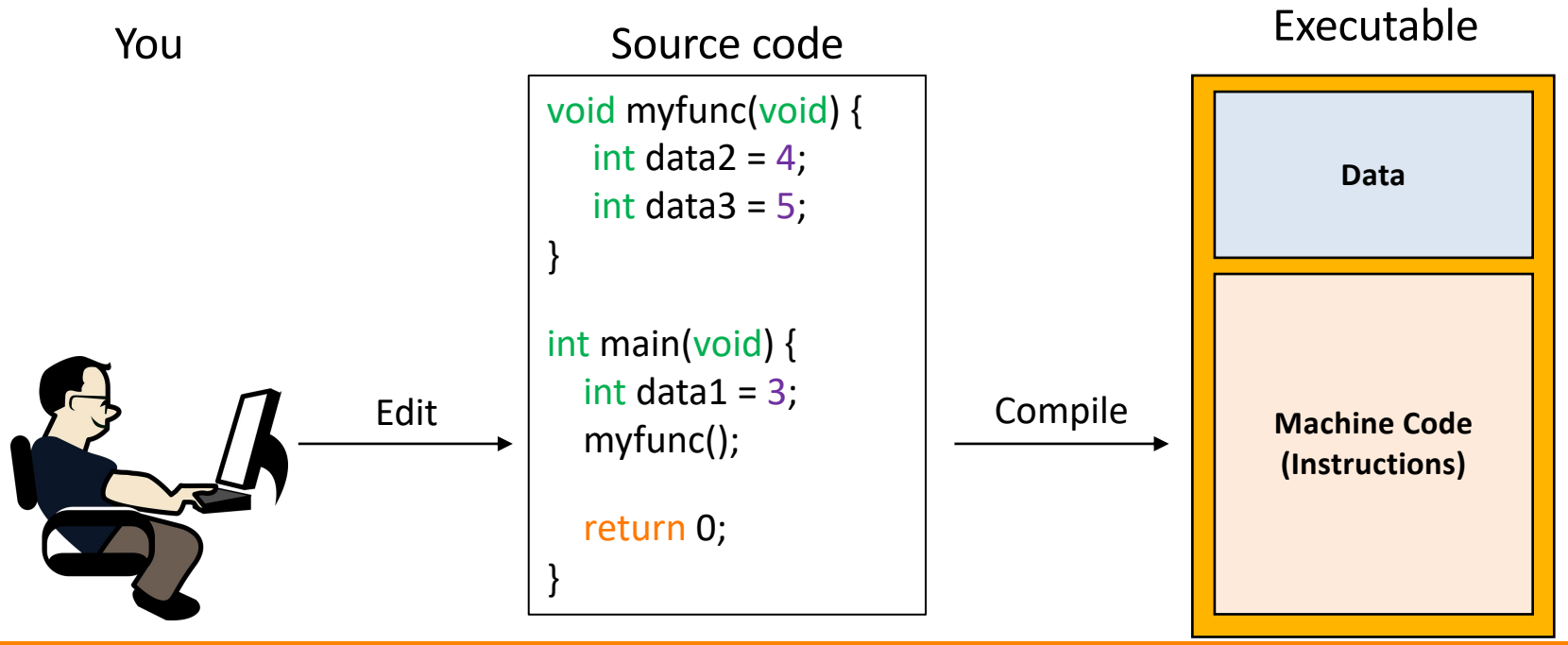
Edit

Source code

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
   int data1 = 3;
   myfunc();

   return 0;
}
```

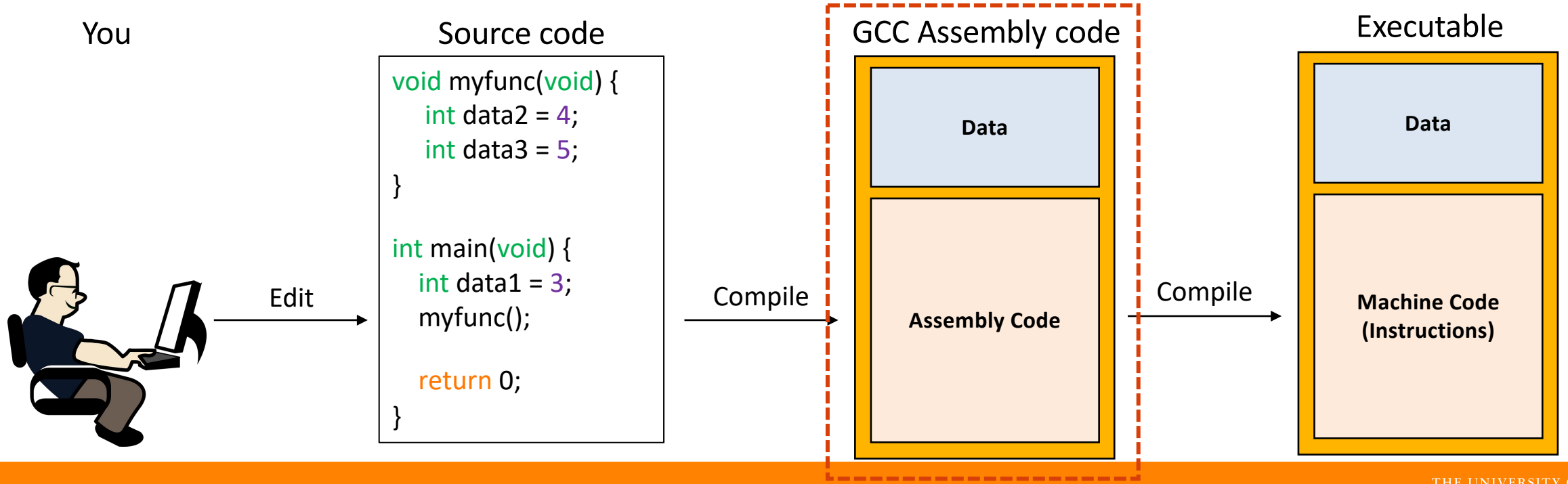
Provide abstraction: a program

- (Computer) Program
  - **Definition:** a set of instructions for an OS to execute
  - An example program for Linux computer



Example: C compilation with GCC

- GCC compilation
  - It converts source code to assembly code ($ gcc -c -S <filename.c>)
  - It then converts the assembly code to instructions
    ($ gcc -c <filename.s> -o <filename.o>; gcc -o <filename.o> -o filename)



Example: C compilation with GCC

- GCC compilation
  - It converts source code to assembly code ($ gcc -c -S <filename.c>)

```
    .file   "example.c"
    .text
    .globl  myfunc
    .type   myfunc, @function
myfunc:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $4, -4(%rbp)
    movl    $5, -8(%rbp)
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   myfunc, .-myfunc
    .globl  main
    .type   main, @function
main:
example.s
```

```
    .size   myfunc, .-myfunc
    .globl  main
    .type   main, @function
main:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $3, -4(%rbp)
    call    myfunc
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   main, .-main
    .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
    .section    .note.GNU-stack,"",@progbits
example.s
```

Provide abstraction: a program

- (Computer) Program
  - **Definition**: a set of instructions for an OS to execute
  - An example program for Linux computer

You

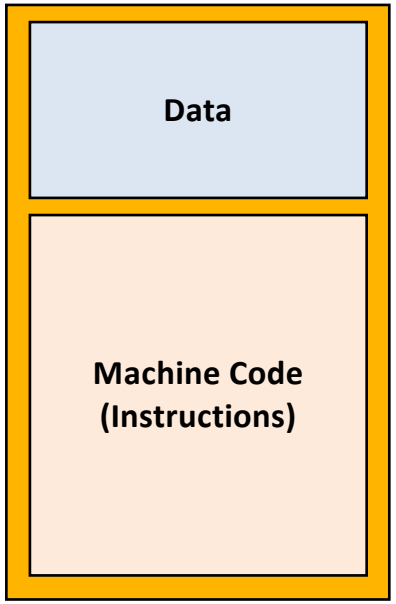
Edit

Source code

```
void myfunc(void) {
    int data2 = 4;
    int data3 = 5;
}

int main(void) {
    int data1 = 3;
    myfunc();

    return 0;
}
```

Compile

Executable



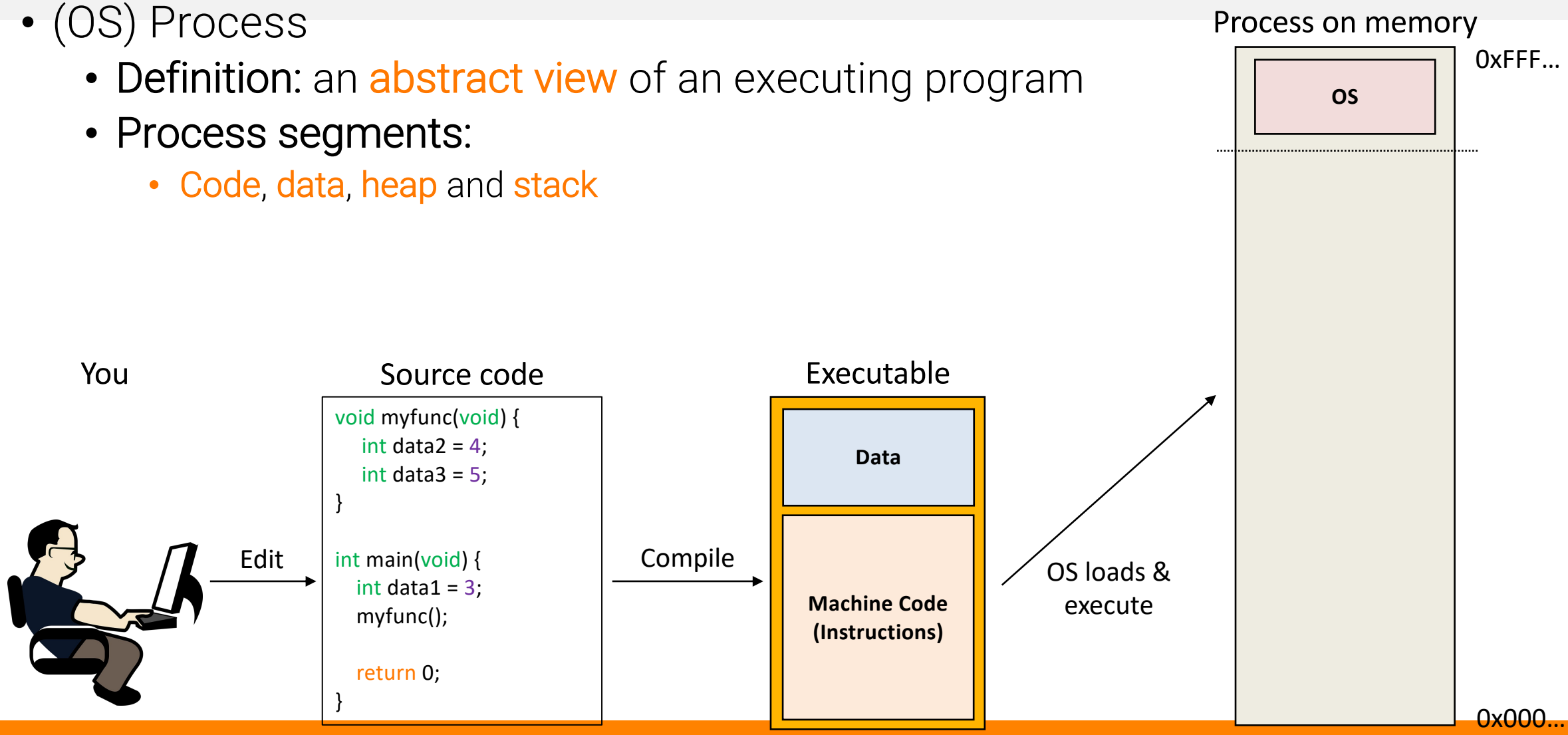
ex. $ hexdump subvert.c

```
(base) os1 ~/temp 1014$ hexdump example
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0002 003e 0001 0000 0400 0040 0000 0000
0000020 0040 0000 0000 0000 1918 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001e 001d
0000040 0006 0000 0005 0000 0040 0000 0000 0000
0000050 0040 0040 0000 0000 0040 0040 0000 0000
0000060 01f8 0000 0000 0000 01f8 0000 0000 0000
0000070 0008 0000 0000 0000 0003 0000 0004 0000
0000080 0238 0000 0000 0000 0238 0040 0000 0000
0000090 0238 0040 0000 0000 001c 0000 0000 0000
00000a0 001c 0000 0000 0000 0001 0000 0000 0000
00000b0 0001 0000 0005 0000 0000 0000 0000 0000
00000c0 0000 0040 0000 0000 0000 0040 0000 0000
00000d0 0704 0000 0000 0000 0704 0000 0000 0000
```

...

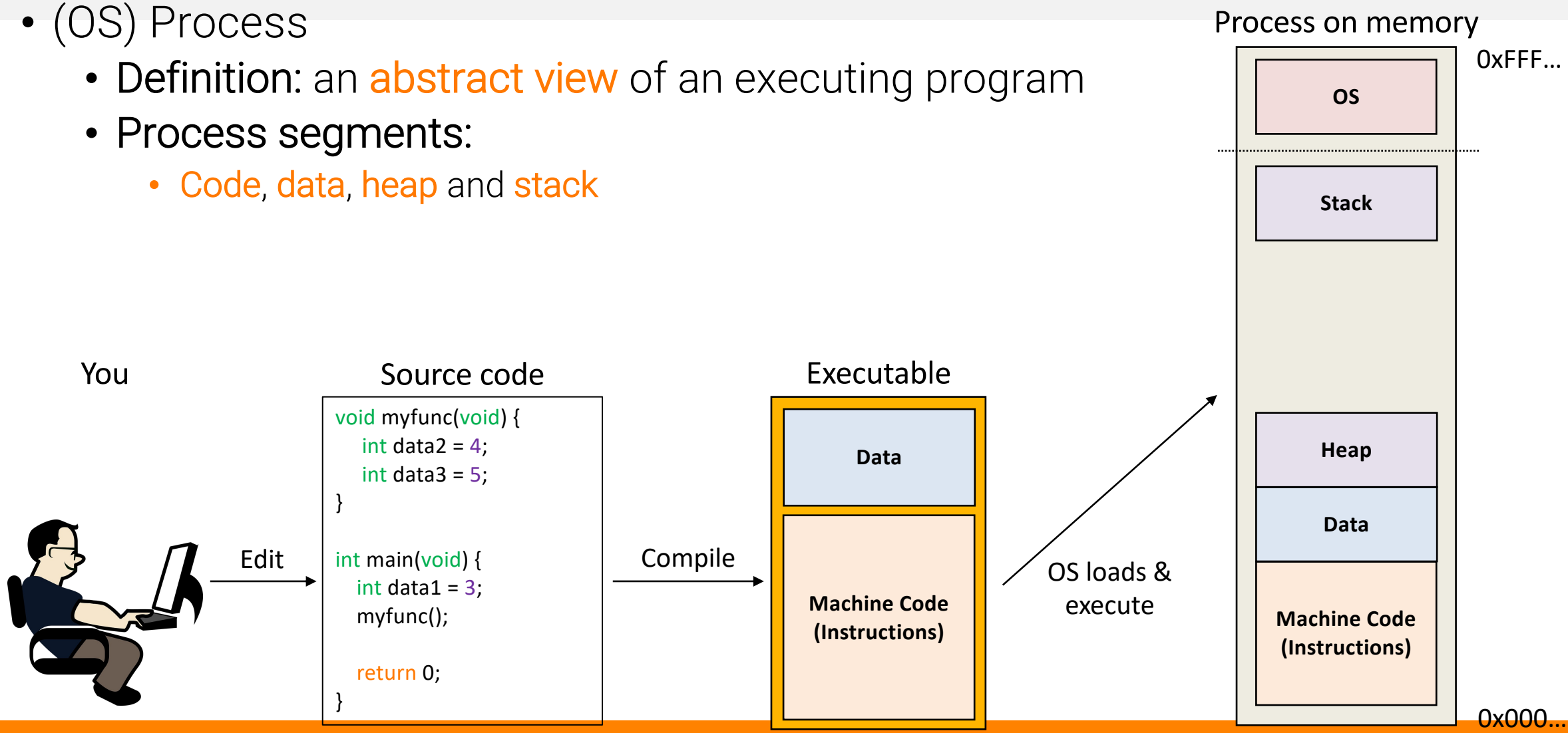
Provide abstraction: a process

- (OS) Process
  - Definition: an abstract view of an executing program
  - Process segments:
    - Code, data, heap and stack



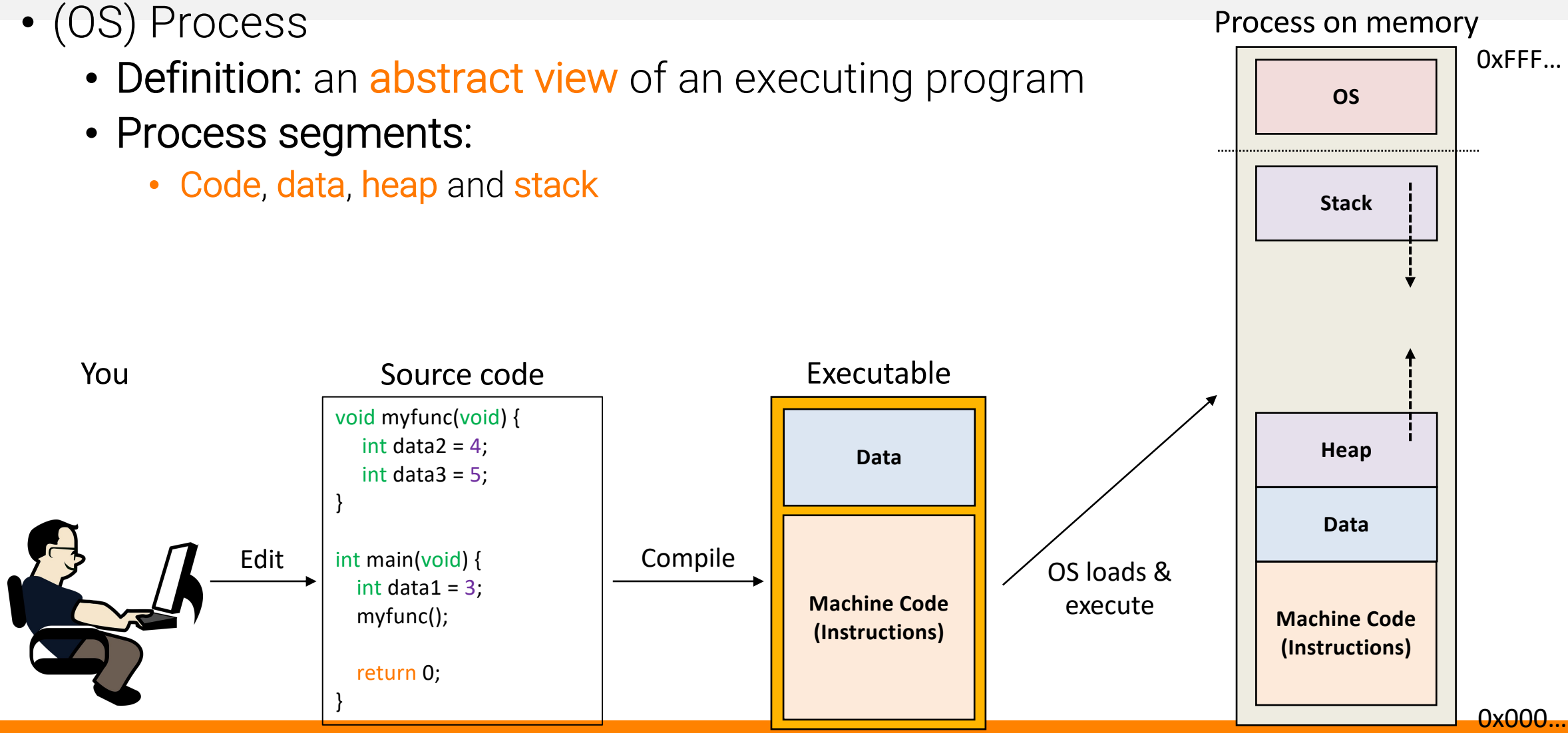
Provide abstraction: a process

- (OS) Process
  - Definition: an abstract view of an executing program
  - Process segments:
    - Code, data, heap and stack



Provide abstraction: a process

- (OS) Process
  - Definition: an abstract view of an executing program
  - Process segments:
    - Code, data, heap and stack



x86-64 Assembly

Overview

Instruction: Opcode + Operands

- You can consider as *Command* + *Arguments*

  Opcode Operands

- *Opcode* to
  - Move data, do arithmetic, control devices, change CPU context ..

- *Operands*
  - Immediate value
    Integer (or float number) constants
  - Register
    - 8 general purpose registers for 32-bit architecture
    - +8 for 64-bit architecture (r9 – r15)
  - Memory
    - Index, base, offset format

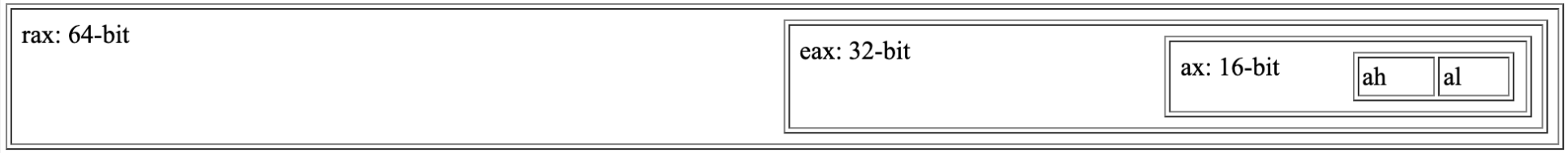
Registers

- Registers
  - Matches the word size (64-bits)
  - Each has its own name
    - Variants of the name allow accessing low order 4-bytes
    - Also 2-byte and 1-byte variants
- Non-integer registers are not discussed in this class

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

Register



- 1972: al and ah are the 8 bit “char” size registers
  - al: the low 8 bits
  - Ah: the high 8 bits

- Intel 8086 (1978), 16-bit processor
  - Register width is 16-bit

```
%ax, %bx, %cx, %dx, %si, %di, %bp, %sp, %ip
%al, %bl, %cl, %dl, %sil, %dil (8-bit regs)
```

- Intel 80386 (1985), 32-bit processor

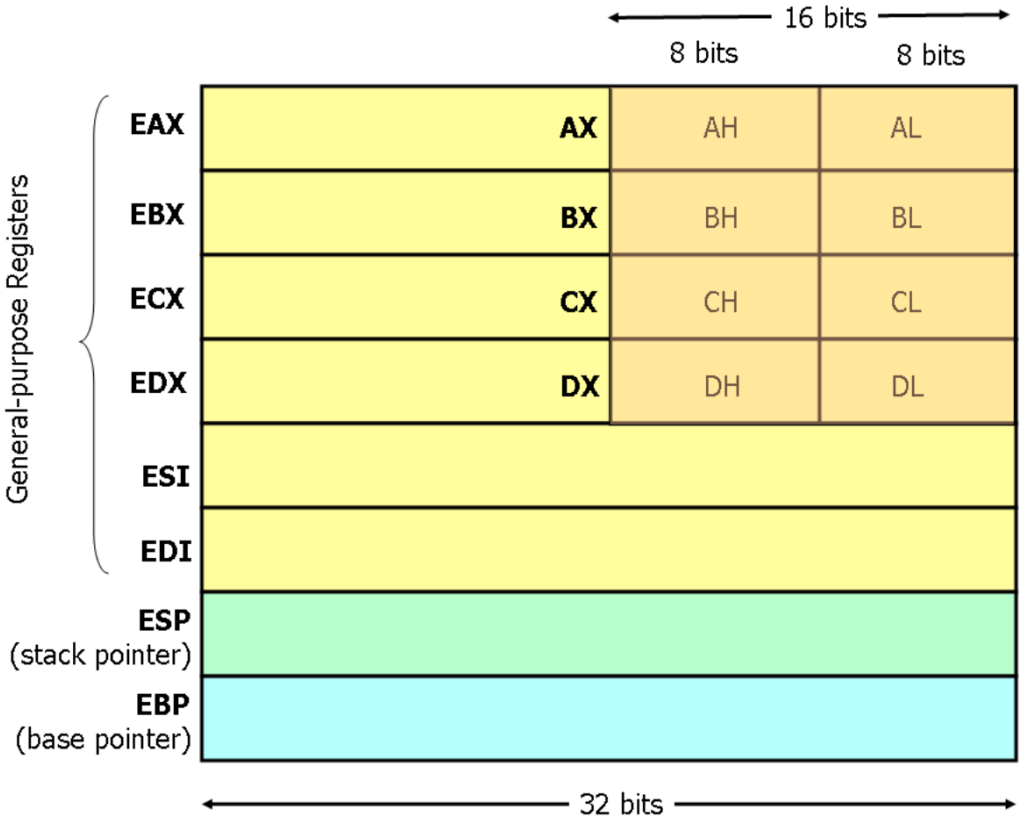
```
%eax, %ebx, %ecx, %edx, %esi, %edi, %ebp, %esp, %eip
```

- Intel Pentium 4 64-bit (Prescott, 2004)

```
%rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %rip
%r8 -- %r15
```

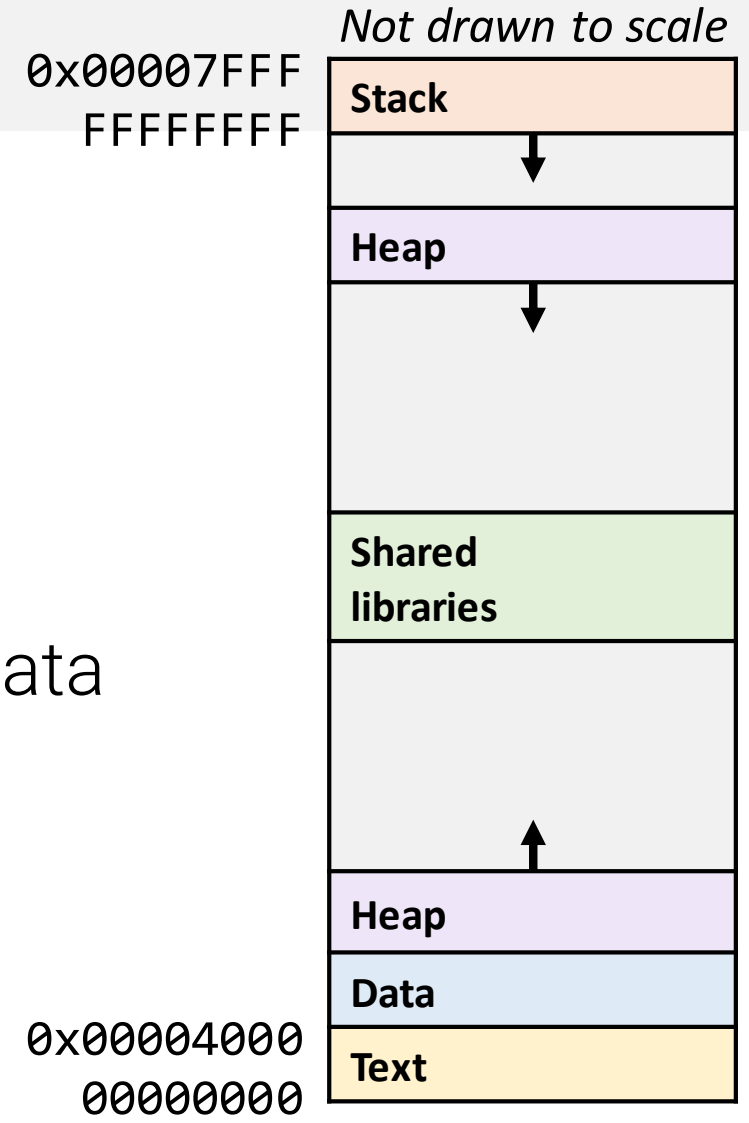
Register

- Modern (i.e 386 and beyond) x86 processors have eight 32-bit general purpose registers
- Historically,
  - EAX: used by a number of arithmetic operations
  - ECX: the counter since it was used to hold a loop index
- most of the registers have lost their special purposes in the modern instruction set
- But, ESP and EBP reserved for special purpose



Memory

- Treat it like a big array of bytes
  - More complicated in practice
- x86-64 is byte addressable
  - Each byte can be directly accessed
- Different locations contain different types of data
  - OS handles arranging these



Intel x86 Assembly Syntax

- AT&T/GNU Syntax
  - [instruction] [source] [destination]
  - mov %esp, %ebp `mov %esp,%ebp`
    - Assign the value of esp to ebp, i.e., `ebp = esp;` in C
  - mov $0x0, %eax `movl $0x0,-0x4(%ebp)`
    - Assign value 0 to eax, i.e., `eax = 0;` in C
    - Assign value 0 to the address pointed by ebp-4, i.e., `ebp[-1] = 0;` (ebp as int *)
  - [instruction] [destination]
  - call 0x80483c0 <printf@plt> `call 0x80483c0 <printf@plt>`
    - Call `printf()`
  - jne 0x804861f <main+127> `jne 0x804861f <main+127>`
    - Jump to 0x804861f if the comparison result was Not Equal (NE)

OP A B
B = A OP B

add %eax, %ebx
ebx = eax + ebx;

Glossaries: Instructions

- mov: move value from src to dst
  - mov %esp, %ebp (`ebp = esp;`)
  - movl: move 4 bytes (move long integer)
  - movw: move 2 bytes (move word integer)
  - movb: move 1 byte (move byte integer)
- call: call the destination function
  - call 0x80483c0 <printf@plt>
  - call 0x80483b0 <strcmp@plt>
- ret: return from a function call
  - ret `0x0804863d <+157>: ret`

Glossaries: Instructions

- Arithmetic Operations
  - add %eax, %ebx (`ebx = eax + ebx;`)
  - add $1, %eax (`eax = eax + 1;`)
  - sub $0x228, %esp (`esp = esp - 0x228;`)
  - xor %eax, %eax (`eax = eax ^ eax;`)

Glossaries: Instructions

- Compare
  - cmp $0x0,%eax
    ```
    cmp    $0x0,%eax
    ```
  - Compare the value 0x0 to the value of %eax
  - Comparison result will be stored in the EFLAGS register
    - Will be used in the jump instruction
- Jump
  - je: jump if equal
    - `If (eax == 0)`
  - jne: jump if not equal
    ```
    jne    0x804861f <main+127>
    ```
    - `if (eax != 0)`
  - jg: jump if greater
    - `If (eax > 0)`
  - jge: jump if greater and equal
    - `If (eax >= 0)`
  - Jmp: jump anyway!
    ```
    jmp    0x8048633 <main+147>
    ```

Dereferencing in x86 Assembly

- Parenthesis means accessing the register as an address
- mov %eax, (%esp) `mov %eax,(%esp)`
  - Regard %esp as int *esp; in C
  - `*esp = eax;` (move the value of eax to the address pointed by %esp)
  - `esp[0] = eax;`
- movl $0x0, -0x4 (%ebp) `movl $0x0,-0x4(%ebp)`
  - Move 0 to (%ebp-4)
  - Regard %ebp as int *ebp; in C, -1 to int* is indeed -4 in address
  - `*(ebp-1) = 0;`
  - `ebp[-1] = 0;`

Can't move between two memory addresses directly

Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

Address	Value
xp	
xy	

Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	???
%rdx	???

Address	Value
0x1234	100
0x1238	200

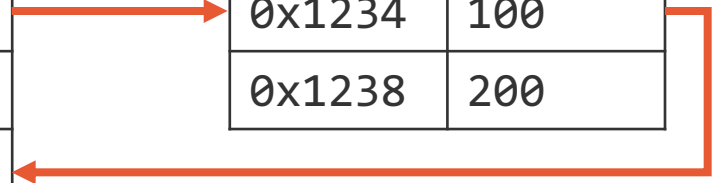
Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	100
%rdx	???

Address	Value
0x1234	100
0x1238	200



Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	100
%rdx	200

Address	Value
0x1234	100
0x1238	200

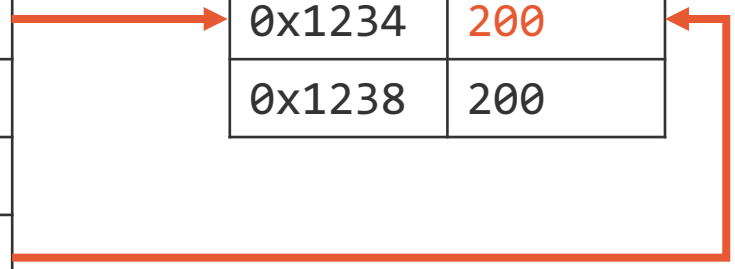
Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	100
%rdx	200

Address	Value
0x1234	200
0x1238	200



Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	100
%rdx	200

Address	Value
0x1234	200
0x1238	100

Example

```
void swap (long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Register	Value
%rdi	0x1234
%rsi	0x1238
%rax	100
%rdx	200

Address	Value
0x1234	200
0x1238	100

Glossaries:
Immediate Value vs. Address

- $0x8048700
  - A number with '$' prefix is regarded as an immediate value

- movl $0x8048700, %edx
  - `edx = 0x8048700;`

Glossaries:
Immediate Value vs. Address

- 0x80486e7
  - A number w/o '$' prefix is regarded as an address
  - In x86 assembly, it refers to the value in that address
- mov 0x80486e7, %eax
  - Move the value stored in the address 0x80486fd to %eax
  - If 0x80486fd stores 32, then `%eax = 32`;
- lea 0x80486e7, %eax `lea 0x80486e7,%eax`
  - LEA: Load Effective Address, load the address of src to dst
  - 0x80486e7 refers to the value stored in the address
  - The address of that value is 0x80486e7
  - Thereby, `%eax = 0x80486e7`;

Load Effective Address Instruction

- **`leaq D(RegRb, RegRi, S), dst`**
  - Stores the calculated address in `dst`
  - Does not access memory
- Used to avoid repeated calculation of memory addresses

Using LEA for Arithmetic

- LEA is used by the compiler to calculate expressions of the form x + k*y
  - k = 1, 2, 4, or , 8
- Faster than using the multiply instruction

```
// C code
long multiply_by_12(long x) {
  return x*12;
}

; ASM code
leaq (%rdi,%rdi,2), %rax
salq $2, %rax
ret
```

Example

```
long func(long x, long y, long z) {
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
func:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  shlq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```