CMPT 295

Unit - Machine-Level Programming

Lecture 15 – Assembly language – Program Control –
Function Call and Stack - Passing Control – cont'd

新年快乐 / 新年快樂
Xīnnián kuàile

Cung Chúc Tân Xuân

Happy Lunar New Year!

Chúc Mừng Năm Mới

过年好 / 過年好
Guò nián hǎo

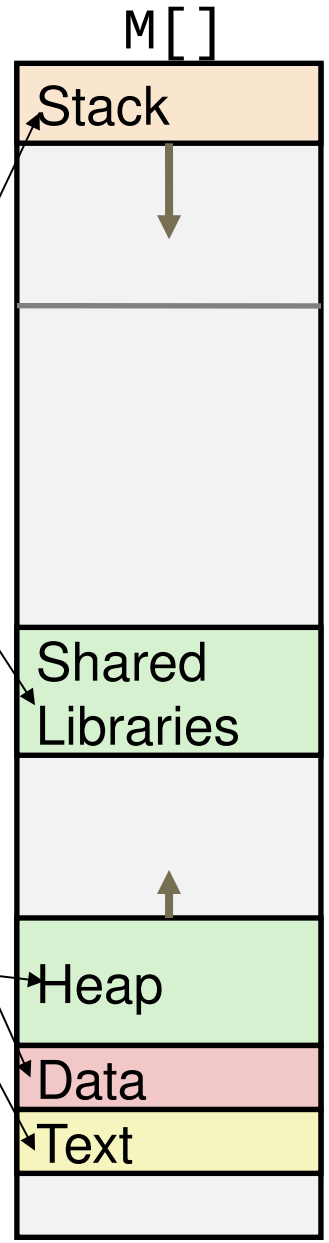
새해 복 많이 받으세요
saehae bog manh-i bad-euseyo

Homework

Memory Allocation Example

Where does everything go?

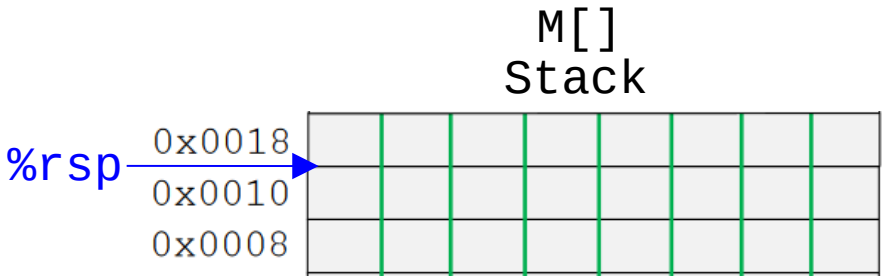
```
#include ...

char hugeArray[1 << 31]; /* 2^31 = 2GB */
int global = 0;

int useless(){ return 0; }

int main ()
{
    void *ptr1, *ptr2;
    int local = 0;
    ptr1 = malloc(1 << 28); /* 2^28 = 256 MB*/
    ptr2 = malloc(1 << 8);  /* 2^8  = 256 B*/

/* Some print statements ... */
}
```



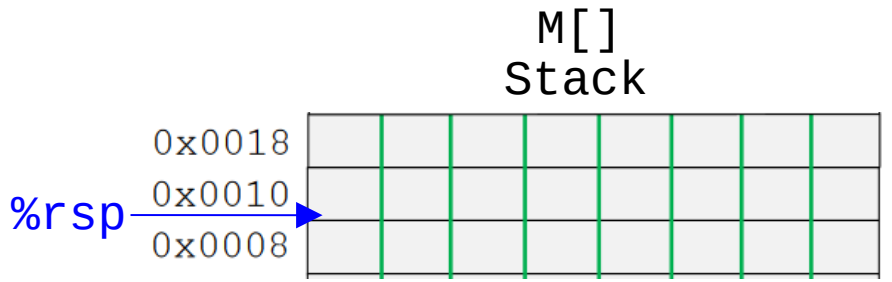
Why 8?

- `pushq src`
  - Fetch value of operand `src`
  - Decrement `%rsp` by 8
  - Write value at address given by `%rsp`
- `popq dest`
  - Read value at %rsp (address) and store it in operand dest (must be register)
  - Increment `%rsp` by 8

1. `%rsp` contains the memory address `0x0018`



2. `%rsp` contains the memory address



`%rsp` contains the memory address

Last Lecture

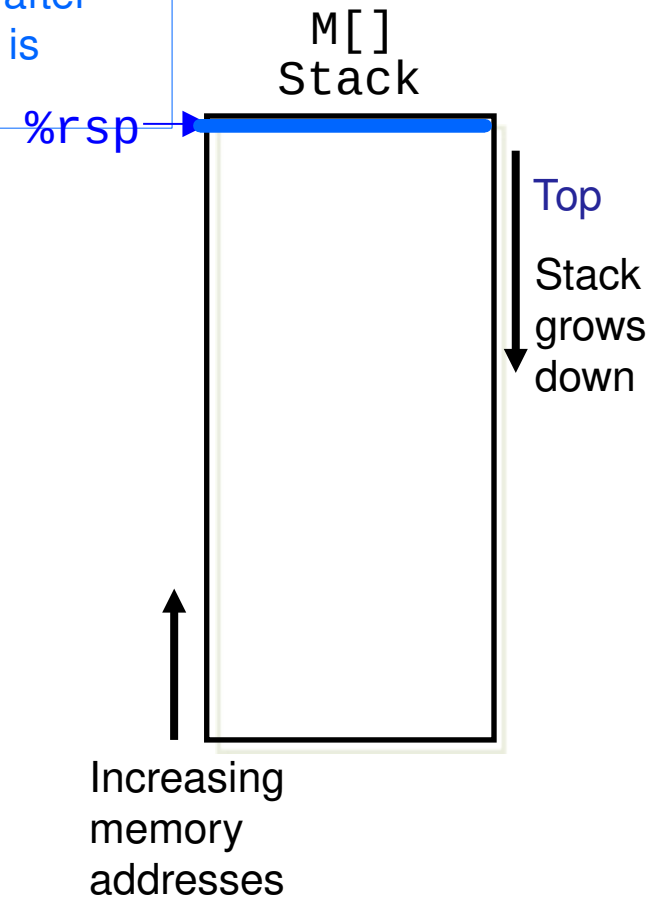
- Function call mechanisms: 1) passing control, 2) passing data, 3) managing local data on memory (stack)
- Memory layout
  - Stack (local variables …)
  - Heap (dynamically allocated data)
  - Data (statically allocated data)
  - Text / Shared Libraries (program code)
- A “stack” is the right data structure for function call / return
  - If `multstore` calls `mult2`, then `mult2` returns before `multstore` returns
- x86-64 stack register and instructions: stack pointer **`%rsp`**, **`push`** and **`pop`**

Passing control mechanism
x86-64 instruction: `call` and `ret`

- `call func`
  - `pushq PC`
  - `set PC to func`
  - `jmp func`

After 1 call …

Effect: return address, i.e., the address of the instruction after `call func` (held in PC) is pushed onto the stack

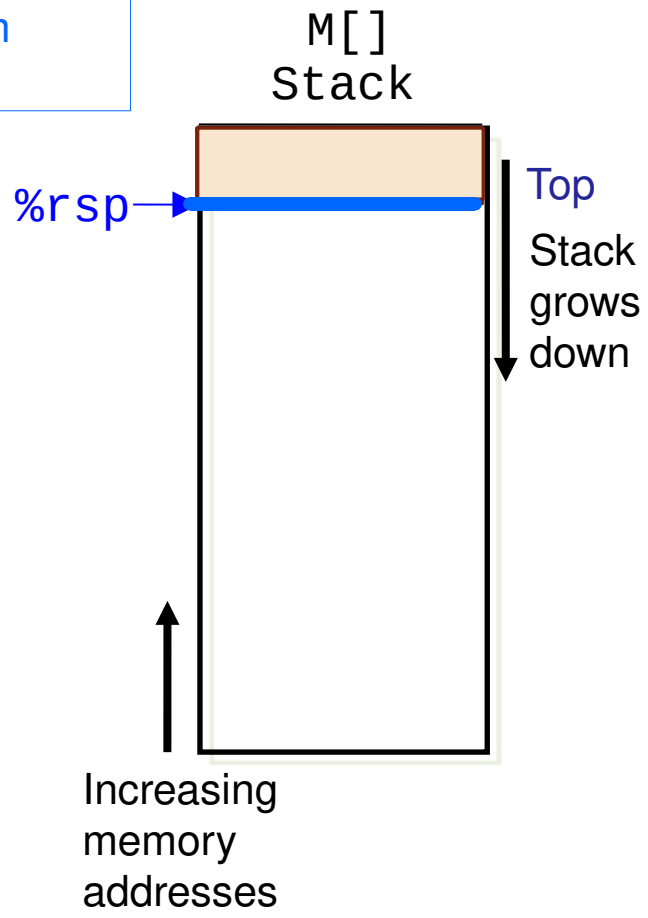


Passing control mechanism
x86-64 instruction: `call` and `ret`

- `ret`
  - `popq PC`
  - `jmp PC`

After returning from the call …

Effect: return address, i.e., the address of instruction after `call func`, is pop'ed from the stack and stored in PC



Example

```
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
    return;
}
```

```
long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

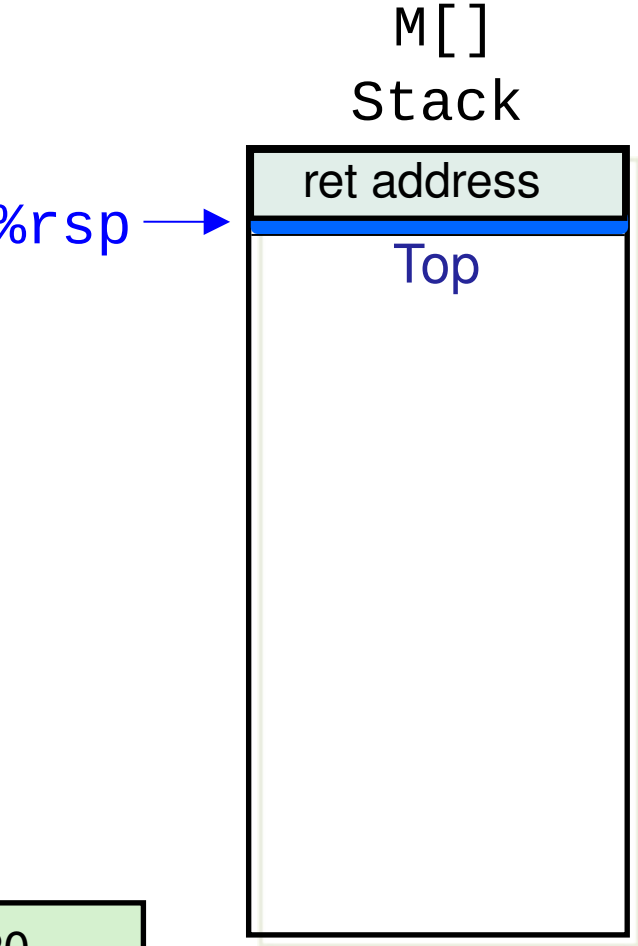
```
0000000000400540 <multstore>:
  400540: push   %rbx                # Save %rbx
  400541: mov    %rdx,%rbx           # Save dest
  400544: callq  400550 <mult2>      # mult2(x,y)
  400549: mov    %rax,(%rbx)         # Save at dest
  40054c: pop    %rbx                # Restore %rbx
  40054d: retq                       # Return
```

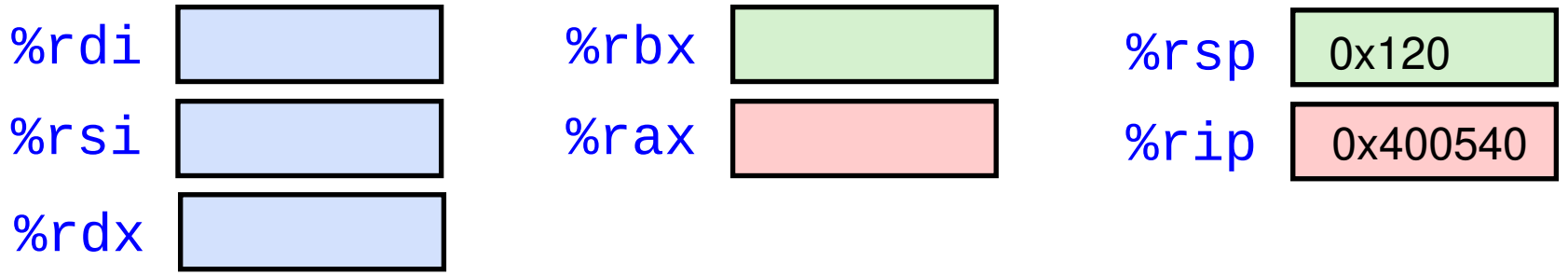
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Example – Steps 1 and 2

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  retq               # Return
```

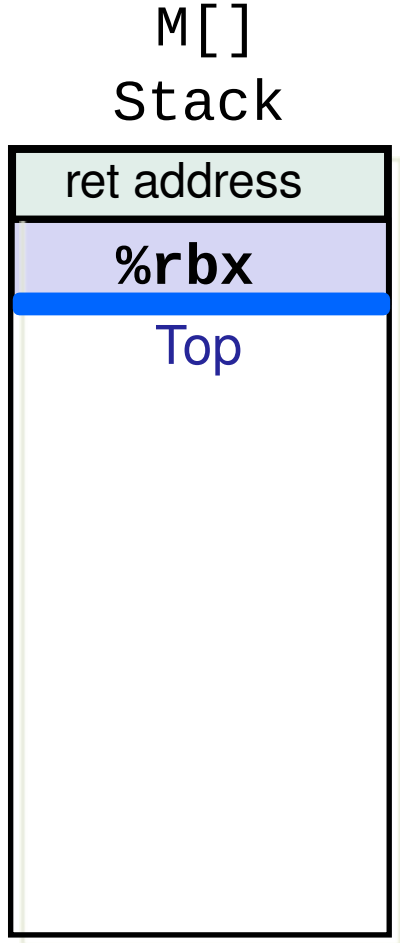


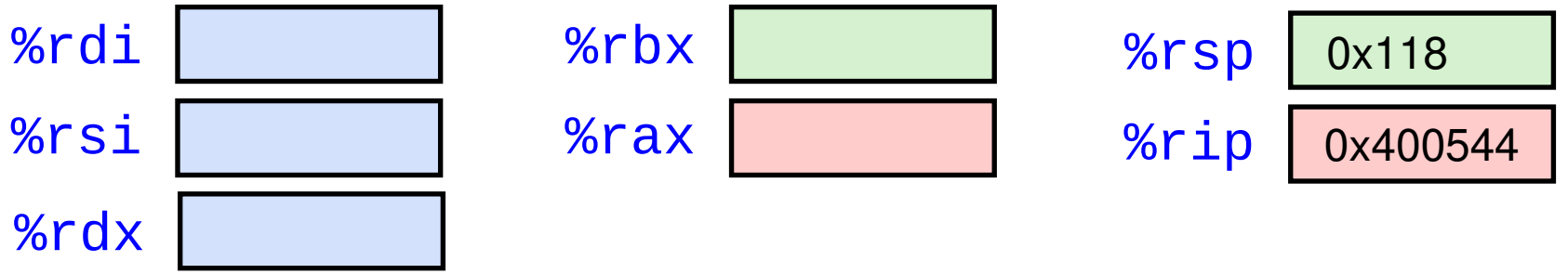


Example – Steps 3 and 4

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  retq               # Return
```

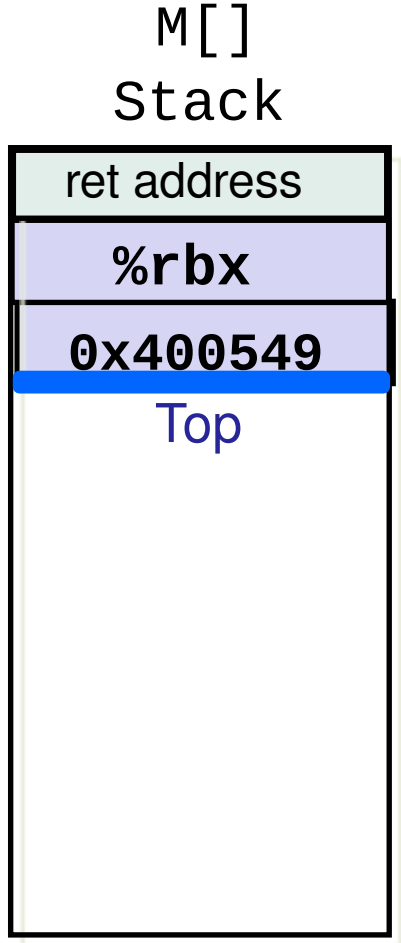


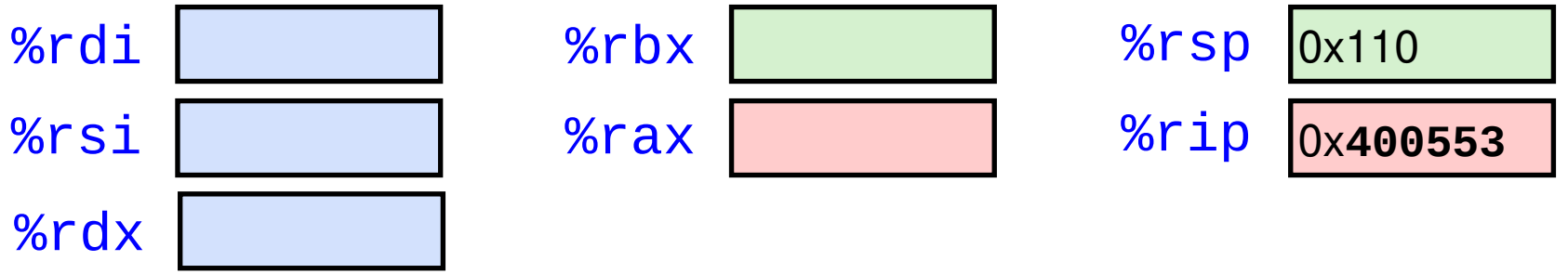


Example – Steps 5 and 6

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  retq               # Return
```

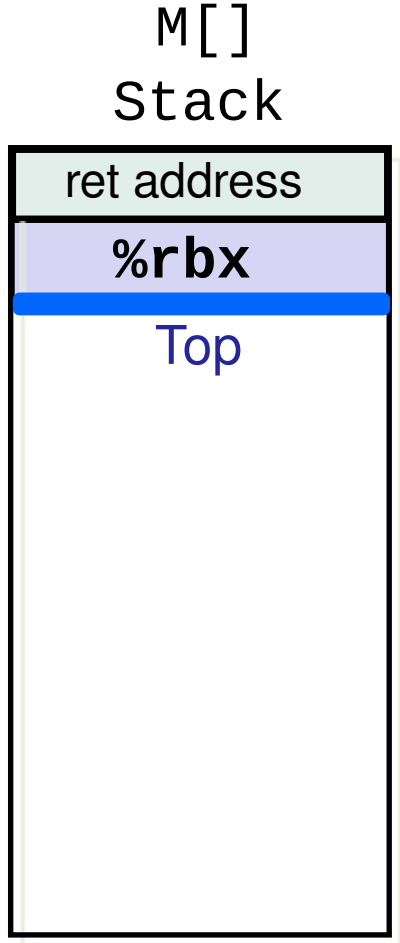


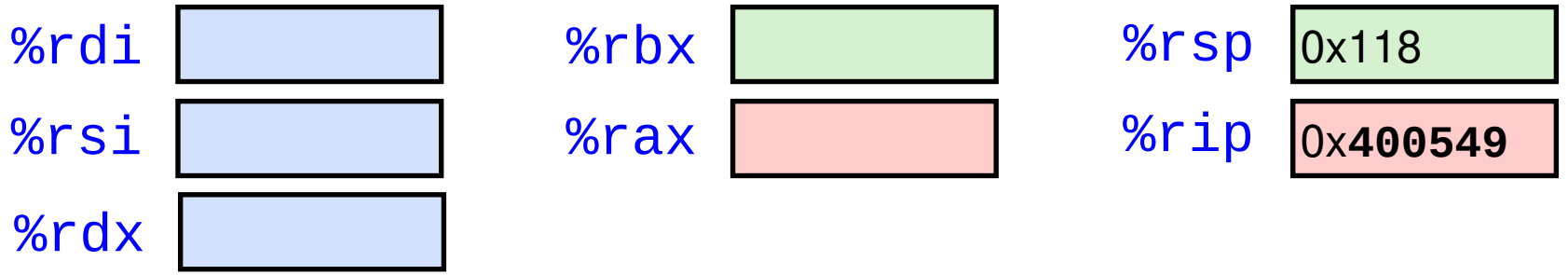


Homework
Example – Steps 7, 8 and 9

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```





Summary

- Function call mechanisms: 1) passing control, 2) passing data, 3) managing local data on memory (stack)
- Memory layout
  - Stack (local variables …)
  - Heap (dynamically allocated data)
  - Data (statically allocated data)
  - Text / Shared Libraries (program code)
- A “stack” is the right data structure for function call / return
  - If `multstore` calls `mult2`, then `mult2` returns before `multstore` returns
- x86-64 stack register and instructions: stack pointer **`%rsp`**, **`push`** and **`pop`**
- x86-64 function call instructions: **`call`** and **`ret`**

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack – Recursion
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations