

Machine Programming with Assembly

COMP201 Lab Session
Spring 2025



**KOÇ
UNIVERSITY**

GDB Recap

- GDB is a debugger for C (and C++), which allows:
 - Run the program up to a certain point,
 - Pause execution and see the current state,
 - Continue execution step by step
- Higher level debugging
 - Simpler to interpret,
 - but not always useful
- What if we want to dive deeper?

Debugging using Assembly Language

- Debugging can be easier if we can see what actually happens under the hood:
 - the individual CPU operations,
 - registers,
 - or the memory.
- To go deeper, one must look at the Assembly code.
- The command in GDB command line: 'disassemble' outputs the assembly translation of the function currently being executed, or the translation of a target function if one is supplied.
 - disassemble
 - disassemble [Function]

Assembly

- A (very) low-level programming language
- Designed for a specific type of processor
- It may be produced **by compiling** source code from a high-level programming language (such as C/C++)
- It can also be written from scratch.
- Assembly code can be converted to machine code using an assembler.

Assembly Language

- Assembly languages differ between processor architectures
- Often similar instructions and operators
- Below are some examples of instructions supported by x86 processors:
 - o **mov** - copy data from one location to another
 - o **add** - add two values
 - o **sub** - subtract a value from another value
 - o **push** - push data onto a stack
 - o **pop** - pop data from a stack (will be covered later)
 - o **jmp** - jump to another execution point
 - o **int** - interrupt a process
 - o **cmp** - compares two operands

Registers

- Registers are data storage locations directly on the CPU
- Usually, the size, or width, of a CPU's registers define its architecture
- In a 64-bit CPU, the registers will be 64 bits wide
- The same is true of 32-bit CPUs (32-bit registers), 16-bit CPUs, and so on.
- Registers are very fast to access and are often the operands for arithmetic and logic operations.
 - `%rbp` and `%rsp` are special purpose registers
 - `%rbp` is the base pointer, which points to the base of the current stack frame
 - `%rsp` is the stack pointer, which points to the top of the current stack frame
 - `%rbp` always has a higher value than `%rsp` because the stack starts at a high memory address and grows downwards.

Understanding Assembly

Consider the following Assembly code:

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -4(%rbp)
movl -4(%rbp), %eax
imull -4(%rbp), %eax
popq %rbp
ret
```

Understanding Assembly

- Normally these are the first 2 instructions of all Assembly codes:

```
pushq %rbp
```

```
movq %rsp, %rbp
```

- The first two instructions are called the function **prologue** or preamble.
- First we **push** the **old base pointer** onto the stack to save it for later.
- Then we **copy** the value of the **stack pointer** to the **base pointer**.
- After this, **%rbp** points to the base of main's stack frame.

Understanding Assembly

```
movl %edi, -4(%rbp)
```

- The first integer argument is passed in the edi register.
- So this line copies the argument to a local (offset -4 bytes from the frame pointer value stored in rbp).

```
movl -4(%rbp), %eax
```

- This copies the value in the local to the eax register.

Understanding Assembly

```
imull -4(%rbp), %eax
```

- Multiply the contents of eax register with eax register

```
popq %rbp
```

- pop original register out of stack

```
ret
```

- return

Let's Revisit

```
square:
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -4(%rbp)
    movl -4(%rbp), %eax
    imull -4(%rbp), %eax
    popq %rbp
    ret
```

Yes, it is just simple squaring function:

```
int square(int num) {
    return num * num;
}
```

Example 1:

What is the equivalent C code?

```
; int f1(int a, int b)
f1:
    leal (%rdi,%rsi), %eax
    subl %esi, %edi
    imull %edi, %eax
    ret
```

Example 2:

What is the x86-64 assembly version of this code?

```
int f2(int a, int b, int c) {  
    int max = a;  
    if (b > max) {  
        max = b;  
    }  
    if (c > max) {  
        max = c;  
    }  
    return max;  
}
```

Example 3:

What is the equivalent C code?

```
; int f3(int num)
f2:
    movl $1, %edx
    movl $1, %eax
    jmp .L2
.L3:
    imull %edx, %eax
    addl $1, %edx
.L2:
    cmpl %edi, %edx
    jle .L3
    rep ret
```

Example 4:

What is the x86-64 assembly version of this code?

```
int f4(int n) {  
    int fib1 = 0;  
    int fib2 = 1;  
    int fib = 0;  
    for (int i = 2; i <= n; i++) {  
        fib = fib1 + fib2;  
        fib1 = fib2;  
        fib2 = fib;  
    }  
    return fib;  
}
```