- CPSC 275: Introduction to Computer Systems

CPSC 275: Introduction to Computer Systems

Fall 2025

- Syllabus
- Schedule
- Resources
- Upload
- Solution

# Lab 6: Getting Started with IA-32 Assembly Programming

## Objectives

The main goal of this laboratory is to:

1. Understand how C code translates into IA-32 assembly instructions.
2. Learn to assemble, execute, and disassemble simple IA-32 programs.
3. Recognize the basic structure and components of assembly language, including directives, labels, and instructions.

This lab introduces the basic instructions of the IA-32 assembly. It does not provide a complete description of the IA-32 architecture but covers enough material to write simple programs from scratch. For more information on these instructions, see IA-32 Reference.

## A Simple Program

Consider the following C program (`num.c`):

```
#include <stdio.h>

void main()
{
    int x = 10;
    int y = 20;

    printf("x = %d y = %d\n", x, y);
}
```

When you compile this program with the following command,

```
$ gcc -m32 -S num.c
```

the compiler will generate assembly code (`num.s`) that may look like this:

```
1       .LC0:
2               .string "x = %d y = %d\n"
3
4               .globl  main
5       main:
```

```
6               pushl   %ebp            # save frame pointer
7               movl    %esp,%ebp       # adjust stack pointer
8
9               movl    $10,%eax
10              movl    $20,%ebx
11              pushl   %ebx
12              pushl   %eax
13              pushl   $.LC0
14              call    printf
15
16              leave                   # restore the current activation
17              ret                     # return to caller
```

Note that it has three distinct parts:

- **Directives** begin with a dot and provide structural information useful to the assembler or linker. You will need to know at least two directives: `.globl` and `.string`. The directive `.globl main` indicates that the label `main` is a global symbol that can be referenced by other code modules. The directive `.string` defines a string constant that the assembler should insert into the output code.
- **Labels** end with a colon and indicate, by their position, the association between names and locations. For example, the label `.LC0:` marks the immediately following string as `.LC0`. The label `main` indicates that the instruction `pushl` is the first instruction of the `main` function. By convention, labels beginning with a dot are temporary local labels generated by the compiler, while other symbols are user-visible functions and global variables.
- **Instructions** include everything else, typically indented to visually distinguish them from directives and labels.

## Assembling the code

Replace the compiler-generated code with the provided example code using a text editor, and save the file. Make sure to comment your code (Lines 9–14).

Compile it with:

```
$ gcc -m32 -o num num.s
```

and run it with:

```
$ ./num
```

## Disassembling the code

Now let us examine the machine code produced by the compiler using the `objdump` utility:

```
$ objdump -d num
```

You may wish to redirect the output to a text file. Examine the section labeled `<main>`. What do you observe?

## Exercise

Write an assembly program (`getbyte.s`) that extracts byte $n$ from a 4-byte word $x$. Here, $n$ is 0, 1, 2, or 3, and byte 0 is the least significant byte. For example, if $x$ = `0x12345678` and $n$ = 1, the output should be `0x56`. Assume that the values of $x$ and $n$ are hardcoded. Ensure your program works for all valid values of $n$, and include comments in your code. **Hint**: Use shift instructions; note that the source operand for shifts must be either an

immediate or the `%cl` register.

Use the following skeleton code to get started:

```
# Program: getbyte.s
# Purpose:
# Author:
# Date:

        .globl  main
main:
        pushl   %ebp            # save frame pointer
        movl    %esp,%ebp       # adjust stack pointer

        ### YOUR CODE SHOULD GO HERE
        ...

        leave                   # restore the current activation
        ret                     # return to caller
```

Compile it with:

```
$ gcc -m32 -o getbyte getbyte.s
```

and run it with:

```
$ ./getbyte
```

Your program should produce the following output when $n = 1$:

```
Byte 1 of 0x12345678 is 0x56.
```

## Handin

When completed, ask your instructor or TA to check your work.

- **Welcome: Sean**
  - LogOut