ECE 273

Computer Organization Laboratory

Assembly Language Programming

Clemson University
Department of Electrical and Computer Engineering
Clemson, SC 29634

Updated January 2014
by Ryan Izard

Table of Contents

Introduction

The purpose of ECE 273 is to teach you the basics of Intel 80x86 assembly language. You will learn in ECE 272 lecture that assembly language (or simply "assembly") is important because it is the principal link between the software world of high-level languages like C and Java and the hardware world of CPU design. Assembly language is the lowest-level, human-readable programming medium we can use to express complete application programs. Assembly language gives full access to the programmable features of the hardware, so a good understanding of it will provide valuable insight into the fundamentals of CPU design, the operation of the datapath, and program execution.

Since the creation of compilers and assemblers, assembly language programming as an art has virtually disappeared from the face of the Earth, so of what use is it to you? There are several major advantages to understanding assembly language. First, compilers translate high-level languages into assembly language, so compiler writers must understand assembly. Operating systems also include critical components written in assembly. Furthermore, embedded and mobile device programming often require knowledge of assembly language. As these technologies become more and more important to the overall performance and flexibility of computer systems, knowledge of the computer at the assembly-language level will prove to be a valuable asset. Even if you spend your entire career programming in high-level languages, a basic understanding of assembly language concepts will give you an insight into your work that will in turn make you more valuable as an electrical or computer engineer.

With these considerations in mind, ECE 273 will not strive to make you a proficient assembly language programmer. However, like most programming languages, you simply cannot grasp the key concepts by mere discussion. Therefore, you will, for a semester, become an assembly language programmer just like the “hackers" of old. ECE 273 is a laboratory class, meaning we will provide an environment for you to gain hands-on experience with the tools and concepts used in the course. This approach also means that you will only get from it what you put into it. The more time you spend working on your program, the more you will learn from it, and the more you will understand about how and why assembly languages works the way it does.

The whole laboratory class provides a study of assembly language from the point of view of a high-level language, namely C. For example, C provides a control structure called the `for` loop, and we will (eventually) discuss how to implement a `for` loop in assembly language. As such, a good knowledge of C is necessary to fully understand and succeed in the laboratory assignments. The goal of the first lab is to introduce you to the tools you will need throughout this course. Most of what you will need to know to be successful in ECE 273 will come from a collection of documents available on the web at http://www.clemson.edu/ces/departments/ece/resources/ECE273Lab.html. This site includes: background material on the Intel 80386, which outlines the features of the CPU and its respective assembly language; the GNU Debugger (GDB), although it is not required (but still encouraged) for this course; and information on navigating your way around the UNIX terminal. The Appendix of this manual also provides resources for successfully completing each lab.

For development and testing, any flavor of Linux should suffice, as they are all based on UNIX. If you are new to UNIX, you may need to seek out a more complete reference at the library or online and spend some time familiarizing yourself with the fundamentals. For each lab, you must be able to log onto a campus UNIX machine and edit files with any of the standard text editors (vi, gedit, nano, pico, etc). However, when working in any Mac Lab, you are encouraged to use Xcode on any of the iMacs. An IDE such as Xcode is an efficient way to work on your laboratory assignments. *Please note that the iMacs in the 321 Riggs Lab or any other Mac Lab at Clemson are not directly compatible with the assembly code in ECE 273.* This means your assembly programs cannot be tested on them; the iMacs with Xcode are simply a convenient interface for editing your code. The code we will be studying is written for 32-bit Linux machines running on Intel 80386 (or x86) hardware – the iMacs are 64-bit Mac OS X machines with Intel hardware. (Because assembly language is a part of the Instruction Set Architecture (ISA), it is specific to not only the OS but also the hardware it runs on.) If you choose to write and test the code on your own 32-bit Linux machine or virtual machine, then you should not have a problem; however, you should test your code on the campus Linux machiens discussed in **Lab 1** and **Appendix C**. **Lab 1** will discuss how to compile and test your 32-bit Linux code from a Mac Lab. You can refer back to it as a reference for **Labs 2 – 6**. If you would like to work on your code remotely, refer to **Appendix C**. *For online courses, this appendix should be used as a guide when working on and testing your code remotely.*

On another note, programmers have a general idea how code should be formatted, but it is important to standardize the formatting for this course. This not only makes assignments easier to grade, but it makes them easier to read and understand. Please be aware that uncommented or improperly commented programs will *not* be accepted, and will result in a reduced grade. See the syllabus for specifics on grading and the **Appendix A** for details on comments. Poorly commented assembly code, due to its low-level and complexity, is difficult to understand and is of little use to third party users. ***Your comments need to relay the purpose of the code and not simply verbalize the instructions.*** Consult your lab instructor if you have any questions about code comments.

Finally, maintain academic honesty. You will be turning in your assignments via email and/or Blackboard, and we will actively work to detect copied programs, including those that have been cosmetically altered. ***Don't do it!*** If you don't understand what is going on, ask your lab instructor for help. Note that each lab has a fully functional solution in **Appendix E**. *Please refrain from referencing the solution unless you have exhausted your resources (e.g. the instructor, the lab manual, and online sources).* Sometimes you may get stumped and find it useful to work backwards from the solutions; however, *submissions copied verbatim from the solution will not be accepted and will be considered academically dishonest*. Remember, it is important to know not just what the working code looks like, but also why it works and how to write your own. You will be assessed via quizzes and/or a final exam to verify you know how to program in assembly, as well as the concepts associated with assembly programming. See the syllabus for details.

Lab 1

Compiling and Testing Assembly Code

Student Objectives

- Discover the procedure to write, test, and debug lab assignments in the lab
- Properly use function header and program header comments
- Learn how to submit lab assignments for evaluation

1.1 – Background

The goal of Lab 1 is simply to introduce you to the basic tools and procedures you will use to write, assemble, link, execute, and debug your programs. The task is simple: create an assembly program and run it to demonstrate what it does. If you are already familiar with the UNIX operating system, this assignment will be trivial.

All of your assignments will consist of two program source files. One is a C program that sets up the assignment – it is referred to as the **driver**. You must not alter this file in any way, or your assignment may not work properly when your lab instructor tests it. The second source file is an assembly language file that implements one or more functions called by the C program. Some of this file will be completed for you – it is referred to as the **assembly stub**. Both files are provided in the lab manual following each lab's introduction and discussion. They can be copied from the lab manual; however, typing them out manually will give your more experience with the structure of C and assembly programs. *Note that code copied from a PDF version of the lab manual may not paste in the same order as it appears in the manual.* Check your driver code carefully for copy-and-paste errors before asking for help.

Lab 1 requires no additional code, other than what is provided in the lab manual. As such, we only need to save, compile, and test it out! To do so from the Mac Lab on campus, we must first find a machine with the same ISA and system call conventions. Fortunately, the College of Engineering and Science (COES) has some Linux machines for us to use – they are called the Apollo machines and there are 16 of them named **apollo01.ces.clemson.edu, apollo02.ces.clemson.edu, …, apollo16.ces.clemson.edu**. Note the number of the machine is a two-digit number from 01 – 16, and any machine listed above will work for compiling and running your code. (Simply pick one that is currently online.)

First, we need to transfer our code (the C driver and assembly files) from the Macs, where they are developed, to the Apollo machines, where we will compile and test our program. If you have had some experience in the terminal, you might be aware of the commands SCP and SFTP, which allow the user to securely transfer files from one machine to another. You are more than welcome to use these commands; however the Macs have a custom program for seamlessly accomplishing this task – **cesmount**. What cesmount does is essentially establish a graphical SSH session using your Clemson username with an Apollo machine. This appears on

the iMac Desktop as a removable disk drive named as your username. The advantage to using cesmount is the following: when working with your code in an IDE, such as Xcode, on the iMacs, you can save your code onto your drive (mounted by cesmount) as if it were a flash drive you had plugged in yourself. Now, when you login to the Apollo machines to test your code, the same directory in your 'cesmount-ed' drive is used as your home directory on any Apollo machine. So, if you save your code on your virtual disk drive, it will automatically be updated/mirrored on the Apollo machine you are using – pretty cool! For example, say you just tested your code on an Apollo machine and realized you need to make a change. Simply open up Xcode (or your IDE of choice), resave it to your drive again, and viola it is updated on the Apollo machines and ready for you to test, almost instantly. There is no need to use a command to transfer your files each time you update them.

That was likely more information than you needed to know in order to do the labs. But, as ECE majors, hopefully you found it interesting and perhaps inspiring. So, let's get started with Lab 1. From here, it is assumed you have just logged into an iMac in the lab. If you have any trouble during the following procedure, please ask the instructor or a neighbor for assistance.

1.2 – Assignment

1. Mount your COES user directory to the desktop. Open the terminal. It should appear as a black icon in the dock; however, if it is not there, you can use Spotlight to locate it. Press command + space; this enables the Spotlight search in the upper right corner of the screen. Type "Terminal" or "iTerm" into the prompt. If the application does not appear, do not worry, it is likely not indexed on that machine. Open Finder; it is a face icon on the far-left in the dock. Select Machintosh HD. From there browse to Macintosh HD → Applications → Utilities → Terminal (or iTerm). In the terminal, type "cesmount". If prompted to accept the connection, type "yes". Enter you Clemson username, then press enter. Enter your Clemson password, then press enter. Note, for security purposes, your password will not be displayed as you type it. Rest assured that it is being received as you type. *If you are unable to login using your Clemson credentials, please notify the instructor – you may not have COES account.* Upon success, cesmount will exit. Browse to your Desktop and verify a disk drive has been mounted with your username as its name.

2. Open your disk drive on the Desktop and create a folder for each lab this semester – Lab1, Lab2, …, Lab6. It is suggested you omit any spaces from the folder and file names to simplify browsing in the terminal later on.

3. Open up Xcode. It should appear as a blue icon with a hammer in the dock; however if it is not there, you can use Spotlight or Finder to locate it. Follow the same procedure in Step 1, but instead search Spotlight for "Xcode" or browse to Macintosh HD → Applications → Xcode in Finder.

4. Create a new file in Xcode for your C driver. On the top menu bar, select File → New → New File → C and C++ → C (.c) File. Click the down arrow to the right of the name field to expand the window. Browse to Desktop → COES User Directory → Lab1.

Name the driver file “lab1drv.c” (or a name of your choosing...this file will not be submitted for grading). Save the file.

5. Similar to Step 4, create a new file in Xcode for your assembly file. On the top menu bar, select File → New File → Other → Assembly (.s) File. Follow the same procedure as in Step 4 to browse to your disk drive's Lab1 folder. Name your assembly file as **username_273_sectionNum_labNum.s**. For example, if I were to create an assembly file for myself and I am enrolled in section 002, I would name my assembly file “rizard_273_002_1.s”. Save the file. **(*Consult your syllabus if the file naming convention requested by your instructor is different from the example above.)***

6. Copy or type the C driver code and the assembly code into their respective files and save them. For Lab 1 there is not an assembly stub but the full solution instead. Use this in your assembly file for Lab 1 only. In all labs, these are found after the introduction and problem statement. Add the required comments to the assembly file. For Lab 1, since we have not discussed how to actually write assembly, only the **program header comments** are required. See the Appendix for how to write these comments. As always, ask the instructor if anything is unclear. Comments are worth a significant portion of each lab assignment's grade. They can also be helpful in debugging your programs.

7. In assembly language, the last line of an assembly file is denoted by a **blank line**. So, in your assembly (.s) file, insert a blank line at the end (i.e. press “Enter” or “Return”) – even after any comments you may have at the end of the file. GCC will report a warning if the last line of the file is not blank.

8. Login to an Apollo machine to compile and test your code. Open your terminal application and SSH into an Apollo machine (01 – 16) by typing **“ssh username@apolloXX.ces.clemson.edu”** where XX is a valid machine number. For example, if I were to login to Apollo 08, I would input “ssh rizard@apollo08.ces.clemson.edu”. Press enter. If you are prompted to accept the connection, type “yes” and press enter. Input your password followed by enter. Just like cesmount, your password will not display while you type it. Upon success, SSH will log you into the Apollo machine of your choice. Enter the command “ls” followed by enter. The file structure shown should be identical to that of your cesmount disk drive.

9. Browse to your Lab1 folder by typing “cd Lab1”. If you have your folder set up or named differently, browse to your folder with your Lab 1 files – the C driver and assembly file created in Xcode.

10. Compile and run your code. To do this, type **“gcc -m32 -o myprog lab1drv.c username_273_sectionNum_1.s”**. For example, if I wanted to compile my files created in the steps above, I would type “gcc -m32 -o myprog lab1drv.c rizard_273_002_1.s”. GCC will create an executable named “myprog”. If the “-o <name>” flag is omitted, your program will be named “a.out”. For those who are new to GCC, the argument immediately following “-o” will be the name of your executable. *Make sure it is not the same name as your C driver or assembly file; otherwise, the source code will be overwritten!* The “-m32” tells GCC that the program it is compiling

should be compiled as a 32-bit program. This is necessary, since the assembly we will learn in this course is 32-bit. If GCC compiled without errors, run your code by typing "./myprog". The output should be a prompt for you to input a string of characters. What does the program do? Hint: Try and read the C driver. You will learn later what the assembly code means.

11. Once you have verified your program is working and properly commented, **submit your assignment** to your instructor via email and/or via Blackboard. You only need to submit your assembly file (.s file). Follow the instructions in your syllabus for submitting your code.

12. When working on any public computer, it is important to keep your private data safe. In ECE 273, we work on department iMacs. Before you leave the lab, be sure to log out of any personal browser sessions (e.g. email, Blackboard, SISWeb, iROAR, etc.). It is also important that you **eject your user drive** mounted via cesmount. To do so, right click the drive on your desktop and click "Eject". If your iMac does not have its right click enabled, hold down the "control" key and click the icon simultaneously. On the pop-up menu, click "Eject". Alternatively, if you have your terminal window open, you may run the "cesunmount" command. Upon success, it will display a message confirming your drive has been removed, and you will see it disappear from the Desktop. If you have any trouble ejecting your drive, please ask the instructor for assistance; otherwise, your data could become compromised if left accessible on the machine.

13. **Read over the documentation** at the course web page. Pay particular attention to instructions on using GDB. It will not be covered directly in this course, although it can be useful (and is a recommended tool) in debugging your programs.

The following is the C driver. Do not modify this code. You are not required to add comments to the driver.

```
/* begin C driver */

#include <stdio.h>

int main(int arg, char **argv)
{
     char buffer[256];
     do {
          int i = 0;
          printf ("Enter a string terminated with a newline\n");
          do {
               buffer[i] = getchar();
          } while (buffer[i++] != '\n');
          buffer[i-1] = 0;
          /* asum() is the function implemented in assembly */
          i = asum(buffer);
          if (i) {
               printf ("ascii sum is %d\n", i);
               continue;
          }
     } while(1);
     return 0;
}

/* end C driver */
```

The following is the assembly solution to the `int asum(char *)` function. You are required to add comments (program and function headers only) to this file. However, you are not required to understand the implementation details of this code at this point in the course. For Labs 2 – 6, you will be given an *assembly stub* file instead of the solutions. The assembly stub will require you apply the topics discussed in each lab in order to form a completed solution to the assignment.

```
/* begin assembly code */

.globl asum
.type asum,@function
asum:
     pushl %ebp
     movl %esp, %ebp
     subl $4, %esp
     movl $0, -4(%ebp)
.L2:
     movl 8(%ebp),%eax
     cmpb $0,(%eax)
     jne .L4
     jmp .L3
.L4:
     movl 8(%ebp),%eax
     movsbl (%eax),%edx
     addl %edx, -4(%ebp)
     incl 8(%ebp)
     jmp .L2
.L3:
     movl -4(%ebp), %eax
     jmp .L1
.L1:
     movl %ebp, %esp
     popl %ebp
     ret

/* end assembly */
/* Do not forget the required blank line here! */
```

Lab 2

Simple Assignments and Arithmetic

Student Objectives

- Learn what registers are, why they are important, and how to use them
- Learn how to declare and initialize global variables
- Discover direct and immediate addressing and how they are used in assembly programming
- Learn how to perform basic operations: move, add, subtract, multiply, and divide
- Learn how to write expression evaluations in assembly from given C code

2.1 – Background

In this lab we will begin to explore the details of assembly language by looking at simple expression evaluation. We will provide you with a C program that calls assembly language routines that you will write. You need not worry about how data is passed between the C and assembly code – we have taken care of that, but you will later learn how to implement such code in assembly. The assignment is straightforward – implement simple arithmetic operations in assembly. More complex programs involving pointers, arrays, data structures, and function calls and returns will be discussed as we progress through the later labs.

To get started, computer programs are composed of two basic elements: (1) memory for storing data, such as variables, and (2) instructions (i.e. the code or "program" itself) for manipulating the data. Assembly language programs have these same features, plus one more – **registers**. Like memory variables, you can store values into registers and use them in computations. These registers are located onboard the CPU itself. Note that the microprocessor and the memory (RAM) are two different entities within the computer. *Registers* are fast data storage units used for temporary variables, whereas *memory* variables (or simply "variables") exist in the computer's memory, which is both more complicated and time-consuming to access. Because they are fast and easier to work with, it is best to use registers as much as possible; however, there are a limited number of them. Despite this limitation, we cannot use only memory variables either. The Intel 80386 architecture places constraints on us as programmers: *we can use up to one memory variable in a single assembly computation*. Furthermore, there is a difference in the instructions provided in assembly language: *we may only perform one computation per instruction or statement*. For example, in C we can say:

```
int a, b, c, d, e;
a = ((b + c) - (d + e)) - 10;
```

Code 2.1.1

The expression in **Code 2.1.1** performs four computations in one statement using four variables (a, b, c, d, and e) and a constant (the number 10). In assembly language we cannot perform such a complex statement. In x86 assembly language, each instruction can perform only one computation at a time and may reference up to one memory variable per computation. At least one of the required data (i.e. arguments to the instruction) must be in a register. To start, four **general purpose registers** provided in the 80386 are **A**, **B**, **C**, and **D**. Thus, the previous example would look like this:

```
.comm a, 4
.comm b, 4
.comm c, 4
.comm d, 4
.comm e, 4
.text
movl b, %eax        # move variable b into register A
addl c, %eax        # add variable c to register A
movl d, %ebx        # move variable d into register B
addl e, %ebx        # add variable e to register B
subl %ebx, %eax     # subtract register B from register A
subl $10, %eax      # subtract 10 from register A
movl %eax, a        # move register A to variable a
```

Code 2.1.2

Note the comments in **Code 2.1.2** above. In assembly, a hash or pound symbol ("`#`") is interpreted as the start of an *inline comment*. Unlike C, double-slashes ("`//`") and block comments ("`/* … */`") cannot be on the same line as assembly code; however, they can be used on lines without assembly code – the program and function header comments, for instance. Placing these types of comments on the same line as assembly code will generate a compile-time error. (To avoid this, use the pound symbol like the example above.)

2.2 – Data Storage and Variables

Let's break this down piece by piece. First of all, in order to declare a variable, we use a statement that will **define a storage location** and assign a name or symbol to that location (or address). Actually, this isn't an instruction at all but an *assembler directive*. These are commands to the assembler program invoked by GCC to perform some action – in this case, reserve memory for a variable. There are several of these directives that can be used to reserve memory. Which one we use, depends on what size block of memory we want to allocate (similar to the data types `char, short, int, and long` in C). In assembly, there are directives used to allocate space for *uninitialized variables*, and directives used in order to reserve memory and *initialize variables*. The most common directive used in this course is `.comm,` which creates a symbol (or *variable* as it is sometimes called) with the name given as the first argument and reserves the number of bytes listed as the second argument. This variable name is actually a placeholder for the address in memory where the space is allocated. At assemble time, all

variable names are replaced by their respective memory addresses. Note that *there is no type information* associated with the memory or the symbol.

Alternatively we could have chosen to initialize the allocated space to some value. In C we could have said:

```
int a;              /* uninitialized */
int b = 10;         /* decimal */
int c = 0x20;       /* hexadecimal */
int d = 'a';        /* ascii */
int e = 040;        /* octal */
int f = 024;        /* C does not have a binary type */
                    /* this is octal */
```

Code 2.2.1

which, in assembly language would be:

```
.comm a, 4          # declare variable ‘a’ as 4 bytes (4B)
b: .int 10          # declare var ‘b’; init to ‘10’
c: .int 0x20        # declare var ‘c’; init to ‘0x20’
d: .int 'a'         # declare var ‘d’; init to ‘a’
e: .int 040         # declare var ‘e’; init to octal ‘040’
f: .int 0b000010100 # declare variable ‘f’
                    # initialize to binary ‘0b000010100’
```

Code 2.2.2

In **Code 2.2.2**, note the syntax for expressing values in different number bases, including the **octal** and **binary syntax**, the latter of which does not exist in C. The symbol created is defined by the **label** to the left of the colon on each line. (We will discuss labels in greater detail in Lab 3.) The value it is initialized to is located to the right of the directive `.int`. Other directives include `.byte`, `.hword`, `.word`, `.quad`, and `.octa` to initialize 1, 2, 4, 8, and 16-byte integers, respectively. Likewise, for floating point numbers, `.float`, `.single`, and `.double` are directives to initialize 4, 4, and 8-byte floating point numbers, respectively. (Note `.float` and `.single` *both* initialize 4-byte floating point numbers.)

2.3 – Moving Data, Addition, Subtraction, and Constants

Before we begin, in x86 assembly, there are two popular syntaxes used – **Intel syntax** and **AT&T syntax**. Although we are writing code for an Intel x86-based processor, *we will use AT&T syntax*. Why? Well, GNU GCC works natively with AT&T syntax. In order for us to compile our programs with GCC, we must use this syntax. There are no pros or cons to one or the other – they are simply different ways of “doing” the same thing. Please note that both syntaxes are directly mapped to the Intel x86 machine language – there are no compute differences at runtime. Now, let’s get started:

Consider the evaluation of the statement `a = ((b + c) - (d + e)) - 10;`, **Code 2.1.1** from **Section 2.1 – Background**. Notice that assembly language does not use the standard mathematical symbols for addition, subtraction, multiplication, division, and so on, like high level languages do. Instead, each operation has its own instruction `addl`, `mull`, `subl`, and `divl`. We will explain how to do each of these during this lab.

In addition to arithmetic operations, in assembly, there is a new operation not available in C – the `movl` or **move** instruction. The majority of assembly language programs have a lot of `movl` instructions, so let's begin our discussion of arithmetic instructions by first talking about the move instruction. As you may have guessed, the move instruction simply moves data from one place to another, and thus the instruction:

```
movl src, dst        # move src to dst
```

Code 2.3.1

is equivalent to the simple C assignment statement:

```
dst = src;
```

Code 2.3.2

The main limitation to all assembly operations, however, is that at least one of `dst` and `src` *must be in a register*. In other words, there can be no more than one memory variable in a `movl` instruction, but they can both be registers if desired.

Notice the example in **Code 2.3.1** and **Code 2.3.2** above contains `dst` and `src` as parameters to `movl`. As you might have imagined, they stand for *source* and *destination*, respectively. As such, to load a memory variable into the A register, we would write:

```
movl variable, %eax      # move the source (variable)
                         # to the destination (the A
                         # register)
```

Code 2.3.3

And, to load the contents of the A register into a variable, we would type:

```
movl %eax, variable      # move the source (the A register)
                         # to the destination (variable)
```

Code 2.3.4

As previously stated, we can also move a register into a register, but *we cannot move a memory variable into another memory variable*. To do this, we must first move one of the memory

variables into a register. Then, we can move that register into the other memory variable. *Any assembly instruction can access main memory no more than once during its execution.*

Notice in **Code 2.3.3** and **Code 2.3.4** the parameter `%eax` used for the A register. This seemingly cryptic syntax specifies that we want to use all 4 bytes of the A register. We will discuss this in more detail in **Section 2.4 – Data Sizes, Register References, and Opcode Suffixes**.

Now that we've masted the move instruction, let's move on to addition and subtraction in assembly language.

In C and other high-level languages, it is fairly common to write code that performs the addition of more than one source and a different destination, all on a single line, as shown in **Code 2.3.5** below:

```
dst = src1 + src2;
```

Code 2.3.5

However, it is not possible to perform such a complex addition in assembly language. What we must do instead is break this addition up into many smaller addition operations. To facilitate this, addition in assembly language works by adding one argument to another, as shown in **Code 2.3.6**:

```
dst = dst + src;          # or equivalently:
dst += src;
```

Code 2.3.6

In assembly language, to perform the simple **addition** in **Code 2.3.6**, we would write:

```
addl src, dst  # add src to dst and store result in dst
```

Code 2.3.7

But remember, as was true for the `movl` instruction, for addition, with respect to **Code 2.3.7**, either `dst`, or `src`, or both *must be a register*. In **Code 2.3.8** below, in order to add two variables, we must first move one to a register, then perform the addition.

```
int a, b;
a += b;
```

Code 2.3.8

The equivalent in assembly is:

```
.comm a, 4      # reserve 4 bytes of space for 'a'
.comm b, 4      # reserve 4 bytes of space for 'b'
movl b, %eax    # first copy variable b to a the A register
addl %eax, a    # add the A register (var b) to variable a
                # this is a = a + b <--> dst = dst + src
```

Code 2.3.9

So, continuing our initial example in **Code 2.3.5**, if we want to add one variable to another and store the result in a different variable, we must first move one into a register, perform the addition, and then copy the result to the desired destination. For instance:

```
int dst, src1, src2;
dst = src1 + src2;
```

Code 2.3.10

is written in assembly language as:

```
.comm dst, 4    # reserve 4B of space for 'dst',
.comm src1, 4   # 'src1',
.comm src2, 4   # and 'src2'
movl src1, %eax# copy variable src1 to register A
addl src2, %eax# add src1 to src2; store the result in A
movl %eax, dst # copy the result to variable dst
```

Code 2.3.11

See, it's that easy – we just need to get accustomed to thinking in smaller steps.

Now, **subtraction** in assembly works just like addition. So, the following operation in C:

```
int a, b, c;
a = b - c;
```

Code 2.3.12

is written in assembly language as:

```
.comm a, 4      # reserve 4 bytes of space for each variable
.comm b, 4
.comm c, 4
movl b, %eax    # copy variable b to register A
subl c, %eax    # subtract c from b (in register A) and
                # store the result in register A
movl %eax, a    # move the result of b - c to variable a
```

Code 2.3.13

Notice in **Code 2.3.12** and **Code 2.3.13** that subtraction (just like addition) takes two arguments where the first is the source and the second is the destination. For both addition and subtraction, it is very important to note the "add to" and "subtract from" functions implemented by `addl` and `subl`, respectively. The destination argument is not simply the destination; the data present in the destination argument will first be used as part of the computation (i.e `dst +/- src`), then it will be overwritten with the result (i.e. `dst = dst +/- src`). *As such, if the original data in the destination argument is important, be sure to `movl` it somewhere else (i.e. copy it) so that it is not lost after the computation.*

Lastly, just as we can specify a **constant** in C to use in a computation, we can specify a constant in assembly language. Constants in assembly are preceded by the `$` symbol:

```
int a;
a = a + 2;
```

Code 2.3.14

is equivalently

```
.comm a, 4
addl $2, a      # $2 is the constant 2
```

Code 2.3.15

Based on what we have discussed thus far, you should be able to go back to the very first example of `a = ((b + c) - (d + e)) - 10;`, **Code 2.1.1** in **Section 2.1 – Background**, and understand how this complex addition and subtraction operation is implemented in assembly. Give it a try!

2.4 – Data Sizes, Register References, and Opcode Suffixes

Now it is time to expand our horizons a little more. The first thing to consider is that the 80386 can operate on several different sizes of data. The primary data sizes are 8, 16, and 32 bits. In support of this, the A, B, C, and D general purpose registers can be **referenced** as 8-bit registers, 16-bit registers or 32-bit registers. To do this, each of the four general-purpose registers we have seen (A, B, C, and D) can be referenced in the following ways:

8-bit: `%ah, %al; %bh, %bl; %ch, %cl; %dh, %dl`

Code 2.4.1

These eight registers in **Code 2.4.1** above reference the A, B, C, and D registers 8 bits at a time. The `h` specifies the high-order 8 bits of the low-order 16 bits of the total 32 bits, while the `l` specifies the low-order 8 bits of the low-order 16 bits of the total 32 bits. That is quite a mouthful and is best explained with a picture.

Data Size	Bits of the "A" General-Purpose Register			
	31, 30, …, 25, 24	***23, 22, …, 17, 16***	***15, 14, …, 9, 8***	***7, 6, 5, …, 2, 1, 0***
32-bit long-word	**%eax**			
16-bit word			**%ax**	
8-bit byte			**%ah**	**%al**

Table 2.4.1

Table 2.4.1 represents a general-purpose register A, B, C, or D. (A is used in the table as an example, but the principle applies to all.) The register has a total of 32 bits (31 down to 0), where it can store data in binary. As depicted in the table, the most significant bit is on the left – bit 31, and the least significant bit is on the right – bit 0. If we want to access or store 8-bit data in the register, we can use either bits 15 through 8 or bits 7 through 0. The former can be accessed by referring to the register as `%ah`, `%bh`, `%ch`, or `%dh`, depending on which register we want to use. Referring to the register as `%al`, `%bl`, `%cl`, or `%dl` can access the latter. As mentioned previously, the `h` stands for the *high-order* bits (15 to 8) of bits 15 to 0; the `l` stands for the *low-order* bits (7 to 0) of bits 15 to 0. So theoretically, if we wanted, we could store two 8-bit values in a single register by storing one using `%ah` and the other using `%al`. As the table illustrates, they would be in two different physical locations within the same register.

What about 16-bit data types? They can be referenced the following ways:

16-bit: `%ax; %bx; %cx; %dx`

Code 2.4.2

These four registers in **Code 2.4.2** represent the least-significant 16 bits of the total 32 available bits in the general-purpose registers. Note that, as shown in **Table 2.4.1**, these are the *exact* same 16 bits used for referencing 8-bit data sizes; only they are being referenced as all 16 at once, as

opposed to 15 to 8 and 7 to 0 separately. The `x` in the syntax stands for *extended*, meaning it extends the number of bits referenced from 8 bits to 16 bits.

Last, but certainly not least is the 32-bit data size. It is the size most frequently used in this course and in most assembly programs. It is also the register size used in the previous move, addition, and subtraction examples in the prior sections of this lab, so its syntax should look familiar. It can be referenced the following ways:

32-bit: `%eax; %ebx; %ecx; %edx`

Code 2.4.3

The syntax in **Code 2.4.3** above represents all 32 bits of the register for the A, B, C, and D general-purpose registers, respectively. The `e` in the register name stands for *extended* and the `x` stands for *extended* as well. Originally, when the first Intel 80XXX processor was developed, there were only 8-bit registers. Therefore, as the family of processors matured and technology increased in sophistication, the new 16-bit processors *eXtended* the 8-bit ones, and when the time came around, new 32-bit processors *Extended* the older 16-bit ones. As seen in **Table 2.4.1**, working with 32-bit data leverages all available bits in the register. However, like explained previously, these same 32 bits can be accessed 16 or 8 at a time, depending on the syntax used to reference the register.

Aside: Although it is not a part of this course, 64-bit system architectures and operating systems – x86_64 – are becoming more prevalent. Their registers work in the same fashion, but to access all 64 bits of information, one must reference them as `%rax`, for example. 32, 16, and 8 bit accesses work the same as described above.

Now, when we refer to a register in an instruction, *the size of the register must match the size of the opcode*. The opcode is merely a fancy name for the bits that characterize the instruction or *operation* being performed. Assembly instructions, in addition to the data they operate on, are also represented in the computer in binary coding – this is called the **opcode**. Note these instructions are *specific to the size* of data we want to work with. In 80386 assembly language, instructions can be used with 1, 2 or 4-byte data, specified with an **opcode suffix** of either `b` (for "byte"), `w` (for "word"), or `l` (for "long-word"), respectively.

Recall that all of the assembly instructions in the earlier examples in this lab have used *4-byte long-words*; thus, all of the opcodes have had an `l` suffix as in `addl, movl, subl`. This is the most common data size we will work with. But, be aware that there are also instructions for other data sizes, such as `addb, divb, movb` for 1-byte words. We need to be careful and match the opcode suffix with the correct register reference. Thus, to match the opcode with the parameters, the instruction:

```
addb $2, %al
```

Code 2.4.4

is an 8-bit operation, where the `b` and `%al` correspond to 8-bit instructions. On the other hand:

```
addw $2, %ax
```

Code 2.4.5

is a 16-bit operation, where the `w` and the `%ax` correspond to 16-bit instruction syntax. **Table 2.4.2** summarizes opcode suffixes:

Data Size	Size in Bytes	Opcode Suffix	Example Use
byte	1	`b`	`addb`
word	2	`w`	`addw`
long-word	4	`l`	`addl`

Table 2.4.2

Remember, the instruction and the data size need to match up in order to compile without errors. For example, if we want to add 32-bit data sizes, the opcode suffix needs to be `l` making `addl`, and the parameters to the instruction `addl` need to be variables declared as 4 bytes or registers using the 32-bit syntax – `%eax` or `%ebx`, for example.

2.5 – Multiplication and Division

In assembly, the multiplication and division instructions are somewhat more complex than the other operations we have discussed. Let's start with **multiplication**. First, there are two versions: multiplication for integers, `imull`, and multiplication of unsigned numbers, `mull`. We will discuss `mull` in detail; however, keep in mind there is an alternative for integers only.

The `mull` instruction has a single operand (which can be a variable or a register). The value of this operand is multiplied *by the A register*, and the result is *placed back in the A register*, and potentially the D register. Yes, that's right, the `mull` instruction can have potentially two destination registers. Also note that one parameter to the multiplication instruction is assumed to be in the A register. So, this means that if we want to multiply the contents of register B and register C we *cannot* do:

```
mull %ebx, %ecx     # this is incorrect and will not
                    # compile
```

Code 2.5.1

Instead of doing **Code 2.5.1**, we must first move the contents of one of the operands to register A and then multiply by register C:

```
movl %ebx, %eax
mull %ecx            # %eax = %eax * %ecx
```

Code 2.5.2

Note in **Code 2.5.2** that the result of register B * register C is placed in the A register, overwriting the value of one of the operands to the multiplication.

With regard to data sizes, multiplication and division are different from other instructions, since they can operate on more than one data size at a time. For example, when we multiply two 8-bit numbers, we can potentially get a 16-bit result. When we multiply two 16-bit numbers we can potentially get a 32-bit result, and when we multiple two 32-bit numbers, we can potentially get a 64-bit result. Thus, in each case, our result can require more space than the operands. To prove this to yourself, try multiplying the maximum integer we can represent in 8 bits by itself. In other words, what is the highest multiplication result we could achieve with two 8-bit values? How many bits does the result require?

For multiplication, in the 8-bit case, the result is placed in `%ah` concatenated with `%al` (denoted `%ah:%al`). This is also known as `%ax`. Convince yourself of this by referring back to **Table 2.4.1**.

For 16 bits, the result is placed in `%dx:%ax`. Note the result is going into two different registers – A and D. This may seem strange, but it was done this way to be compatible with pre-32-bit hardware.

Finally, in the 32-bit case, the result is placed in `%edx:%eax`. The higher-order bits of the result are in `%edx`, while the lower-order bits are placed in `%eax`. So, if we multiply two numbers that result in a number that can be represented with 32 bits (a number less than or equal to 2^{32}-1), then the entire result will be in the A register. If this is not the case, and a very large answer is generated, then the result would *overflow* into the D register. This principle is true for 8-bit and 16-bit operations as well, with their respective destination registers. *The point is: when working with large numbers, do not forget to retrieve your data from both result registers.* In this course, we will work with relatively small numbers, so this will not be of great concern, but it is still very important to remember; otherwise, data can be lost, resulting in inaccurate results.

So, what does this mean with regard to data sizes? This implies that if we would like to implement **Code 2.5.2** where we multiply a 32-bit value by another 32-bit value, the result will potentially be 64 bits and occupy *both* the A register and the D register, `%edx:%eax`. If the result is small enough, it will only be in the A register, but if it is large, it could occupy both the A and the D registers. **Table 2.5.1** illustrates multiplication for 8, 16, and 32-bit values:

Assembly Instruction	Operation Performed
`mulb` X_{8bit}	`%ax = %al *` X_{8bit}
`mulw` X_{16bit}	`%dx:%ax = %ax *` X_{16bit}
`mull` X_{32bit}	`%edx:%eax = %eax *` X_{32bit}

Table 2.5.1

Notice in **Table 2.5.1** that for the 32-bit multiplication, the result is placed into `%edx:%eax` as explained above. As the single operand to the `mul` instruction, we supply `X`, which is either a 32, 16 or 8-bit register or a 32, 16 or 8-bit variable in main memory.

Now, that we have discussed multiplication, we can move on to **division**. Like multiplication, division also has its peculiarities. The format of the divide instruction is similar:

```
divl <divisor>
```

Code 2.5.3

Like the `mull` instruction, `divl` assumes the dividend and the destinations of the quotient and remainder based on the opcode suffix. **Table 2.5.2** illustrates division for 8, 16, and 32-bit values:

Assembly Instruction	Operation Performed
`divb` X_{8bit}	`%al = %ax /` X_{8bit}`, %ah = remainder`
`divw` X_{16bit}	`%ax = %dx:%ax /` X_{16bit}`, %dx = remainder`
`divl` X_{32bit}	`%eax = %edx:%eax /` X_{32bit}`, %edx = remainder`

Table 2.5.2

For example, as shown in **Table 2.5.2** above, in 32-bit division, the divisor – X_{32bit} – is supplied as the parameter to `divl`; the dividend – `%edx:%eax` – is assumed to be in the A register and D register combined (for a possible 64-bit value); after the calculation, the quotient of the operation is placed in the A register – `%eax`; and finally, the remainder is placed in the D register – `%edx`. This pattern is the same for 16-bit division, but note the differences for 8-bit division – like multiplication, it *does not* use the D register and the A register combined – ~~`%dl:%al`~~ (this is incorrect), but instead it requires the A register alone – `%ah:%al` or `%ax`.

Note that just like multiplication, division allows for mixed data sizes. 8-bit division takes a 16-bit dividend, 16-bit division takes a 32-bit dividend, and 32-bit division takes a 64-bit dividend. You might have been wondering why we can generate a 64-bit result in multiplication if the 32-bit hardware and ISA do not directly support it. With 64-bit-compatible division, a 64-bit result from a multiplication (along with clever programming) can be used in later calculations, despite the ISA being 32-bit.

Please note that when we divide by a 32 or 16-bit number, we must make sure `%dx` or `%edx` contains the correct value, which is *usually zero*. This is very important, since if the division we need to perform is just `%eax / X`$_{32bit}$, the `divl` instruction will not know we do not want to include data in the D register – `%edx`. If there is irrelevant data there, it will be incorporated into the division operation, thus making our intended dividend much larger or maybe even negative depending on where the junk data bits are in the D register. In order to avoid this, we must always *clear out* the D register by performing the following operation:

```
movl $0, %edx       # this ensures the D register has no
                    # data in it before a 32-bit/32-bit
                    # division
```

Code 2.5.4

This same principle of clearing the dividend (shown in **Code 2.5.4** above) applies to 16, and 8-bit division, the upper (most significant) bits of the dividend must be cleared unless there is valid data in them for the division being performed. *By the end of this course, many program bugs will take a while to fix, and this is the problem 25% of the time.*

Let's work an example using 32-bit division in assembly language. Let's assume we have the following C code fragment that we wish to implement in assembly language:

```
int dividend, divisor, quotient, remainder;
...
quotient = dividend / divisor;
remainder = dividend % divisor;     # "%" in C is the
                                    # modulus operator.
                                    # It returns the
                                    # remainder of the
                                    # division instead
                                    # of the quotient.
```

Code 2.5.5

To convert **Code 2.5.5** to assembly language, recall that the `divl` instruction assumes the dividend is in the `%edx` and `%eax` registers, while it takes the divisor as its single argument. Also recall that for 32-bit division, the quotient is stored in `%eax` and the remainder is stored in `%edx`. Thus, the following assembly code in **Code 2.5.6** implements the C in **Code 2.5.5**:

```
     .comm dividend, 4
     .comm divisor, 4
     .comm quotient, 4
     .comm remainder, 4
     ...
     movl $0, %edx            # clear the D register
     movl dividend, %eax      # initialize the dividend
     divl divisor             # perform the division
     movl %eax, quotient      # grab the quotient
     movl %edx, remainder     # grab the remainder
```

Code 2.5.6

2.6 – Assignment

This is the specification of what the assembly functions need to perform. Do not copy or type this code. Use it as a reference when writing the assembly.

```
/* begin assignment specification */

int digit1;
int digit2;
int digit3;
int diff;
int sum;
int product;
int remainder;

dodiff() {
     diff = (digit1 * digit1) + (digit2 * digit2) - (digit3 *
     digit3);
}

dosumprod() {
     sum = digit1 + digit2 + digit3;
     product = digit1 * digit2 * digit3;
}

doremainder() {
     remainder = product % sum;
}

/* end assignment specification*/
```

The following is the C driver. Do not modify this code. You are not required to add comments to the driver.

```
/* begin C driver */

#include <stdio.h>
extern int digit1;
extern int digit2;
extern int digit3;
extern int diff;
extern int sum;
extern int product;
extern int remainder;

int main(int argc, char **argv)
{
     for (digit1 = 0; digit1 < 10; digit1++) {
          for (digit2 = digit1; digit2 < 10; digit2++) {
               for (digit3 = digit2; digit3 < 10; digit3++) {
                    dodiff();
                    if (diff == 0)
                         printf("%d%d%d PT\n",
                              digit1,digit2,digit3);
                    dosumprod();
                    if (sum && product) {
                         doremainder();
                         if (remainder == 0)
                              printf("%d%d%d ED\n",
                                   digit1,digit2,digit3);
                    }
               }
          }
     }
     return 0;
}
/* end C driver */
```

The following is the assembly stub to the driver. You are required to fully comment and write the assembly code to model the specification code. Insert your code where you see `/* put code here */` **and** `/* declare variables here */`**. Do not modify any other code in the file. Note the last line of the file must be a blank line to compile without warnings.**

```
/* begin assembly stub */

.globl dodiff
.type dodiff, @function
dodiff:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

.globl dosumprod
.type dosumprod, @function
dosumprod:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

.globl doremainder
.type doremainder, @function
doremainder:
```

```
/* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

/* declare variables here */

/* end assembly stub */
/* Do not forget the required blank line here! */
```

The following is what the correct program output should look like.

```
000 PT
011 PT
022 PT
033 PT
044 PT
055 PT
066 PT
077 PT
088 PT
099 PT
123 ED
138 ED
145 ED
159 ED
167 ED
189 ED
224 ED
235 ED
246 ED
257 ED
268 ED
279 ED
333 ED
345 PT
345 ED
347 ED
357 ED
369 ED
448 ED
456 ED
459 ED
466 ED
578 ED
579 ED
666 ED
678 ED
789 ED
999 ED
```

Lab 3

Control Statements

Student Objectives

- Learn about flags and how they are set/reset in assembly operations
- Learn what labels are in assembly and how they are used
- Discover how to jump to labeled parts of code based on flags
- Understand the compare instruction and how it differs from C comparisons
- Learn and apply conditional jumps to the result of a compare operation
- Learn how to decompose complex C conditional statements into assembly
- Discover how to unconditionally jump and learn how to jump to different segments

3.1 – Jumps, Labels, and Flags

In this lab we continue our tour of assembly language by adding **control statements**. Control statements are coding structures that direct the flow of a program (the order in which the code is executed). In assembly language there are no `if ... then ... else`, `while`, or `for` statements. Instead, there are two basic primitives: jumps and conditional jumps (actually, there *are* some looping instructions, but we'll cover those after we understand the basics). The **unconditional `jmp`** instruction is, for all practical purposes, a `GOTO` statement. You might recall from higher-level programming classes that the `GOTO` statement is something which we should avoid. While this is largely true for high-level language applications – as it can result in what some call "spaghetti code" – jump statements remain a primary tool of assembly language.

With jumps, where are we "jumping" to, you might ask? The answer is in the **label**. Labels can be likened to mile markers in your program. These markers are named starting with a letter and can be up to 30 characters in length. A label denotes a specific location in memory where a part of your executable code resides. In practice, we can think of labels as marking the line numbers of our code. For example:

```
Line    Instruction
142     myLabel:
143     movl a, %eax
144     mull %eax
…       …
…       …
180     jmp myLabel
```

Code 3.1.1

In **Code 3.1.1**, there is a label called `myLabel` at line 142 of some assembly code. When the code is executed and line 180 is reached, the `jmp` instruction will force execution of the program to move to the location of `jmp`'s argument – `myLabel` – which is at line 142. Recall, assembly code, like C, executes from top to bottom. We can now extend this definition to be: assembly code executes from top to bottom *unless a jump occurs*, forcing execution to move elsewhere. So, when the jump occurs on line 180, execution will move to `myLabel`, thus causing lines 143, 144, and so on… until another jump is reached, to be executed. When a label is reached during code execution, it is simply bypassed to the next instruction – labels mean nothing to the executable code except places to which we can jump. We can use as many as we desire. In fact, as we will demonstrate in the Lab 3 examples, labels can help our conceptual understanding of assembly control statements, even though our code does not jump to them. (See the `do…while` loop example later on.) On a final note, notice the syntax for writing labels – there is a colon (":") after the label – `myLabel:`, in the previous example. This colon defines the location of the label. When a label is used in a jump, notice the colon is omitted, since we are not defining a label inside the jump; instead, we want to go to the label, which has been defined elsewhere.

While the `jmp` instruction has its merit, the more interesting control primitive is the **conditional jump**. A conditional jump works like the following pseudo-code:

```
if (condition) then goto label
```

Code 3.1.2

The trick in **Code 3.1.2** is in the `condition`. There are very few *pre-defined conditions* you can jump on, and all of them depend on something called flags. A **flag** is a single bit of a special register in the CPU called the *status register or flags register*. Each bit in this register has a special meaning and most of them tell us something about the results of a previous computation. There are three main flags we use: the *zero flag*, the *sign flag*, and the *carry flag*. Certain instructions cause these flags to be set according to the results of the instruction. Which instructions change certain flags varies from one machine to another, but arithmetic instructions, like `add` and `mul`, almost always do. There is also a special instruction called a **compare** instruction (`cmp`) which *sets* the flags without actually changing any of the other registers in the CPU. We'll talk more about that in a moment.

As we said, the flags are set according to the result of certain instructions. As an example, if we execute an `add` instruction and the result is less than zero, then the sign flag is set (indicating a negative result), and the zero flag is cleared. If the result is greater than zero, then both flags will be cleared, and if it is equal to zero, the zero flag is set and the sign flag is cleared. The carry flag indicates if a carry out occurred in the highest order bit position and thus is data dependent. Most machines also have an *overflow flag* which indicates the computation has produced a result that cannot be correctly represented in the current data size. As you may have guessed by now, one of the differences between signed and unsigned computations is the way flags are set.

Conditional jump instructions allow you to test the current state of the various flags. Each combination has its own instruction. For example, `jz label` will jump to label if the zero flag

is set, while `jnz label` will jump if the zero flag is not set. The zero flag signifies a result of zero, "0". These and others are shown in **Table 3.1.1** below:

```
jc label        # jump if carry
jnc label       # jump if not carry
jz label        # jump if zero
jnz label       # jump if not zero
js label        # jump if sign (negative)
jns label       # jump if not sign (positive)
jo label        # jump if overflow
jno label       # jump if not overflow
jpo label       # jump if parity is odd
jpe label       # jump if parity is even
```

Table 3.1.1

While these certainly have merit, usually when we are writing programs, we want to implement control statements like:

```
if (a > b) then ...
```

Code 3.1.3

Statements similar to **Code 3.1.3** are what the `cmp` instruction is for. The **`cmp` instruction** compares two values by subtracting the first operand from the second to set the flags. Unlike the `sub` instruction, the result of the subtraction in the `cmp` instruction is simply discarded – it is *not saved* in a register or a memory variable. **Code 3.1.4** shows the operation of the `cmpl` instruction for long-word data sizes:

```
cmpl src, dst       # (dst - src) __ 0?
                    # How does the result compare to zero?
                    # We can jump based on this result.
```

Code 3.1.4

After executing the `cmpl` instruction, the flags in the status register can be checked in order to determine the relationship between the two values being compared. A set of special conditional jump instructions are provided in **Table 3.1.2** to make this easy to do:

```
je label        # jump if equal
jne label       # jump if not equal
jg label        # jump if greater than
jng label       # jump if not greater than
jl label        # jump if less than
jnl label       # jump if not less than
jge label       # jump if greater or equal
jnge label      # jump if not greater or equal
jle label       # jump if less or equal
jnle label      # jump if not less or equal
```

Table 3.1.2

Thus to implement the following C `if` statement:

```
int a, b;
if (a > b) {
     ... code ...
}
... more code ...
```

Code 3.1.5

with an assembly conditional statement, we would write:

```
     .comm a, 4
     .comm b, 4
     movl a, %eax         # must have at least 1 argument
                          # in a register
     cmpl b, %eax         # is (%eax - b) __ 0? Set flags
     jng done             # if (%eax - b) !> 0, goto done
     ... code ...
done:
     ... more code ...
```

Code 3.1.6

Notice that we execute a jump if *not greater than* in **Code 3.1.6**. If `a` is greater than `b` then we *do not want* to jump, but we *do want* to execute the code block. Note that the `cmp` instruction subtracts `b` from `%eax` and then we check if the result of the subtraction was negative or not. If the result is negative or zero, `b` must be greater than or equal to `a` – the opposite of our `if (a > b)` statement – thus we want to skip `... code ...`. This is done because we want to jump over `... code ...` to `... more code ...` if the result is less than or equal to zero. Let's look at some other typical high level language control statements and how they would be implemented in assembly language.

To start, the following simple C `if...else` statement:

```
int a, b;
if (a > b) {
     ... code block 1 ...
} else {
     ... code block 2 ...
}
... more code ...
```

Code 3.1.7

translates to:

```
     .comm a, 4
     .comm b, 4
     movl  a, %eax
     cmpl  b, %eax
     jng else                  # jump if (a - b) !> 0
     ... code block 1 ...
     jmp more                  # skip else when 'if' is true
else:
     ... code block 2 ...
more:
     ... more code ...
```

Code 3.1.8

While the following C `while` loop:

```
int a, b;
while (a > b) {
     ... code ...
}
... more code ...
```

Code 3.1.9

translates to:

```
        .comm a, 4
        .comm b, 4
while:
        movl a, %eax
        cmpl b, %eax
        jng more                    # jump if (a - b) !> 0
        ... code ...
        jmp while                   # continue to loop
more:
        ... more code ...
```

Code 3.1.10

And the following C `for` loop:

```
int i;
for (i = 0; i < 100; i++) {
        ... code ...
}
... more code ...
```

Code 3.1.11

translates to:

```
        .comm i, 4
        movl $0, i
for:
        cmpl $100, i
        jnl more                    # jump if (i - 100) !< 0
        ... code ...
cont:
        incl i                      # i = i + 1 (see Section 3.4)
        jmp for                     # jump to try and loop again
more:
        ... more code ...
```

Code 3.1.12

And last but not least, the following C `do...while` loop:

```
int a, b;
do {
     ... code ...
} while (a > b);
... more code ...
```

Code 3.1.13

translates to:

```
     .comm a, 4
     .comm b, 4
do:                        # will be executed regardless of
     ... code ...          # the condition evaluation
cont:
     movl a, %eax
     cmpl b, %eax
     jg do                 # jump back to do if (a - b) > 0
more:                      # automatically break if false
     ... more code ...
```

Code 3.1.14

In all loops, it is important that the loop variable be *written to memory* just before the jump back to the top, so that when it is checked by the compare statement the correct value is used. For example:

```
int a, b;
while (a > b) {
     ... code ...
}
... more code ...
```

Code 3.1.15

translates to:

```
        .comm a, 4
        .comm b, 4
    while:
        movl  a, %eax
        cmpl  b, %eax
        jng  more              # jump to more if (a - b) !> 0
        ... code ...
        movl  %eax, a          # save the present value of a from
                               # %eax to a (in main memory)
        jmp while              # keep looping
    more:
        ... more code ...
```

Code 3.1.16

Notice in examples **Code 3.1.8**, **Code 3.1.10**, **Code 3.1.12**, and **Code 3.1.16** that we can implement the C statement `break` as `jmp <label>`, and the C statement `continue` as `jmp <label>`. In these examples, we have used labels such as `do`, `more`, and `for`. We have done this in order to make it easier to see the structure of these control statements (and how they relate to their C counterparts) in **Code 3.1.7**, **Code 3.1.9**, **Code 3.1.11**, and **Code 3.1.15**, respectively. In actuality, we as programmers can use whatever label names we desire.

3.2 – Complex and Compound Conditional Expressions

We must also consider different kinds of conditional expressions. In our examples we have only shown the comparison of two variables or one variable to a constant. In reality, we can and sometimes need to compare combinations of variables in a single conditional expression. We can *simplify* these more complex cases to the familiar comparison of two simple expressions by (1) evaluating/simplifying the complex expressions, placing the results in registers, and then comparing the registers, and by (2) breaking up complex statements into multiple statements/comparisons. This is best understood with some examples.

The following is a **complex** `if` statement. The left hand side (LHS) and right hand side (RHS) need to be simplified before the check for equality can be done. As shown in the assembly translation, once this simplification is performed, the operation becomes trivial and can be likened to our previous examples in **Section 3.1 – Jumps, Labels, and Flags**.

```
int a, b, c, d;
if ((a + b) == (c - d)){
     ... code block 1 ...
}
... more code ...
```

Code 3.2.1

The above C complex if statement in **Code 3.2.1** translates to:

```
.comm a, 4
.comm b, 4
.comm c, 4
.comm d, 4
movl a, %eax
addl b, %eax        # perform the addition, save to A
                    # register
movl c, %ebx
subl d, %ebx        # perform the subtraction, save to B
                    # register
cmpl %ebx, %eax     # is ((a+b)-(c-d)) __ 0?
jne more            # (jne    -->      !=  )
... code ...
more:
     ... more code ...
```

Code 3.2.2

As shown in **Code 3.2.2**, in order to evaluate a complex conditional statement, we simply need to reduce the LHS and RHS of the expression in order to perform the comparison and conditionally jump.

Another situation we can encounter is a **compound conditional**, which uses `AND` and/or `OR` operators (`&&` and `||`). In the case of compound conditionals, it depends on the nature of the high level language we are implementing into assembly on how to proceed with the translation. Some languages require that the entire expression be evaluated, regardless of the outcome of prior evaluations. If this is the case, we must first reduce each and every comparison to a one or a zero (true or false). Then, these reductions can be combined with the logical operators `and`, `or`, and `not` until a single result is obtained. This final result can then be compared to zero (using `cmpl` and its corresponding *jump* instruction).

Thankfully, in C (and many other languages), compound conditionals use what are called **short circuit operators**, which means that the conditional expression is evaluated *left to right*, and as soon as the outcome is *known*, the evaluation *stops*. This will occur when a true (i.e. "1") is found to the *left* of an `||` or a false (i.e. "0") is found to the *left* of an `&&`. In either of these

cases, the entire conditional, because it is evaluated one simple expression at a time, can be thought of as a set of *nested* `if` statements. These can be implemented in the same manner we have already shown in **Section 3.1 – Jumps, Labels, and Flags**. For example, this compound `AND` statement in **Code 3.2.3**:

```
int a, b, c, d;
if (a > b && c < d) {
     ... code ...
}
... more code ...
```

Code 3.2.3

can be simplified to:

```
int a, b, c, d;
if (a > b) {         # the nexted if statements form
     if (c < d) {    # a logical AND
          ... code ...
     }
}
... more code ...
```

Code 3.2.4

Notice how the compound `AND` is broken into (and is equivalent to) two nested `if` statements in **Code 3.2.4**. This can be readily translated into assembly language by applying techniques previously disucussed for non-compound conditional statements. The only difference is we check two conditions instead of one:

```
        .comm a, 4
        .comm b, 4
        .comm c, 4
        .comm d, 4
        movl a, %eax
        cmpl b, %eax    # (a - b) <= 0 ?
        jle more        # break out of the if statement if true
                        # if false, proceed to 2nd conditional
        movl c, %eax
        cmpl d, %eax    # (c - d) >= 0 ?
        jge more        # break out of the if statement if true
code:                   # if both jumps fail, execute "code"
                        # i.e. both sides of the AND are true
        ... code ...
more:                       # execution automatically flows
        ... more code ...   # into "more" if "code" was
                            # executed with conditional jump
```

Code 3.2.5

An `OR` is a bit trickier:

```
int a, b, c, d;
if (a > b || c < d) {
        ... code ...
}
... more code ...
```

Code 3.2.6

and can be simplified to:

```
int a, b, c, d;
if (a <= b) {               # if "opposite" is true
        if (c >= d) {       # and if "opposite" is also true
                goto more;  # then neither "original"
        }                   # condition is true, so skip "if"
}
code:                       # labels can also be used in C
        ... code ...
more:                       # goto will jump to label "more"
        ... more code ...
```

Code 3.2.7

Notice, in **Code 3.2.7** we dug into our repertoire of ECE knowledge and pulled out *De Morgan's Laws.* We took our initial compound `OR` conditional, negated each side's condition, and changed the `OR` to an `AND`. (Remember, an `AND` within an `if` statement is equivalent to a series of nested if statements.) Another thing to note is we are performing the opposite operation inside the nested if statements – we are skipping `... code ...` and going to the `more` label. Essentially, what we are checking for, is if both original conditions (from **Code 3.2.6**: `a > b || c < d`) have *failed* (i.e. evaluated to false); if so, then neither one of them were true, so execution should now skip to `more`, thus breaking the `if` statement. Said another way, the simplified code checks the opposite conditions, so we need to perform the opposite operation if the opposite conditions succeed. Take a moment to convince yourself that what we have done so far will work. We can now translate our nested `if` statements and `goto` into assembly language:

```
      .comm a, 4
      .comm b, 4
      .comm c, 4
      .comm d, 4
      movl a, %eax
      cmpl b, %eax
      jg code              # 1st is true, so execute code
      movl c, %eax         # 1st is false if we get here
      cmpl d, %eax
      jl code              # 2nd is true, so execute code
      jmp more             # both are false if we get here
code:
      ... code ...
more:
      ... more code ...
```

Code 3.2.8

Notice in this implementation , **Code 3.2.8**, we jump to `code` if either of the two original conditions are true – this implements the desired `OR` operation. However, also note the inclusion of an unconditional jump to `more`. This is the `goto` statement we included in our C simplification. It will only get executed if the previous conditional jumps fail – if neither `a > b` nor `c < d`. In this case, where neither of the conditions evaluate to true, we want to skip to `more`. This necessitates the inclusion of the unconditional jump in this implementation. If it were to be omitted, notice that `code` would get executed regardless of the results of the conditional jumps, which is an *incorrect implementation*. Let's consider an alternative translation:

```
        .comm a, 4
        .comm b, 4
        .comm c, 4
        .comm d, 4
        movl a, %eax
        cmpl b, %eax
        jg code              # 1st is true, so execute code
        movl c, %eax         # 1st is false if we get here
        cmpl d, %eax
        jge  more            # 2nd is false too, so goto more
code:                        # 1st or 2nd is true if we get to code
        ... code ...
more:
        ... more code ...
```

Code 3.2.9

Notice the implementation in **Code 3.2.9** is more efficient than the previous implementation in **Code 3.2.8**. We replaced the second conditional jump and the unconditional jump with a single conditional jump. The difference is in the condition upon which we perform the jump. Recall from the previous implementation that the *unconditional* jump to `more` will only get executed if `c < d` is not true. Notice the second conditional jump only results in a skip over the single unconditional jump to the `code` label. This means we can eliminate the unconditional jump if we *reverse the conditional jump* – jumping to more if `c < d` is false, meaning jump if `c >= d` to the more label. *Why is this the case?* This is the second (and *final*) condition being evaluated. If any of the previous conditions (`a > b` in this case) were true, we would have jumped already; but since they were false (thus allowing the code to execute down to the last compare and jump), we can assume that if this final check is also false, then none were or will be true – we can skip to `more`. *This assumption can only be done if it is the last condition being evaluated in a compound conditional expression*.

3.3 – if … then … else Conditional Expressions

Perhaps one of the most daunting conditional expressions is the `if ... then ... else` control statement. Writing this in assembly may appear difficult at first glance, but as we will see in this section, the `if ... then ... else` statement is merely a special application of the skills and techniques we have learned thus far. Consider what this conditional expression is providing: it is very similar to the simple `if` statements we have worked with earlier in this lab; the only difference is we will now control or limit access to *multiple* blocks of instructions. In the following example, we will examine an `if ... then ... else` style control statement in detail and walk through how to write its equivalent in assembly.

Consider the following nested `if ... then ... else` control statement in **Code 3.3.1**:

```
int a, b, c, d;
if (a > b) {
     if (c < d) {
          ... code block 1 ...
     }
     else if ((c == d) && (a != 0)) {
          ... code block 2 ...
     }
     else if (c > d) {
          ... code block 3 ...
}
else {
     ... code block 4 ...
}
... more code ...
```

Code 3.3.1

When working with complex and compound nested control structures, the best place to begin is typically at the top. So, let's approach this example one step at a time. We first need four memory variables. Then we want to compare two of them:

```
     .comm a, 4
     .comm b, 4
     .comm c, 4
     .comm d, 4
     movl a, %eax
     cmpl b, %eax   # evaluate first if: is (a-b) <= 0?
     jle  block4    # if is false, so break out to else
     ...
     ...
     ...
block4:
     ... code block 4 ...
more:
     ... more code ...
```

Code 3.3.2

Notice in **Code 3.3.2** that we have not filled in the details of the nested `if` statements yet – we are writing *framework code*, where we will insert the pieces (in place of the `...`'s) one at a time. Continuing this approach, let's venture inside the first `if` and add in `if (c < d)`:

```
        .comm a, 4
        .comm b, 4
        .comm c, 4
        .comm d, 4
        movl a, %eax
        cmpl b, %eax    # evaluate 1st if: is (a-b) <= 0?
        jle  block4     # if is false, so jump out to else
             movl c, %eax   # evaluate 2nd if: is (c-d) < 0?
             cmpl d, %eax   # if is true, so jump to block1
             jl block1
             ...
             ...
             ...
        ...
        ...
        ...
block1:
        ... code block 1 ...
        jmp more        # no more ifs/else to eval. if in block1
block4:
        ... code block 4 ...
more:
        ... more code ...
```

Code 3.3.3

Notice how we are indenting our assembly code to mimic its C specification code. This will be helpful in both writing our code, as well as reading it at a later time.

Also, consider how the `block1` label is placed outside the nested `if` structure in **Code 3.3.3**. If you look closely at **Code 3.3.1**, you will notice that when the `if` statement guarding `... block 1 ...` is true, then when `... block 1 ...` finishes executing, execution should resume at `more`. Thus, we do not need to bother with any other comparisons – we simply need to unconditionally jump to `more`. Now, if we wanted, we could have included the code inside the `block1` label within the nested `if` structure, instead of towards the end. This is a design decision and assembly code can work either way – but be careful with jumps – some implementations will require more jumps than others and are thus more difficult to read and less efficient. In general, the best approach with regard to the design of your program's flow should be the implementation that is most readable.

Now, what should the next step be in our translation? Take a look at the remaining `else if` conditionals in **Code 3.3.1**. They also compare `c` and `d`, where each comparison is mutually exclusive. In other words, only one of `c < d`, `c == d`, and `c > d` can be true given `c` and `d`. As such, it does not matter the order in which we evaluate these three control statements. *Note:*

*this is **not** true in general – many control statements must be evaluated sequentially to preserve the logic and intention of the original code.* When we wrote the assembly comparison for `c < d` in **Code 3.3.2**, we used the `cmpl` instruction to perform the comparison. Recall, the `cmpl` instruction sets flags (e.g. sign, zero, overflow, carry, etc) to describe the result of the comparison. These flags are only set or cleared upon (1) an invocation of `cmpl` or (2) performing any mathematical operation (e.g. `addl`, `subl`, `mull`, and `divl`). So, we can simply insert additional jumps for the other `else if` statements comparing only `c` and `d` without having to compare `c` and `d` again – the flags will still be set from the initial comparison assuming neither (1) nor (2) have occurred since that initial comparison. Knowing this and that our comparisons are mutually exclusive, let's write the fourth if statement, `c > d`, saving the third (and most complex) for last:

```
        .comm a, 4
        .comm b, 4
        .comm c, 4
        .comm d, 4
        movl a, %eax
        cmpl b, %eax    # evaluate top if: is (a-b) <= 0?
        jle  block4     # if is false, so jump out to else
             movl c, %eax   # evaluate 1st if: is (c-d) < 0?
             cmpl d, %eax   # if is false, so jump to
             jl block1
             jg block3      # evaluate 3rd if: is (c-d) > 0?
             ...            # no need for another cmpl here
             ...
        ...
        ...
        ...
block1:
        ... code block 1 ...
        jmp more
block3:
        ... code block 3 ...
        jmp more
block4:
        ... code block 4 ...
        # no need to jump to more (it's immediately below)
more:
        ... more code ...
```

Code 3.3.4

Notice, like we described, we saved ourselves a line of code in **Code 3.3.4** (and made our implementation more efficient) by using the comparison for `c < d` in the `c > d` comparison – we simply inserted a different condition to jump on after the first jump. Now, all we have left to implement is the second condition in the inner `if...then...else` statements – namely `(c`

`== d) && (a != 0)`. Like the `c > d` comparison, we can use the previous comparison, since nothing other than jumps have been inbetween the initial comparison and the jump. But, we cannot simply use this comparison alone – we need to implement the compound `AND` technique we discussed in **Section 3.2 – Complex and Compound Conditional Expressions** to also check if `a != 0`. Doing so, we can now complete our assembly translation with:

```
        .comm a, 4
        .comm b, 4
        .comm c, 4
        .comm d, 4
        movl a, %eax
        cmpl b, %eax    # evaluate top if: is (a-b) <= 0?
        jle  block4     # if is false, so jump out to else
             movl c, %eax    # evaluate 1st if: is (c-d) < 0?
             cmpl d, %eax    # if is false, so jump to
             jl block1
             jg block3       # evaluate 3rd if: is (c-d) > 0?
             jne more        # evaluate 2nd if: is (c-d) == 0?
             movl a, %eax
             cmpl $0, %eax   # evaluate 2nd if: is (a-0) != 0?
             je more
block2:
     ... code block 2 ...
     jmp more
block1:
     ... code block 1 ...
     jmp more
block3:
     ... code block 3 ...
     jmp more
block4:
     ... code block 4 ...
     # no need to jump to more (it's immediately below)
more:
     ... more code ...
```

Code 3.3.5

Since we chose to place `... block 2 ...` immediately below the last comparison in **Code 3.3.5**, the `block2` label is completely optional. However, we included it to make the code easier to read. Also, notice that `block4` does not need an unconditional jump to `more`. It is the last label before `more`, so execution will automatically progress into `more` as it proceeds from top to bottom.

Take some time to walk through and truly understand the example discussed in **Section 3.3 – if ... then ... else Conditional Expressions**. It might seem complicated, but when approached one step at a time, `if ... then ... else` statements can be simple to implement in assembly language.

3.4 – Special Looping Instructions

We will next consider three special instructions included in the 80386 instruction set specifically for implementing loops. Consider a typical C loop of the form:

```
int i;
do {
     ... code ...
} while(--i);        /* decrement i, then check if i > 0 */
... more code ...
```

Code 3.4.1

Normally, we would implement **Code 3.4.1** as:

```
.comm i, 4
movl i, %ecx
do:
     ... code ...
     decl %ecx            # %ecx = %ecx - 1
     jnz do               # go to 'do' if zero flag not set
more:
     movl %ecx, i              # save decremented i from %ecx
     ... more code ...
```

Code 3.4.2

Notice the new instruction in **Code 3.4.2** – `decl %ecx`. This instruction is the **decrement instruction**, and is equivalent to subtracting one from its argument. Recall flags are set for *any arithmetic* or *compare* instructions, so the `decl` instruction *also sets flags*. When the zero flag is set, the loop has reached zero, and the `while` condition fails.

As an aside, note the efficiency of loops that count down to zero instead of up to some number. Zero has the special property that it will set the zero flag when its value is reached. Counting up to a number will typically not set any flags (unless going from a negative to zero or a positive number). In other words, the `decl` instruction, when looping to zero, performs the decrement and the comparison (`cmpl`) instruction by setting the zero flag automatically. Also note that we are keeping `i` in the `%ecx` register during the loop and saving it to the variable `i` after the loop breaks, which will produce faster code.

The use of `decl` seems very efficient and quite simple too, but the previous implementation can be further improved. There is a special **loop instruction** called the `loop` instruction. We can use a `loop` to take the place of *both* the `decl` and `jnz` instructions. For instance, the previous example in **Code 3.4.2** could also be written as:

```
.comm i, 4
movl i, %ecx
do:
     ... code ...
     loop do              # decrement %ecx with 'loop'
                          # if zero flag not set, go to 'do'
... more code ...
```

Code 3.4.3

As shown in the comments in **Code 3.4.3**, the use of `loop` *requires* that the loop count *down to zero* and that the loop variable be *kept in the* `%ecx` *register*. The C register is often referred to as the **counting register** because of the many instructions that use it as a counter – the `loop` instruction, for example. There are variations on `loop` that allow the instructions to both decrement the loop variable and test for conditions other than the zero flag. Read more about these in the Intel 80386 instruction set reference.

3.5 – Long Distance Jumps

One last thing to consider about conditional jumps is that they are *short jumps*. This means they can only jump short distances in programs – namely, they can only jump within the program's current segment. As a brief background, when programs are running on a computer, they only have access to a certain amount of memory (instructions and variables) at a time. The **current segment** is defined as any address that can be referenced from the program's index register. When a label address outside the range of the index register is requested, a new segment selector must be loaded to access this part of the executable code. The **jmp (long distance jump) instruction** can modify the segment selection register, and will automatically do so in order to access a label outside the current segment. So, for example, if we want to jump if equal to `label`, and `label` is out of our segment (very far away in our program), we must do the following:

```
je label  # this code will not work if 'label' is too far
          # away to access with a conditional jump
```

Code 3.5.1

must be written as:

```
          jne nojmp            # skip the long distance jump
          jmp label            # go out of the current segment
     nojmp:                    # go here to not jump to 'label'
          ... more code ...
```

Code 3.5.2

Code 3.5.2 solves the problem posed in **Code 3.5.1** because `jmp` can jump anywhere in memory. Note that the jump condition to `nojmp` is negated from the original (`je label` becomes `jne nojmp`). If we do not want to jump to label, we need to check for the opposite condition and skip over the long distance jump.

Note that the `jmp` instruction for long distance jumps is the same `jmp` instruction used for unconditional jumps. We are simply stating its requirement for long distance jumps in this section.

Now, there is an instruction that implements this trick for us, and it is called the `JUMPS` instruction. It is placed immediately before any conditional jump that is out of range, and when the assembler is run on the source code, the change is automatically made if there is a need to do so.

3.6 – Assignment

This is the specification of what the assembly functions need to perform. Do not copy or type this code. Use it as a reference when writing the assembly.

```
/* begin assignment specification */

void classify(void)
{
     /*
     Covert the following C code to equivalent assembly.
     Everywhere you see "return" replace with "jmp return" Don't
     forget to declare "match" in the area at the bottom. Note
     that the C driver defines i, j, k, and tri_type, so the
     assembly code does not need to.
     */

     if (i == 0 || j == 0 || k == 0)
     {
          tri_type = 0; /* can't be a triangle */
          return;
     }
```

```
/* count matching sides */
match = 0;
if (i == j) match += 1;
if (i == k) match += 2;
if (j == k) match += 3;

/* select possible scalene triangles */
if (match)
{
     if (match == 1)
     {
          if ((i+j) <= k)
          {
               tri_type = 0;  /* Not a triangle */
               return;
          }
     } else
     {
          if (match != 2)
          {
               if (match == 6)
               {
                    tri_type = 1; /* Equilateral triangle */
                    return;
               }
               else
               {
                    if ((j+k) <= i)
                    {
                         tri_type = 0; /* Not a triangle */
                         return;
                    }
               } /* end of "if (match == 6)...else" */

          } else /* else for "if (match != 2)...else" */
          {
               if ((i+k) <= j)
               {
                    tri_type = 0; /* Not a triangle */
                    return;
               }

          } /* end of "if (match != 2)...else" */

     } /* end of "if (match == 1)...else" */
```

```
            tri_type = 2;  /* Isosceles triangle */
            return;

      } /* end of "if (match)" */

      /* check to see if this is a triangle */
      if ((i+j) <= k || (j+k) <= i || (i+k) <= j)
      {
            tri_type = 0; /* Not a triangle */
            return;
      }

      tri_type = 3; /* Scalene triangle */
      return;

} /* end of void classify(void) */

/* end assignment specification */
```

The following is the C driver for the assembly file. You are not required to comment this code. Please do not modify this file.

```
/* begin C driver */

#include <stdio.h>

int i, j, k, tri_type;

int main(int argc, char **argv)
{
     printf("Enter the size of triangle side 1 -> ");
     scanf("%d",&i);
     printf("Enter the size of triangle side 2 -> ");
     scanf("%d",&j);
     printf("Enter the size of triangle side 3 -> ");
     scanf("%d",&k);


     classify();

     switch(tri_type)
     {
          case 0:
               printf("This is NOT a triangle.\n");
               break;
          case 1:
               printf("This is an Equilateral triangle.\n");
               break;
          case 2:
               printf("This is an Isosceles triangle.\n");
               break;
          case 3:
               printf("This is a Scalene triangle.\n");
               break;
          default:
               printf("Invalid triangle type returned.\n");
     }
     return 0;
}

/* end C driver */
```

The following is the assembly stub to the driver. You are required to fully comment and write the assembly code to model the specification code. Insert your code where you see `/* put code here */` **and** `/* declare variables here */`**. Do not modify any other code in the file. Note the last line of the file must be a blank line to compile without warnings.**

```
/* begin assembly stub */

.globl classify
.type classify,@function
classify:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */

return:
     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

/* declare variables here */

/* end assembly stub */
```

The following are the cases you should test to ensure correct operation.

```
./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 3
Enter the size of triangle side 3 -> 10
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 10
Enter the size of triangle side 2 -> 3
Enter the size of triangle side 3 -> 3
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 10
Enter the size of triangle side 3 -> 3
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 1
Enter the size of triangle side 2 -> 2
Enter the size of triangle side 3 -> 10
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 10
Enter the size of triangle side 2 -> 1
Enter the size of triangle side 3 -> 2
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 1
Enter the size of triangle side 2 -> 10
Enter the size of triangle side 3 -> 2
This is NOT a triangle.

./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 3
Enter the size of triangle side 3 -> 5
This is an Isosceles triangle.

./lab3
Enter the size of triangle side 1 -> 5
Enter the size of triangle side 2 -> 3
```

```
Enter the size of triangle side 3 -> 3
This is an Isosceles triangle.

./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 5
Enter the size of triangle side 3 -> 3
This is an Isosceles triangle.

./lab3
Enter the size of triangle side 1 -> 4
Enter the size of triangle side 2 -> 4
Enter the size of triangle side 3 -> 4
This is an Equilateral triangle.

./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 4
Enter the size of triangle side 3 -> 5
This is a Scalene triangle.

./lab3
Enter the size of triangle side 1 -> 0
Enter the size of triangle side 2 -> 4
Enter the size of triangle side 3 -> 5
This is NOT a triangle.


./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 0
Enter the size of triangle side 3 -> 5
This is NOT a triangle.


./lab3
Enter the size of triangle side 1 -> 3
Enter the size of triangle side 2 -> 4
Enter the size of triangle side 3 -> 0
This is NOT a triangle.
```

Lab 4

Addressing Modes: Arrays and Pointers

Student Objectives

- Differentiate between data, operands, result, and post-op location
- Learn about addressing modes in the Intel 80386
    - Register addressing
    - Immediate addressing
    - Direct addressing
    - Register indirect addressing
    - Direct indexed addressing
    - Base indexed addressing
- Apply addressing modes to implement arrays, pointers, and data structures

4.1 – Addressing Data in the CPU

In this lab we begin to explore different addressing modes. An addressing mode is the means by which the computer selects the *data* that is used in an instruction. The addressing mode is determined by the syntax with which you specify the instruction's *operand(s)*. Let's begin by clarifying and expanding upon these terms.

First, **data** are numerical values that are used in instructions to compute a result. Data can be of any type or even a memory address. An **operand** is the form in which a piece of data is presented to the instruction. Consider **Code 4.1.1**. Here, we give some cases where data and operands are defined and used in assembly instructions.

```
movl $3, %eax        # 3 is data; $3 is the only operand
movl %eax, %ebx      # 3 is data; %eax is the only operand
addl $1, %ebx        # 1 and 3 are data;
                     # $1 and %ebx are operands
addl %eax, %ebx      # 3 and 4 are data;
                     # %eax and %ebx are operands
```

Code 4.1.1

In **Code 4.1.1**, notice that for each instruction, the *result* and where it is stored are *not considered data or operands*. The **result** of an instruction is the new value that is produced from the data; *where* the result of the instruction is stored is, for lack of a better word, the **post-op** (or **post-operation**) **location**.

The terms *data* and *operand* only apply to information the instruction uses to compute a result (i.e. the *source* information or parameters). However, sometimes a destination parameter – such as

`%ebx` in each `addl` instruction in **Code 4.1.1** – is considered as data and an operand. Recall that for many assembly instructions, addition included, the *destination* parameter contains information necessary to compute the result and is thus treated as a *source* until the result is computed and stored. For example, consider **Code 4.1.2**, taken from the last line of **Code 4.1.1**:

```
addl %eax, %ebx      # %ebx          = %ebx    +    %eax
                     # post-op loc   = operand +    operand
                     # 7             = 4       +    3
                     # result        = data    +    data
```

Code 4.1.2

As shown **Code 4.1.2**, `%ebx`, the destination parameter, is not only the post-op location but is also an operand. The important thing to note is that certain components of an assembly instruction (e.g. the destination parameter in **Code 4.1.2**) can take on more than one role in the operall operation. When `%ebx` is an operand, it contains data; when it is a post-op location, it contains the result. Both term are applicable to this instruction, but only at particular points in time – namely the start and at the conclusion of the instruction, respectfully.

Now, as insight into the *result*, in well-designed assembly code, the result value and its corresponding post-op location will likely be data and an operand in a future instruction. Otherwise, why would we compute the result in the first place? These concepts of operands, data, results, and post-op locations apply to nearly every assembly instruction, including multiplication, division, and even compares and jumps.

So, now that we have clarified these terms, let's talk about the addressing mode of an instruction. The **addressing mode** of an instruction describes the relationship between the operands and the data. In other words, it details *how* we use the operands to retrieve the data desired for use in the instruction. In all, there are *six ways* in which we can reference or *address* information within an instruction: (1) register addressing, (2) immediate addressing, (3) direct addressing, (4) register indirect addressing, (5) direct indexed addressing, and (6) base indexed addressing. We will consider each of these over the course of this lab. They will give us insight into and allow us to work with advanced data types, such as arrays, pointers, and structures. Addressing modes will also give us the background necessary to work with data on the stack, which is the topic of Labs 5 and 6.

4.2 – Simple Addressing – Register, Immediate, and Direct

The previous example (reproduced in **Code 4.2.1** below) illustrates the two simplest addressing modes, **register addressing** and **immediate addressing**. Register addressing means the *data is in a register*, and immediate addressing means the *data is supplied as part of the instruction* – a numerical constant.

```
movl %ecx, %eax      # register addressing – register '%ecx'
movl $4, %ebx        # immediate addressing – constant '4'
addl $3, %ebx        # register addressing – register '%ebx'
                     # immediate addressing – constant '3'
```

Code 4.2.1

These two are fairly straightforward – a value or constant as an operand is a form of immediate addressing, while a register as an operand is register addressing. Let us now consider variables:

```
addl %eax, var       # var is an address to the data, %eax
```

Code 4.2.2

In **Code 4.2.2**, the instruction `addl` is summing `var` and the A register. `var` is a symbol that represents a memory location or address. This address in memory contains the data we say is "stored" in the variable `var`. The *use of variables or symbols* as operands is called **direct addressing**, which means the memory address of the data is supplied with the instruction. One might ask, how is the address supplied to the instruction? When the assembler translates the program into executable machine code, it assigns an address to the symbol `var`, and it is that address that is included with the instruction in the program memory. Thus, the operand is the memory address represented by the symbol `var`, and the data is the contents of memory at that address.

4.3 – Declaring and Initializing Arrays

Now, before we discuss other forms of addressing, let's set the stage and look at a different kind of variable – namely, arrays. Consider the following fragment of C code:

```
int a[10];          /* an array of 10 integers */
int i;
main()
{
      ...
      ...
      a[0] = 10;
      a[4] = 20;
      a[i] = 30;
      ...
      ...
}
```

Code 4.3.1

As shown in **Code 4.3.1** above, the code declares an array and assigns values to certain elements of the array. So far, we have not worked with arrays in our assembly programs. To start, let's discuss how to declare or allocate space for an array in assembly language. Recall the `.comm` assembler directive we have studied:

```
.comm i, 4          # allocate 4 bytes of space for 'i'
```

Code 4.3.2

We can use the same memory-allocation mechanism demonstrated in **Code 4.3.2** to allocate memory for arrays. Remember, the second parameter to `.comm` is the amount of space we want to reserve. In the case of the integer `i`, we need four bytes of space. Now, the array `a` is an array of ten integers. Each integer needs its own four bytes of space; so, it follows that array `a` requires 40 bytes of space:

```
.comm a, 40         # allocate 40 bytes for a 10-int array
```

Code 4.3.3

In other words, the assembly code given in **Code 4.3.3** declares ten long words of memory, but does not initialize them to any value. To allocate and initialize arrays, we can use the `.fill` directive as shown in **Code 4.3.4**:

```
a: .fill 10, 4, 0    # allocate 10*4 bytes at label 'a'
                     # fill each 4-byte unit with '0'
```

Code 4.3.4

In general terms, the `.fill` assembler directive works as follows:

```
<label>: .fill <length>, <size>, <value>
```

Code 4.3.5

where in **Code 4.3.5** `<length>` is the length of the array, `<size>` is the size in bytes of each element, and `<value>` is the value to set to each `<size>` unit in `<length>`. Note, `<value>` may be omitted. The default for the `.fill` directive is to initialize each unit to zero if `<value>` is not present.

What if we want to initialize different elements of an array to different values? If we want to set the first three elements of array `a` to the values `10`, `20`, and `30` and the rest to zero, we could write:

```
a:
     .fill 1, 4, 10      # a[0] = 10
     .fill 1, 4, 20      # a[1] = 20
     .fill 1, 4, 30      # a[2] = 30
     .fill 7, 4, 0       # a[3 - 9] = 0
```

Code 4.3.6

Think of the label `a` in **Code 4.3.6** as being an address in memory. This address `a` and onward is being defined or filled with whatever we choose to put after the `a` label. In other words, `a` is a symbol that is equal to the *starting address* of the first word within it. Let's try and visualize what is going on here:

Address	0x100	0x104	0x108	0x10C	0x110	0x114	0x118	0x11C	0x120	0x124
Data	10	20	30	0	0	0	0	0	0	0
Position	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Figure 4.3.1

Remember, the symbol `a` is simply an address in memory to represent the start of the integer array we are defining – `0x100`, in this example. As shown in **Figure 4.3.1**, to reference the 0th element, address `0x100+0*4` is accessed; to reference the 1st element, address `0x100+1*4` is accessed; to reference the 2nd element, address `0x100+2*4` is accessed; etc. When we initialized this space in memory for `a`, the first `.fill` in label `a` says to set a single 4-byte size block of memory to the value `10`; the next `.fill` in label `a` sets the next 4-byte block to `20`; the third `.fill` sets the

following 4-byte block to `30`; while the last `.fill` sets the 7*4 or 28-byte block of memory to 0. We can envision each successive `.fill` as taking up `<length>*<size>` bytes of the `a` label. These four invocations of `.fill` complete the initialization of the 40-byte or ten integer array `a`.

4.4 – Working with Arrays – An Application of Direct Addressing

So, now that we have allocated and initialized an array in assembly, how can we access array elements for use in assembly instructions? Consider the first line of C code within `main` of our example in **Code 4.3.1** (reproduced as **Code 4.4.1** below). We can access an array element and assign it a value using the square bracket notation:

```
int a[10];          /* an array of 10 integers */
int i;
main()
{
      ...
      ...
      a[0] = 10;      # Use [X] to access element X
      a[4] = 20;
      a[i] = 30;
      ...
      ...
}
```

Code 4.4.1

In **Code 4.4.1**, the `a[0] = 10;` access is simple, since this refers to the first element of the array. We can access it by simply using *direct addressing*, which as discussed in detail in **4.2 – Simple Addressing – Register, Immediate, and Direct**, appears as follows:

```
movl $10, a         # a is at 0x100, move '10' into the
                    # address 0x100 in main memory
```

Code 4.4.2

Remember that the symbol `a` in **Code 4.4.2** is the address of the first word of the array – `0x100` in our example. So the `movl $10, a` instruction simply moves the number `10` into the first long word of the array at the address of `a` – let's say `0x100` for these examples. Why the first *long word*? The answer is due to the instruction suffix used in the instruction. Recall from **Lab 2, Section 2.4 – Data Sizes, Register References, and Opcode Suffixes**, that in the Intel 80386, we can work with data of three different sizes – 1-byte, 2-byte, and 4-byte. Each of these is denoted with an instruction suffix of `b`, `w`, and `l`, respectively. In our example, the `mov` instruction has the `l` suffix, so we are storing the constant `10` as a long word (or as 4 bytes) at address `a`.

The second example in **Code 4.4.1**, `a[4] = 20;`, is a little more tricky, since we do not have a symbol defined for the 4th element. We can, however, use *direct addressing* once more, and specify an *offset from the start* of the array as follows:

```
movl $20, a+16      # a is 0x100, 0x100 + 16 bytes = 0x110
```

Code 4.4.3

Recall that each element of `a` is 4 bytes, so the element `a[4]` *begins* at address `a+4*4 = a+16`, as shown in **Code 4.4.3**. Note this is still classified as *direct addressing*. The + symbol tells the assembler to add the value following it, `16`, to the address of `a`, and then include this sum with the assembly instruction as the address in memory that we wish to access. In the case of **Code 4.4.3**, the sum included with the `movl` instruction is `0x100 + (decimal) 16 = 0x110`, which is the starting address of `a[4]`. After the code's assembly process, there is no indication that the address supplied was originally a label or a computed value (such as `a+16`). In fact, if we wanted to read from or write to some fixed memory address, we can exlude the label altogether and write:

```
movl $20, 110H      # move '20' into address 0x110 (a[4])
                    # the trailing 'H' is for 'hexadecimal'
```

Code 4.4.4

The use of labels simplifies our programs by allowing us to refer to memory addresses with symbols or variables, rather than referencing the addresses directly, as depicted in **Code 4.4.4**. Fortunately, assemblers allow us to use variables and labels instead of hexadecimal addresses. In **Code 4.4.3**, the use of variables and labels help us keep track of addresses. Not only do they make assembly code easier to read, but they can also reduce the likelihood of errors by letting the assembler substitute the addresses at compile-time.

4.5 – Working with Arrays – Direct Indexed and Based Indexed Addressing

Continuing with our previous example in **Code 4.3.1** and **Code 4.4.1**, we come to the last line of our excerpt, `a[i] = 30;`.

```
int a[10];          /* an array of 10 integers */
int i;
main()
{
      ...
      ...
      a[0] = 10;
      a[4] = 20;
      a[i] = 30;      # set the "ith" element of a to 30
      ...
      ...
}
```

Code 4.5.1

This array access in **Code 4.5.1** is different from the previous two examples we have discussed. Notice, we have an indexing variable – `i` – that determines which element of `a` to access. Because `i` is a variable – meaning it can change during program execution – it is *not possible* for the assembler to know the address of the data at `a[i]` when the program is assembled. The address of this data must be computed at runtime.

As such, the address cannot be directly specified in the instruction. In order to do this type of addressing, the Intel 80386 provides us with a couple more addressing modes, which are referred to as **indexed addressing** modes. As we will show in the following examples, these forms of addressing are very useful when working with arrays and structures.

In indexed addressing, the processor uses the contents of an *indexing register*, along with a data size constant, to compute the memory address of the data the instruction wishes to access. There are two index registers onboard the Intel 80386 for performing index addressing – `%esi` and `%edi`. `%esi` stands for **source index** and `%edi` stands for **destination index**. We may use the two interchangeably when it comes to working with arrays. The index registers must contain the index of the array element we desire to access – the variable `i` in our example.

Aside: In actuality, there is no requirement necessitating the use of the source or destination index registers in indexed addressing – any general purpose register will work as an index; however, indexing registers are preferred, since their primary purpose is to maintain an index without utilizing an artithmetic general purpose register (i.e. %eax, %ebx, %edx, or %edx).. Furthermore, although it is not common practice, indexing registers can also be used in computations for which we have learned to use the arithmetic general purpose registers.

Consider the following C fragment:

```
.comm a, 40
.comm i, 4
movl $10, a                 # a[0] = 10
movl $20, a+16              # a[4] = 20
movl i, %edi                # move the index, 'i', into an
                            # index register, '%edi'
movl $30, a(, %edi, 4)      # a[i] = 30;
```

Code 4.5.2

In **Code 4.5.2**, the last instruction, `movl $30, a(, %edi, 4)`, uses indexed addressing. Let's discuss the syntax in greater detail. First, there are two types of indexed addressing – **direct indexed** addressing and **based indexed** addressing. They each take the *general* form:

```
<variable>(<base>, <offset>, <size>)
```

Code 4.5.3

What distinguishes direct indexed and base indexed addressing are the components of this general form in **Code 4.5.3** that are utilized. **Direct indexed addressing** *omits* the `<base>` parameter and appears as follows:

```
<variable>(, <offset>, <size>)
```

Code 4.5.4

Consider the structure of direct indexed addressing provided in **Code 4.5.4**. This syntax provides the program with the information it needs to calculate the address of the memory location accessed by the instruction. This address is computed by the program at runtime, which takes the `<offset>` and multiplies it by the `<size>`. The `<offset>` is one of the index registers – `%edi` or `%esi` – and the `<size>` is the size of each element of the array we are dealing with. In our example, we are working with an *integer array*; and since integers are 4 bytes in x86 assembly language, the size of each element of the array `a` is also `4` bytes.

Similar to *direct addressing*, `<variable>` as given in **Code 4.5.4** is a label or symbol, which represents an address in main memory. In our example, this label is variable `a`. To visualize this, let's construct a sample memory map where the start of our array `a` is at address `0x100`.

Address	0x100	0x104	0x108	0x10C	0x110	0x114	...	...	0x120
Data	10	20	---	---	---	---	---	---	30
Position	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[...]	a[...]	a[i]

Figure 4.5.1

If we wish to access `a[i]` and set it equal to the value `30`, we need to first move our index into an index register, and then we can perform direct indexed addressing:

```
movl i, %edi              # move 'i' to an index register
movl $30, a(, %edi, 4)    # 0x100+(i*4)
```

Code 4.5.5

Code 4.5.5 takes the address of `a` – `0x100` – and adds to this the value of the `%edi` register multiplied by `4`, which is `<offset> * <size>`. If the indexing variable `i` is `8`, then the address accessed is:

```
0x100 + (8 * 4) = 0x120  # see Figure 4.5.1
```

Code 4.5.6

In **Code 4.5.6**, the computation performed by the program to access `a[i]` is shown for `i = 8`. This can be validated by a close examination of the memory map given in **Figure 4.5.1** above.

What we have discussed so far is only direct indexed addressing. Recall, there is another form or indexed addressing called **base indexed addressing**. It takes the form:

```
(<base>, <offset>, <size>)
```

Code 4.5.7

In **Code 4.5.7** `<variable>` is omitted for *based indexed addressing*. (Contrast this to *direct indexed addressing*, which omits the `<base>` as shown in **Code 4.5.4**.) Based indexed addressing uses a *base register* to store the address of the start of the array we wish to access. So, what is the difference between the two forms of indexed addressing? In many cases, there is not a difference. In fact, we can write the previous example (see **Code 4.5.5**) using base indexed addressing:

```
movl i, %edi              # move the index, 'i', into an
                          # index register, '%edi'
movl a, %ebx              # move the start of the array 'a'
                          # into the base index register
movl $30, (%ebx, %edi, 4)# a[i] = 30;
```

Code 4.5.8

Let's discuss how the address of `a[i]` is computed in **Code 4.5.8**. Similar to direct indexed addressing, base indexed addressing uses an index for the `<offset>` (provided via an indexing register – `%edi` or `%esi`). It then computes the address to be accessed by multiplying the `<offset>` by the `<size>`; this is then added to the `<base>`. In this specific example, the `<base>` is the address aliased as array `a` or `0x100`, the `<offset>` is `i` in `%edi`, and the

<size> is 4 bytes (i.e. the size of each element in an integer array). Assuming i = 8 once again, this computation can be validated using the memory map given in **Figure 4.5.1** and is shown in **Code 4.5.9** below:

```
0x100 + (8 * 4) = 0x120  # see Figure 4.5.1
```

Code 4.5.9

Note that for both forms of indexed addressing, the <size> used to compute the address (seen in **Code 4.5.5** and **Code 4.5.8**) is not preceeded by a $. The $ symbol is only used to specify a constant that is used *directly as a parameter in an assembly instruction*. In either form of indexed addressing, <size> is an intermediate value and is thus taken literally in the computation of the address; the inclusion of the $ symbol *will result in a compile time error*. Furthermore, <size> must be a number and cannot change durring runtime (i.e. it cannot be a register or a variable). Contrast this to <offset> which can change at runtime via an indexing register.

Although base indexed and direct indexed addressing can oftentimes be used interchangeably, there are some cases when we *do* need to use base indexed addressing, and these cases involve structures. We will discuss this specific case set in greater detail in **Section 4.7 – Putting it all Together – An Example using Arrays, Pointers, and Structures**.

4.6 – Working with Pointers – Register Indirect Addressing

Like indexed addressing, there is another situation in which the address of the data is not known at assemble time, and that is when a program uses a *pointer*. In this situation, the address of the data is specified entirely in a register. For this case, the Intel 80386 provides us with a final form of addressing – **register indirect addressing**. In register indirect addressing a *register* holds the address of the data to be addressed, as necessitated by **Code 4.6.1** below:

```
int *p;              /* declare an integer pointer */
main()
{
      ...
      ...
      *p = 40;       /* set the value at the ptr to '40' */
      ...
      ...
}
```

Code 4.6.1

which can be written in assembly language as:

```
.comm p, 4          # allocate space for the pointer
movl p, %ebx        # move address in 'p' to register B
movl $40, (%ebx)    # set value at 'p' to '40'
```

Code 4.6.2

In **Code 4.6.2**, the variable `p` is a pointer that contains the address where the integer resides in memory. Informally, the parenthesis around the register say, "go to the location/address in `%ebx` and write `40` there." So, if `p` contains the address `0x100`, this example of register indirect addressing *dereferences* the B register and writes the constant `40` at address `0x100`. In assembly (and other types of) programming, the term **dereference** means to obtain the address of some data located elsewhere in memory.

An offset can also be applied to a pointer in assembly language. This can be useful when traversing an array or accessing an element within a structure. To use register indirect addressing with an offset, we simply include to the left of the parenthesis the offset in bytes to add to the address stored in the register being dereferenced. For example, if `p` is a pointer to the base of an integer array, to set the `0th` element of the array to `50`, and the `4th` element to `100` using register indirect addressing, we would write the assembly code as shown in **Code 4.6.3**:

```
movl p, %ebx        # move address in 'p' to register B
movl $50, (%ebx)    # set value at 'p' to '50'
movl $100, 16(%ebx) # set value at 'p[4]' to '100'
```

Code 4.6.3

One might think that using a variable as a pointer would look more like `movl $40, (p)`, but in 80386 assembly this is *not* the case. Recall from **Section 2.3 – Moving Data, Addition, Subtraction, and Constants** that *no more than one main memory access can be performed in a single assembly instruction*. The invalid assembly instruction `movl $40, (p)` accesses main memory to read the data at the address of variable `p`, then accesses main memory again to read the data at the address stored in `p`, which is one memory access too many. We must first move the pointer to a register, and then perform register indirect addressing on the register.

Note, although we have used the B register in our example for register indirect addressing (and indexed addressing in **Section 4.5 – Working with Arrays – Direct Indexed and Base Indexed Addressing**), we may dereference *any general purpose register*, including the indexing registers. The B register is most commly used, since its name is likened to the *base* of the array being accessed.

4.7 – Putting it all Together – An Example using Pointers, Arrays, and Structures

Let's work a final example. Here, we will show how we can use a combination of the addressing modes we have discussed in order to implement a more complex (and relatively typical) case involving arrays and structures. Consider the following C code:

```
int *ap;                    /* pointer to an array */
struct {                    /* structure with two integers */
     int a;
     int b;
} *asp;                     /* pointer to the structure */
int i;                      /* indexing/counting variable */

main()
{
     ...                    /* ap and asp "malloc'd" */
     ...
     ap[3] = 50;
     ap[i] = 60;
     asp[i].b = 70;
     ...
     ...
}
```

Code 4.7.1

Code 4.7.1 can be translated into assembly language as:

```
.comm ap, 4
.comm asp, 4
.comm i, 4

movl ap, %ebx
movl $50, 12(%ebx)              # ap[3] = 50
movl i, %edi
movl $60, (%ebx, %edi, 4)       # ap[i] = 60
                                # ap is still in %ebx
movl asp, %ebx                  # i is still in %edi
movl $70, 4(%ebx, %edi, 8)      # asp[i].b = 70
```

Code 4.7.2

Notice, in **Code 4.7.2** we have three global variables, all of which are 4 bytes and two of which are pointers. In programing, the size of a pointer is based on the size of the address space the

pointer needs to be able to map (or "point to"). In our case, we are working with the Intel 80386, which is a 32-bit processor; thus, pointers are all 4 bytes (i.e. 32 bits).

Let's get started. As seen in **Code 4.7.1** the first line of `main` sets the third element of `ap` to `50`. Recall from C programming that pointers and arrays are essentially the same thing. We can chose to allocate space for an array statically (as we have in our previous examples), or we can declare a pointer to an array, and then allocate the space for the array dynamically using tools such as `malloc`. We should assume that sufficient space for the arrays `ap` and `asp` has been allocated somewhere in main prior to the lines we will examine, as `malloc` is a topic beyond the scope of this course.

Let's first take a look at `ap[3] = 50`, which in our assembly translation in **Code 4.7.2** is:

```
movl ap, %ebx
movl $50, 12(%ebx)
```

Code 4.7.3

In the excerpt given in **Code 4.7.3**, `ap` contains the address of the array where we then want to set element `3` to the value `50`. We can use register indirect addressing combined with an offset to accomplish this. Unlike direct addressing, which specifies the offset with a + sign, register indirect addressing applies a constant to the left of the parenthesis of the register being dereferenced. In this case, we need to skip `12` bytes in order to get to the third element of `ap`. Why 12 bytes? Well, `ap` is a pointer to an integer array, and as we know, integers are 4 bytes – `4` bytes * `3` indices is `12` bytes. For example, assume the *zeroth* element of `ap` is at address `0x100`:

Address	0x100	0x104	0x108	0x10C	0x110	...
Data	---	---	---	50	---	...
Position	`ap[0]`	`ap[1]`	`ap[2]`	`ap[3]`	`ap[4]`	`a[...]`

Figure 4.7.1

As shown in **Figure 4.7.1**, an offset of `12` bytes from address `0x100` is `0x10C`, which corresponds to `ap[3]`. That is all we need to do to accomplish the first line of our C code in **Code 4.7.1**.

Now how about the second line? Here, we want to perform `ap[i] = 60`, where we set an *unknown* element of `ap` to the value `60`. Recall that in indexed addressing, the variable we index with is not known when the program is assembled; thus, we need to use either *direct indexed addressing* or *base indexed addressing* to access the `ith` value of `ap` during program runtime.

```
movl i, %edi
movl $60, (%ebx, %edi, 4)
```

Code 4.7.4

In **Code 4.7.4**, notice that to implement the second line of our C code in **Code 4.7.1**, we do not need to move `ap` into the B register – it is already there from translating the previous line of C. However, we do need to use one of our indexing registers to take `i` into account. Recall from **Code 4.7.3** that the address to which the array `ap` is pointing to is in the B register. We wish to access the `ith` element, so from what we know about base indexed addressing, we can do this by incrementing `<offset> * <size>` bytes from the address supplied in `<base>`. This is `0x100 + %edi * 4`, in this example. Remember, the size is `4` bytes, since we are working with an integer array.

Now, let's take a look at the final line of **Code 4.7.1**: `asp[i].b = 70`. Here, we wish to increment to the `ith` element of `asp`, access its `b` sub-element, and then set it to the value `70`.

This is the most complex operation of this example, and is the case we hinted at previously where base indexed addressing *must* be used. Why is this so? Recall the trick we used from register indirect addressing. We can insert a constant offset to the left of the parenthesis as an additional increment into the address calculated. With direct indexed addressing, the `<variable>` parameter is present, so we cannot use an additional offset; however, base indexed addressing allows us to exploit this trick. Let's walk through how it is done.

To start, recall that `asp` is a pointer to a structure – in this case, it is an array of structures. Each of these structures is of size `8` bytes and contains two integers. We wish to access the `ith` element of the structure array and set its `b` component to `70`. The same principles we have learned about base indexed addressing are applied here. The starting address of the `asp` array is placed in the B register, and the index variable is moved into an indexing register. Notice we already moved `i` into `%edi` for the previous C instruction, and since `i` has not changed, we can reuse it again.

Going further, recall the `<size>` parameter of indexed addressing is the size of each element in the array. In this case, we are working with an array of structures, each of which are `8` bytes. So, as a result, the `<size>` is `8`. Here is where we apply our technique from register indirect addressing – because we need to get to the `b` element, we need to add `4` to the address computed. When programs are compiled with structures, the elements are allocated sequentially in memory in the order they are declared; so, `b` is after `a`, and since both `a` and `b` are integers, we need to skip over `a`, which is `4` bytes. The entire calculation is as follows in **Code 4.7.6**:

```
<base> + (<offset> * <size>) + <additional offset>
```

Code 4.7.6

or specifically, `0x100 + (i * 8) + 4`. In the following memory map in **Figure 4.7.2**, we assume `i` to be the index `2`. So, to get to the desired location, the base indexed addressing simplifies to **Code 4.7.7**:

```
0x100 + (2 * 8) + 4 = 0x114
```

Code 4.7.7

The resulting calculation in **Code 4.7.7** can be verified and visualized with the memory map of the `asp` structure array in **Figure 4.7.2**:

Address	0x100	0x104	0x108	0x10C	0x110	0x114
Data	---	---	---	--	---	70
Position	`asp[0].a`	`asp[0].b`	`asp[1].a`	`asp[1].b`	`asp[i].a`	`asp[i].b`

Figure 4.7.2

Thus, the final assignment in our C code in **Code 4.7.1** can be implemented in assembly as follows in **Code 4.7.5**:

```
movl asp, %ebx
movl $70, 4(%ebx, %edi, 8)     # base + (i*structSize)
                               # + offsetOfB
                               # = 0x100 + (2 * 8) + 4
                               # = 0x110 + 4 = 0x114
                               # = asp[2].b, when 'i = 2'
```

Code 4.7.5

4.8 – Summary of Addressing Modes

Addressing Mode	Example	Data is...	Notes
Immediate	`movl $0, %eax`	A constant	
Register	`movl %eax, %ebx`	In a register	
Direct	`movl var, %eax`	Retrieved from the address of a variable	Can include an offset from the variable using +
Register Indirect	`movl (%ebx), %eax`	Retrieved from the address stored within a register	Can include an offset in front of the parenthesis. Used for dereferencing pointers
Direct Indexed	`movl var(, %edi, 4), %edx`	Retrieved from the address of a variable plus an offset multiplied by a constant size	Used for indexing into arrays
Base Indexed	`movl (%ebx, %esi, 8), %eax`	Retrieved from the address stored within the base register plus an offset multiplied by a constant size.	Can include an additional offset in front of the parenthesis. Used for indexing into arrays and structures

Table 4.8.1

4.9 – Assignment

This is the specification of what the assembly functions need to perform. Do not copy or type this code. Use it as a reference when writing the assembly. Please refer to the Appendix for a reference on ASCII code.

```
/* begin assignment specification */

AtoI()
{
     sign = 1; /* initialize sign to positive */

     /* skip over leading spaces and tabs */
     while (*ascii == ' ' || *ascii == '\t')
          ascii++;

     /* check for a plus or minus sign */
     if (*ascii == '+')
          ascii++; /* found a plus sign */
     else if (*ascii == '-')
     {
          sign = -1; /* found a minus sign */
          ascii++;
     }

     *intptr = 0; /* stores the value calculated below */

     /* skip to the one's place of the digit */
     for(i = 0; ascii[i] >= '0' && ascii[i] <= '9'; i++);
     i--; /* back up to the one's place */

     multiplier = 1; /* set multiplier for one's place */
     for(; i>=0 ; i--)
     {
          *intptr += multiplier * (ascii[i] - '0');
          multiplier *= 10; /* inc multiplier by factor of 10 */
     }

     /* multiply in the sign */
     *intptr = *intptr * sign;
}

/* end assignment specification */
```

The following is the C driver for the assembly file. You are not required to comment this code. Please do not modify this file.

```
/* begin C driver */

#include <stdio.h>

char input[10];
int output;
char *ascii;
int *intptr;
int sign, multiplier, i;

int main(int argc, char **argv)
{
     printf("Enter a number: ");
     fgets(input, 10, stdin);
     ascii = input;
     intptr = &output;

     /* this is the function we will write in assembly */
     AtoI();

     printf("\nASCII is: %s\nInteger is: %d\n",input,output);


     return 0;
}

/* end C driver */
```

The following is the assembly stub to the driver. You are required to fully comment and write the assembly code to model the specification code. Insert your code where you see `/* put code here */`**. Do not modify any other code in the file. Note the last line of the file must be a blank line to compile without warnings.**

```
/* begin assembly stub */

.globl AtoI
.type AtoI,@function
AtoI:
     /* prolog */
     pushl %ebp
     movl %esp, %ebp
     pushl %ebx
     pushl %esi
     pushl %edi

     /* put code here */

return:
     /* epilog */
     popl %edi
     popl %esi
     popl %ebx
     movl %ebp, %esp
     popl %ebp
     ret

/* end assembly stub */
```

The following are the cases you should test to ensure correct operation. Your program should accept an integer as input, in the form of a character string. The driver does not check for all input cases (e.g. junk in the input string), but correct operation should produce the following results:

```
./lab4
Enter a number: 32
ASCII is: 32
Integer is: 32

./lab4
Enter a number: -64
ASCII is: -64
Integer is: -64

./lab4
Enter a number: +128
ASCII is: +128
Integer is: 128

./lab4
Enter a number: 0
ASCII is: 0
Integer is: 0

./lab4
Enter a number: +0
ASCII is: +0
Integer is: 0

./lab4
Enter a number: -0
ASCII is: -0
Integer is: 0

./lab4
Enter a number: <space><tab><space><tab>256
ASCII is:           256
Integer is: 256
```

Lab 5

Subroutines and the Stack

Student Objectives

- Learn about the "stack" and how it is used in programming
- Learn about stack registers and how they are used to maintain the stack
- Learn how to implement functions/subroutines in assembly language
- Learn about the prolog and epilog and their important role in assembly language
- Learn how to create and work with local variables
- Differentiate between iterative techniques and recursive techniques
- Apply recursion to assembly programming

5.1 – Why Use Subroutines?

In this lab, we will discuss subroutine structure. As you might know from C, **subroutines** (**i.e. functions**) are the primary mechanisms for building programs out of smaller blocks. Rather than creating a program with all its code in `main()`, it oftentimes makes sense to break code into smaller functional blocks. Consider the following example implementations of `main()`:

```
/* no functions used */
int main(void) {
     ...
     ...
     ...code...
     ...
     ...
     ...more code...
     ...
     ...
     ...even more code...
     ...
     ...
     ...code... /* repeat */
     ...
     ...
     ...code... /* repeat */
     ...
     ...
     return 0;
}
```

```
/* functions used */
int main(void) {
     ...
     ...
     code_func();
     moreCode_func();
     evenMoreCode_func();
     code_func(); /* reuse */
     code_func(); /* reuse */
     return 0;
}
void code_func() {
     ...
}
void moreCode_func() {
     ...
}
void evenMoreCode_func() {
     ...
}
```

Code 5.1.1

Notice in the comparison in **Code 5.1.1** that functional programming allows easy reuse of code and can also increase readability. Rather than repeating or copying `...code...` three times in `main()`, we can compose a single function, `code_func()`, and call it multiple times instead. Furthermore, functional programming can also be a means to interface programs to the operating system and program library. For example, system calls such as `malloc()`, `printf()`, and `fork()` are C library functions that allow a program to interface with the operating system and its libraries.

5.2 – Calling and Returning from Subroutines

In assembly language, any program label can be the beginning of a subroutine. Thus, any label you can jump to (using `jmp`, `je`, `jne`, `jle`, `jg`, etc), you can alternatively `call` as a subroutine. The difference is that when you **call** a subroutine, the computer *saves the return address*, so that later when a `ret` instruction is executed, the program **returns** to the address of the instruction *immediately after the* `call`. You might be asking, "How does the `ret` instruction know the address of the instruction immediately after the `call`? The `call` saves this address, but how does `ret` access this saved information?" The answer involves something called the stack, which we will discuss in great detail in the following sections. Before we do that, let's go through a simple `call` and `ret` example:

```
0x100:    jmp some_label       call some_label
0x104:    ...code...           ...code...
0x108:    ...                  ...
0x10C:    ...                  ...
0x110:    some_label:          some_label:
0x114:         ...                  ...
0x118:         ...                  ...
0x11C:         jmp ...code...       ret   # go to ...code...
0x120:
```

Code 5.2.1

As shown in **Code 5.2.1**, `call` and `ret` are used to shift execution of the program to the label `some_label` and then resume execution at the *address immediately following* the `call` – `...code...` in this case. Notice the line numbers to the left of each line of code. These represent the addresses in memory these lines of assembly code reside at. Since the Intel 80386 uses a 32-bit instruction set architecture, each assembly instruction takes up 4 bytes of memory; thus, each line is represented in increments of four. Now, let's consider this example in more detail. When the `call` instruction is executed, the address of `some_label` is read, and then a jump occurs to that address – `0x110`, in this example. Later on, when the `ret` instruction is reached within `some_label`, the program jumps to the address of the line immediately after the `call` – address `0x104`.

With this example in mind, the processor allows us to *nest* subroutine calls. This means one subroutine can call another, which in turn can call another, and so on. Likewise, recursive

subroutines (i.e. functions that call themselves) can be implemented. For each subroutine `call`, we need to have a corresponding `ret` (or return) to go back to the line of code beneath the `call`. Each time a subroutine returns, it only returns to the subroutine that called it last. For example:

```
0x100:    call function_1
0x104:    ...after_f1_call...
0x108:    ...
0x10C:    function_1:
0x110:         ...
0x114:         call function_2
0x118:         ...after_f2_call...
0x11C:         ...
0x120:         ret   # go to ...after_f1_call...
0x124:    function_2:
0x128:         ...
0x12C:         ret   # go to ...after_f2_call...
```

Code 5.2.2

Here, in the nested `call` in **Code 5.2.2**, the `call` at line `0x100` causes execution to jump to `function_1` at line `0x10C`, which in turn `calls function_2` at line `0x114`. This `call` causes execution to jump to `function_2`. Now, when the `ret` in `function_2` is reached at line `0x12C`, it will return to the code `...after_f2_call...`. After this return, the code will resume execution at line `0x118` until it reaches the `ret` at line `0x120`. This return will jump to the code `...after_f1_call ...`.

So, considering we can nest subroutine calls, how does the processor keep track of where to return to for a particular `ret`? In other words, how do we know which `call` a certain `ret` corresponds to? The answer lies in *the stack*.

5.3 – An Introduction to the Stack

In programming, the **stack** is a data storage mechanism that has two basic features: (1) the ability to add data to it, and (2) the ability to remove data from it. Each of these features is implemented via the assembly instructions `push`, `pop`, respectively. Before we discuss how these instructions work and how to use them in our assembly code, let's first visualize the stack in memory:

Address	Contents
0x0800	
...	
0x1000	
0x1004	
0x1008	
0x100C	
0x1010	
0x1014	
0x1018	
0x101C	
0x1020	
0x1024	XXXX

Register	Contents
%eax	
%ebx	
%ecx	
%edx	
%esi	
%edi	
%ess	0x0800
%ebp	0x1030
%esp	0x1024

Figure 5.3.1

In **Figure 5.3.1**, we have a stack shown on the left, and on the right are some registers – some should be familiar and some might not be – we will discuss the last three registers shortly. For the time being, let's ignore the registers and focus our attention on the stack itself.

First, the **stack** is a segment of main memory (RAM) that has been allocated to a program by the operating system. The purpose of the stack is to provide a means and a location for the program to store data critical to its successful execution. In a 32-bit computer, and as shown above, the stack is composed of 4-byte blocks of memory, where data can be stored and retrieved. Based on what we have discussed about function calls and returns, what data does a program need to *remember* when a function is called? It needs to store the *return address of the* `call` – the address where the assembly instruction after the `call` instruction is located. As we have alluded to, this return address is stored on the stack when the `call` is executed. The `call` instruction determines the address of the next line of code (4 bytes from the `call`), and then it saves this address on the stack. Later on, when the return is executed, the `ret` instruction reads the return address the `call` had already saved on the stack, then the program's execution jumps to that address.

It is important to note that the stack resides in a *different segment* of memory than the code itself. The assembly code is *not located on the stack*, but it instead *utilizes the stack* to store information and data generated during execution and function calls.

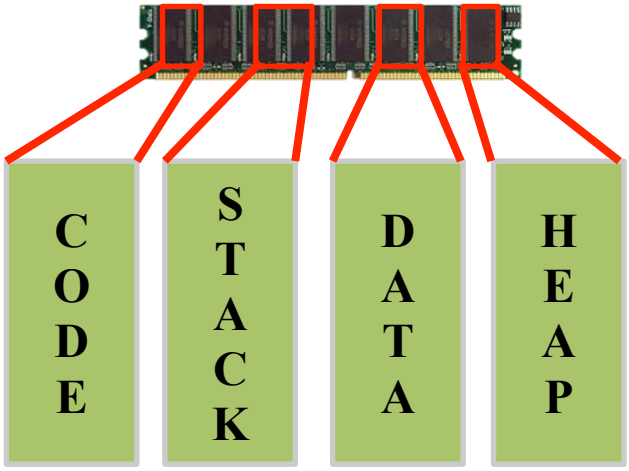


Figure 5.3.2

As shown in **Figure 5.3.2**, programs consist of *four independent blocks of memory* – (1) the **code segment**, where the instructions are located; (2) the **stack segment**, where function calls, returns, and temporary data are stored; (3) the **data segment**, where global variables and other static variables (those which are known and allocated at compile-time) are stored; and (4) the **heap segment**, where dynamically allocated variables are instantiated and stored. In a 32-bit computer, there are 2^{32} distinct addresses in memory – `0x0` through `0xFFFFFFFF`; and, each of these four segments of memory are mapped to four sets of distinct addresses. For example, the code segment is assigned to a different range of addresses than the stack segment, the data segment, or the heap segment. The program's instructions are located in and running from the code segment, while the stack, data, and heap segments *are used by the instructions* in the code segment to perform function calls/returns and to access variables, respectively. *Note: This is a common point of confusion when learning about and working with the stack. The stack is separate from the code in memory. The code is designed to use the stack to run.*

[As an aside, we mentioned there are 2^{32} distinct addresses in memory in a 32-bit computer. For those of us who have tried to upgrade the RAM in 32-bit computer, this is why there is a limit of 4 gigabytes (i.e. 2^{32} bytes) of RAM. A 32-bit operating system or 32-bit hardware is not capable of addressing more memory than 2^{32} bytes of memory.]

Now that we know what the stack is, as well as its purpose, let's explore how the stack operates. Consider our simple example of a `call` and a `ret`. As we now know, the return address is stored on the stack. However, what we have not discussed is *how* this process occurs. Consider some key information we as programmers might want to know about the stack. As we know, the stack is a segment of main memory the program uses to execute function calls and returns. What might we need to know about our stack space? It would be nice to know what the bounds of our stack space are. Since the stack is only a small chunk of the available RAM in the system, we need to know where it begins and where it ends. The **%ess register** (or **stack segment register**) provides us with one piece of that information. It is a pointer to what we call the **bottom of the stack**. The bottom of the stack is the *lowest address of our stack*. Likewise, the **%esp register** (or **stack pointer register**) provides us with the upper bound – a pointer to the **top of the stack**.

The top of the stack is the *highest address of our stack*. **Figure 5.3.3** below illustrates these two new registers.

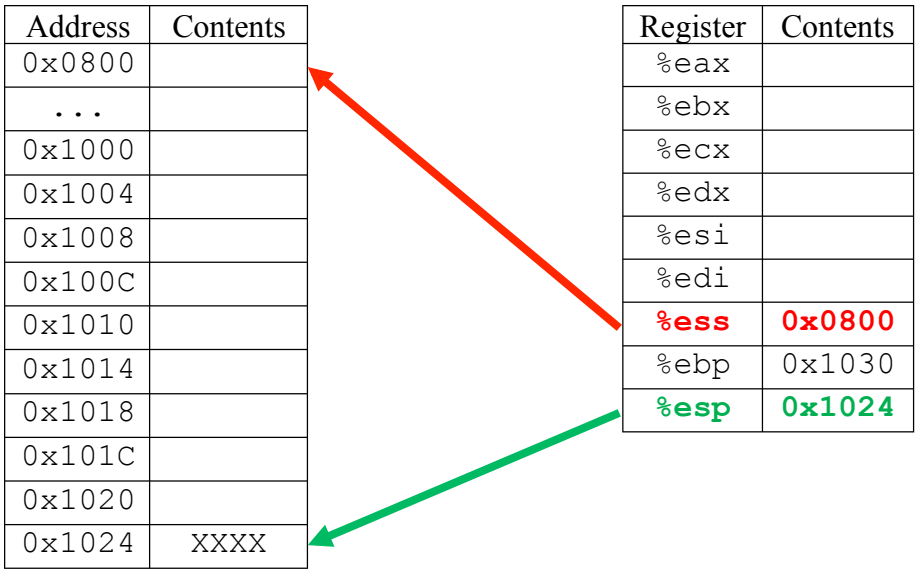


Figure 5.3.3

Note: In our stack illustrations, the bottom of stack is towards the top of the page, while the top of the stack is towards the bottom of the page. Do not confuse the top of the page with the top of the stack or the bottom of the page with the bottom of the stack. The bottom of the stack is the ***lowest stack address*** *and the top of the stack is the* ***highest stack address****.*

At program runtime, the operating system initializes the stack segment to the lowest address of the stack and the stack pointer to the highest address of the stack. In the example above, the stack pointer register (`%esp`) can point to any memory address from its initial value of `0x1024` all the way down to `0x0800` (the value of `%ess`). The stack segment register is static, while the stack pointer register is dynamic. Because the stack pointer can change, it is said to roam the stack. It will point to the current top of the stack or the *most recently added piece of data*. During program execution, as data is *added to the stack*, the value of the *stack pointer decreases by 4 bytes* to the limit of the stack segment.

So, with knowledge of the `%ess` and `%esp` register, we can now look at how `call` and `ret` actually work. When a function is called, the `call` instruction, as we know, places the return address of the next line of code to the stack. To do this, the `call` will (1) decrement the stack pointer (`%esp`) by 4 bytes, and then (2) write the return address at the location of the new value in `%esp`. Consider the following example:

```
0x100:    call my_func
0x104:    ...
0x108:    ...
0x10C:    my_func:
0x110:         ...
0x114:         ret
```

Code 5.3.1

In **Code 5.3.1** the call to `my_func` at address `0x100` will (1) decrement the stack pointer from `0x1024` to `0x1020`, then (2) place address `0x104` (the return address) on the stack at the new location of `%esp` – `0x1020`. At this point, program execution will then jump to `my_func` – address `0x10C`, as shown in **Figure 5.3.4** below:

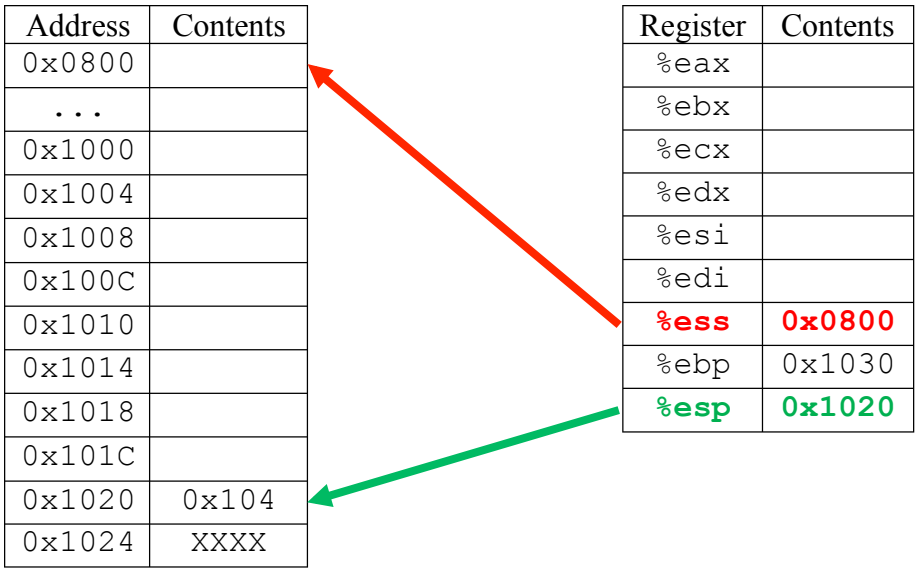


Figure 5.3.4

Consider the same example, where now we return to the calling function. After the jump to `my_func`, the intermediate code is executed and the `ret` is reached at address `0x114`. At this point, the `ret` performs the opposite operation as the `call`. The `ret` will (1) determine the current location of the stack pointer (`0x1020`) and read the data at this address on the stack – `0x104`. Next, the `ret` will (2) increment `%esp` 4 bytes to address `0x1024`. Finally, after this operation is complete, `ret` will resume program execution at the address it read from the stack – `0x104`.

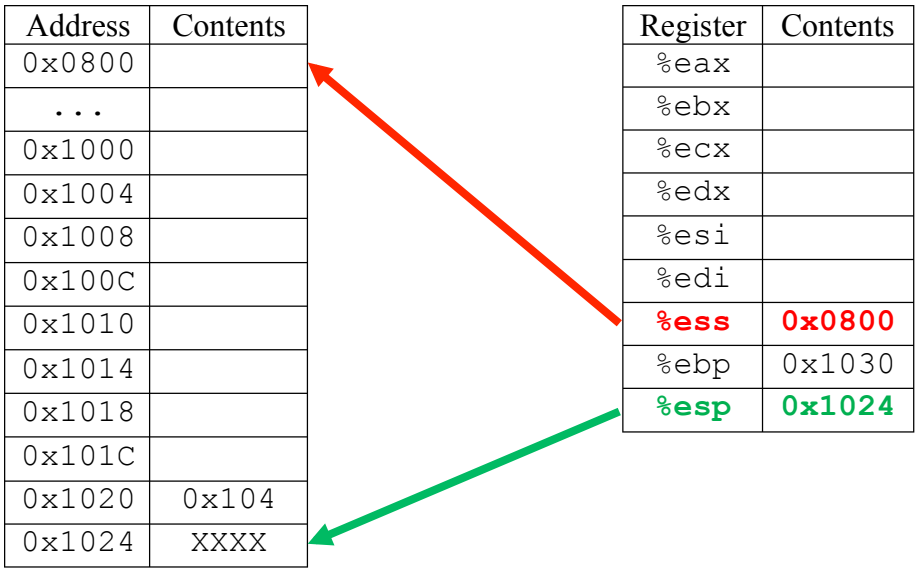


Figure 5.3.5

Notice in **Figure 5.3.5** that the address `0x104` remains on the stack even after the return from `my_func`. This data will only be removed from the stack when it is overwritten by the next function call's return address or other such data saved to the stack. This might seem rather "unclean," but when working in microprocessors, efficiency is key. It is not necessary to clear the data from the stack after it is removed/copied.

As we can see, `ret` relies on the fact that the stack pointer is pointing to the correct address on the stack – the location where the `call` placed the return address data. If `%esp` were to have been accidentally moved to `0x101C`, `ret` would have read the data at that stack location, moved `%esp` to `0x1020`, and attempted to jump to the address it read from location `0x101C`. Note, in the figure above, there does not appear to be any data on the stack at `0x101C` through `0x0800`; however, it is important to know that the stack is *never* initialized or cleared to zero before the program or function is executed. As such, in our latter (and unfortunate case), `ret` will read whatever "garbage" data is at `0x101C` and try to jump or return to the address it read. Chances are, the address `ret` reads is not an address within the code segment of our program's memory. Thus a segmentation fault (a.k.a. seg. fault) will likely occur. When a program encounters a segmentation fault, it has tried to access an unauthorized memory address – an address not within the scope of the program's allotted memory. The Intel 80386, when used in 32-bit or "protected" mode, will stop the operation of a program that accesses other programs' memory/data. This is a great feature, since it prohibits one program from interfering with the execution of another.

As we can see, the stack is a very powerful tool in functional programming, but we have just explored the surface of what the stack is capable of and used for. Let's take a look at how we can not only call and return from functions but also manually store and retrieve data from the stack.

5.4 – Pushing To and Popping From the Stack

As we will see, it is often desirable to use the stack as a place to store data. So, why might we want to store something on the stack? To start, consider the number of general purpose registers within the Intel 80386; there are eight – `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`, and `%esp`. As we know, the first four of these registers are used to perform computations and store results. The next two are indexing registers (discussed in Lab 4), and the final two are registers reserved for use on the stack. (We will discuss `%ebp` in the following section.) What if we need to perform a computation that requires keeping track of more *intermediate values* than the general purpose registers can support? In this case, we could use the stack as a place to store any temporary data in excess of the space available in the general purpose registers.

To write data to the stack, the Intel 80386 processor provides us with the `push` instruction. The **`push` instruction** (or `pushl` for 32-bit values) takes a single argument. This argument is the data to be written to the stack. A `push`, when it is executed, will (1) *subtract* 4 bytes from the stack pointer, and (2) write the argument of `pushl` to the new address contained in the `%esp` register. Notice, this is very similar to the `call` instruction; however, `call` includes a jump to its argument, while `push` simply saves its argument to the stack. We can think of the `call` instruction as being a combination of a `push` and a `jmp`.

Contrary to `push`, to read data from the stack, the processor provides us with the **`pop` instruction**. Like `push`, `pop` (or `popl` for 32-bit values) takes a single argument. This argument is the location where we would like to read or *copy* the data from the stack. `popl` will (1) copy the data at the location in the stack pointer to the argument of `popl`, and (2) *increment* the stack pointer 4 bytes to the next location on the stack.

To illustrate how `push` and `pop` work, let's consider the following situation where we perform a calculation, run out of general purpose registers, and need to save an additional intermediate value. We decide to save the current value of `10` in `%eax` to the stack so that we might reuse `%eax` for another intermediate calculation.

```
0x100:    call my_func
0x104:    ...
0x108:    my_func:
0x10C:         ...
0x110:         pushl %eax
0x114:         ...
0x118:         popl %eax
0x11C:         ...
0x120:         ret
```

Code 5.4.1

Let's assume we have already called `my_func` in **Code 5.4.1**, and the return address `0x104` is on the stack. At address `0x110`, we need to save `%eax` (which contains the number `10`) to the

stack for later use. To do so, we `push` it to the stack, which decrements the stack pointer 4 bytes from `0x1020` to `0x101C` and then writes the contents of the A register to this address on the stack. The following stack representation in **Figure 5.4.1** depicts this `push`:

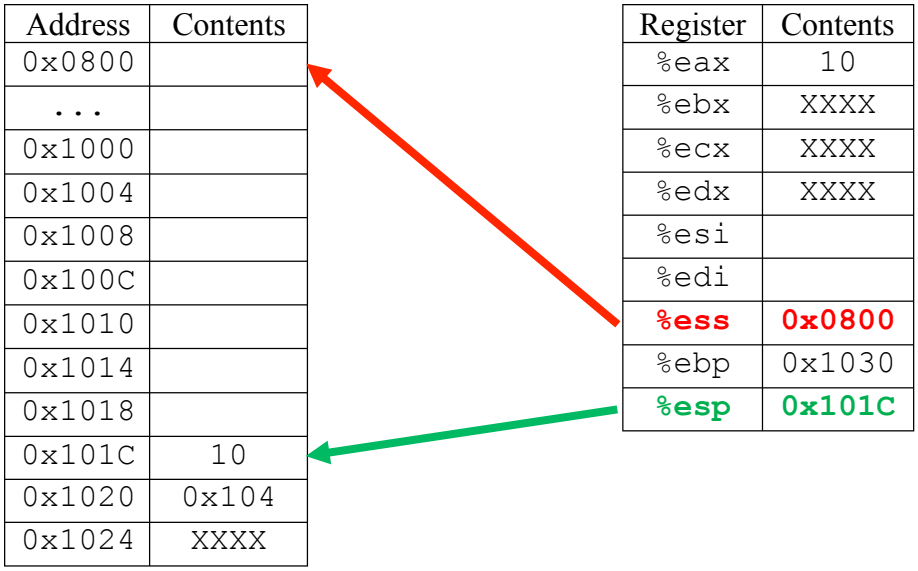


Figure 5.4.1

After the `push`, we are safe to use `%eax` for our additional calculations, with knowledge its original value is safely saved on the stack at address `0x101C`. The calculations are then performed overwriting `%eax` with a value not equal to `10`. Next, our code reaches line `0x118`, where we want to retrieve the A register's previous value to finish our calculation. To do this, we execute a `pop` to `%eax`, which will determine the location of the stack pointer (`0x101C`), read the value stored there (`10`), write this value to the argument of `pop` (`%eax`), and then increment the stack pointer 4 bytes to the next location on the stack.

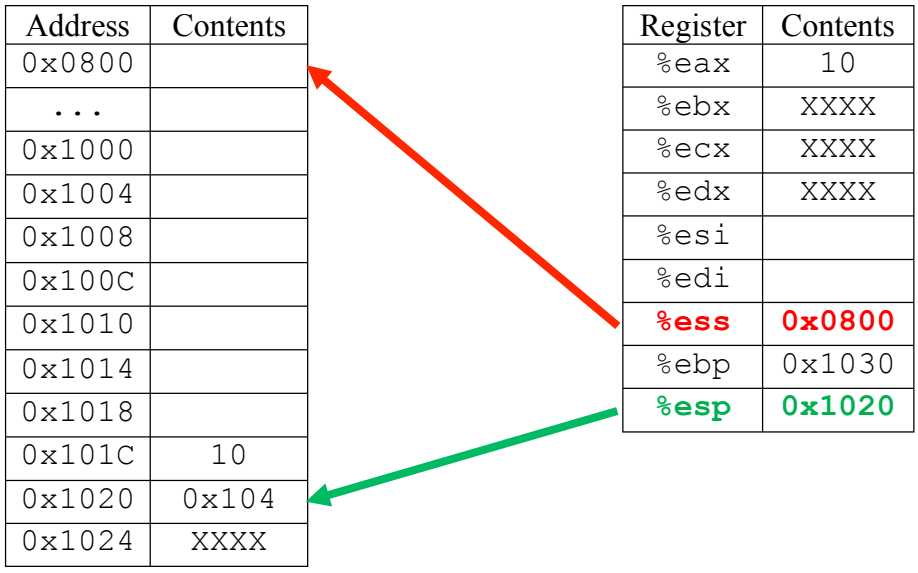


Figure 5.4.2

As we can see in the **Figure 5.4.2** above, the stack pointer is pointing to the return address at the conclusion of the `push` and `pop` instructions. *As a general rule, for every `push` in a function, there must be a corresponding `pop`.* Had we forgotten to `pop` `10` off the stack to `%eax`, not only would our calculation have been wrong, but the `ret` instruction would have tried to return to address `10` in memory, which most certainly is not within the scope of the code segment for our program. This again demonstrates the low-level (or "dumb") nature of assembly instructions – there is no built-in error-checking against assembly code. For example, a `ret` will simply read the address at the stack pointer, increment `%esp`, and then attempt to jump to the address it read. It is our responsibility as assembly programmers to ensure the stack pointer is pointing to the correct data (e.g. the correct return address, in this case).

As we can observe, `push` simply adds a new data value to the top of the stack, and `pop` removes a data value from the top of stack. What makes the stack a stack is that the `push` and `pop` follow a **"last-in, first-out"** or **LIFO** policy. Thus, if we `push` the numbers `10`, `20`, and `30` onto the stack, and then `pop` the stack three times, we will get the numbers `30`, `20`, and `10`, in that order, as shown in **Code 5.4.2** below:

```
pushl $10
pushl $20
pushl $30
...
...
popl %ecx    # put 30 in %ecx
popl %ebx    # put 20 in %ebx
popl %eax    # put 10 in %eax
```

Code 5.4.2

As we have seen and will see in the following sections, the order in which we `push` and `pop` data to and from the stack is crucial in successfully implementing functional programs.

5.5 – Stack Frames: Caller and Callee Responsibilities

As we have discovered, functions are implemented by use of the stack in assembly language. Each function operates in what is called its *stack frame*. A **stack frame** is the address range on the stack a particular function uses. For example, the basic function call and return scenario used at the beginning of this lab has a stack frame of 4 bytes, since the only thing stored on the stack for that particular function was the return address placed there by the `call`. Likewise, the `push` and `pop` example in the previous section has a stack frame of 8 bytes – 4 for the return address placed on the stack by the `call` and 4 for the `push` of the `%eax` register. Stack frames are typically larger than this, since most functions contain local variables stored on the stack. In addition to allocating space for local variables, stack frames contain special data critical to the execution of and successful return to the calling function. The return address is certainly part of this data; however, there are additional pieces to the puzzle of functional programing we have yet to discuss – namely *setting up* and *tearing down* the stack.

To set the stage, let's think back to *nested function calls*. What if a program has data in some registers and needs to `call` a function? It would not be very "fair" (or efficient for that matter) to leave the data in the registers and prohibit the next function from using them. In Intel x86 assembly, to ensure all functions have safe read/write access to the registers in the CPU, both the calling function (the "**caller**") and the function that gets called (the "**callee**") *split the task* of saving CPU register contents *to the stack*. The caller will save data it needs to remember after the function call is complete. By convention, it is the responsibility of the caller to save the contents of the `%eax`, `%ecx`, and `%edx` registers, if the caller has anything in these registers it needs to use after the function call completes. In a similar manner, by convention, it is the responsibility of the callee to save the contents of the `%ebx`, `%ebp`, `%esi`, and `%edi` registers, if it needs to use any of these registers during its execution. This is summarized in **Table 5.5.1** below:

Caller Should Save	Callee Should Save
`%eax`	`%ebx`
`%ecx`	`%ebp`
`%edx`	`%esi`
`---`	`%edi`

Table 5.5.1

The contents of general purpose registers are saved on a *need-only basis*. The reason for this is that it consumes more time and power to save registers that the caller does not care about or that the callee will not overwrite.

Although these facets of the computer are hidden away from us in high-level languages, we as assembly programmers need to keep track of what registers we use and will need to use in the future when conducting function calls and returns. Let's go through an example:

```
0x100:    ...
0x104:    movl varl, %eax
0x108:    movl ptr1, %ebx
0x10C:    movl var2, %edx
0x110:    movl count, %edi
0x114:    ## what do we need to save here?
0x118:    ...
0x11C:    call my_func
0x120:    ...
0x124:    ...
0x128:    movl (%ebx, %edi, 4), %ecx
0x12C:    addl %eax, %edx
...       ...
0x150:    my_func:
0x154:         ## What do we need to save here?
0x158:         movl x, %eax
0x15C:         movl y, %edi
0x160:         ...
0x164:         ret
```

Code 5.5.1

In this scenario, we have an assembly program that is running and needs to call the function `my_func`. The goal is to determine what registers need to be saved at addresses `0x114` and `0x154` of the code given in **Code 5.5.1**.

Let's begin with the caller and complete the code starting at address `0x114`. To do so, let ask the question: What registers contain data prior to the function `call` and are used after it? In other words, *pretending the function `call` is not even there*, what registers after the `call` *rely on* data present in the same registers before the `call`? We can see that the code after the `call` uses registers `%ebx` (`ptr1`), `%edi` (`count`), `%eax` (`var1`), and `%edx` (`var2`). These two lines of code operate under the assumption that the registers will not be modified during the function `call`. Recall that the caller must save `%eax`, `%ecx`, and `%edx` if it will need their contents after the `call`; thus, we need to `push` `%eax` and `%edx` to the stack before we `call` `my_func`. Notice we do not need to save `%ecx`. Although it is used after the function call, it is a destination, and data is not read from it to compute a result. So, the following in **Code 5.5.2** is required by the caller:

```
0x114:    pushl %eax   ## needed after the function call
0x118:    pushl %edx   ## needed after the function call
0x11C:    call my_func
```

Code 5.5.2

We're halfway there; now let's work on what the callee needs to save. By convention, the callee is required to save `%ebx`, `%ebp`, `%esi`, and `%edi`, if it will need to use them. As we can see in `my_func`, the `%eax` and `%edi` registers are overwritten; thus, the callee needs to include a

`push` of the `%edi` register. Notice that the `%eax` register is not saved – that is the responsibility of the caller. Also, notice that the `%ebx` register is used by the caller after the function return but not pushed by the callee. Since the callee does not overwrite `%ebx`, this register will still contain `ptr1` after the function return. As such, only one `push` is necessary in the callee, as shown in **Code 5.5.3**:

```
0x150:    my_func:
0x154:         pushl %edi
```

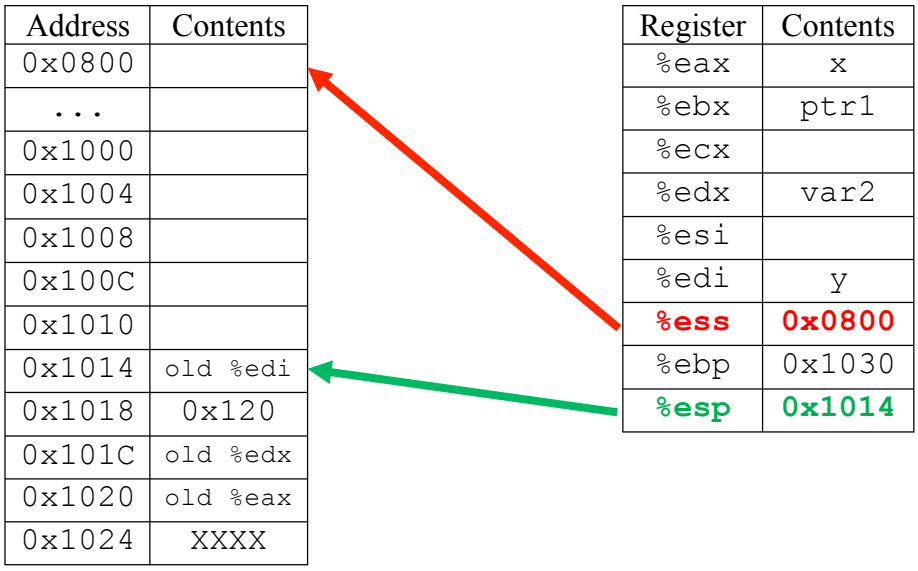
Code 5.5.3

Putting it all together, we obtain **Code 5.5.4**:

```
0x100:    ...
0x104:    movl varl, %eax
0x108:    movl ptr1, %ebx
0x10C:    movl var2, %edx
0x110:    movl count, %edi
0x114:    pushl %eax
0x118:    pushl %edx
0x11C:    call my_func
0x120:    ...
0x124:    ...
0x128:    movl (%ebx, %edi, 4), %ecx
0x12C:    addl %eax, %edx
...       ...
0x150:    my_func:
0x154:         pushl %edi
0x158:         movl x, %eax
0x15C:         movl y, %edi
0x160:         ...
0x164          ret
```

Code 5.5.4

with its corresponding stack and registers, current as of line `0x160`:



Address	Contents
0x0800	
...	
0x1000	
0x1004	
0x1008	
0x100C	
0x1010	
0x1014	old %edi
0x1018	0x120
0x101C	old %edx
0x1020	old %eax
0x1024	XXXX

Register	Contents
%eax	x
%ebx	ptr1
%ecx	
%edx	var2
%esi	
%edi	y
%ess	**0x0800**
%ebp	0x1030
%esp	**0x1014**

Figure 5.5.1

Notice in **Figure 5.5.1**, the `push %edx` was actually not necessary, since `var2` in `%edx` was not changed during the execution of `my_func`. However, the caller needs to save any register (of those it is responsible for) if it has any important data in it. This model operates under the assumption that the caller does not have knowledge of how the callee is written – a "better be safe than sorry" model. In fact, this is very accurate for library or system calls of which the internal code is often hidden from users; so, we cannot assume certain registers will not be used or overwritten.

Now, let's expand this idea to form a fully functional solution, since what we have done so far has omitted a key component. We know that both caller and callee subroutines need to save some of the general purpose registers, if they are important or will be overwritten, respectively. The question becomes, how do we *restore* what we saved? For example, if the caller needs to save the `%eax` and `%edx` registers to the stack and the callee needs to save `%edi` (as in our example above), how can we put the values we saved back in the registers when the function `my_func` returns?

Recall that the stack is a *LIFO data structure*. This means that the last item to be pushed or saved to the stack must be the first to be removed or popped. Let's step through what we have done so far and see what is missing. As shown in **Code 5.5.4**, before the call to `my_func`, we `push %eax`, then `%edx`, and then we `call my_func`, which automatically pushes the return address to the stack. Following this, we `push %edi` to the stack when executing `my_func`. The next stack operation we see in our code is the `ret` at address `0x164`. Remember, `ret` does not know to search for and read the *correct* return address from the stack. It simply reads the data at the address in the stack pointer register. Recall that the stack pointer is decremented for each `push` or `call` and incremented for each `pop` or `ret`. Thus, based on our code so far, `%esp` will point to the old value of `%edi`, which was the last thing we saved to the stack. `ret` will attempt to

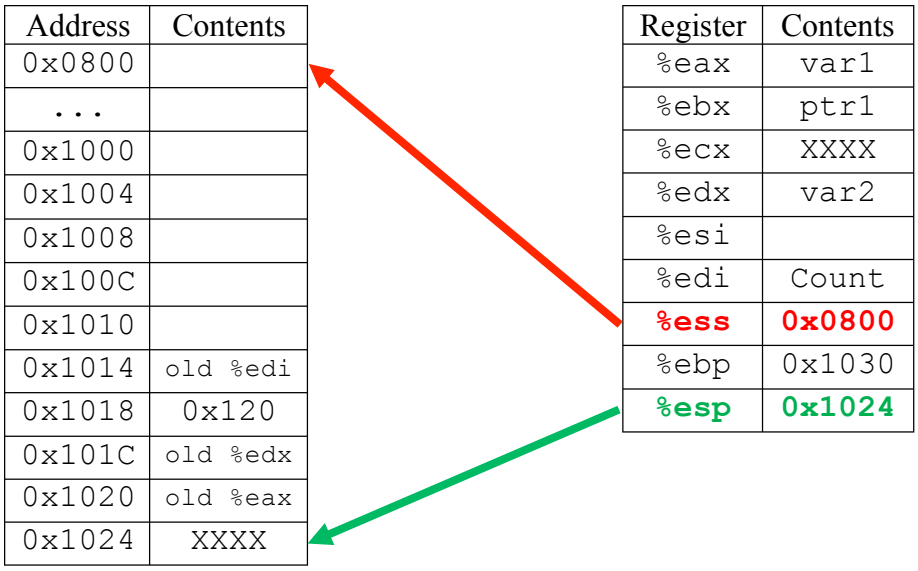
jump to the old value of `%edi`, which is not what we want to happen. As shown in the stack in the previous figure, **Figure 5.5.1**, the correct return address is actually 4 bytes higher on the stack than the old value of `%edi`, so we need to first remove or `pop` this old value of `%edi` from the stack (which will increment `%esp` 4 bytes) in order to access the return address with the `ret` instruction.

Likewise, after the return from `my_func`, we need to restore the `%eax` and `%edx` registers we pushed. To do so, we must follow the LIFO policy and `pop` the `%edx` register, then `pop` the `%eax` register. Like `ret`, `pop` will only remove the data the stack pointer is currently pointing to, then increment `%esp` by 4 bytes to the next element on the stack. Continuing our example, after the `ret` jumps to the return address, the stack pointer is pointing to our old value of `%edx`, so we `pop` it off, followed by `%eax`, which results in the assembly given in **Code 5.5.5**:

```
0x100:    ...
0x104:    movl varl, %eax
0x108:    movl ptr1, %ebx
0x10C:    movl var2, %edx
0x110:    movl count, %edi
0x114:    pushl %eax
0x118:    pushl %edx
0x11C:    call my_func
0x120:    popl %edx   ## By LIFO, this must mirror the pushes
0x124:    popl %eax
0x128:    movl (%ebx, %edi, 4), %ecx
0x12C:    addl %eax, %edx
...       ...
0x150:    my_func:
0x154:         pushl %edi
0x158:         movl x, %eax
0x15C:         movl y, %edi
0x160:         popl %edi   ## Only 1; LIFO order does not matter
0x164          ret
```

Code 5.5.5

with the following stack, current from address `0x12C`:



Address	Contents
0x0800	
...	
0x1000	
0x1004	
0x1008	
0x100C	
0x1010	
0x1014	old %edi
0x1018	0x120
0x101C	old %edx
0x1020	old %eax
0x1024	XXXX

Register	Contents
%eax	var1
%ebx	ptr1
%ecx	XXXX
%edx	var2
%esi	
%edi	Count
%ess	**0x0800**
%ebp	0x1030
%esp	**0x1024**

Figure 5.5.2

Notice in **Figure 5.5.2**, the stack still contains the data we just popped off it, but the top of the stack (`%esp`) is back where it was before any of our code executed (`XXXX` at address `0x1024`). Also note, since we pushed/saved and popped/restored our registers, they have been persevered for each stack frame (i.e. each function `call`). *This successfully implements the idea that each function should have access to all general purpose registers without the concern of overwriting a prior function's data.*

5.6 – Stack Frames: The Prolog, Epilog, and Local Variables

The act of saving registers and restoring them *within a given function* is significant enough to have its own terminology. These lines of code are called the prolog and the epilog of a function. The purpose of the subroutine **prolog** is to "set up" the stack frame for the function – meaning save applicable registers and allocate space for local variables. The purpose of the function **epilog** is to "tear down" the stack frame for the function – meaning de-allocate local variable space, restore registers saved in the prolog, and finally return from the function. The prolog and epilog skeleton of a function is shown below in **Code 5.6.1**:

```
my_func:
     ## prolog: push registers and get space for local var's
     ...
     ...
     ...
     ## epilog: restore local var's space and pop registers
     ret
```

Code 5.6.1

We have already discussed saving and restoring general purpose registers; however we have not discussed *local variables*.

If we look back to the programs we have written thus far in Lab 1 through Lab 4, each of these assignments have used *global variables* to save data from registers. As described earlier in **Section 5.2 – An Introduction to the Stack**, global variables and data are stored in the data segment of memory. In addition, global variables are allocated at compile-time, while local variables are allocated during run-time. As we know from C and other functional programming languages, local variables apply only to the scope of the function or the body of loops they are declared within. An interesting aspect about local variables is that there can be more than one instance of a given local variable name in a program; this is not true for global variables, which must be uniquely identifiable within the scope of the entire program. Thus, it follows that these variables of limited scope be stored someplace only the function that uses them will access – the stack.

The stack, as we know, is a contiguous segment of memory used by subroutines for their successful execution. It has a top and a bottom. The top of the stack is indicated by the stack pointer, and as data is added to the stack, `%esp` slowly approaches the bottom of the stack (indicated by `%ess`).

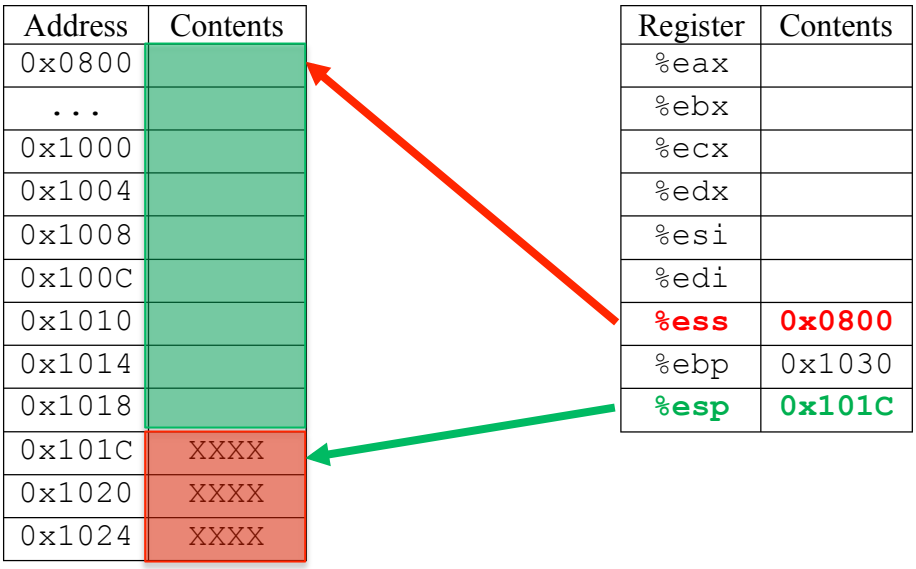


Figure 5.6.1

In the stack in **Figure 5.6.1** above, we can see there are three items present on the stack, which are shaded in **red**. These are pieces of data saved to the stack by the current or perhaps a previous subroutine, and thus should not be overwritten. By nature of the stack, the addresses below (lower than) the stack pointer are available to store data and are shaded in **green**. As the stack pointer gets closer to the beginning of the stack segment, as items are pushed or function calls are made, there is a decreasing amount of available space on the stack, as seen below in **Figure 5.6.2**. This can occur when many sequential stack frames are created but not torn down until a later time, as is often done in recursive programming with nested function calls. This is not a bad condition to encounter in a program. What is of concern is running out of stack space.

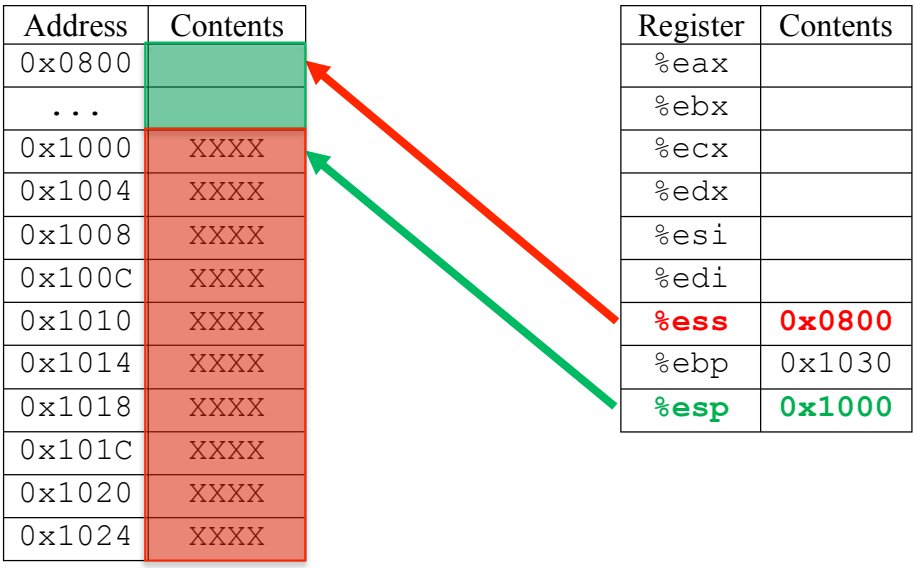


Figure 5.6.2

When the stack pointer reaches and attempts to decrement lower than the stack segment, there is no more space on the stack, and the program encounters a **stack overflow**, shown below in **Figure 5.6.3**. A stack overflow is a fatal event, which results in the termination of the program by the operating system.

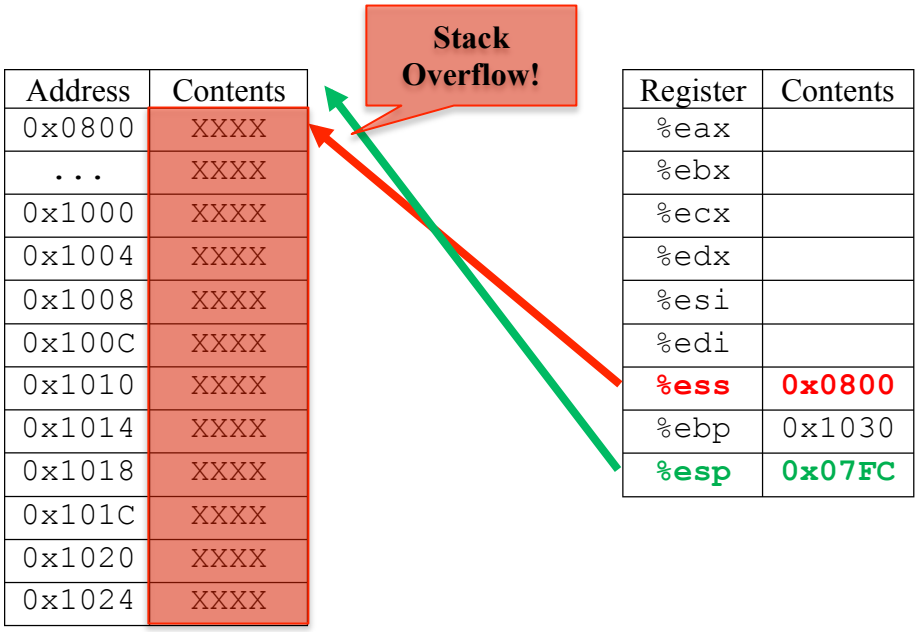


Figure 5.6.3

On the other hand, as data is removed from the stack (items are popped, rets are executed, etc), the stack pointer moves farther away from the beginning of the stack segment, which means there is an increasing amount of available space on the stack. This behavior can be seen by viewing **Figures 5.6.1** and **5.6.2** in reverse order.

So, what do the previous figures have to do with local variables? As we mentioned before, local variables are stored on the stack. In order to use the stack as a space to store local variables, we need to understand that the free space on the stack is located below (at a lower address from) the stack pointer. We can take advantage of this free space to use for local variables by manually decrementing the stack pointer into this unused space. To aid in this process, we have another general purpose register at our disposal – the **base pointer register** or **`%ebp`**. The idea is simple – (1) save the contents of `%ebp` to the stack, (2) save the stack pointer address in `%ebp`, (3) subtract from the stack pointer the number of bytes required to store the local variable(s), and (4) access these variables via an offset to either the base pointer register or the stack pointer register. This is a rather difficult concept to explain without an example, so consider the following...

Let's say we have a function that requires three local variables that are integers – `int a`, `int b`, and `int c`. This means, we will need to get some space on the stack to store these integers. How much space do we need? For three integers, we need 3 integers * 4 bytes/integer = `12 bytes`. As shown in **Figures 5.6.1** and **5.6.2**, on the stack, unused memory is located at addresses below the top of the stack – pointed to by the stack pointer, `%esp`. So, we can subtract `12` from `%esp` to

move the top of the stack down three integer's worth of space. The addresses of this newly allocated memory will become synonymous with these three local variables.

```
my_func:
     ## prolog: push registers and get space for local var's
     pushl %ebx         ## used in ...
     pushl %ebp         ## save prior stack frame's %ebp value
     movl %esp, %ebp    ## set %ebp = %esp = 0x1010
     subl $12, %esp     ## get space for 3 integers
     ...
```

Code 5.6.2

Consider a stack like **Figure 5.6.1** prior to the execution of `my_func`. With all functions, the first event that occurs on the stack is the return address written by the `call` (designated as `ret` addr in **Figure 5.6.4** below). In **Code 5.6.2**, following the `call`, `my_func` (somewhere in `...`) requires the use of `%ebx`, so its register contents need to be pushed to the stack in the prolog. (Recall, the callee is required to save `%ebx`, `%ebp`, `%edi`, and `%esi`, as necessary.) Next, we need to get the stack ready for the three integers, which are local variables. To do so, we first `push` the base pointer register's contents to the stack. Since local variables will require the use of `%ebp`, we need to save its contents from the previous stack frame in the prolog.

After all the push instructions are complete, the stack pointer is pointing to the last item pushed (address `0x1010` in **Figure 5.6.4**). We are now ready to get space for `a`, `b`, and, `c`. We first need to mark where `%esp` is currently pointing. This is done by moving the contents of `%esp` into `%ebp`. (Note, we have saved `%ebp` in our prolog, so it is acceptable to write a new value in it.) Now, to get the space we need for our local variables, we need move the top of the stack `12 bytes` from its current location (`0x1010`); however, we do not want to overwrite our prolog – the old values of `%ebp`, `%ebx`, and the return address. So we need to *subtract 12 bytes from the stack pointer*, which will move it down the stack to address `0x1004` – computed from `0x1010` minus `0x00C` (decimal `12`). In essence, the stack addresses below `%ebp` to the limit of `%esp` – `0x100C`, `0x1008`, and `0x1004` – are now reserved for local variables `a`, `b`, and `c`, respectively.

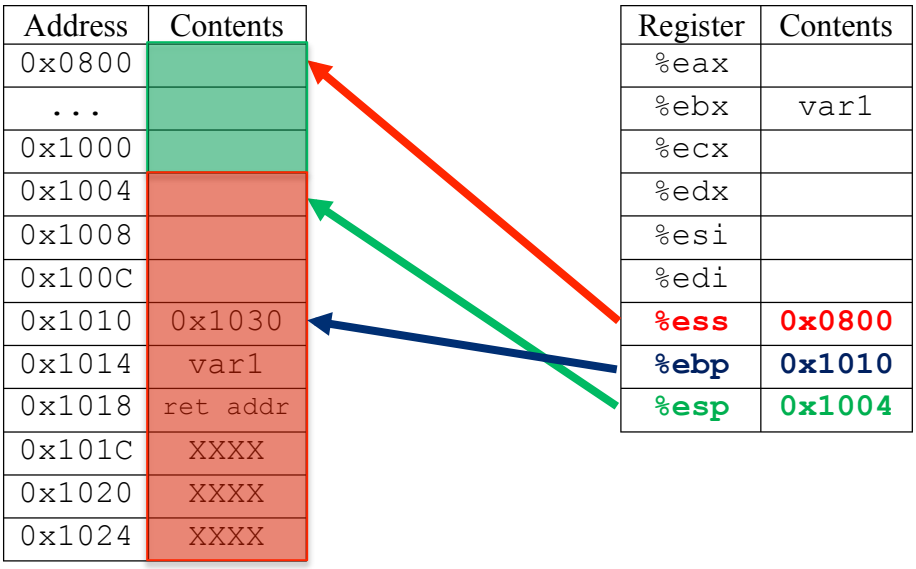


Figure 5.6.4

The question then becomes: *How do we access this newly acquired space?* Recall, in Lab 4 we discussed various addressing modes of the Intel 80386 processor. A couple of these techniques applied to pointer and arrays – namely register indirect addressing and base-indexed addressing. As such, it is worth noting the array-like structure of the stack. Notice how the stack is a sequence of addresses, each of which has some data stored in it. Also, notice the pointers that are used with the stack – `%esp` and `%ebp`. We can apply either of these addressing technique to the stack and its registers in order to access or modify stack contents.

Continuing our example, let's say we want to perform the following C operation shown in **Code 5.6.3** in assembly:

```
my_func() {
      int a;
      int b;
      int c;
      a = 0;
      b = 10;
      c = b;
      ...
}
```

Code 5.6.3

In assembly, local variables are not given alphanumeric (i.e. human-readable) names. They are referred to by their addresses on the stack. Refer to **Figure 5.6.4** – the addresses `0x100C`, `0x1008`, and `0x1004` correspond to variables `a`, b, and `c`, respectively. The first variable to be declared is the first to receive space on the stack. In our example, `a` was declared before `b`, which was declared before `c`. This means, when we subtract `12` bytes from `%esp`, the first 4 bytes allocated (`0x100F` through `0x100C`) are used for our first integer – `int a`. Likewise, bytes 5 –

8 allocated (0x100B through 0x1008) are used for our second integer – int b. And finally, bytes 9 – 12 of the 12 bytes allocated (0x1007 through 0x1004) are used for our third and last integer – int c. The function that uses these local variables (my_func) must be written to refer to these addresses in order to use any of the local variables – *they cannot be referenced by name*.

With this information, let's now convert **Code 5.6.3** to assembly. We already have the space allocated, as shown in the prolog in **Code 5.6.2**, so all we need to do is perform the assignments. Variable a is the first to be declared, so it is at an offset of 4 bytes from the pointer %ebp, as shown in **Figure 5.6.4**. Which form of addressing could we use to reference this memory reserved for int a? Register indirect addressing fits the bill perfectly. Recall that register indirect addressing is used for pointers, where the address of the data's location is in a register and is dereferenced using parenthesis with an optional offset. So to set a = 0, we can write the following:

```
my_func:
     ...
     movl $0, -4(%ebp)     ## set a = 0
     ...
```

Code 5.6.4

Notice the -4 in **Code 5.6.4**. The base pointer register is pointing to address 0x1010 on the stack. The memory we reserved for our local variables starts at address 0x100C, which is below %ebp. A positive 4 would move in the direction of the registers saved in the prolog, which could overwrite important data and later cause an error. Now, let's use the same methodology to set b = 10:

```
my_func:
     ...
     movl $0, -4(%ebp)     ## set a = 0
     movl $10, -8(%ebp)    ## set b = 10
     ...
```

Code 5.6.5

As we can see in **Figure 5.6.4**, variable b's space is an additional 4 bytes from the memory used for variable a on the stack, which means it is a total of 8 bytes lower than the address in register %ebp. Now, to set c = b, we need to access b, then set the address of c equal to it. To do this, we need to copy the contents from stack address 0x1008 (the address of b) to address 0x1004 (the address of c). We will certainly need to use the mov instruction, but be careful – at least one argument to any instruction must be a register or constant. So, the act of c = b actually takes two assembly instructions, as seen in **Code 5.6.6**:

```
my_func:
     ...
     movl $0, -4(%ebp)     ## set a = 0
     movl $10, -8(%ebp)    ## set b = 10
     movl -8(%ebp), %ebx   ## use B register as intermediate
     movl %ebx, -12(%ebp) ## set c = b
     ...
```

Code 5.6.6

We are almost done – so far we have allocated space for local variables and accessed these local variables on the stack. To finish up, we need to de-allocate the stack space we have used for `a`, `b`, and `c`. In other words, we need to set `%esp` back to where it was before we manually moved it `12` bytes. Why is this so? Well, if we were to progress right into the epilog and `pop` the registers we saved in the prolog, `pop` would reference the current location of the stack pointer, and put variable `c` into `%ebp`, then variable `b` into `%ebx`, and finally `ret` would attempt to return to the address `0` – the value of `a`. This would certainly result in a segmentation fault. To avoid this problem, we must relocate `%esp` back to where it was after the last `push` in the prolog (address `0x1010`). This way, the `pop` and `ret` instructions will reference the correct stack addresses each time. Thus, we need to insert a `mov %ebp, %esp` before any of the pops in the epilog:

```
my_func:
     ## Prolog ##
     pushl %ebx
     pushl %ebp
     movl %esp, %ebp
     subl $12, %esp

     ## Function Code ##
     movl $0, -4(%ebp)
     movl $10, -8(%ebp)
     movl -8(%ebp), %ebx
     movl %ebx, -12(%ebp)

     ## Epilog ##
     movl %ebp, %esp
     popl %ebp
     popl %ebx
     ret
```

Code 5.6.7

The full implementation of this example is shown in **Code 5.6.7**. Note how the epilog is a mirror image of the prolog – the last register pushed is the first to be popped. As discussed, this is due to the LIFO nature of the stack. Likewise, notice how in the prolog we save the stack pointer in the base pointer register, and in the epilog we restore the original stack pointer from the base pointer by copying it back into `%esp`. This essentially de-allocates the space we attained on the stack for

the local variables, *no matter how much space we used*. In this case, the stack pointer is incremented by `12` bytes by simply moving the base pointer's value into it.

As we saw from this example, register indirect addressing allows us to read from and write to local variables. This idea can be extended to work with base-indexed addressing as well. The only difference is the inclusion of an indexing register – typically `%esi` or `%edi`. One might wish to use this form of addressing over register-indirect when working with loops and local variables. **Code 5.6.8** below is another way to implement **Code 5.6.7** using base-indexed addressing and a loop.

```
my_func:
     ## prolog: push registers and get space for local var's
     pushl %edi
     pushl %ebx
     pushl %ebp
     movl %esp, %ebp
     subl $12, %esp
     movl $-1, %edi
     loop:
          set_a:
               cmpl $-1, %edi
               jne set_b
               movl $0, (%ebp, %edi, 4)    ## set a = 0
               decl %edi
               jmp loop
          set_b:
               cmpl $-2, %edi
               jne set_c
               movl $10, (%ebp, %edi, 4)   ## set b = 10
               decl %edi
               jmp loop
          set_c:
               incl %edi  ## get b (-2)
               movl (%ebp, %edi, 4), %ebx  ## set %ebx = b
               decl %edi  ## go back to c (-3)
               movl %ebx, (%ebp, %edi, 4)  ## set c = %ebx
     done:
     ...
```

Code 5.6.8

As we can see, this implementation in **Code 5.6.8** is of course not the most practical or efficient method for this simple example. However, it illustrates how base-indexed addressing can be used with the stack. In either form of referencing data on the stack, we may use either the base pointer or the stack pointer register with an offset to access local variables. Our examples have focused on accessing local variables by applying a negative offset to `%ebp`. The same principle can be applied to `%esp`; however instead of subtracting from `%esp`, we would need to add a positive offset to

access each variable. Also note, if this latter approach is used, the first variable (`a` in our case) will be the farthest away from `%esp`, meaning to access `int a`, we would need to add an offset of `0` to `%esp`, to access `int b`, we would need to add an offset of `4` to `%esp`, and finally to access `int c`, we would need to add an offset of `8` to `%esp`. The equivalence of these two methods can be verified by examining **Figure 5.6.4** once more and applying the offsets.

So, in summary of this section, the prolog and epilog are essential features of any assembly language subroutine or function. They work together to initialize the stack for the function and then restore the stack to its original condition after the function executes. The prolog optionally saves the `%ebx`, `%ebp`, `%edi`, and `%esi` general purpose registers by pushing them to the stack. (`%eax`, `%ecx`, and `%edx` are saved by the caller.) These registers only need to be saved if the function that is about to execute will overwrite their contents. This is important, since we want the caller to have its original data in these registers when the function returns. In any case, all registers that are pushed to the stack in the prolog need to be popped from the stack in the epilog. This is done in reverse (or mirror) order from the prolog, due to the LIFO nature of the stack. Finally, the prolog is also used to allocate space on the stack for local variables. This is done after all registers have been saved and is implemented by subtracting the number of bytes required for the local variables from the stack pointer register. This effectively increases the stack space of the function's stack frame. Local variables are accessed on the stack using either `%ebp` or `%esp` in register indirect or base-indexed addressing.

5.7 – Putting it all Together

Now that we have discussed function calls and returns, pushing and popping data, saving registers to the stack, allocating and working with local variables, and the prolog and the epilog of subroutines, let's apply this newfound knowledge to an example where we have a couple C functions that are called within a program:

```
int a = 0;
int b = 20;
int sum = 0;
int avg = 0;
...
...
...
// Calculate the average of numbers a and b
get_average();
...
...
...
void get_average(void) {
     int quantity;
     get_sum();
     quantity = 2;
     avg = sum / quantity;
     return;
}
...
...
...
void get_sum(void) {
     int a_loc = a;
     int b_loc = b;
     sum = a_loc + b_loc;
     return;
}
```

Code 5.7.1

In **Code 5.7.1**, we use global variables and function calls to calculate the average of `a` and `b`. The code calls `get_average()`, which in turn calls `get_sum()` in order to compute the average. In general (and as many of us have been taught) the use of global variables should be limited to those cases in which we absolutely need them. We would normally get around this by writing `average = get_average(int a, int b)` and `sum = get_sum(int a, int b)`, where `average` and `sum` each take arguments and return values; however, we have not discussed how to implement parameters and returns in assembly – this is a topic for **Lab 6**. So, for **Lab 5**, even though it is not a good programming practice in general, we will use global variables. Also note the unnecessary use of local variables in `get_average()` and `get_sum()`. We will

use local variables in this example to demonstrate how we could use them in a more complicated situation.

So, let's get to it! Let's start by declaring and initializing our global variables. These are stored on the data segment in memory, which as depicted in **Figure 5.3.2** is in a different memory segment than the stack. For the purposes of this example, we will give variables in the data segment addresses starting with `0x8XXX`, assembly instructions in the code segment with memory addresses starting with `0x1XXX`, and stack data in the stack segment with addresses starting with `0x3XXX`. It is important to distinguish between the location of code, global variables, and stack data in RAM.

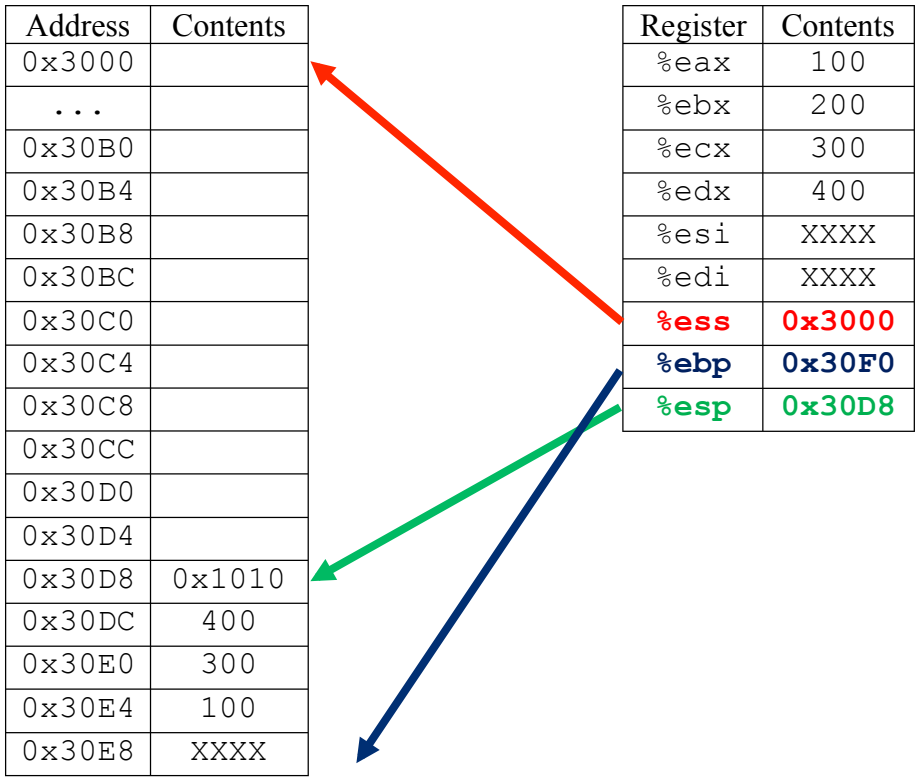
```
0x8000:      a:
                  .int 0
0x8004:      b:
                  .int 20
0x8008:      sum:
                  .int 0
0x800C:      avg:
                  .int 0
...          ...
...          ...
...          ...
```

Code 5.7.2

Now, let's add in the `call` to `get_average`. Since we do not know what happens in-between the function `call` and the global variable declarations, we should assume there is data important to the caller in the general purpose registers. Recall that by convention, it is the responsibility of the caller to save the `%eax`, `%ecx`, and `%edx` registers, if it needs the data in them when the function it is about to `call` returns (**Table 5.5.1**). As such, let's save these registers before we `call get_average`.

```
0x8000:     a:
                 .int 0
0x8004:     b:
                 .int 20
0x8008:     sum:
                 .int 0
0x800C:     avg:
                 .int 0
...         ...
...         ...
...         ...
...         ## Save caller's responsible registers, if necessary
0x1000:     pushl %eax
0x1004:     pushl %ecx
0x1008:     pushl %edx
0x100C:     call get_average
...         ...
```

Code 5.7.3



Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	
0x30C0	
0x30C4	
0x30C8	
0x30CC	
0x30D0	
0x30D4	
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	100
%ebx	200
%ecx	300
%edx	400
%esi	XXXX
%edi	XXXX
%ess	0x3000
%ebp	0x30F0
%esp	0x30D8

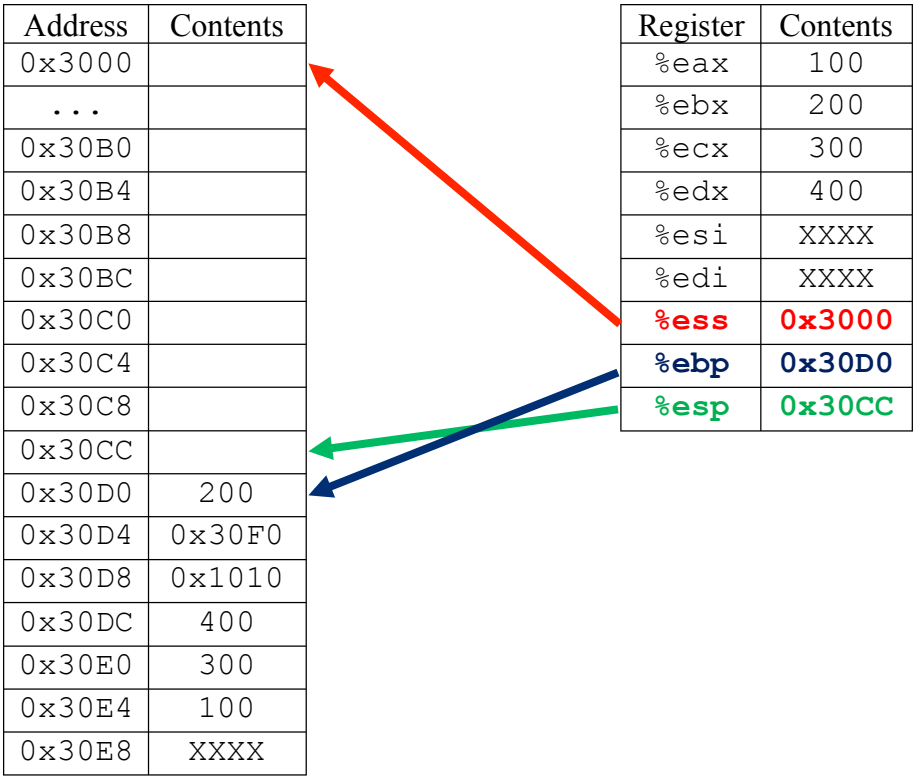
Figure 5.7.1

As we can see in **Figure 5.7.1**, the stack is now ready to execute `get_average`. Next, let's write the function prolog for `get_average` shown in **Code 5.7.4** and **Figure 5.7.2**:

```
...         ...
```

```
...             ...
...             ...
0x1040:         get_average:
                     ## Prolog: save callee's responsible
                     ## registers, if necessary, and get space
                     ## for local variables
0x1044:              pushl %ebp
0x1048:              pushl %ebx
0x104C:              movl %esp, %ebp
0x1050:              subl $4, %esp
...                  ...
```

Code 5.7.4



Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	
0x30C0	
0x30C4	
0x30C8	
0x30CC	
0x30D0	200
0x30D4	0x30F0
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	100
%ebx	200
%ecx	300
%edx	400
%esi	XXXX
%edi	XXXX
%ess	**0x3000**
%ebp	**0x30D0**
%esp	**0x30CC**

Figure 5.7.2

We will use the `%ebp` and `%ebx` registers after the function `call`, so as the callee, we must save them to the stack before we overwrite them. We will also use an integer local variable `quantity`, which is 4 bytes. As such, we must make room for this local variable on the stack by subtracting `4` from the stack pointer register.

Next, we need to `call get_sum` from within `get_average`. There is nothing important to `get_average` in the `%eax`, `%ecx`, or `%edx` registers, so we do not need to save them this time. **Code 5.7.5** and **Figure 5.7.3** show the code and stack at this point in the program's execution:

```
...             ...
...             ...
...             ...
0x1040:         get_average:
                     ## Prolog: save callee's responsible
                     ## registers, if necessary, and get
                     ## space for local variables
0x1044:              pushl %ebp
0x1048:              pushl %ebx
0x104C:              movl %esp, %ebp
0x1050:              subl $4, %esp
                     ## Call get_sum; no need to save registers
0x1054:              call get_sum
0x1058:              ...
```

Code 5.7.5

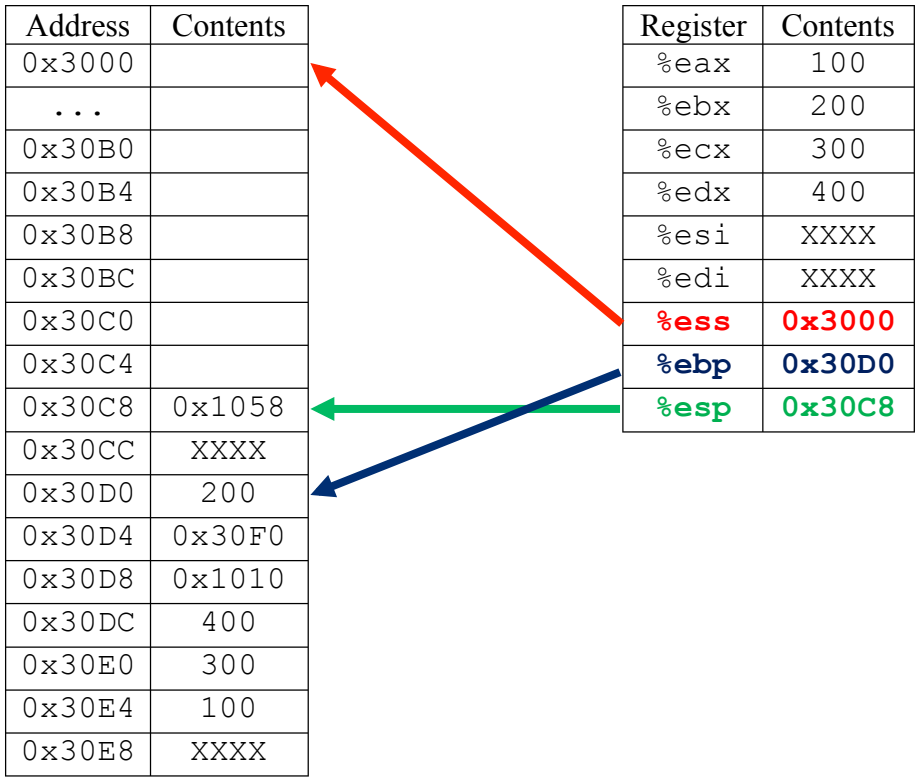


Figure 5.7.3

Now, we're ready to write the `get_sum` function. Let's say it's at address `0x1080` in the code segment of memory. In `get_sum`, we will need to save the `%ebp` register, since we have two local variables which will require `%ebp` to be rewritten. We will not use the `%ebx` register or either of the two indexing registers, so we do not need to save them. **Code 5.7.6** shows the initial assembly code for `get_sum`, and **Figure 5.7.4** depicts the corresponding stack:

```
...             ...
...             ...
...             ...
0x1080:         get_sum:
                     ## Prolog: save callee's responsible
                     ## registers, if necessary, and get
                     ## space for local variables
0x1084:              pushl %ebp
0x1088:              movl %esp, %ebp
0x108C:              subl $8, %esp
...                  ...
```

Code 5.7.6

Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	
0x30C0	
0x30C4	0x30D0
0x30C8	0x1058
0x30CC	XXXX
0x30D0	200
0x30D4	0x30F0
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	100
%ebx	200
%ecx	300
%edx	400
%esi	XXXX
%edi	XXXX
%ess	**0x3000**
%ebp	**0x30C4**
%esp	**0x30BC**

Figure 5.7.4

Now, we're ready to do some calculations. Following the specification code in **Code 5.7.1**, we need to move global variable `a` into local variable `a_loc` and global variable `b` into local variable `b_loc`. Since `a_loc` is declared before `b_loc`, it is the highest address/location in the `8` bytes

of space we obtained on the stack. After we set our local variables, we want to compute the sum. **Code 5.7.7** and **Figure 5.7.5** depict this portion of the program:

```
...             ...
...             ...
...             ...
0x1080:         get_sum:
                     ## Prolog: save callee's responsible
                     ## registers, if necessary, and get
                     ## space for local variables
0x1084:              pushl %ebp
0x1088:              movl %esp, %ebp
0x108C:              subl $8, %esp
                     ## Set local variables and computer sum
0x1090:              movl a, %eax
0x1094:              movl b, %ecx
0x1098:              movl %eax, -4(%ebp)
0x109C:              movl %ecx, -8(%ebp)
0x10A0:              movl -4(%ebp), %edx
0x10A4:              addl -8(%ebp), %edx
0x10A8:              movl %edx, sum
...                  ...
```

Code 5.7.7

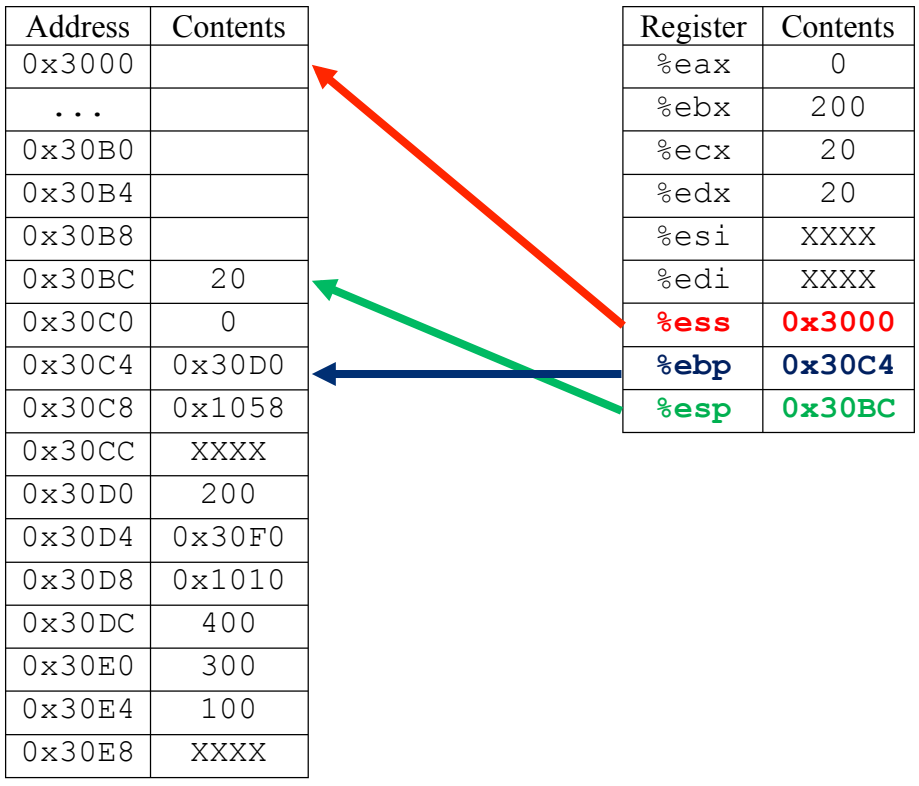
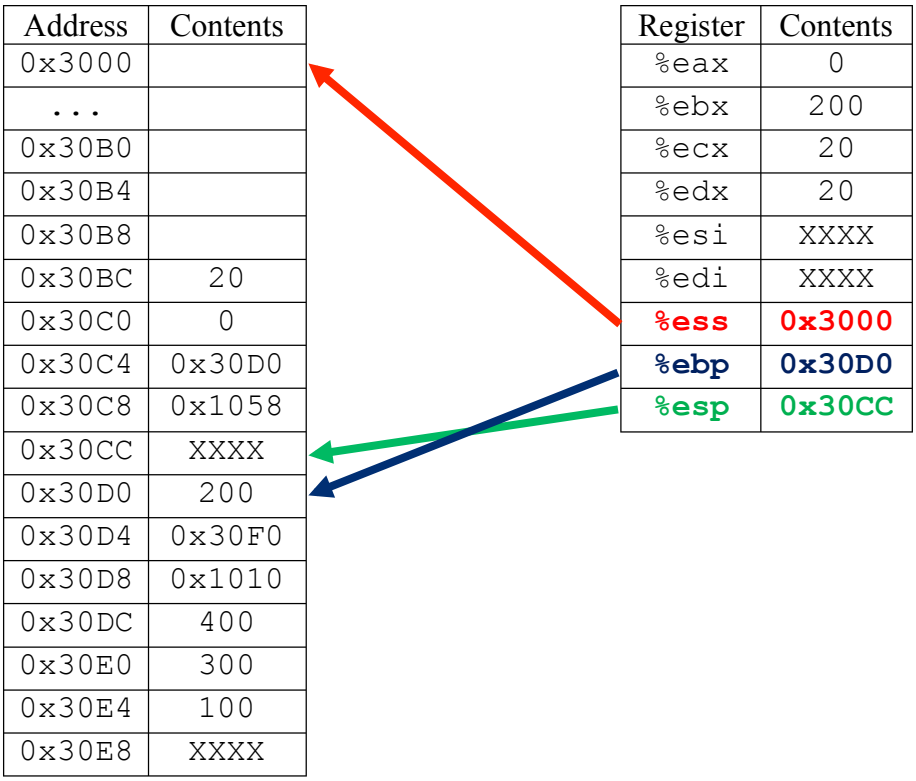


Figure 5.7.5

Before we return from `get_sum`, we need the epilog to undo what the prolog did. We must first de-allocate the space used for our local variables, and then `pop` off any registers we saved to the stack. At this point we can return to the caller – `get_average`. **Code 5.7.8** and **Figure 5.7.6** show the inclusion of the epilog to `get_sum`:

```
...           ...
...           ...
...           ...
0x1080:       get_sum:
                   ## Prolog: save callee's responsible
                   ## registers, if necessary, and get
                   ## space for local variables
0x1084:            pushl %ebp
0x1088:            movl %esp, %ebp
0x108C:            subl $8, %esp
                   ## Set local variables and computer sum
0x1090:            movl a, %eax
0x1094:            movl b, %ecx
0x1098:            movl %eax, -4(%ebp)
0x109C:            movl %ecx, -8(%ebp)
0x10A0:            movl -4(%ebp), %edx
0x10A4:            addl -8(%ebp), %edx
0x10A8:            movl %edx, sum
                   ## Epilog: restore space for local var's,
                   ## restore registers, and return
0x10AC:            movl %ebp, %esp
0x10B0:            popl %ebp
0x10B4:            ret
```

Code 5.7.8



Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	20
0x30C0	0
0x30C4	0x30D0
0x30C8	0x1058
0x30CC	XXXX
0x30D0	200
0x30D4	0x30F0
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	0
%ebx	200
%ecx	20
%edx	20
%esi	XXXX
%edi	XXXX
%ess	**0x3000**
%ebp	**0x30D0**
%esp	**0x30CC**

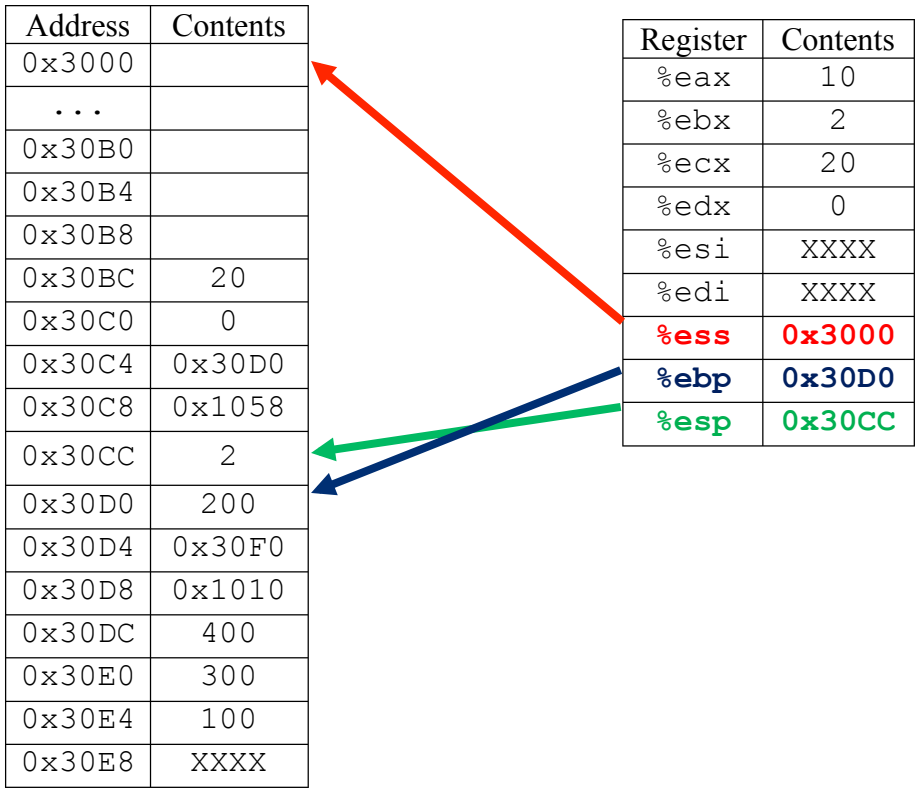
Figure 5.7.6

Notice the stack of **Figure 5.7.6**; although we have returned, the contents of the stack remain. They will not be overwritten until another function utilizes that stack space. Furthermore, notice how the single `mov` instruction de-allocates our local variable space on the stack. No matter how much space we use for local variables, simply setting `%esp` to `%ebp` will restore it. Also, saving the base pointer in the prolog allows us to restore it to where it was in the previous stack frame – this applies to all registers we save in the prolog.

Now we are ready to pick up where we left of at address `0x1058` in `get_average`. This is the address of the instruction below the `call`, and is the address `ret` read from the stack. Here, we will simply compute the average by dividing the `sum` by the `quantity`, which is our local variable set to value `2`. Notice that since we saved the `%ebx` register in the prolog, we are safe to overwrite it in this function.

```
...             ...
...             ...
...             ...
0x1040:         get_average:
                     ## Prolog: save callee's responsible
                     ## registers, if necessary, and get
                     ## space for local variables
0x1044:              pushl %ebp
0x1048:              pushl %ebx
0x104C:              movl %esp, %ebp
0x1050:              subl $4, %esp
                     ## Call get_sum; no need to save registers
0x1054:              call get_sum
                     ## Perform calculation and set avg
0x1058:              movl $2, (%esp)  ## reference using %esp
0x105C:              movl sum, %eax
0x1060:              movl $0, %edx  ## don't forget to clear D
0x1064:              movl -4(%ebp), %ebx  ## or ref. using %ebp
0x1068:              divl %ebx
0x106C:              movl %eax, avg
...                  ...
```

Code 5.7.9



Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	20
0x30C0	0
0x30C4	0x30D0
0x30C8	0x1058
0x30CC	2
0x30D0	200
0x30D4	0x30F0
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	10
%ebx	2
%ecx	20
%edx	0
%esi	XXXX
%edi	XXXX
%ess	**0x3000**
%ebp	**0x30D0**
%esp	**0x30CC**

Figure 5.7.7

As shown in **Code 5.7.9** and **Figure 5.7.7**, we have now calculated the average of `a` and `b` and saved the result in `avg`. Now, we are ready to return from `get_average` to the caller. To do so, we need to write our epilog, which will de-allocate the `4` bytes of space used on the stack for our local variable `quantity`, restore the general purpose registers we saved, and finally return to the caller:

```
...           ...
...           ...
0x1040:       get_average:
                   ## Prolog: save callee's responsible
                   ## registers, if necessary, and get
                   ## space for local variables
0x1044:            pushl %ebp
0x1048:            pushl %ebx
0x104C:            movl %esp, %ebp
0x1050:            subl $4, %esp
                   ## Call get_sum; no need to save registers
0x1054:            call get_sum
                   ## Perform calculation and set avg
0x1058:            movl $2, (%esp)  ## reference using %esp
0x105C:            movl sum, %eax
0x1060:            movl $0, %edx  ## don't forget to clear D
0x1064:            movl -4(%ebp), %ebx  ## or ref. using %ebp
0x1068:            divl %ebx
0x106C:            movl %eax, avg
                   ## Epilog: restore local var space,
                   ## restore registers, and return
0x1070:            movl %ebp, %esp
0x1074:            popl %ebx
0x1078:            popl %ebp
0x107C:            ret
```

Code 5.7.10

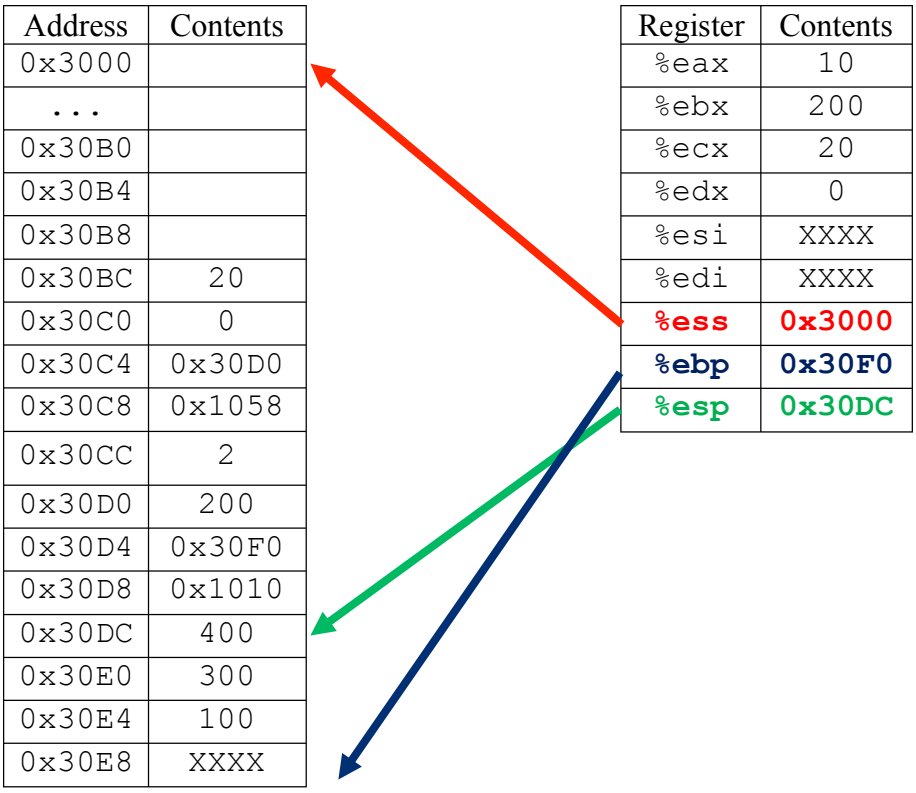
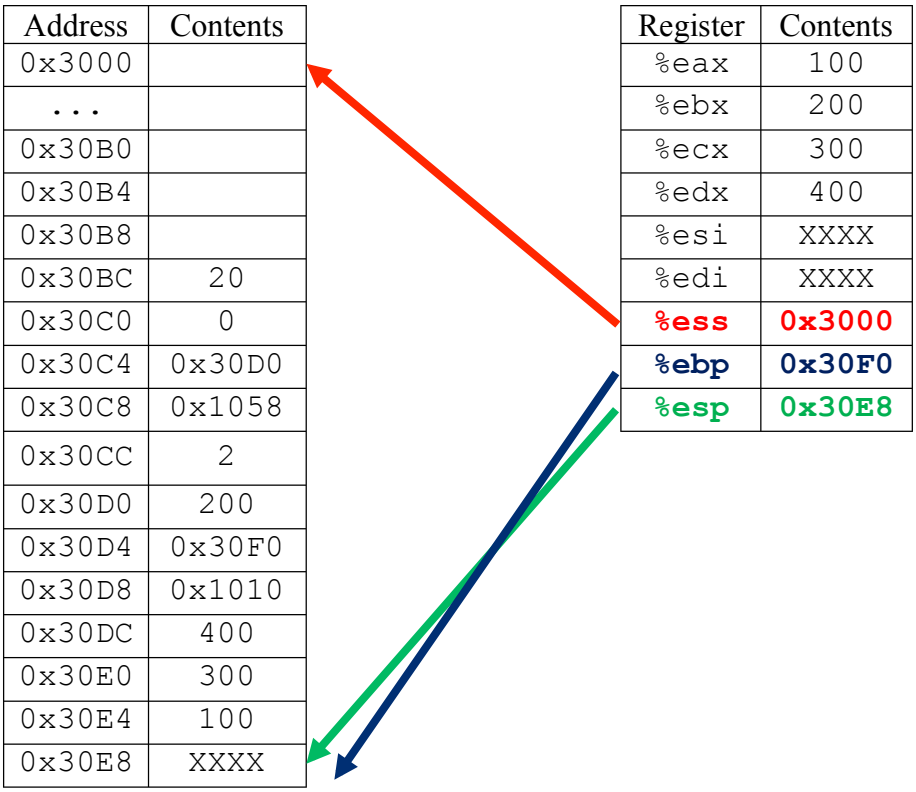


Figure 5.7.8

Now, the average has been computed, and execution has been moved to the instruction at address `0x1010` in memory. This instruction is immediately after the `call` to `get_average`. Before we called the function, we saved the registers which contained important data. We now need to restore them to general purpose registers, as shown in **Code 5.7.11**. It is a good thing we saved them, since **Figure 5.7.8** shows neither `%eax`, `%ecx`, or `%edx` – the registers the caller is responsible for saving – are what they were before we called `get_average`.

```
0x8000:     a:
                 .int 0
0x8004:     b:
                 .int 20
0x8008:     sum:
                 .int 0
0x800C:     avg:
                 .int 0
...         ...
...         ...
...         ...
...         ## Save caller's responsible registers, if necessary
0x1000:     pushl %eax
0x1004:     pushl %ecx
0x1008:     pushl %edx
0x100C:     call get_average
            ## Restore saved registers in LIFO order
0x1010:     popl %edx
0x1014:     popl %ecx
0x1018:     popl %eax
...         ...
```

Code 5.7.11



Address	Contents
0x3000	
...	
0x30B0	
0x30B4	
0x30B8	
0x30BC	20
0x30C0	0
0x30C4	0x30D0
0x30C8	0x1058
0x30CC	2
0x30D0	200
0x30D4	0x30F0
0x30D8	0x1010
0x30DC	400
0x30E0	300
0x30E4	100
0x30E8	XXXX

Register	Contents
%eax	100
%ebx	200
%ecx	300
%edx	400
%esi	XXXX
%edi	XXXX
%ess	**0x3000**
%ebp	**0x30F0**
%esp	**0x30E8**

Figure 5.7.9

And we are done! Again, notice how all data is still present on the stack in **Figure 5.7.9**. It will not be overwritten until another function is called, at which point the same process will occur once more.

In general these are the procedures to follow when calling and returning from functions:

1) Pre-call: `push` all important registers the caller is required to save to the stack.
2) Call: call automatically saves the return address of the code to the stack.
3) Prolog: `push` all registers the callee is required to save to the stack, if needed later. Get space for local variables by saving the stack pointer in the base pointer, then decrementing the stack pointer.
4) Mid-call: Access local variables with an offset to either `%ebp` or `%esp`. Perform arithmetic, implement loops, or anything necessary to the function.
5) Epilog: Free space for local variables by restoring the original stack pointer from the base pointer register. `pop` all registers pushed in the prolog in LIFO order.
6) Return: Automatically retrieve the return address from the stack (indicated by `%esp`) with the `ret` instruction. Resume execution at the address read. This address will/should be the instruction after the `call` in the code segment of memory.
7) Post-call: `pop` all registers that were pushed prior to the function `call`, in LIFO order.

5.8 – A Note about Recursion

In programming, there are two ways to accomplish a repetitive task: (1) iterate through a particular block of code, and (2) recurse through a particular function containing the code we want to execute repeatedly. **Iterative programming** involves loops. Such an example would be a basic `while(true)` loop used to keep a program running or perhaps a `for` loop to sum an array. **Recursive programming** is the act of calling a function within itself to perform the same instructions. In either case, repetition needs to be bounded by a condition upon which the code will stop looping or recursing. This condition is often referred to as the **base case** – the condition that signals we are done looping or recursing. In loops, this base case is implemented using the compare instruction, as discussed in **Lab 3**. This is also true in recursive programming; however, the base case will not only cause the function to stop calling itself, but it will also cause the function to start returning into the prior function call.

For example, consider two implementations of a C fragment to sum an array `x` of length `10`:

```
/* Iterative Method */
int x[10]; // global var's
int sum = 0;
int i;
...
for (i=0;i<10;i++) {
     sum = sum + x[i];
}
...
```

```
/* Recursive Method */
int x[10]; // global var's
int sum = 0;
int i = 0;
...
find_sum();
...
void find_sum() {
     if (i > 9) // base case
          return;
     sum = sum + x[i];
     i++;
     find_sum(); // call again
     return;
 }
```

Code 5.8.1

In **Code 5.8.1**, the left fragment is an iterative example for calculating the sum of a 10-element array. This code can be implemented in Intel 80386 Assembly using the techniques learned in **Labs 2, 3, and 4**. On the right is the recursive example. Notice how the program, at some point, calls the function `find_sum()`. This function will sum the elements of the array from `i=0` to `i=9`, but in a recursive manner. Let's walk through this process. The first time `find_sum()` is called, the index variable `i=0`. After `find_sum()` is called, the base case is first analyzed. We need to see if we are done summing the array. Since `i=0`, this is less than or equal to `9`, so the `if` statement evaluates to `false` and we add `x[i=0]` to the `sum`. Next, we increment `i` and `call find_sum()` again. This is where it gets interesting. We know from the prior sections of this lab that each function `call` in assembly has its own stack frame. This `call` to `find_sum()` will initialize a new stack frame (i.e. save the return address, save necessary registers, etc.) for the upcoming execution of `find_sum()`. From our studies of the stack, we know this new stack frame will be located below (at a lower address from) the stack frame for the previous `call` to `find_sum()`. Recursive functions do not use the same stack frame – they each have their own. (However, programming in assembly can sometimes allow one stack frame to access data from another. We will see this in **Lab 6**.) Continuing our example, execution will jump to `find_sum()`, but this time `i=1`. The same process will occur as with `i=0`, and another stack frame will be created below the `i=1` stack frame for the upcoming execution of `find_sum()` with `i=2`. The recursive `call` will occur over and over until `i=10`, at which point, the base case will be true, and the stack frame for `i=10` will return to where `find_sum()` was called in the stack frame for `i=9`. The code will start executing at the next instruction below the `call`, which is return. This returns to the stack frame for the `i=8` case, and the process continues – going back to previous stack frames higher on the stack – until `find_sum()` finally returns into the main program when the `i=0` stack frame concludes. The following **Code 5.8.2** is the assembly version of **Code 5.8.1**:

```
/* Iterative Method */
.comm x, 40 ## global var's
sum:
     .int 0
.comm i, 4
...
movl $0, i
movl i, %edi
continue:
     cmpl $10, %edi
     jge done
     movl i, %edi
     movl x(,%edi,4), %eax
     addl %eax, sum
     incl %edi
     jmp continue
done:
movl %edi, i
...
```

```
/* Recursive Method */
.comm x, 40 ## global var's
sum:
     .int 0
i:
     .int 0
...
call find_sum
...
find_sum:
     pushl %edi ## prolog
     cmpl $9, i ## base case
     jg done
     movl I, %edi
     movl x(,%edi,4), %eax
     addl %eax, sum
     incl %edi
     movl %edi, i
     call find_sum ## call again
     done:
          popl %edi ## epilog
          ret
 ...
```

Code 5.8.2

As we can see, the assembly iterative method on the left looks familiar. The assembly recursive method on the right calculates the same result but does so via recursion. Notice, each time a `call` to `find_sum` is made, a new stack frame is created below the existing frame on the stack. As we can imagine, had this recursion go on for far greater than `10` elements of array `x`, the free stack space could significantly reduce as more and more stack frames from the `find_sum` function calls are added to the stack. If we were to never check the base case using recursion, our code would eventually encounter a stack overflow after it runs out of stack space. See **Figures 5.6.1 – 5.6.3** for an illustration of how the stack grows and shrinks during recursive calls. **Figure 5.6.1** shows the initial stack, and **Figure 5.6.2** depicts the stack as more frames are added to it (after more recursive calls to `find_sum`). After the base case is reached, the stack will return (as the `find_sum` function instances return) to its original state in **Figure 5.6.1**. **Figure 5.6.3** depicts the stack overflow, which could happen if we forget or incorrectly implement a base case check.

One might wonder, why even use recursion if assembly code can be implemented iteratively? Sometimes it is easier to visualize a problem recursively than it is iteratively. There are also many cases where is it computationally less complex to use a recursive implementation over an iterative implementation. Furthermore, there are languages other than C that are purely functional languages, meaning they are not imperative and do not contain looping structures (e.g. `for`, `while`, `do...while`, etc.). These languages often require the use of recursion in order to

perform repetitive computations. *But for our purposes, recursion, although it is not always as efficient as iteration, is an excellent way to learn how to program with the stack.*

5.9 – Assignment

The assignment for this lab asks us to compose assembly code that calculates the Fibonacci sequence of length `n+1` given `n`. For example, the Fibonacci sequence of `n = 3` is: `0 1 1 2`, the Fibonacci sequence of `n = 5` is `0 1 1 2 3 5`, and likewise for `n = 6` is `0 1 1 2 3 5 8`. In general, the calculation of this sequence from position `1` to position `n + 1` is calculated by `n + 1 = n + (n - 1)` – the next digit is the sum of the preceding two digits. This calculation could be performed iteratively, where we keep track of the previous two digits in order to determine the next; however, it can also be implemented recursively, where we use global and local variables to maintain the previous digits and calculate the Fibonacci sequence. *It is important that you write a recursive implementation of this assignment. Not only is this how the specification code is written, but creating an iterative solution will not allow you to sufficiently practice with the stack.*

This is the specification of what the assembly functions need to perform. Do not copy or type this code. Use it as a reference when writing the assembly. This is a recursive implementation; do not write an iterative solution.

```
/* begin specification code */

/* Convert the procedure exactly as given using the local
variables local_var and temp_var on the stack. */

void Fib(void)
{
     int local_var;
     int temp_var;
     local_var = global_var;
     if(local_var == 0) return;
     else if(local_var == 1) return;
     else {
          global_var = local_var - 1;
          Fib();
          temp_var = global_var;
          global_var = local_var - 2;
          Fib();
          temp_var = temp_var + global_var;
          global_var = temp_var;
     }
     return;
}
/* end specification code */
```

The following is the C driver for the assembly file. It works by calling the function `Fib()` in the assembly file. You are not required to comment this code. Please do not modify this file.

```
/* begin C driver */

/* This is the skeleton C program. It will ask for a number, then
    * print the Fibonacci sequence up to that number plus one.
    * NOTE: You will need to use the stack for the local
      variables local_var and temp_var of the function Fib().
      global_var is a global variable.
*/

#include <stdio.h>

void Fib(void);

int global_var; /* this is a GLOBAL variable */

int main(void)
{
     int i;
     int number;
     int fib_subscript;
     printf("Please enter a number: ");
     scanf("%d",&number);
     printf("Fibonacci sequence from subscript 0 to %d :\n",
     number);
     for(i=0; i<=number; i++) {
          global_var = i;
          Fib();
          fib_subscript = global_var;
          printf("%3d ", fib_subscript);
     }
     printf("\n");
}
/* end C driver */
```

The following is the assembly stub to the driver. You are required to fully comment and write the assembly code to model the specification code. Insert your code where you see `/* put code here */`**,** `/* prolog */`**, and** `/* epilog */`**. Do not modify any other code in the file. Note the last line of the file must be a blank line to compile without warning.**

```
/* begin assembly stub */

.globl Fib
.type Fib,@function
Fib:
     /* prolog */

     /* put code here */


     return:
          /* epilog */
          ret

/* end assembly stub */
```

The following are the cases you should test to ensure correct operation. Your program should accept an integer as input. The driver does not check for all input cases (e.g. junk in the input), but correct operation should produce the following results:

```
./lab5
Please enter a number: 1
Fibonacci sequence from subscript 0 to 1 : 0 1

./lab5
Please enter a number: 2
Fibonacci sequence from subscript 0 to 2 : 0 1 1

./lab5
Please enter a number: 3
Fibonacci sequence from subscript 0 to 3 : 0 1 1 2

./lab5
Please enter a number: 4
Fibonacci sequence from subscript 0 to 4 : 0 1 1 2 3

./lab5 Please enter a number: 5
Fibonacci sequence from subscript 0 to 5 : 0 1 1 2 3 5

./lab5
Please enter a number: 6
Fibonacci sequence from subscript 0 to 6 : 0 1 1 2 3 5 8

./lab5
Please enter a number: 7
Fibonacci sequence from subscript 0 to 7 : 0 1 1 2 3 5 8 13

./lab5 Please enter a number: 8
Fibonacci sequence from subscript 0 to 8 : 0 1 1 2 3 5 8 13 21

./lab5
Please enter a number: 9
Fibonacci sequence from subscript 0 to 9 : 0 1 1 2 3 5 8 13 21 34

./lab5 Please enter a number: 10
Fibonacci sequence from subscript 0 to 10 : 0 1 1 2 3 5 8 13 21
34 55
```

Lab 6

Subroutine Parameters and Returns

Student Objectives

- Learn about how to pass arguments/parameters to assembly functions
- Learn how to return values from assembly functions
- Learn how to implement parameters and returns with recursive functions

6.1 – Introduction to Parameters

Parameters or arguments are the primary means of communicating between subroutines. In C, we pass parameters using the parenthesis notation. **Code 6.1.1** shows a C function that takes two integer arguments – `arg1` and `arg2`.

```
void my_func(int arg1, int arg2) {
     ...
}
```

Code 6.1.1

In x86 assembly, there are two main methods for passing parameters: **(1) on the stack**, and **(2) in registers**. When parameters are passed to a function, the caller must provide them to the callee; so, naturally there must be some form of agreement between the two functions. If the caller places the arguments on the stack, the callee must know to read them from the stack. Likewise, if the callee requires that its arguments be provided via registers, the caller must know to place the arguments in registers. Another issue that arises from either method is the placement of *multiple parameters*. In **Code 6.1.1**, there are two arguments. If they are passed using the stack, which order should they appear? And likewise, if they are passed using registers, which register should be assigned to a particular argument? Fortunately, compilers like GCC hide these fine points from us in high-level languages like C, where they implement the handshaking between functions for us. However, now that we are assembly programmers, we must take these details into account.

6.2 – Parameters on the Stack

Let's consider the first method for passing parameters – **the stack**. *By default, GCC passes function parameters using the stack.* The caller must push the arguments to the stack in the reverse order that they would appear in C. For example, **Code 6.1.1** could be implemented in assembly as follows:

```
## Save any registers here
...
## Pass arguments in reverse order
movl arg2, %eax
movl arg1, %ebx
pushl %eax
pushl %ebx
call my_func
addl $8, %esp
```

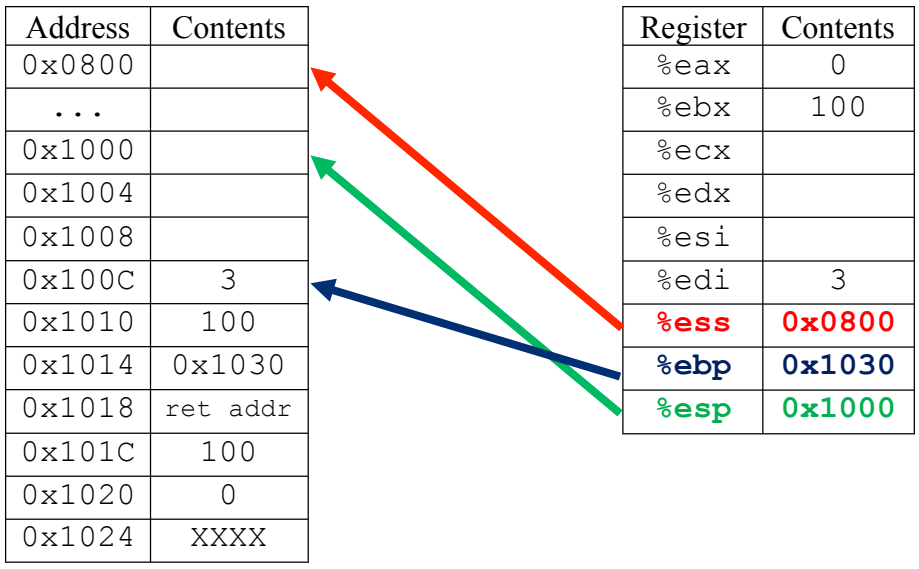
Code 6.2.1

*As a brief aside, notice the last line of **Code 6.2.1**. We mentioned in **Lab 5** that in general, for every `push` there needs to be a corresponding `pop`; however, here we add `8` bytes to `%esp`. Recall that the `popl` instruction reads the value at the current location of the stack pointer and then increments `%esp` 4 bytes. With a single `addl` instruction, we can increment the stack pointer back to where it was before we pushed any number of arguments. This is equivalent to popping each argument off individually but with a single, more efficient instruction. It is important to note that this method does not read data, as `pop` does. As we know, `pop` copies data from the stack to its argument and increments the stack pointer, while `add` skips this step and simply increments the stack pointer.*

Now back to our example, the question is, how does the callee receive these arguments on the stack? Recall that with local variables, we can apply an offset to either `%ebp` or `%esp` (with register indirect or base-indexed addressing) to access data on the stack. This same concept can be applied to retrieving arguments from the stack. The trick is to apply the appropriate offset to reach the arguments. This offset will depend on the prolog of the function, as well as the number of arguments pushed to the stack. Let's consider an example:

```
pushl %eax
pushl %ebx
call my_func
...
my_func:
     pushl %ebp
     pushl %ebx
     pushl %edi
     movl %esp, %ebp
     subl $12, %esp
     ...
```

Code 6.2.2



Address	Contents
0x0800	
...	
0x1000	
0x1004	
0x1008	
0x100C	3
0x1010	100
0x1014	0x1030
0x1018	ret addr
0x101C	100
0x1020	0
0x1024	XXXX

Register	Contents
%eax	0
%ebx	100
%ecx	
%edx	
%esi	
%edi	3
%ess	**0x0800**
%ebp	**0x1030**
%esp	**0x1000**

Figure 6.2.1

Consider an assembly fragment in **Code 6.2.2** and its corresponding stack in **Figure 6.2.1**. The caller passes `0` and `100` to the stack via `%eax` and `%ebx`, respectively. The function `my_func` is called, and the prolog of `my_func` executes, which saves three registers to the stack and allocates space for three integer local variables. To read in the arguments, the callee must use either `%ebp` or `%esp` with an offset. The base pointer is the most common practice for retrieving argument, although it is also possible with the stack pointer. So, from the base pointer, we can see that there are three items pushed to the stack from the prolog (12 bytes), the return address of the function (4 bytes), and then the arguments appear. This means to reach the first argument, we must apply an offset of `16` to `%ebp`, and to reach the second argument, we must apply an offset of 16 bytes plus 4 bytes or `20` bytes. **Code 6.2.3** shows the assembly required to access these arguments from within `my_func`.

```
pushl %eax
pushl %ebx
call my_func
...
my_func:
     pushl %ebp
     pushl %ebx
     pushl %edi
     movl %esp, %ebp
     subl $12, %esp
     ## Access/retrieve arguments
     movl 16(%ebp), %eax
     movl 20(%ebp), %ebx
     ...
```

Code 6.2.3

As shown, we add a positive offset to the base pointer in order to retrieve the arguments from the stack. Notice that since we offset from the base pointer, we can ignore the space used for local variables; however, if we were to use the stack pointer, then we would need to account for the local variable space in our offset as well.

6.3 – Parameters in Registers

Let's consider the second method for passing parameters to functions – **general purpose registers**. *As stated earlier, by default, all functions in assembly and C use the stack to pass parameters*. The reason is there are a limited number of registers at our disposal. What if we need to pass more than a few arguments, or perhaps we need to pass a large data structure to a function? The registers cannot be used (entirely) in these cases – we would have to pass some, if not all of the parameters using the stack. The use of registers also requires more coordination between the caller and the callee. Accessing arguments on the stack is a sequential and calculable operation, while accessing arguments in registers requires thought as to which register will contain which argument – there is no convention for this, unlike the stack which uses a reverse-order policy.

These drawbacks aside, in simple cases where there are only a few arguments to pass, we may use the general purpose registers. Like the stack though, the caller and callee must agree upon which argument will be in which register. The benefit to using registers is that passing arguments is very straightforward, since there are no offsets to worry about. Furthermore, accessing parameters in registers is far more efficient than the stack, since the latency of a main memory access will not be present.

So, why are registers not the primary means of parameter passing? The answer lies in the complexity involved with determining which arguments go in which register. In high-level languages like C, there are many standard, as well as custom libraries we use to perform common computational tasks – e.g. `math.h`, `stdlib.h`, and `stdio.h`. These libraries are designed to work with as many users as possible. In order to broaden the compatibility of C

functions, the stack is the method by which parameters are passed. So for example, the function `printf(...)` will expect GCC to compile the user's code such that all of its arguments are pushed to the stack in reverse order. No matter what code invokes `printf(...)`, it will expect its arguments to be on the stack. As such, the vast majority of assembly functions are written to use the stack for passing parameters. Likewise, the C driver files we have used for all our labs contain code that calls our assembly functions. Any C code that directly calls an assembly function will place its arguments on the stack (when assembled from C by GCC). So, with relevance to this lab, we must write our assembly code such that it uses the stack to pass parameters.

Having said the above, there is a way to force GCC to compile C functions and use registers as the method for passing parameters. This is done using the GCC keyword `__attribute__`. This is included with a C function declaration (i.e. with the function prototype) and supplies the compiler with additional information about your function, in order to optimize how it is called by other functions and how it performs at runtime. `__attribute__` takes a double-parenthesis, comma-delimited list of function specifications and appears before the function declaration. As of GCC version 4.8.0, `__attribute__` has many function specifications, but the ones of interest to us at this time are `fastcall` and `noinline`. `fastcall` is a function attribute specification that tells GCC to pass the first argument in register `%ecx`, the second argument in register `%edx`, and all other arguments on the stack. This technique will only work if the arguments are of *integral type*, which means they must be a standard type (`char`, `int`, etc.) and not user-defined (e.g. `structs`). When GCC compiles our code, depending on the flags used, it might attempt to optimize any code that does not have any side effects. In other words, if our code does not use any standard output to the user, GCC will attempt to make it faster; this optimization process includes taking functions that are small, removing the call and return, and placing the code within the function in the code that called it. Functions that are purposefully written in or optimized in this manner are said to be *inline functions*. To stop GCC from converting our functions to inline form (where they are not actually called and thus take no arguments) we include another function specification called `noinline`. **Code 6.3.1** demonstrates how to write a C function and force it to place its arguments in the `%ecx` and `%edx` registers, with the remainder of the parameters placed on the stack.

```
__attribute__((fastcall, noinline)) void my_func(int
x, int y);
```

Code 6.3.1

As we can see, the `__attribute__` keyword contains the function specifications `fastcall` and `noinline`, which will cause `int x` to be placed in register `%ecx` and `int y` to be placed in register `%edx`. Note, this is the C function prototype for `my_func`, which tells GCC at compile-time to not only supply `my_func` with arguments in the aforementioned registers, but also assemble `my_func` in such a way that it expects its arguments to be in the proper registers. Without this addition to the function prototype, GCC will place `int y` followed by `int x` on the stack prior to calling `my_func`.

For those who would like to learn more about attributes in GCC, refer to the GNU GCC online documentation. Attributes are powerful tools to optimize and customize C code and can be used for functions, variables, and structure/union data types:

For Function Attributes:
http://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Function-Attributes.html#Function-Attributes

For Variable Attributes:
http://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Variable-Attributes.html#Variable-Attributes

For Type Attributes:
http://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Type-Attributes.html#Type-Attributes

Attributes are not a part of this lab's assignment; however, it is important to understand how C and assembly are related and how they interact during function calls and returns.

6.4 – Subroutine Returns

It is often the case where we wish to create a function that can communicate a result, status, event, etc. from the callee function to the caller function. In most programming languages, this is accomplished via **subroutine returns**. In C, to specify a return for a function, we include the return type in the function prototype and in the function itself.

```
int my_func(void);
...
...
...
int my_func(void) {
     ...
     ...
     ...
     return some_int;
 }
```

Code 6.4.1

As shown in **Code 6.4.1**, the function `my_func` takes no parameters but returns an integer – specified by the `int` preceeding the function prototype and the function definition.

Also in C, to use a return value, we can do a variety of things – we can set a variable to the return or check the return within a conditional expression, for example.

```
int number;
...
number = my_func();
```

```
int number;
...
if (my_func()) {
    ...
    return some_int;
}
```

Code 6.4.2

As we can see in **Code 6.4.2**, we set the return of `my_func` to the integer `number` and demonstrate the use of `my_func` within an `if` statement – `if my_func` is greater than zero, then the `if` statement evaluates to `true`.

So, the question is, how does this translate to x86 assembly? Like function parameters in assembly, function returns are implemented a particular way, by convention. In x86 assembly language, *all function returns are placed in the A register* (i.e. `%al`, `%ah`, `%ax`, or `%eax`). As such, the caller will expect the return value to be in the `%eax` register after the callee executes the `ret` instruction and execution jumps back to the caller. This means, it is the responsibility of the callee to make sure the return value is placed in the A register before executing ret.

```
...
addl %ecx, var
movl var, %eax
## Epilog
...
```

Code 6.4.3

As seen in **Code 6.4.3**, the callee is performing some operation and needs to return the value in the variable `var`. To do so, the callee moves the contents of `var` to the A register, and then gets ready to return by executing the epilog (as discussed in Lab 5). It's that simple!

Sometimes, an explicit `movl` of a value to the A register is not necessary. Say you wish to return the quotient of a division. As we know, the quotient is stored in the `%eax` register after a 32-bit division. If this is what we wish to return, then it is already in the A register, and no explicit move is necessary.

```
...
divl var
## Want to return the quotient
## %eax contains the quotient
## %edx contains the remainder
...
## ...no other instructions with A reg as dest...
...
## Epilog
...
```

Code 6.4.4

As we can see in **Code 6.4.4**, the value we wish to return to the caller is already in the `%eax` register, so we do not need to include an explicit move to the A register. *Note, this only holds true if the most recent calculation with the A register as a destination is the desired return value.* If a subsequent instruction overwrites the return value (the quotient) with another value, then we must explicitly move the quotient back into the A register before we execute the epilog and return – otherwise, the caller might not get the correct quotient from our function.

6.5 – Subroutine Assembler Directives

Now that we are well-versed in writing assembly language subroutines, we need to learn how to communicate to the assembler that we have a new subroutine for other fuctions or parts of our program to `call`. After all, what use is our function if it cannot be referenced from our C driver file, for instance?

To allow assembly subroutines to be called or referenced from functions *in other files* (like our C driver code), we need to (1) give the label of the function global scope and (2) tell the assembler that the label of the function can be called with the `call` instruction. We can accomplish each of these tasks using the `.globl` and `.type` assembler directives:

```
        .globl myFunc
        .type myFunc, @function
myFunc:
        ## Prolog
        ...
        ...
        ## Epilog
        ret
```

Code 6.5.1

In **Code 6.5.1**, we have a sample function/subroutine called `myFunc` that we wish to allow to be executed as a function and refereced from other files of our program. Let's first take a look at `.globl`, which is provided in its general form in **Code 6.5.2** below:

```
.globl <label>
```

Code 6.5.2

As seen in **Code 6.5.2**, `.globl` takes a single argument, and as you might have guessed, it gives its argument – the label `myFunc` in **Code 6.5.1** – global scope in the program. For this course, where we have C code that calls assembly code *located in another file*, the inclusion of `.globl` is imperative in order for our code to compile. Otherwise, `main()`, in our C code, would never be able to "see" our assembly subroutine code `myFunc` in the separate file. The behavior of `.globl` can be likened to the `#include <header_file.h>` compiler directive in C. Like

`.globl`, `#include` instructs the compiler to allow the file the directive is issued from to have access to the functions and variables of the file given by `<header_file.h>`.

Also in **Code 6.5.1**, notice the new assembler directive `.type`. In general, `.type` takes the form given below in **Code 6.5.3**:

```
.type <label>, <label_type>
```

Code 6.5.3

The purpose of `.type` is to inform the assembler about what kind of label `myFunc` is. Possible values of `<label_type>` are `@notype`, `@object`, and `@function`, which are used for undefined types, frequently referenced labels, and functions, respectively. The only one of these types of interest to us when composing subroutines is the type `@function` – this tells the assembler that `<label>` is a valid label to reference with the `call` instruction. Theoretically, any label can be called in an assembly program; however, the use of `.type` protects against labels being called that are not actually written as functions. Labels that are not written as functions internally (with a prolog, epilog, return, etc) but referenced with the `call` instruction could overwrite important data in general purpose registers from other function calls and cause undesired results. (Recall the callee responsibilities discussed in **Lab 5, Section 5.5 – Stack Frames: Caller and Callee Responsibilities**.)

So, with knowledge of the assembler directives `.globl` and `.type`, take a look at the assembly stubs from previous lab assignments. As you will notice, in the assembly files, each subroutine called from the C driver file includes the `.globl` and `.type` assembler directives to allow our assembly code's label to (1) be referenced from an external file, the C driver file, and (2) be called using the `call` instruction, respectively.

6.6 – Assignment

This is the specification of what the assembly functions need to perform. Do not copy or type this code. Use it as a reference when writing the assembly. This is a recursive implementation; do not write an iterative solution.

```
/* begin specification code */

/* Convert the procedure exactly as given using local variables
where needed. */

int Factorial(int n)
{
     if (n == 0 || n == 1) {
          return 1;
     }
     else {
          return n * Factorial(n-1);
     }
}
/* end specification code */
```

The following is the C driver for the assembly file. It works by calling the function `int Factorial(int)`, which you will write in the assembly file. You are not required to comment this code. Please do not modify this file.

```
/* begin C driver */

/* This is the C driver. It will input a number, then
    print the Factorial of that number.
    NOTE: You will need to use the stack for the local variables
    and return the answer in the appropriate register.
*/
#include <stdio.h>

int main(void);
int Factorial(int);

int main(void)
{
     int i;
     int number;
     int answer;

     int fib_subscript;
     printf("Please enter a number: ");
     scanf("%d",&number);  answer = Factorial(number);
     printf("The factorial of %d is %d.\n", number, answer);
}

/* end C driver */
```

The following is the assembly stub to the driver. You are required to fully comment and write the assembly code to model the specification code. Insert your code where you see `/* put assembler directives here */`**,** `/* put code here */`**,** `/* prolog */`**, and** `/* epilog */`**. Do not modify any other code in the file. Note the last line of the file must be a blank line to compile without warning.**

```
/* begin assembly stub */

/* put assembler directives here */
Factorial:
     /* prolog */

     /* put code here */

     return:

     /* epilog */

     ret

/* end assembly stub */
```

The following are the cases you should test to ensure correct operation. Your program should accept an integer as input. The driver does not check for all input cases (e.g. junk in the input), but correct operation should produce the following results:

```
./lab6 Please enter a number: 0
The factorial of 0 is 1.

./lab6 Please enter a number: 1
The factorial of 1 is 1.

./lab6 Please enter a number: 2
The factorial of 2 is 2.

./lab6 Please enter a number: 3
The factorial of 3 is 6.

./lab6 Please enter a number: 4
The factorial of 4 is 24.

./lab6 Please enter a number: 5
The factorial of 5 is 120.

./lab6 Please enter a number: 6
The factorial of 6 is 720.

./lab6 Please enter a number: 7
The factorial of 7 is 5040.
```

Appendix A

Code Comments

As a future engineer, just as it is important for you to compose code that works well, it is equally important that your work and your programs are easy to interpret by others – be it an instructor, your peers, or your future colleagues. You will be expected to include three types of comments in your lab assignments – a program header, function headers, and in-line comments.

The **program header** is a comment block at **the top of each of your files** that gives the following information and is formatted in the following fashion:

```
/*NAME    Ryan Izard
  COURSE  ECE 273
  SECION  400
  DATE    11.06.12
  FILE    example_program_header.s
  PURPOSE This is an example of a program header. The
          Purpose section gives an overview of what
          The program does and what programming skills the
          lab assignment is designed to develop.
*/
```

The **function header** is a comment block at **the top of each function** that gives the following information and is formatted in the following fashion. Note, for this course, there are not any arguments or returns until Lab 6. Do not confuse the use of global variables with arguments or returns.

```
/*FUNCTION     myFunction
  ARGUMENTS    Give a list of the parameters and data types
               the function takes as input
  RETURNS      Give the return type, if any, of the
               function
  PURPOSE      This is an example of a function header. The
               purpose section gives an overview of what
               the function does and any other
               implementation details a 3rd party viewer
               would need to know (such as any improvements
               that need to be made, etc.)
*/
```

In-line comments are commentary **within the code** itself. These comments need to be detailed and frequent enough for a third party to be able to easily interpret and understand your code and its purpose. In-line comments should describe the *big picture* of what the line(s) of code is/are trying to accomplish. In-line comments *should not* state simply the operation the line(s) is/are performing. For example:

```
movl $2, %eax          # square 2
mull %eax
movl %eax, result      # save the result of 2^2
```

Note the comment `square 2` applies to the first two lines of code; and, the comment `save the result of 2^2` applies to the third line. They tell the reader what you, the programmer, are trying to accomplish – take the square of the number 2 and save it. The comments *do not* say "move 2 into the A register," "multiply the A register," "move the A register into the variable result." In-line comments of this form are of little to no use to a programmer, since he or she can simply read the code itself to figure that out. Instead, in-line comments are designed to give a higher-level overview of what the code is doing. If you have any questions on this, please ask.

Similar to comments, the aesthetics or appearance of your code needs to be pleasing to the eye. This increases readability and reduces fatigue when reading code. It is expected that you indent properly and make your code uniform throughout. For example:

```
Label_1:
     movl $2, %eax        # square 2
     mull %eax
     movl %eax, result    # save the result of 2^2
Label_2:
     ...more assembly code...
```

Appendix B

Useful Terminal Commands

Being comfortable working in the terminal is a necessary skill for both Electrical and Computer Engineers. Although GUIs are great for the end-user, many seasoned programmers and system developers find it more straightforward and less cumbersome to work within the terminal. It can be an intimidating experience at first, but with a few handy commands in your toolbox, the terminal can quickly become an efficient and easy environment to work in.

1) Press the **`TAB`** key when you are typing the name of a file, program, etc. in your working directory. This will ask the terminal to predict what you are trying to type and fill it in for you. Predictions are pulled from the contents of the working directory and the environment variables. For example, if you just compiled your program with the `gcc` command `gcc -o myprog drv.c asm.s`, you can run it by typing `./m`. Don't press `ENTER` yet, but press `TAB` instead. The terminal will look in your current folder for files starting with the letter `m` and try and fill it in. Chances are it's the only file present starting with `m`, so it will turn into `./myprog` without you having to type the whole thing. If there is more than one choice beginning with `m`, you can press `TAB` twice to have the terminal give you the options. From there, you can continue typing which file you want and press `TAB` again to prompt the terminal to search for a matching file based on the additional letters you've given it. Go ahead and give it a try! It will save a lot of typing (and typos from occurring) and will quickly become second nature with practice.

2) At any time, you can press the **`UP`** and **`DOWN`** arrow keys to browse through recently entered commands. If you just compiled your program using the `gcc` command given in (1) above, you don't have to retype it again to recompile. Just press the `UP` arrow until you see it again. Not only can this help save time and increase productivity, it can also help you recall something you typed earlier in the case you have forgotten.

3) The command **`pwd`** prints your current working directory. If you just logged into an Apollo machine, you will be in the `/users/yourUserID` directory.

4) The **`ls`** command shows the contents of your current working directory. Give this command in any folder to see what is inside.

5) The **`cd`** command changes to another directory. If you are in `/users/yourUserID` and you do an `ls` and see that you have a `lab1` folder, you can go into it by typing `cd lab1`. If you are am in `/users/yourUserID/lab1` and you want to go back to your `/users/yourUserID` folder, then you can run `cd ..`. This double-dot tells the `cd` command to go up one folder.

6) Pressing **ctrl + c** (i.e. **^c**) will stop the execution of any program running in the foreground. So, if your program gets into an infinite loop, or like our Lab 1 program, requires the user manually kill it, `ctrl + c` will stop the program from running.

7) The **mkdir** command will create a new directory in your current working directory. So, if you are in the `/users/yourUserID` directory and you want to make a `lab2` folder, you would enter `mkdir lab2`. You can give the `ls` command here to reveal your newly created `lab2` folder.

8) The **rm** command will remove a file. If you are in `/users/yourUserID/lab1`, and you want to remove the `a.out` executable that `gcc` creates, you would run `rm a.out`. `rm` can also be used to remove folders. It works in the same way, but it requires the recursion flag **-r**. So, if you are in the `/users/yourUserID` folder and you want to delete the `lab1` folder inside (full directory: `/users/yourUserID/lab1`), you would run `rm -r lab1`. This will delete your `lab1` folder and everything inside it!

9) The **cp** command will copy files. If you are in `/users/yourUserID/lab1`, and you want to copy the driver file named `drv.c` to your `/users/yourUserID` folder, then you would do `cp drv.c /users/yourUserID`. The first argument to `cp` is the source and the second is the destination. `cp` can also work recursively to copy a folder and its contents. If you are in the `/users/yourUserID` folder and you want to copy all of what is in `/users/yourUserID/lab1` into the `/users/yourUserID/Labs` directory, you would do `cp -r lab1 Labs`.

10) The **scp** command securely copies your files from one machine to the other. Refer to Appendix C on Working Remotely for further instructions.

11) **"<quotes>"** When you have a file name, directory name, program name, or anything with a space in it, you need to encapsulate the name with quotes. Linux interprets a `SPACE` as break between two distinct arguments. If you have a folder called `Lab 1` and you want to navigate to it, you would need to type `cd "Lab 1"`. An alternative to using quotes is to use a backslash, `\`. Everywhere there is a `SPACE` as a part of a name, you put a `\` to tell the terminal to expect a `SPACE` immediately following the `\`. The same example would be `cd Lab\ 1`. The `\` goes immediately before the `SPACE` in the name. If you are looking for more detail as to why we need to do this – the terminal, like mentioned before, interprets a `SPACE` as a delimiter between arguments. Every word separated by a `SPACE` typed on the command line is an argument. The command `cd "Lab 1"` has argument `0` as `cd` and argument `1` as `"Lab 1"`. (We used quotes here, but the same would work with the backslash method.) If you did `cd Lab 1`, there would be three arguments – `cd`, `Lab`, and `1`. The `cd` command only expects a single argument, so it would throw an error stating it was used improperly if "`...`" or a `\` were not used.

Appendix C

Working Remotely

It can sometimes be helpful to work on our assignments away from the lab. For online courses, this is a requirement to debug and test your work. There are a couple methods for working outside of the lab– (1) using your own Linux machine or (2) logging into the CES Apollo machines remotely.

Editing with a Local Machine

If you would like to use your own computer, you must use a version of Linux. Since all assignments will be evaluated on the CES Apollo or equivalent machines, it is recommended you develop and test your code on an equivalent machine. As of May 2013, CES Apollo machines are running Ubuntu Linux x86_64 with kernel version 3.2.0-41-generic. In case of an update, to find the most recent kernel and OS information, login to the Apollo machines using the instructions below in **Remotely Logging In** and enter the command `uname -a`. Information will be displayed about the currently running kernel and operating system.

You may either run Ubuntu natively or install a virtual machine. To install Ubuntu on your machine, refer to www.ubuntu.com for documentation. If you would like to use a virtual machine on top of your existing operating system, refer to www.virtualbox.org. VMWare and Parallels are alternatives to VirtualBox; however, VirtualBox is free and highly recommended. If you have any difficulties installing Linux, please ask your instructor *before* assignments are due.

Next, before we install libraries, we must first set up the root password. If you already have root access, you may skip this step. Upon a fresh install of Ubuntu, the root password is an unknown random hash. To set the root password to something you will remember, execute the `sudo passwd` command in the terminal. It will prompt for the new password and once again to confirm it.

Now, most modern computers and operating systems are 64-bit; however, the assembly language taught in this course is 32-bit. As such, you must make sure you have the 32-bit C libraries installed on your Ubuntu system. To do so, open the terminal and type `sudo apt-get install gcc-multilib`. This will install all standard 32-bit libraries for compiling 32-bit code with `gcc` and will require the root password.

At this point, we are ready to go. Simply use the text editor of your choice and follow the instructions for Lab 1 to compile your code. Although development of your assignments is more streamlined on your own machine, it is recommended you follow the `scp` instructions in the last section to test your code for compatibility on the machines it will be graded on.

Editing with a CES Apollo Machine

To access the CES Apollo machines remotely, we must first make our computer appear as if it is on campus from the point of view of any campus machine. To do this, we can either choose to VPN onto campus or use the `ssh` utility (or an equivalent program). This will allow us to login to an Apollo machine to write, test, and debug our assignments. If you would like to VPN, please do so, and disregard command **(1)** below; proceed from **(2)**. The basic command structure (typed in the terminal) is the following:

1) `ssh ClemsonUserId@access.ces.clemson.edu`

Where `ClemsonUserId` is your Clemson University username. If prompted to accept an RSA key fingerprint, type `yes`. Your password is your Clemson University password. This will create a tunnel between your personal computer and the access machine.

From the access machine, we must then `ssh` again:

2) `ssh ClemsonUserId@apolloXX.ces.clemson.edu`

where `ClemsonUserId` is again your Clemson University username and `XX` is any two-digit number in the range `01, 02, 03, ..., 14, 15, 16`. Also, if again prompted to accept an RSA key fingerprint, type `yes`. Your password is also your Clemson University password. This will log you into the Apollo machine of your choice. Note, regardless of the machine number you choose, your account information, files, etc. will be mirrored onto that machine. In other words, if one machine does not grant you access (for potentially a variety of reasons), give another number a try. Your files should be accessible from every CES computer. *If your password fails (and you are certain you are using your Clemson password), you might not have a CES account. Please email your instructor as soon as possible for assistance. Setting up an account takes time and resources and might not get done outside the typical hours of a work day. Please do not wait until the last minute to resolve any account issues.*

At this point, you can choose the text editor of your choice (`vi`, `vim`, `pico`, `nano`, etc.) and work on your assignments. Note, this method is not graphical; however, it can make you more proficient in the Linux terminal. If you would prefer to use an IDE or a GUI-based application to edit your code, you will need to copy your files from your local machine to an Apollo machine for testing (see **Testing Your Code on a CES Apollo Machine**, below)

To close any `ssh` session, give the `exit` command. If you followed **(1)**, you will need to close your Apollo and access `ssh` sessions. If you VPNed and began with **(2)**, you will only need to close the Apollo `ssh` session.

Testing Your Code on a CES Apollo Machine

If you chose to edit your code on your local or personal machine, you should transfer it to an Apollo machine for testing. Likewise, after you have completed an assignment and fully tested it on an Apollo machine, you should transfer your assembly file from the Apollo machines to your personal computer for submission.

To copy your files from your local machine to an Apollo machine, open a terminal on your *local machine* and type:

```
scp labXassembly.s labXdriver.c
ClemsonUserId@apolloXX.ces.clemson.edu:labXfolder
```

where `labXassembly.s` and `labXdriver.c` are the assembly and driver files for the lab you wish to compile and test. `ClemsonUserId` is your Clemson University username and `XX` is any two-digit number in the range `01, 02, 03, ..., 14, 15, 16`. If prompted to accept an RSA key fingerprint, type `yes`. Your password is your Clemson University password. This will copy your assembly file and driver file from your local machine's current working directory (i.e. the location you ran `scp` from) to the Apollo machine in your folder of choice. Replace `labXfolder` with the path you wish to save your files to on the Apollo machines in your `ClemsonUserId` directory.

To copy your assembly file from an Apollo machine to your local machine, open a terminal on your *local machine* and type:

```
scp ClemsonUserId@apolloXX.ces.clemson.edu:labXfolder/labXassembly.s .
```

where `ClemsonUserId` is your Clemson University username and `XX` is any two-digit number in the range `01, 02, 03, ..., 14, 15, 16`. If prompted to accept an RSA key fingerprint, type `yes`. Your password is your Clemson University password. This will copy your assembly file from the Apollo machine to your personal computer's current working directory (i.e. the location you ran `scp` from). Replace `labXfolder` with the path of your file on the Apollo machines in your `ClemsonUserId` directory. Also replace `labXassembly.s` with the name of the assembly file you wish to copy. Note – do not forget the `SPACE` and `PERIOD` after the `labXassembly.s`.

In general `scp` takes `X` arguments. The first argument through argument `X-1` is/are the source file(s). The last argument (i.e. argument `X`) is the destination. If you would like to place the file you are copying into a different destination than your current working directory, replace the `.` with the destination directory. For example, if you would like to paste your assembly file on your desktop, you might type:

```
scp ClemsonUserId@apolloXX.ces.clemson.edu:labXfolder/labXassembly.s
                        /home/username/Desktop
```

where `username` is your local Ubuntu username.

If you would like to copy multiple files from a remote source to a local destination, simply *encapsulate the paths of each file in quotes* after the colon, `:`.

For example:

```
scp ClemsonUserId@apolloXX.ces.clemson.edu:”labXfolder/labXassembly.s
                    labXfolder/labXdriver.c” .
```

Appendix D

ASCII Code

ASCII (American Standard Code for Information Exchange) is a way to represent characters, both visible and hidden, on a computer. Standard ASCII assigns an 8-bit (or 1-byte) binary code to a given character. These codes can be referenced in their *decimal* equivalents of `0` to `127`, as well as in *octal* and *hexadecimal*. For example, the string:

```
A B C 1   2    3
```

is represented in ASCII as:

```
65  32  66  32  67  32  49  9  50  9  51
```

where `A` is ASCII `65`, `B` is ASCII `66`, `C` is ASCII `67`, `1` is ASCII `49`, `2` is ASCII `50`, `3` is ASCII `51`, a `space` is ASCII `32`, and a `tab` is ASCII `9`. Notice the invisible characters have ASCII values as well.

The following table from *www.asciitable.com* shows the ASCII-coded values for common characters. For this course, ASCII will be required in Lab 4.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Appendix E

Assignment Solutions

Please refer to these solutions only once you have exhausted all your resources – the instructor, the lab manual, and your neighbor (for group assignments) – and you feel working backwards might help. Any submissions copied verbatim or in part from this appendix will not receive credit. Your instructor wants to see you succeed and understand these assignments, so ask for help! Note there are many ways to complete each assignment; the solutions simply demonstrate one correct implementation for each.

Lab 1 Solution

```
/* begin assembly code */

.globl asum
.type asum,@function
asum:
     pushl %ebp
     movl %esp, %ebp
     subl $4, %esp
     movl $0, -4(%ebp)
.L2:
     movl 8(%ebp),%eax
     cmpb $0,(%eax)
     jne .L4
     jmp .L3
.L4:
     movl 8(%ebp),%eax
     movsbl (%eax),%edx
     addl %edx, -4(%ebp)
     incl 8(%ebp)
     jmp .L2
.L3:
     movl -4(%ebp), %eax
     jmp .L1
.L1:
     movl %ebp, %esp
     popl %ebp
     ret

/* end assembly */
/* Do not forget the required blank line here! */
```

Lab 2 Solution

```
/* begin assembly solution */

.globl dodiff
.type dodiff, @function
dodiff:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */
     movl $0, %edx
     movl digit1, %eax
     mull %eax
     movl %eax, %ebx
     movl digit2, %eax
     mull %eax
     movl %eax, %ecx
     movl digit3, %eax
     mull %eax
     addl %ebx, %ecx
     subl %eax, %ecx
     movl %ecx, diff

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

.globl dosumprod
.type dosumprod, @function
dosumprod:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */
     movl digit1, %eax
     addl digit2, %eax
     addl digit3, %eax
     movl %eax, sum
```

```
     movl $0, %edx
     movl digit1, %eax
     mull digit2
     mull digit3
     movl %eax, product

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

.globl doremainder
.type doremainder, @function
doremainder:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     /* put code here */
     movl $0, %edx
     movl product, %eax
     divl sum
     movl %edx, remainder

     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

/* declare variables here */
.comm digit3, 4
.comm digit1, 4
.comm digit2, 4
.comm diff, 4
.comm sum, 4
.comm product, 4
.comm remainder, 4

/* end assembly solution */
```

Lab 3 Solution

```
/* begin assembly solution */

.globl classify
.type classify,@function

classify:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp

     movl $0, match
     movl i, %eax
     cmpl j, %eax        # compare j to i (eax)
     jne match2
     addl $1, match

     match2:
          cmp k, %eax
          jne match3
          addl $2, match

     match3:
          movl j, %eax
          cmpl k, %eax
          jne if_level1_num1
          addl $3, match

     if_level1_num1:
          # if(match)
          movl match, %eax
          cmpl $0, %eax       # compare match with 0
          je if_level1_num2   # if they're equal, skip this if

          # if(match == 1)
          cmp $1, %eax        # compare match with 1
          jne else_level2     # if not equal, jump to the else
          # if they are equal, perform this nested if statement
          movl i, %ebx
          addl j, %ebx        # ebx = i + j
          cmpl k, %ebx        # compare (i+j) (the ebx reg.) to k
          jg if_level1_fin    # if it's greater than, jump to the
          2 if movl $0, tri_type   # tri_type = 0
          jmp return          # jump to the return
```

```
else_level2:
     # if(match != 2)
     movl match, %eax
     cmp $2, %eax         # compare 2 to match
     je else_level3       # if !=, jmp else_level3
     # if(match == 6)
     cmpl $6, %eax        # compare 6 to match
     jne else_level4      # jump if not equal to else_level4
     movl $1, tri_type    # tri_type = 1
     jmp return           # jump to return

else_level4:
     movl j, %eax
     addl k, %eax         # eax = j+k
     cmpl i, %eax         # compare (j+k) to i
     jg if_level1_fin     # if >, jump to the end of main if
     movl $0, tri_type    # tri_type = 0
     jmp return           # jump to return

else_level3:
     movl i, %eax
     addl k, %eax         # eax = i+k
     cmp j, %eax          # compare (i+k) to j
     jg if_level1_fin     # if >, jump to the end of main if
     movl $0, tri_type    # tri_type = 0
     jmp return           # jump to return

if_level1_fin:
     # this is the very bottom of the 1st main if, before
     # the second if
     movl $2, tri_type    # tri_type = 2
     jmp return           # jump to return

if_level1_num2:
     # if((i+j)<=k || (j+k) <= i || (i+k) <= j)
     movl i, %eax
     addl j, %eax         # eax = i + j
     cmpl k, %eax         # compare (i+j) to k
     jle inside           # if (i+j) <= k, jump to inside
     # or
     movl j, %eax
     addl k, %eax         # eax = j + k
     cmpl i, %eax         # compare (j+k) to i
     jle inside           # if (j+k) <= i, jump to inside
     # or
     movl i, %eax
     addl k, %eax         # eax = i + k
```

```
          cmp j, %eax          # compare (i+k) to j
          jle inside           # if (i+k) <= j, jump to inside
          movl $3, tri_type    # tri_type = 3
          jmp return           # jump to return

     inside:
          movl $0, tri_type    # tri_type = 0
          jmp return           # jump to return

return:
     /* epilog */
     movl %ebp, %esp
     popl %ebx
     popl %ebp
     ret

/* variable declarations */
.comm match, 4

/* end assembly solution */
```

Lab 4 Solution

```
/* begin assembly solution */

.globl AtoI
.type AtoI,@function
AtoI:
     /* prolog */
     pushl %ebp
     movl %esp, %ebp
     pushl %ebx
     pushl %esi
     pushl %edi

     /* put code here */
     movl $1, sign

     ## while(*ascii == ' ' || *ascii == '\t') ascii++;
     CheckWhiteSpace:
          movl ascii, %eax # put the char array pointer in %eax
          cmpb $32, (%eax) # comp this element to a space
          jne CheckTab # if not equal, check if it's a tab
          addl $1, ascii # if equal, move ahead one space
          jmp CheckWhiteSpace # reloop and check next space

     CheckTab:
          movl ascii, %eax # put the char array pointer in %eax
          cmpb $9, (%eax) # comp this element to a tab
          jne CheckPlus # if not equal, goto the next checks
          addl $1, ascii # if equal, move ahead one space
          jmp CheckWhiteSpace # reloop and check next space

     ## if(*ascii == '+') ascii++;
     CheckPlus:
          movl ascii, %eax
          cmpb $43, (%eax)
          jne CheckMinus
          addl $1, ascii

     ## else if(*ascii == '-') { sign = -1; ascii++; }
     CheckMinus:
          movl ascii, %eax
          cmpb $45, (%eax)
          jne ChecksDone
          movl $-1, sign
          addl $1, ascii
```

```
ChecksDone:
     movl intptr, %eax
     movl $0, (%eax)
     movl $0, i

LoopSearch:
     movl $0, %eax
     movl ascii, %ebx
     movl i, %esi
     movb (%ebx, %esi, 1), %al
     cmpb $48, %al
     jl LoopSearchDone
     cmpb $57, %al
     jg LoopSearchDone
     addl $1, i
     jmp LoopSearch

LoopSearchDone:
     addl $-1, i
     movl $1, multiplier

LoopCalculate:
     cmp $0, i
     jl FinishUp
     movl $0, %ecx
     movl ascii, %ebx
     movl i, %esi
     movl $0, %eax
     movb (%ebx, %esi, 1), %al
     subb $48, %al
     mull multiplier
     movl intptr, %ebx
     addl %eax, (%ebx)
     movl multiplier, %eax
     movl $10, %edx
     mull %edx
     movl %eax, multiplier
     addl $-1, i
     jmp LoopCalculate

FinishUp:
     movl intptr, %ebx
     movl (%ebx), %eax
     mull sign
     movl %eax, (%ebx)
```

```
     return:
          /* epilog */
          popl %edi
          popl %esi
          popl %ebx
          movl %ebp, %esp
          popl %ebp
          ret

/* end assembly solution */
```

Lab 5 Solution

```
/* begin assembly solution */

.globl Fib
.type Fib,@function
Fib:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp,%ebp
     subl $8,%esp

     /* put code here */
     movl global_var, %eax
     movl %eax, -4(%ebp)
     cmp $0,-4(%ebp)
     je return
     cmp $1,-4(%ebp)
     je return

     movl -4(%ebp), %eax
     subl $1, %eax
     movl %eax, global_var
     call Fib
     movl global_var, %eax
     movl %eax, -8(%ebp)
     movl -4(%ebp), %eax
     subl $2, %eax
     movl %eax, global_var
     call Fib
     movl global_var, %eax
     movl -8(%ebp), %ebx
     addl %ebx, %eax
     movl %eax, global_var

     return:
          /* epilog */
          movl %ebp, %esp
          popl %ebx
          popl %ebp
          ret

/* end assembly solution */
```

Lab 6 Solution

```
/* begin assembly solution */

.globl Factorial
.type Factorial,@function
Factorial:
     /* prolog */
     pushl %ebp
     pushl %ebx
     movl %esp, %ebp
     subl $4, %esp
     /* put code here */
     /* get 'number' (the argument) from stack */
     movl 12(%ebp), %ecx
     /* ecx is the argument now */
     cmp $0, %ecx
     je ret_one
     cmp $1, %ecx
     je ret_one
     /* save the variable locally so that after the function call
        we can perform our math
     */
     movl %ecx, -4(%ebp)
     /* decrement then call */
     decl %ecx
     pushl %ecx
     call Factorial
     movl -4(%ebp), %ecx
     /* the return was placed in %eax so we just multiply by
        our local */
     mull %ecx
     jmp return
     ret_one:
          movl $1, %eax
     return:
          /* epilog */
          movl %ebp, %esp
          popl %ebx
          popl %ebp
          ret

/* end assembly solution */
```