**Lab 10: Introduction to ARM Assembly Language**

**Due date: This lab is due during your lab session meeting for the week of April 02-06.**

# Overview

The first part of this lab details the process that translates a high-level program into machine language instructions. We use the GNU Compiler Collection (gcc) to reveal this step-by-step process The key understanding that you should develop from this section is how to look at a program file in different stages of the process and what the purpose of each step is.

The second part of this lab part describes three programs that you must write in ARM assembly language.

There are links to references for ARM assembly at the end of this document that should be very helpful.

# Part 1 - Translating from high level code to assembly code to machine code

**Note, Part 1 is an extended example of creating a C program and following it through translation all the way to execution. This example shows each step in the process. It is highly recommended you read through Part 1. Part 2 sets forth the deliverables for this lab.**

ARM is a widely used instruction set architecture (ISA). Because ARM is energy efficient and easy to license it is used in mobile devices including most Android phones, all iPhones and iPads, your Raspberry Pi, and many other devices such as TVs and DVD players. In this lab you will write ARM assembly programs for the Raspberry Pi (RPi) and run them on your Pi.

## Step 1: Create a Simple C Program

Open the nano editor on your RPi and write the following C program and name it hello.c.

[hello.c](http://courses.cs.purdue.edu/_export/code/cs25000:fall2016:labs:lab7?codeblock=0)

#include <stdio.h>

#define X 14

#define Y 12

#define SUM(A,B) (A+B)

int main() {  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello world**\n**");  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("SUM=%d**\n**", SUM(X,Y));  
 return 0;  
}

Then compile hello.c and run it by typing the two commands

gcc -o hello hello.c  
./hello

gcc -o hello hello.c does the following steps to generate the executable hello:

1. C preprocessor: First, the file hello.c is passed to a C preprocessor, which expands or evaluates all of the preprocessor directives (e.g. #include, #define, #if and #pragma) and then generates the file hello.i.
2. Compilation: The hello.i program is compiled, yielding the assembly language program hello.s.
3. Assembly: The hello.s program in assembly code is assembled with /usr/bin/as. This generates an object file hello.o.
4. Linker: The hello.o program is linked with other libraries and a file crt0.c with the code for start(). This generates an executable file named hello.

## Step 2: The C Preprocessor

The C preprocessor evaluates all constructions that start with “#”, such as #include, #define, #if, #ifdef, etc. Then, it generates a new C program hello.i. For example, the line #include <stdio.h> will insert the file /usr/include/stdio.h in the original hello.c to produce hello.i. You can tell the compiler to stop just after generating the hello.i file with the -E option.

Run gcc -E -o hello.i hello.c and look at how stdio.h has been placed inline into your code and the constant values for X and Y in the second printf statement have been bound (substituted in) already.

## Step 3: The Compiler

The compiler compiles the C program hello.i and translates it into a program in assembly language hello.s. The file hello.s has the assembly instructions in ASCII, so they are still human readable.

You can use the -S option to stop the compiler just after generating hello.s. This is useful sometimes to learn assembly language by writing small C programs and looking at what the compiler produces. Examine hello.s.

**Note**: Feel free to use the gcc -S command as a learning tool. However, when asked to write a program in assembly language, do not just turn in the output produced by gcc -S. As you can see here, the hello.s file contains many lines that are not necessary. Also, the output generated by a compiler is much more difficult to read than a program written by a person. It is easy to tell when a program is generated by a compiler.

## Step 4: The Assembler

The assembler translates the assembly instructions in ASCII in hello.s into an object file named hello.o that has the assembly instructions in binary. The object file is not yet executable because it has references to functions that are not defined in the object file, such as calls to printf().

The assembler is located in /usr/bin/as, and is invoked from gcc when compiling.

To generate an object file, type the following command:

gcc -c hello.c

This command generates the file hello.o. You may examine the contents with the command nm. Type the command:

nm -S hello.o

This command prints the list of functions and global variables defined in hello.o. The symbols that have a “U” are undefined symbols and the symbols with “T” are defined symbols. When defined, the first number is the address relative to the object file and the second is the size in hexadecimal. You can look at hello.o as a hex dump with the command xxd hello.o.

## Step 5: The Linker

The linker is a program that takes object files as its input, as well as static libraries (\*.a) and shared libraries (\*.so), and combines them together into an executable file. The linker assigns starting addresses to each object file, and then assigns an address to each defined symbol. The linker then stores the address of a defined symbol in the machine instructions of the object files that have this symbol as undefined.

To generate an executable, type:

gcc -o hello hello.c

You can use the nm command to find the symbols that are defined and undefined in an executable:

nm hello

## Step 6: Library Dependences

Sometimes, the symbols in the executable are marked as undefined because they are defined in a shared library. A shared library is not added to the executable file. Instead, shared libraries are loaded into computer memory once and then to be used by as many programs as may need its code. they are either already in computer memory because they were used by a program that was run earlier and the operating system has not removed, or they are loaded into memory at **runtime** (when the program is loaded), if not already in memory.

To find out the shared libraries needed by a program, use the ldd command. Type:

ldd hello

Also, you can see the symbols that a library defines using the nm command. For example, type:

nm -D /lib/arm-linux-gnueabihf/libc.so.6

Since libc contains a very large number of symbols, you can type the following to display only the symbols that contain printf() and malloc():

nm -D /lib/arm-linux-gnueabihf/libc.so.6 | grep printf  
nm -D /lib/arm-linux-gnueabihf/libc.so.6 | grep malloc

## Step 7: Writing a simple ARM Assembly Program

This is an example of taking your assembly code and making an executable file from it.

Copy the following assembly program and save it as test1.s. You need not copy all the comment statements indicated with /\* \*/:

[test1.s](http://courses.cs.purdue.edu/_export/code/cs25000:fall2016:labs:lab7?codeblock=1)

|  |
| --- |
| .data  /\* Assemble the following statements into the data section. \*/  print\_string: .asciz "Hello World\n"  /\* Assembler directive to assemble the ASCII string constant \*/ /\* into consecutive **byte** addresses followed by a zero **byte**. \*/    .text  /\* Assemble the following statements into the text section \*/  .global main /\* Make the symbol "main" visible to the linker \*/  main:  /\* main: is a label for the start of this program. \*/  /\* main: stands for the address of the first instruction of the program in memory. \*/  push {r4-r9, fp, lr}  /\* Push registers r4 through r9 that printf will use, along with fp and lr \*/  ldr r0, =print\_string  /\* Load r0 with the address symbolized by "print\_string", the first **byte** \*/  /\* of the Hello World string in the data segment. Printf will use this. \*/  bl printf  /\* Branch with link to "printf". Saves pc into lr then branches to printf. \*/  pop {r4-r9, fp, pc}  /\* Restore registers r4 through r9, fp and pc allowing graceful return \*/  /\* from printf. Without the push and pop a segmentation fault occurs. \*/ |

To assemble and run the program, type:

gcc -o test1 test1.s ./test1

# Part 2 - Graded Assignment

Write the following simple programs in ARM assembly language. You cannot use gcc -S as a “template”. **You have to write your own programs by hand in assembly.**

1. Write a program larger.s that reads two numbers from stdin, and prints the larger of the two numbers. The numbers can be positive, negative, or zero. To input the numbers to your program, you can use 'scanf' or something similar. Type one number followed by the enter key, then type the second number followed by the enter key, and then print the result to the terminal by using 'printf' or something similar. An example of what the output should look like is shown below.

pi@raspberrypi:~/cs250/lab7-src$ ./larger

-5

7

7

pi@raspberrypi:~/cs250/lab7-src$ ./larger

1501

0

1501

2. Write a program length.s that reads a string from stdin and prints the length of the string. You cannot use the C library function “strlen” in this assignment. The length of the string is guaranteed to be less than 100 characters long, and there will be no white space. Type the string in the terminal and output the length to the terminal. An example output is shown below.

pi@raspberrypi:~/cs250/lab7-src$ ./length

HelloWorld!

11

pi@raspberrypi:~/cs250/lab7-src$ ./length

12345678901234567890

20

3. Write a program sumarray.s that reads 5 numbers from stdin and prints the sum of the numbers. The number of inputs is arbitrary and could be many more, so use a loop to read in the 5 numbers. For example, a loop would look like this. In other words, do not copy/paste the scanf and format character 5 times in your program.

while x < 5 input[x] = scanf("%d") x++

An example of the output is shown below.

pi@raspberrypi:~/cs250/lab7-src$ ./sumarray

1

2

3

4

5

15

pi@raspberrypi:~/cs250/lab7-src$ ./sumarray

1

-2

3

-4

5

3

# Submitting your work

Turn in this lab by attaching your code at the end of this document and upload the entire document to Blackboard.

A TA will, in person, ask you to run your code on your Raspberry Pi, perform test cases with your code, and look at your source code. Your TA will also ask you questions about your work for 40% of the score for this lab.

# Grading Criteria

|  |  |  |
| --- | --- | --- |
|  | **Criteria** | **Points** |
| 1 | larger.s functions correctly and is nicely commented on each line (20) and question (10) | 30 |
| 2 | length.s functions correctly and is nicely commented on each line (20) and question (10) | 30 |
| 3 | sumarray.s functions correctly and is nicely commented on each line (20) and question (10) | 30 |
| 4 | Student is able to explain what each part of the code does in lab and answer questions from the TA. | 10 |
|  | Total Score |  |

## References

[ARM Instruction Set](http://courses.cs.purdue.edu/_media/cs25000:tutorials:arm_inst.pdf)

Here are some other good references that can help you in your programming

* [ARM assembly for beginners](http://courses.cs.purdue.edu/cs25000:tutorials:tutorial8)
* [ARM Assembly Tutorial 1](http://www.peter-cockerell.net/aalp/html/frames.html)
* [ARM Assembly Tutorial 2](http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/)

[ARM Assembly Language Reference Sheet](http://www.cs.purdue.edu/homes/cs250/LectureNotes/arm-ref.pdf)

# ATTACH YOUR CODE HERE

<attach code here>