# CS061 – Lab 9

**Assembly Code Generation**

## 1 High Level Description

The purpose of this lab is to demonstrate what happens when a C or C++ compiler generates assembly instructions from C/C++ source code, and how optimizations impact the generation of code.

## 2 Our Objectives for This Week

1. Exercise 1 – Simple “Hello, World” in C
2. Exercise 2 – Simple “Hello, World” in C++
3. Exercise 3 – Unrolling Loops

## 3 Exercises

Introduction

At the beginning of this course we discussed the role of assembly language. Assembly code is the language of a specific processor (or specified as an ISA). When high-level programming languages such as C/C++, Python, Java, etc. are run, they must be first converted to the ISA for the platform they will run on.

For instance, the following is a simple C/C++ function that determines, recursively, whether or not a string is a palindrome. It is similar to the C++ code presented for programming assignment 5:

```
int is_palindrome(char *first, char *last) {
    if (first >= last) return 1;
    if (*first != *last) return 0;
    return is_palindrome(++first, --last);
}
```

If we compile this code using something like GCC or CLang, it would first generate assembly in the code of the platform we are compiling for. Most likely x86_64 or Arm, since those are the most popular processors for laptops and desktop computers. However, someone created a C/C++ compiler for the LC-3.

The code generated for the code above is in the file with this lab called palindrome_generated.asm. Open this file and take a look. Lines 19 through 87 are the code for this recursive method. That's a total of 68 lines of code, not including blank lines. I have written a similar method in pure assembly that takes only 33 lines of code. That's half the code, and less code means faster execution times.

This phenomenon is why assembly code was important to understand in the past. Now, however, compilers have gotten smarter and more complex, and can optimize the code they generate to be more efficient. Today, the code generated is likely to be much more efficient than the code written manually by humans.

The rest of this lab will explore the concept of code generation and optimization with some simple examples. We'll use some simple C/C++ codes to generate assembly. Normally we would do this using the GCC or CLang C/C++ compilers using a command similar to:

```
c++ -o program.asm -S program.c
```

The -o flag tells the compiler to name the output program.asm, and the -S flag says stop at the code generation phase of compilation. The output is then the program, after it has been optimized, in assembly.

Since not all students have access to a compiler on their computers, we'll use a web tool instead.

**For the following exercises, enter the text directly into this document, or write them into a separate document or on paper.**

Exercise 1

Now let's explore how optimizations impact the way code is generated from a simple "Hello, World" program in the C programming language. Go to the online compiler at https://godbolt.org/. First, at the top toward the middle of the page, select the C programming language (C++ is selected by default). Next, select x86-64 gcc 9.4 just next to where you selected the C0..g language (x86_64 gcc 12.1 is selected by default). Then replace the C code on the left side with the following simple program:

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
}
```

Once you've typed in this code, you should see the assembly code generated by GCC version 9.4 for the x86_64. Now, we did not study this ISA in this class, but it is similar to LC-3. The main part to focus on here is the line that says "call puts". This is similar to the JSR instruction in LC-3. It backs up the current PC and sets the PC to the address of the C function called `puts` (this version of puts is different than that in LC-3).

Ex 1. Q1. How many lines of assembly code are generated for the code above?
10 lines of code

Ex 1. Q2. List all the labels you see in the generated code. In x86_64, labels have a colon after them.
.LC0

Notice that it did not call the subroutine `printf`, which is what we would expect. The `printf` routine stands for formatted print. You can pass this method a formatting string and values you want to print to the console. It is similar to cout in C++.

In this case, since we're only printing a plain string, and not any values of a variable, the compiler recognizes that it would just be faster to call `puts` instead, which printf eventually calls once it figures out what to print.

Now, change the code in the C editor to the following:

```
#include <stdio.h>

int main() {
    printf("Hello, world! %d \n", 10);
}
```

Ex 1. Q 3. How many lines of assembly code are generated for this new code?
2 extra lines

Ex 1. Q 4. Do you see a call to the `printf` or `puts` in the generated code?
yes

Since we are actually sending a value for the printf subroutine to format, the code generated actually calls `printf` instead of `puts`.

Now let's explore what happens if we tell the C compiler to do further optimizations. Both GCC and Clang allow programmers to specify how much and what type of optimizations to do. It is beyond the scope of this lab to go into much more detail about the types of optimizations. However, let's explore the result of doing different amounts of optimizations on this code.

Ex 1. Q 5. Write down the x86_64 code generated for the C code above.

```
.LC0:
        .string "Hello, world! %d \n"
main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 10
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

At the top left of the godbolt website, next to the assembly target (x86-64 gcc 9.4) there is a text input box labeled  “Compiler options”…. Change the contents of this text input box with -O1. To be clear, that is a single -, a capital O and the number 1. This tells the compiler to do some basic code generation optimization.

Ex 1. Q 6. Write down the x86_64 code generated for the same C code with the -O1 compiler option

```
.LC0:
        .string "Hello, world! %d \n"
main:
        sub     rsp, 8
        mov     esi, 10
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        add     rsp, 8
        ret
```

Ex 1. Q 7. How many lines of assembly code were generated with the compiler options -O1?

1 less line from last time

Now do the same thing with the compiler option -O2, and write the result below.

Ex 1. Q 8. Write down the x86_64 code generated for the same C code with the -O2 compiler option

```
.LC0:
        .string "Hello, world! %d \n"
main:
        sub     rsp, 8
        mov     esi, 10
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Ex 1. Q 9. What is the difference between the code generated using the -O2 and the -O1 compiler options?

-O1 tends to use the mov instruction constantly while the -O2 uses xor

Now do the same thing with the compiler option -O3, and write the result below.

Ex 1. Q 10. Write down the x86_64 code generated for the same C code with the -O3 compiler option

```
.LC0:
        .string "Hello, world! %d \n"
main:
        sub     rsp, 8
        mov     esi, 10
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Ex 1. Q 11. What is the difference between the code generated using the -O3 and the -O2 compiler options?

There's no difference

At some point there is not much more optimization you can do for such a simple program.

Exercise 2

Now let's explore how optimizations impact the way code is generated from a simple "Hello, World" program in the C++ programming language. Go to the online compiler at https://godbolt.org/. First, at the top toward the middle of the page, select the C++ programming language (C++ is selected by default, but C might be selected if you're continuing from the previous exercise). Next, select x86-64 gcc 9.4 just next to where you selected the C ++ programming language (it may already be selected from the last exercise). Then replace the C/C++ code on the left side with the following simple program:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!\n";
```

```
}
```

Ex 2. Q 1. How many lines of assembly code are generated for this new code?
39 lines

Ex 2. Q 2. How does the size of this assembly code compare to the size of the code generated for the C version of “Hello, World”?
The size of the code is way larger than the simple C language code.

Ex 2. Q 3. What line of the assembly do you see a call to the `operator<<` in the generated code?
Line 8

As you can see, the C++ code generates more assembly than the same type of program in C. This is a common criticism of C++. In general, it produces about 10% more code that C over larger programs. In most cases the design features of C++ outweigh the extra code, making it a good choice for many projects. In other cases, especially smaller cases on smaller hardware such as microcontrollers, C makes more sense.

Finally, for this exercise, on the generated code side, go to “Output…” and uncheck “Demangle Identifiers”. You should see a change in the C++ object names from something that is somewhat recognizable, to something that really does appear to be mangled.

Ex 2. Q 4. What has the name of the `operator<<` subroutine been changed to? Write it mangled name below

```
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
```

The programming languages C and C++ use the same tool called a linker to build the final executable. As a matter of fact, most compiled languages use this same linker. Because of this single tool, names of classes and subroutines must follow the naming conventions of the C programming language, which does not allow characters such as ‘:’, ‘<’ or ‘>’. To ensure that all subroutines are unique, C++ mangles the names of classes and objects, such as std::cout, using legal subroutine characters. This quirk of the programming tools can sometimes make debugging difficult.

Exercise 3
In this final exercise we’ll explore one more common and simple optimization technique used by compilers. It’s called unrolled loops. Many of you have used this technique in your code and assumed it was the inefficient way to do certain loops. The reality is, however, that it can produce more efficient code.

Let’s look at what unrolling a loop is, and then we’ll talk about its efficiency.

Go to the online compiler at https://godbolt.org/. You may use either the C or the C++ programming language. Either language will have the same result. Make sure to select x86-64 gcc 9.4 just next to where you selected the C or C++.

Now in the C/C++ window, type the following code:

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 3; i++) {
        printf("Hello, World!\n");
    }
}
```

Ex 3. Q 1. How many times do you see the call to the `puts` or `printf` subroutine?

One time

Ex 3. Q 2. Given the code above, does this number of calls to `puts` or `printf` make sense? Why or why not?

Only one call for puts does make sense because right after that call, an add function is called and loops for the amount of times it needs according to the c++ loop.

Now let's unroll the loop and see what happens. In the "Compiler options…" input type the following options

```
-O -funroll-loops
```

Wait for the code to be regenerated.

Ex 3. Q 3. Now how many times do you see calls to the `puts` or `printf` subroutine?
3 times

Ex 3. Q 4. Do you see any instructions that are used for looping, such as anything that starts with `br`?

No

The code generated after adding the -O -funroll-loops compiler options no longer use branching, and instead copy the contents of the loop body and repeat them the proscribed number of times in the loop. In this example, the calls to `puts` or `printf` have been copied four times, because the compiler can tell at compile time how many times the loop should be executed. If the number 4 were changed to an unknown value at compile time, the loop could not be unrolled.

Now let's observe what happens when we increase the number of interactions in the loop. Change the line of the C/C++ code that has for (int i = 0; i < 4; i++) by replacing the 4 with a 5.

Ex 3. Q 5. How many times do you see the call to the `puts` or `printf` subroutine?
5 times

Keep incrementing the value to the right of the '<' and stop when you stop seeing the loop being unrolled.

Ex 3. Q 6. At what value does the unrolling stop?

It stops at 5 times

For this compiler, there is no gain in performance for unrolling loops from this value until 19 iterations. After that it starts unrolling again in a slightly different way.

Change the limit of the loop to 20.

Ex 3. Q 7. Does the compiler now unroll the loop?
Yes

Ex 3. Q 7. How many times do you see the call to the `puts` or `printf` subroutine?
10 times

Ex 3. Q 7. Is there still a loop involved in the code generated? (Hint: are the any instructions that start with br)
There's no instructions that start with br.

We have now seen what loop unrolling is. The question is, how does it increase the performance of a program (or how does it decrease the amount of time it takes for the code to run)? This question will be answered in a later class in greater detail, but for now all you need to know is that branching can negatively impact performance. By unrolling the loops you can avoid this performance hit.

## 4 Submission

Your TA will maintain a Google Sheet for questions and demos.
Demo your lab exercises to your TA ***before you leave the lab***.
If you are unable to complete all exercises in the lab, demo the rest during office hours *before* your next lab to get full credit. You can demo during the next lab with a 3 point deduction.
*Office hours are posted on Canvas (under Syllabus -> Office Hours) and Discord (#office-hours).*

## 5 So what do I know now?

... more about what happens when you compile your code in languages like C and C++ :)