# CMSC 216 Project #5 Summer 2021

**Assembly Language Programming** **Due: Tue, Jul 20, 11:55 pm / Fri, Jul 23, 11:55 pm**

## 1 Overview

In this project you will write AVR assembly code that corresponds to C code provided. There are two deadlines associated with this project:

- Tue, Jul 20, 11:55 pm - Your code must pass the first two public tests (PUBLIC 1, PUBLIC 2) in the submit server. That is the only requirement for this deadline. Unlike previous projects, this first deadline is worth 1.5% of your course grade (NOT .5%). Notice you can still submit late for this part.
- Fri, Jul 23, 11:55 pm - Final deadline for the project. Notice you can still submit late (as usual).

**IMPORTANT: You must implement this project individually. You may not work with others.**

## 2 Objectives

To practice writing assembly code that corresponds to C code.

## 3 Grading Criteria

Your project grade will be determined by the following:

| | |
|---|---|
| Results of public tests | 50% |
| Results of secret tests | 38% (8% checks comments) |
| Manual (TA) Style Check (indentation (4 pts), meaningful comments (8 pts)) | 12% |

## 4 The Makefile

In the code distribution you'll see two .c files for each function you need write; you'll fill in the third (a .S file).

- `x_reference.c` - A reference C implementation of what you should write.
- `x_driver.c` - A main() that invokes your code (or the reference, depending on the Makefile) with various inputs to print the output.
- `x.S` - Here is where you will define the function that in AVR assembly mimics the reference code.

The Makefile has several rules. For example, for palindrome.S:

```
make palindrome_s     /* builds executable */
make palindrome_s.run /* runs the program using simulator */
make palindrome_s.gdb /* runs the debugger */
```

Although the base name of the assembly file is palindrome, we use palindrom_s. You can also see the expected output for a test based on the reference code (C code). For example, for palindrome.S:

```
make palindrome_r.run /* runs the reference code (C code) */
```

Notice the use of _r. By the way, the driver code contains the public tests.

## 5 Specifications

For each C routine, create an assembly program that produces the same output, given identical input. Your primary goal should be to produce a working version of each program. Your secondary goal is to make it no more complicated than it needs to be.

The input is provided by main(); you will not be expected to read integers or halt (cli/sleep) at the end of your functions. You need only implement the routine, and the `_driver.c` will invoke the routine as needed.

### 5.1 Palindrome

Ask if it seems unclear. Palindromes are words or phrases that are the same forward as backwards, the C code shows an example of how to do it.

You may optimize the code, since array subscript operations are not straightforward. (You need to get to a pointer to read from memory, the pointer is in X, Y, or Z, and although you can access at a constant offset from a pointer ("LDD Rd, Z+q") there is no single AVR instruction for array subscripting (variable pointer + variable offset).)

### 5.2 Fibonacci

Given $n$ as a (`uint16_t`) parameter, it returns the $n$th Fibonacci number. This routine **must** be implemented in a recursive fashion otherwise you will lose most of the credit.

### 5.3 Integer Square Root

Compute the square root of an integer, using the bitwise algorithm at `https://en.wikipedia.org/wiki/Integer_square_root`.

You will need the logical shift instructions. To shift left by one, "lsl r24". To shift left by two, repeat the instruction. Same for shifting right.

### 5.4 Reverse Prefix Sum

Transform an array by adding the value at an index to the sum of the remainder of the array. Return the sum. The return value may be a 16-bit integer, but the array is of 8-bit integers. (Do not concern yourself with behavior on overflow, just assume it will not happen.) The reverse_prefix_sum routine **must** be implemented in a recursive fashion otherwise you will lose most of the credit.

### 5.5 Rules

1. You may refer as much as you like to the output of avr-gcc -S. You may run "`make isqrt_reference.s`" for example. We expect you to use this information only to help with individual fragments, for example, to learn how to add two words together, call shift instructions, find which instructions to execute, etc. We expect you to avoid copying any code from the compiler output.
2. (Restated...) You are expected to write the assembly code; this is preparation for exam questions. You can look at the compiler's output if you need inspiration, but turning in something conspicuously similar to the compiler's output will lose (at least) style points. The output of the compiler is inefficient, uncommented, and has an identifiable style that is more complicated than what humans would write. Notice that you don't need to look at the compiler's output in order to implement the project.
3. Don't change the C code.
4. You don't have to worry about the maximum length of 80 for a line of code. (You might write long comments.)
5. Your comments should be descriptive of the intent, not of the action. A comment per line after a semicolon descriptive of what is happening is necessary. (I.e., "stash n for later use" is an ok comment. "move r24 into r20" is not.) A description of the variable that each register represents is useful.
6. Obey the callee-save and caller-save conventions; you must not clobber registers that the C driver does not expect you to clobber. If you have close control over only calling functions that you have person-

ally written, you may be able to get away with skipping saving some caller-save registers; do not take advantage of this feature.

7. Do not use rcall to call functions.

### 5.6 Style grading

For this project, some style guidelines are obviously different, as you are writing in assembly language, not C. Please pay close attention to these guidelines:

1. Reasonable and consistent indentation is still required. Your editor should help. Convention is to indent all instructions by one "tab", to not indent labels, and to align per-line comments starting at the same column. Our examples are generally formatted this way.
2. Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).
3. Your code must be thoroughly commented. Assembly language can easily become unreadable without proper documentation, so it is absolutely necessary that you comment your code.
4. Use appropriate whitespace, especially between blocks of instructions that are performing different tasks.
5. There should be no need for global variables. Using global variables for format strings (e.g., .asciz "%d") is fine.
6. Obey the callee-save and caller-save conventions; you may not clobber registers that the C driver does not expect you to clobber.
7. Your entire assembly code needs to be commented. At least 70% of lines having instructions must have comments, otherwise you will lose credit.
8. Make sure you leave at least one space between an instruction and the semicolon that starts a comment. You need comments next to the instructions (on the same line).
9. You can use either the strlen (from the C string library) or your own (from the assembly exercise). You don't need to include any files to use the C string library; you can call strlen as its linked using avr-gcc.
10. Even if you are not planning to implement a function for a while, you need to provide an empty shell (i.e., name and ret instruction) that will allow the submit server to build your code.
11. Commonly seen Assembly mistakes can be found at:

    `http://www.cs.umd.edu/~nelson/classes/resources/assembly_avr/#mistakes`

## 6 Other

1. For your code to build in the submit server, you need to add to each .S file a shell of the function you need to implement (similar to the provided fibonacci.S file).
2. Feel free to use code examples provided in lecture.
3. Submit often so you have a copy of your code in the submit server.
4. Adding your own targets to the Makefile is fine, but do not modify the ones provided.
5. For reverse_prefix_sum you need two format strings: "%d" and " %d" (space before %). A test will fail if you are not using the correct format string. See the reference code to see when to use each format string.

6. We cannot provide an extension because you get a ”no more processes” message. in the grace system. We have no control on how many people are using the system at a particular time. This is why is so important you don’t wait until the last minute to submit or work on your project.

7. If you get a ”no more processes” error, you may try to log on to another grace node (e.g., grace2, grace3, etc.)

## 7 Submission

Submit as usual.

### 7.1 Deliverables

The only files we will grade are your .S source files.

## 8 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.