Homework 4

Revision **0**

Instructions

*Note: Late HW is **not** accepted! Put your **"last name, first name," the course number (3744), and the HW number** in the **top right hand corner** of the first page of **all** HW assignments. Also for **all** homework, use file name **`HWx.pdf`**, where x is replaced by the homework number, e.g., **`HW3.pdf for homework 3`**. Do **NOT** put your social security number or your UF ID number on your HW. Include **all** program listings (if any), i.e., list files. When possible, **verify** your program solutions with the simulator.*

This homework will compare/contrast coding in C and Assembly. When programming in Assembly, you explicitly write each instruction that you want the processor to execute. It requires you to think in terms of the language that the processor understands, which isn't usually intuitive. When you program in C, a compiler converts the C into Assembly and then machine code. Writing code in C (or other high-level programming languages) is more human-friendly, using more normal language and algebraic expressions to accomplish your programming goals. The compiler job is to convert your C code into a language that is understood by the processor, i.e., Assembly and ultimately machine code. For example, instead of manually storing values in registers R16-R30, the compiler decides where in memory to store values.

Variables:

In C code, you must initialize a variable (a value that can change) of the type needed (int, uint8_t, uint16_t, char, float, double, etc.) to be a placeholder for the value you are trying to use. For example, the following initializes a variable **`temp`** of type unsigned 8-bit integer to an initial value of hex 0; the compiler (not you) decides where in memory to store and reference this value.

```
uint8_t temp = 0x00;
```

Functions:

In Assembly language, you have used subroutines for distinct (and often repeated) functions. You have probably passed parameters to these subroutines by loading values into specific registers used inside the subroutine just before the subroutine call and returned values from the subroutine by using a different (or the same) register. The C equivalent of a subroutine is called a **function**. Unlike Assembly, in standard C you have to declare a function (with a **prototype**) before you can use it. You can define the function either before or after the main routine (called **main**). (I suggest that you place your functions after the end of the main routine.) A function itself has the following format:

```
type functionName(type parameter1, type parameter2)
{
     //define the contents of the function
}
```

For example, if you had a function called "ADD" that took in two unsigned 8-bit integers, added them, and returned their sum as an unsigned 8-bit integer, a possible prototype for this function follows.

```
uint8_t ADD(uint8_t num1, uint8_t num2);
```

The function itself would perform the required mathematical operations necessary to find the average of two numbers. If you define a function after main, you must declare it before main (with a prototype). (I always want you to use a prototype, even if your code will work without it.) You can call the function in main, passing the necessary parameters to the function. For functions that have a return value other than **void** (nothing), you can declare a variable of the same type in main and simply set the function equal to the variable, as shown below.

```
result = ADD(37, 73);       // function call
```

Do to the prototype defined above, `result` must be initialized as type `uint8_t`.

Compiler:

A compiler is used for creating machine (object) code (and other intermediate files) for high level languages (like C) just as an assembler is used to create machine code for Assembly language. To show how the compiler makes decisions about how to generate the assembly code, we need to refer to some documentation to know where it stores certain values.

From the AVR doc42055 we see that the compiler uses registers starting from r24 to hold the returned values from functions. So looking in r24 directly after the function call, we would see the result of the function. The following is available in section 6 of this manual.

Return values use the registers r25 through r18, depending on the size of the return value. The relationship between the register and the byte order is shown in Table 6-1:

Table 6-1. Relationship between the register and the byte order.

Register	r19	r18	r21	r20	r23	r22	r25	r24
Byte order	b7	b6	b5	b4	b3	b2	b1	b0

C versus Assembly:

When you code in C, you must rely on a compiler to convert your code into machine language. There are several levels of optimization available in C (for options such as minimizing object code size or maximizing execution speed), but you can probably write more efficient code in Assembly.

This lack of control for creating the final object code when you write in a high level language like C is the reason that sometimes writing in Assembly is preferred. Only Assembly language gives you full control over the efficiency of the program that will ultimately run on a microprocessor.

In this assignment will compare code written in Assembly to Assembly language code that is **generated** from your C code.

Homework 4

Revision **0**

Assignment:

1. Write two programs, one in Assembly language (asm file) and one in C (c file). Each should call the subroutine/function "average" that takes two 8-bit inputs and returns the average of those two inputs.
   a. With no compiler optimization, how many assembly instructions does the compiler generate to accomplish the same results as your assembly code? What about when the compilation is optimized for size? (See the notes below describing how to set the optimization level. Also note that this is referring to the assembly generated from the C code, not the C code itself.)
   b. In the C program, watch r24 in a **Watch Window** in Atmel Studio and record the values or r24 directly after the function call. Concatenated, this should hold the result of your `avg` function.
   c. Run the C version and store the result of your average function into a variable, i.e.,
      ```
      uint8_t temp = avg(10, 20);
      ```
      At what address does the compiler stores the variable temp?
   d. To what type of memory is it stored?
2. Write another two programs, one in Assembly and one in C as before. This program is a subroutine or function to interface with the keypad in your kit. Compare the assembly code as done in the previous section to note the length differences that occur when the needed code becomes larger.

Include readable copies of the code for the Assembly language, the C program, and the Assembly language program generated from the C program in your pdf file homework submission.

Notes:

- To divide by 2 in assembly you can use an arithmetic shift right.
- In order to view the Assembly generated by this C code, place a breakpoint on the last instruction in main. When the code hits that breakpoint go to **Debug → Windows → Disassembly**.
- If you want to be able to step through the C code, you must turn off all optimization. At the top right of your screen, select the picture of a chip (next to the word ATxmega128A1U) then select **Toolchain → AVR/GNU C Complier → Optimization → Optimization Level: None (-O0)**