CMSC 216 Arduino (AVR) Assembly Basics Exercise Summer 2022

Assembly Basics Exercise **Due: Thu, Jul 7, 11:55 pm**

1 Objectives

The goal of this exercise is to start to prepare you to write your own functions using assembly. Assembly has a different vocabulary from C and Java (instructions on registers instead of expressions on variables), and a different structure (branches and labels instead of conditionals and blocks). This exercise is mostly about practicing the vocabulary of loads, compares, and branches, and about following the calling convention.

It can be tricky to put together a working assembly function without knowing the useful instructions first. We will provide the high-level structure of some simple routines; all you have to do is fill in the actual instructions that accomplish the goal.

In this exercise, we will fill in a template, one (or maybe two) instruction(s) at a time. It will be up to you to use the documentation and our in-class examples to find the appropriate assembly instructions and operands to perform the task described in the template.

You can discuss this exercise with other classmates, but you may not exchange any code nor write code together.

2 Grading Criteria

Your assignment grade will be determined with the following weights:

Results of public tests 20 pts

3 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. We take academic integrity matters seriously. Please do not post assignment solutions online (e.g., Chegg, github) where others can see your work. Posting code online can lead to an academic case where you will be reported to the Office of Student Conduct.

4 Context: Use of Assembly in Practice

Assembly language is used in two key situations. First, when a machine first starts up, assembly language routines read in the operating system. You might call this the "boot loader". Second, when software needs to interact with hardware, it uses special instructions (in, out) or special registers that aren't exposed to C code. Here, we're working in the context of a C program, where these routines will be called from C, but we won't do anything the compiler couldn't.

You'll also see "hand tuned" assembly called from C when a routine needs to be optimized for size, speed, or to ensure specific timing. Only extremely important or frequently used code gets this treatment, but on small microcontrollers without much memory for instructions, optimization for size is common.

5 Template Style

You will fill in the code in a template that uses the following conventions.

Comments led by three semicolons are descriptive background. They represent context, or show the prototype of the function as it will be called by C code. Comments led by two semicolons represent an instruction you're meant to fill in. Leave all template comments in your solution. You may add comments after the line of code using a single semicolon.

Labels are already included. Labels represent the beginnings of functions and branch targets. Numeric la-

bels are branch targets and need not be unique because the assembler will find the nearest label having that number. For example, `brge 1f` will search "forward" for the next label "1".

The ".global" declarations tell the assembler that the label should be exported, which allows the linker to connect a function call in the C driver file to this function implementation. You can think of it like "extern" in C, but we've provided it for you in the template. We can use directives like these to declare global variables as well.

In some cases, the comment describing what to do will require two related instructions (e.g., add/adc, cp/brge, clr/clr) to complete. It is permitted, but not recommended, to deviate somewhat from these instructions, or even to find one instruction that accomplishes two-comments-worth of the exercise.

Assembly is generally formatted so that labels are flush-left and instructions are indented by a "tab". Indentation makes the labels more visible. The assembler, unlike make, does not *need* a literal tab character; indentation can be accomplished with spaces and indentation is optional to the language (unlike Python or Fortran). Emacs "assembler-mode" will try to help with this formatting; other editors may do the same.

6 Documentation

See http://www.cs.umd.edu/~nelson/classes/resources/assembly_avr/

7 Functions

7.1 Five

This function will return the value "5" as a 16-bit integer. You will need to figure out where the return value belongs and how to load an immediate (constant) into it.

```
;;; Five
    .global Five
Five:
    ;;; uint16_t Five(): return 5 as a uint16_t
    ;;  load immediate
    ;;  load immediate or clear
    ;;  return
```

Note that until you fill in the return instruction, the processor will just keep going, kind of like a switch/case without a break in C.

7.2 Max

This implementation of Max relies on the first parameter having the same location as the return value.

```
;;; Max
    .global Max
Max:
    ;;; uint8_t max(uint8_t x, uint8_t y): return the greater of the arguments
    ;; compare x to y
    ;; if x >= y, branch 1f
    ;; copy y into return value / first param slot.
1:
    ;; return
```

7.3 Strlen

Note that the return value is a 16-bit integer and a pointer is a 16-bit value. It will sometimes take two instructions to alter values.

```
;;; Strlen
    .global Strlen
Strlen:
    ;;; uint16_t Strlen(char *arg)
```

```
    ;; copy argument to X (r27:26) pointer
    ;; initialize return value to zero
2:
    ;; load X with post-increment
    ;; if loaded value was zero, branch 1f (label 1, forward)
    ;; increment return value
    ;; jump 2b (label 2, backward)
1:
    ;; return
```

8 Using the simulator and debugger

The makefile provides useful rules for executing the simulator and debugger.

To run the exercise in the simulator:

```
% make exercise.run
```

This should be equivalent to running "simavr exercise", but includes some command line arguments that should be redundant on grace.

To run the exercise in the simulator, but configure it to wait for avr-gdb (gdb compiled to debug AVR code) to connect so that you can single-step, use:

```
% make exercise.gdb
```

You'll then want to use typical gdb commands, like "break Five", "break Strlen", "continue", "step", etc. Particularly useful to us is "info registers".

Note that gdb doesn't expect the Harvard architecture used by the AVR, so addresses above 0x800000 are really just memory addresses in the normal 16-bit range. This is just to separate the data memory address space from the instruction memory address space. This is why, when running "info registers" you'll see the stack pointer (SP) has a value like "0x800aef". It is really just "0xaef" but in data memory, not in instruction memory.

9 Submitting your assignment

Use the submit command as if you were submitting a class project.