ECE 375: Computer Organization and Assembly Language Programming

Lab 3 – Introduction to AVR Simulation with Atmel Studio

SECTION OVERVIEW

Complete the following objectives:

- Look at a sample AVR assembly program.
- Learn how to use the Atmel Studio simulator to step through the sample program.
- Learn how to interact with the Processor, I/O View, and Memory windows.
- Begin learning about basic assembly program structures (such as loops).

PRELAB

To complete this prelab, you may find it useful to look in the AVR Starter Guide. If you consult any online sources to help answer the questions below, you **must** list these sources as references in your prelab.

1. What are some differences between the debugging mode and run mode of the AVR simulator? What do you think are some benefits of each mode?
2. What are breakpoints, and why are they useful when you are simulating your code?
3. Explain what the I/O View and Processor windows are used for. Can you provide input to the simulation via these windows?
4. The ATmega128 microcontroller features three different types of memory: data memory, program memory, and EEPROM. Which of these memory types can you access by using the Memory window of the simulator?

   (a) Data memory only
   (b) Program memory only
   (c) Data and program memory
   (d) EEPROM only
   (e) All three types

PROCEDURE

Introduction

Writing assembly code requires a lot of attention to detail. Despite your best intentions, it is easy to introduce subtle bugs into your program, especially when you are first learning assembly language programming. Often, you will try to run your code on your `mega128` board, observe that it isn't working, but have a difficult time identifying which portion of your program isn't working as intended. In other words, **simple mistakes can result in behavior that is significantly different than what you intended, which is often confusing or frustrating.** Therefore, it is very useful to know how to to simulate your code, so that you can provide "controlled" input and observe that each portion of your code is working as you intended.

In this lab, you will learn to use the simulator to step through a sample program, observing and recording key values along the way. Be sure to write down your answers to all **bolded questions** as you proceed through the lab, as you will need them at the end of the lab.

Creating Breakpoints

To get started, complete the following steps to establish a new project with the sample program included:

1. Open Atmel Studio, and create a new Assembler project.
2. Download the sample assembly program given on the lab webpage (`Lab3Sample.asm`), and include it into your project.
3. Make sure that the sample program builds without any errors.

In the sample program you just downloaded, there are several lines that have the comment "SET BREAKPOINT HERE". Before you simulate this sample code, you will need to add a breakpoint to every one of these lines. Do the following for each line that has a "SET BREAKPOINT HERE" comment:

- Right-click on the line of code, and then select **Breakpoint** → **Insert Breakpoint**.
- If done correctly, a red dot will appear to the left of the instruction, and the entire line will be highlighted in red.

For this lab to work correctly, it is very important that all required breakpoints are set. Once you have completed this step, and verified that all required breakpoints are in place, you are ready to begin simulating the sample code.

Preparing for Simulation

To start up the simulator, select **Debug → Start Debugging and Break** from the Atmel Studio menu bar (or just press Alt+F5).

Since this is your first time launching the simulator for this particular Atmel Studio project, you will be prompted to configure your `.asmproj` file with the message "Please select a connected tool and interface and try again." Press **Continue**, and then open the dropdown menu below the text "Selected debugger/programmer", and click on **Simulator**.

Then, you can press **Ctrl+S** to save this change to the project file, and then switch back over to the code editor tab, which should still have the sample program open. Finally, you can select **Debug → Start Debugging and Break** again and the simulator will properly start.

Once the simulator has started, you should see several new view tabs in the right-side window of Atmel Studio.

- If you do not see "Processor Status" anywhere, go to **Debug → Windows → Processor Status** to bring it up. It should auto-dock in the right-side window, if not you can drag the window over to dock it manually.
- If you do not see "I/O" anywhere, go to **Debug → Windows → I/O** to bring it up. Again, it should dock automatically, but you can dock it manually if needed.
- If you do not see any "Memory" windows open, go to **Debug → Windows → Memory → Memory 1** to bring one up. This memory view is traditionally docked in the lower-right panel of Atmel Studio.

Before you continue, **have your TA verify** that you have set all required breakpoints, and that you have opened the Processor Status, I/O, and Memory windows correctly.

Simulating the Sample Code – Part 1

1. We began simulation by pressing "Start Debugging and Break", so the simulator is currently in debugging (line-by-line) mode, paused at the very first instruction of our program (it should be `rjmp INIT`). This line is highlighted in yellow, which means it is the **next** instruction to be executed.
2. Since the simulator hasn't actually run any lines of code yet, take this opportunity to observe the default/initial values of some of the I/O registers you have already seen in the previous lab.
   - **What is the initial value of DDRB?**
   - **What is the initial value of PORTB?**
   - **Based on the initial values of DDRB and PORTB, what is Port B's default I/O configuration?**
3. Press **Debug → Step Into** (F11) to execute the first instruction and step forward to the next line. Observe that the flow of the program has moved to the line directly following the `INIT` label.
4. Next, press **Debug → Continue** (F5) to enter run mode. The simulator will run until it encounters the next breakpoint.
5. At this point (Breakpoint #1), the simulator has just finished executing the 4 lines of code that initialize the stack pointer. The stack pointer's value is displayed (in hexadecimal representation) in the Processor Status window.
   - **What 16-bit address was the stack pointer just initialized to?**
6. Press **Continue** again. The program has now advanced another two lines (to Breakpoint #2), and the general purpose register `r0` has just been initialized to a certain value. The contents of general purpose registers like `r0` are also displayed in the Processor Status window.
   - **What are the current contents of register `r0`?**
7. Press **Continue** again. The program flow has continued for several lines, and then paused again at the end of the first iteration of a loop structure (Breakpoint #3), which began at the `LOOP` label. The next instruction to be executed will test whether to move the program flow back up to `LOOP` (i.e., it tests whether the loop will continue). Press **Continue** again. The simulator is still pointing to the same instruction, which means that another iteration of the loop was run. Keep pressing **Continue** until the simulator stops at the first breakpoint outside of the loop (Breakpoint #4).
   - **How many times did the code inside the loop structure end up running?**
   - **Which instruction would you modify if you wanted to change the number of times that the loop runs?**
   - **What are the current contents of register `r1`?**
8. Press **Continue** again. The program flow is now within another loop structure (Breakpoint #5). Keep pressing **Continue** until the program stops at the first breakpoint outside of the `LOOP2` loop (Breakpoint #6).

- **What are the current contents of register `r2`?**

9. Press **Continue** to advance to the final breakpoint (Breakpoint #7).

   - **What are the current contents of register `r3`?**

Inserting Values into Memory

Before you advance the simulator beyond Breakpoint #7, you will need to manually insert some values into the Data Memory. Go to the Memory window, and select **data IRAM** from the "Memory:" dropdown. You should now see a table of two-digit hexadecimal values (bytes), aligned so that the first entry (top left) is the contents of the first location in Data Memory (at address $0100). The entry to its right is the contents of location $0101, then to its right is $0102, etc.

The Memory window allows you to type values directly into memory, as long as the simulation is currently paused in line-by-line mode. Before stepping through any more of the sample code (i.e., before beginning the `FUNCTION` subroutine), complete the following steps:

1. Enter the value you observed in r0 into location $0100.
2. Enter the value you observed in r1 into location $0101.
3. Enter the value you observed in r2 into location $0102.
4. Enter the value you observed in r3 into location $0103.

Simulating the Sample Code – Part 2

Once you have placed the correct values into the correct data memory locations, you can resume stepping through the sample code.

1. Press **Step Into** (F11) to move into the `FUNCTION` subroutine.

   - **What is the value of the stack pointer now that your program flow has moved inside of a subroutine?**

2. Press **Step Out** (Shift+F11), which will run the rest of the subroutine all at once and pause at the first instruction after the function call (which happens to be the end of `MAIN`).

   - **What is the final result of `FUNCTION`? (What are the hexadecimal contents of memory locations $0105:$0104?)**

STUDY QUESTIONS / REPORT

For this lab, a full report write-up **is not required**. Instead, simply submit your answers to the **bolded questions** that were asked throughout the lab. For your convenience, the questions that you need to answer are shown below:

1. What is the initial value of DDRB?
2. What is the initial value of PORTB?
3. Based on the initial values of DDRB and PORTB, what is Port B's default I/O configuration?
4. What 16-bit address (in hexadecimal) is the stack pointer initialized to?
5. What are the contents of register r0 after it is initialized?
6. How many times did the code inside of `LOOP` end up running?
7. Which instruction would you modify if you wanted to change the number of times that the loop runs?
8. What are the contents of register r1 after it is initialized?
9. What are the contents of register r2 after it is initialized?
10. What are the contents of register r3 after it is initialized?
11. What is the value of the stack pointer when the program execution is inside the `FUNCTION` subroutine?
12. What is the final result of `FUNCTION`? (What are the hexadecimal contents of memory locations $0105:$0104)?

CHALLENGE

The `FUNCTION` subroutine featured in the sample code accepts two 16-bit values as parameters, and also returns a 16-bit value as its result. To complete the challenge for this lab, provide detailed answers to the following questions:

1. What type of operation does the `FUNCTION` subroutine perform on its two 16-bit inputs? How can you tell? Give a detailed description of the operation being performed by the `FUNCTION` subroutine.
2. Currently, the two 16-bit inputs used in the sample code cause the "`brcc EXIT`" branch to be taken. Come up with two 16-bit values that would cause the branch NOT to be taken, therefore causing the "`st Z, XH`" instruction to be executed before the subroutine returns.
3. What is the purpose of the conditionally-executed instruction "`st Z, XH`"?