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. Therefore, it is very useful to know how to simulate your code, so that you can provide “controlled” input and observe that each portion of your code is working as 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. Begin by completing the following steps:

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.

Creating Breakpoints

In the sample code you have 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 for all required breakpoints to be 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 if manually if needed.
- If you do not see any “Memory” windows open, go to Debug → Windows → Memory → Memory 1 to bring it 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, this is a good time 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 can you tell about how Port B is configured by default?
3. Press Debug → Step Into (F11) to execute this instruction and move onto the next one. Observe that the flow of the program has moved to the line labelled INIT.
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. Look at the Processor Status window - what value (in hexadecimal) 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. Look in the Processor Status window - what are the contents of register r0?
7. Press Continue again. The program flow has now continued for several lines, and you are at the end of the first iteration of a loop structure (Breakpoint #3). Specifically, the next instruction to be evaluated will test whether or not 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).
8. How many times did the code inside the LOOP structure end up running? What instruction would you modify if you wanted to change the number of times that the loop runs?
9. Now that the program has moved beyond the first loop, another register is done being initialized. What are the contents of register r1?
10. 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 this second loop (Breakpoint #6).
11. Now that the program has moved beyond the second loop, another register is finished being initialized. What are the contents of register r2?
12. Press Continue to advance to the final breakpoint (Breakpoint #7). What are the contents of register r3 at this point?

Inserting Values into Memory

Before you run any more instructions in the simulator, you will need to manually insert some data into the Data Memory. Go to the Memory 1 window, and make
sure data IRAM is selected in the “Memory:” dropdown. This aligns your view with location $0100 of the Data Memory. The top-left value should be “00”, which is the hexadecimal contents of location $0100. The value directly to its right is location $0101, then $0102, etc.

The Memory window allows you type data directly into the contents of 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.

- 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 into this subroutine call?
- Press Step Out (Shift+F11), which will run the rest of the subroutine all at once, and halt again at the first instruction after the function call (which happens to be the end of MAIN). At this point, the result of FUNCTION is saved into memory locations $0104 and $0105. Show the result to your TA, and if it is correct they will check you off for the implementation portion of this lab.

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:

- What is the initial value of DDRB?
- What is the initial value of PORTB?
- Based on the initial values of DDRB and PORTB, what can you tell about how Port B is configured by default?
- What value (in hexadecimal) is the Stack Pointer initialized to?
- What are the contents of register r0 after it is initialized?
- How many times did the code inside the “LOOP” structure end up running? What instruction did you modify if you wanted to change the number of times that the loop runs?
- What are the contents of register r1 after it is initialized?
- What are the contents of register r2 after it is initialized?
- What are the contents of register r3 after it is initialized?
- What is the value of the Stack Pointer when you are inside the FUNCTION subroutine?
- What is the final result of FUNCTION? (What are the contents of memory locations $0105:$0104)?

CHALLENGE

The FUNCTION subroutine featured in the sample code takes two 16-bit values as input, and generates a 16-bit result. Answer the following to receive extra credit on this lab:

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 operands 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”?