
ECE 375 LAB 3

Simulation

Lab Time: Thursday 1000-1200

Eric Prather

INTRODUCTION

IMPORTANT: This lab does not require a writeup. This is stated explicitly in the Lab 3 Handout:

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:

PRELAB QUESTIONS

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?

In computer science, debug and run mode are two distinct sets of preprocessor directives and execution environments. Code compiled for a debug build will often not have certain subroutines condensed and comes with additional reflection data, whereas the final release tends to be fully compressed and optimized. Debug builds additionally possess additional metadata related to the program code allowing for advanced execution behaviors such as break-pointing and member value inspection. In C, common debugging software which uses this metadata include the Visual Studio Profiler and the GDB command line utility. The assembly equivalent of this, for the purposes of this class, is the AVR simulator's debug and release modes. The AVR debug mode allows the management of the program through metadata which, at a core level, can make inferences about and modify program state using an external set of metadata (not unlike the .NET framework reflection library) superimposed on the simulated hardware events. However, despite these clear advantages of debug mode, no debug compilation can ever *perfectly* represent a release build, so testing should always be done in the latter as well to prepare best for the deployment to the hardware.

2. What are breakpoints, and why are they useful when you are simulating your code?

Breakpoints are metadata flags that are used to correlate source-code lines with compiled machine code execution states. A developer can insert a breakpoint in program code and then a debug compiler will automatically generate the necessary metadata for that breakpoint to be observed in the execution environment. The most common use of breakpoints is to pause programs mid-execution so programmers can inspect the state of the program's stack and all of the data at certain pre-defined points while searching for the source of an error.

3. Explain what the I/O View and Processor windows are used for. Can you provide input to the simulation via these windows?

The IO windows show the status of the simulated IO pins and their associated registers (e.g. DDRX, PORTX, PINX). The processor window shows the state of the processor components. IO can be provided by the IO view. The simplest way to do this is by clicking on the bit checkboxes,, but you can also type in values in the value fields. Not all registers are immediately exposed through the I/O window. The processor window is similar, but instead of displaying special external or IO registers on the board, it displays those on the inside of the processor. You can similarly provide values to the registers like with the I/O window.

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

The memory window of the simulator shows **(e) all three types** of memory. This is despite the fact that the program memory is not accessible by the microcontroller itself, so it does not make sense to show it in the debugger. Program memory wouldn't be useful to a debugger anyway. EEPROM is also not pertinent. It is static over the course of the simulation. Nevertheless, the inclusion of all of these types of memory is valuable for the sake of learning.

LAB QUESTIONS




1. What is the initial value of DDRB?

The initial value of the register DDRB is 0b000000:

Name	Address	Value	Bits
 PORTB	0x38	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
 DDRB	0x37	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
 PINB	0x36	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

2. What is the initial value of PORTB?

The initial value of port b is 0b00000000

Name	Address	Value	Bits
 PORTB	0x38	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
 DDRB	0x37	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
 PINB	0x36	0x00	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

3. Based on the initial values of DDRB and PORTB, what is Port B's default I/O configuration?

Using the values from questions 1 and 2, we can determine that Port B's default I/O configuration is **input with no pullup on all pins**.

4. What 16-bit address (in hexadecimal) is the stack pointer initialized to?

After hitting the next breakpoint, the stack pointer is set to 0x10FF.

Processor Status	
Name	Value
Program Counter	0x0000004A
Stack Pointer	0x10FF
X Register	0x0000
Y Register	0x0000
Z Register	0x0000
Status Register	I T H S V N Z C
Cycle Counter	5
Frequency	1.000 MHz
Stop Watch	5.00 μ s

5. What are the contents of register r0 after it is initialized?

After r0 is initialized, it contains the value 0xFF. This is odd because I would have expected an instructor called “clear” to set it to 0x00, but I guess that just goes to show what I know.

6. How many times did the code inside of LOOP end up running?

After waiting at breakpoint #2, I pressed “continue” 5 times until I reached breakpoint #4. During this time, I landed on breakpoint #3 4 times. From this, I can tell that the loop executed four (4) times. The last “continue” did not jump back to the beginning of the loop, which is why I don’t count it as five total loop executions.

7. Which instruction would you modify if you wanted to change the number of times that the loop runs?

I would modify the instruction “ldi i, \$04” to make the loop run a different number of times. The third word on this line should change to modify it. Every time the loop executes, the value is decremented by one. Once the value is zero, the loop does not repeat.

Unnumbered Question: What are the current contents of register r1?

At breakpoint #4, register 1 is 0xAA

8. What are the contents of register r1 after it is initialized?

This question is never asked in the actual body of the lab- it just appears in the list of questions at the end. Did you mean “What are the current contents of r1”? If so, that is answered in the item labeled “Unnumbered Question”.

Important: This question appears directly after numbered question #7, which is highly suspect as an error, because it has nothing to do with the other two questions in the block of questions numbered 6 to 7. Screenshot:

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).

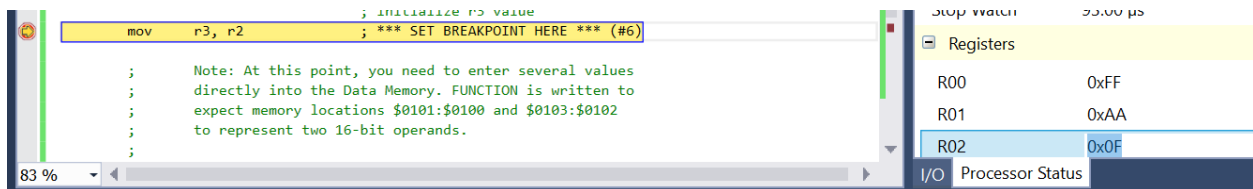
- How many times did the code inside the loop structure end up running?
- Which instruction would you modify if you wanted to change the number of times that the loop runs?
- What are the current contents of register r1?

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).

9. What are the contents of register r2 after it is initialized?

This question does not occur in this lab. Did you mean the similar question “What are the current contents of register r2”? If so, the answer is as follows:

r2 contains 0x0F.



This question appears in the text as:

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.
8. Press **Continue** again. The program flow is now within another loop structure (Breakpoint #5). Keep pressing **Continue** until the first breakpoint outside of the LOOP2 loop (Breakpoint

©2020 Oregon State University

Winter 2020

Lab 3 – Introduction to AVR Simulation with Atmel Studio

- 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?

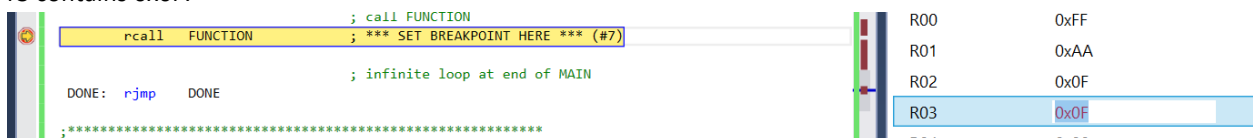
STUDY QUESTIONS / REPORT

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

10. What are the contents of register r3 after it is initialized?

This question does not occur in this lab. Did you mean the similar question “What are the current contents of register r2”? If so, the answer is as follows:

r3 contains 0x0F.



This question appears in the text as:

- What are the current contents of register r3?

The value of the stack pointer inside of this function is 0x10FD

The screenshot shows the AVR Studio IDE. On the left, the assembly code window displays the following code:

```

;-----
; Func: FUNCTION
; Desc: ???
;-----
FUNCTION:
    ldi    XL, $00
    ldi    XH, $01
    ldi    YL, $02
    ldi    YH, $01

```

On the right, the I/O View window is open, showing the current state of the hardware registers:

Name	Value
Program Counter	0x0000005B
Stack Pointer	0x10FD
X Register	0x0000
Y Register	0x0000
Z Register	0x0000

The final result of the function is 0x0eba.

```
|data 0x0100 ff aa 0f 0f 0e ba 00 00 00 00 00 00 00 00 00 00 00 00 ÿa...o..
```

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.

```
// X = $0100, Y = $0102, Z = $0104
FUNCTION(*X, *Y, out Z):
    A = *X
    X++ // $0102
    B = *Y
    Y++ // $0104
    B = B + A
    *Z = B
    Z++ // $0106
    A = X
    B = Y
    B = B + A (carry from last sum included)
    *Z = B
    Z++ // $0108
    If(no carryover bit)
        Return;
    Else
        *Z = XH
```

FUNCTION sums the values at X and Y in memory and stores them at Z in memory, where X, Y, and Z are contiguous 16-bit values from \$0100 to \$0105. **Next**, it adds *Z, *Y, and the carryover- storing the *new* carryover in a special register pin. This second sum is stored in the byte following Z's original address (because z was earlier post-incremented). **Lastly**, if there is a carryover bit, it writes the *next* 8 bits following where the previous sum was stored with the most significant 8 bits of X.

Sums two inputs, stores the result, then sums the result and the second input. Writes to designated output area. All bytes are in reverse endian (rightmost bytes are more significant than leftmost bytes).

Page 6

OUTPUT = INPUT 1 + 2* INPUT2

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.*

Adding 0xFF and 0xFF would do the trick, because it would overflow in the addition operation. If we want the extra statement to execute, that would work nicely.

3. *What is the purpose of the conditionally-executed instruction “st Z, XH”?*

It is possible that the sum results in a number that is **too big to be stored in 2 bytes**. In order to output the whole number, which may take three bytes, the new most significant byte is tacked onto the “beginning” of the sum if necessary.