# REQUIREMENTS NOT MET

N/A. All Requirements are met in this lab.

# PROBLEMS ENCOUNTERED

A problem encountered in this lab was figuring out why my table data wasn't being stored at the correct starting address in program memory. After researching for some time, the error was the result of not having realized that the address was given in data memory, being 0xAAAA, and had to be converted to program memory by dividing by 2. Since the program memory was indexed by word instead of byte like in data, this causes a long journey to find where the table was stored. Another problem was figuring out how to point to program memory that was larger than a word. This was solved by using the elpm for the RAMPZ register instead of the regular lpm.

# FUTURE WORK/APPLICATIONS

This lab was a good introduction into Atmel Studio and the ATxmega128A1U processor. This lab is to become the foundation of more complex assembly programs. If given more time, the code of the lab could have been more organized and have a much neater layout to further reduce the likelihood of mistakes and further enhance the understanding of the program. Additionally, I could have used better instructions to make the code run more efficiently or learn to write more complex programs.

# PRE-LAB EXERCISES

i. Which type of memory alignment is used for program memory in the ATxmega128A1U? Byte-alignment, or word-alignment? What about for data memory?

Word-alignment is used for program memory while byte-alignment is used for data memory.

ii. If you were to use the Memory debug window of Atmel Studio To verify that some datum was correctly stored at address 0xBADD within program memory of the ATxmega128A1U, which address would you specify within the debug window?

Since the address was 16-bit and placed in program memory, there is a left shift on the address so therefore the address that ought to be looked for in the debug window is 0x75BA.

iii. In which section of program memory is address 0xF0D0 located?

The address 0xF0D0 in program memory is located in the Application Table section according to the manual.

iv. Which assembler directive places a byte of data in program memory? Which assembler directive allocates space within data memory? Which assembler directives allow you to provide expressions (either constant or variable) with a meaningful name?

The DB assembler directive places a byte of data in program memory. The BYTE assembler directive allocates space within data memory. The EQU and SET assembler directives allow you to provide expressions with a meaningful name.

v. When using the internal SRAM (not EEPROM), which memory locations can be utilized for the data segment (.dseg)? Why?

The memory locations for the data segment starts at 0x2000 and ends at 0x3FFF as the architecture was designed that way. The .dseg allows for writing or allocating space for data memory such as temporary variables.

vi. Which instructions can be used to read from (flash) program memory? For each instruction, list which registers can be used as an operand.

The instructions lmp and elmp, can be used to read from program memory. Register Z is used as an operand for the lmp instruction family, together ZL and ZH can be used as an address pointer to write or read program memory.

vii. In the context of pointing an index to a specific program memory address within an XMEGA AU architecture, explain why and how the address value should first be altered. Similarly, in the context of pointing an index to a specific data memory address, explain why the address value should not be altered.

Pointing an index to a specific program memory address means that a 16-bit value to be moved to an 8-bit range, because this is not possible, the address value should first be broken from 16-bit address to two 8-bit address in little endian and then left shift both addresses. In contrast, pointing an index to a specific data memory address should not require any alteration to the address value because it is moving a 16-bit value to an 16-bit range, it can fully fit within.

viii. Which assembly instruction can be used to load data directly from the I/O memory of XMEGA AU microcontrollers? Which assembly instruction can be used to store data directly to the I/O memory?

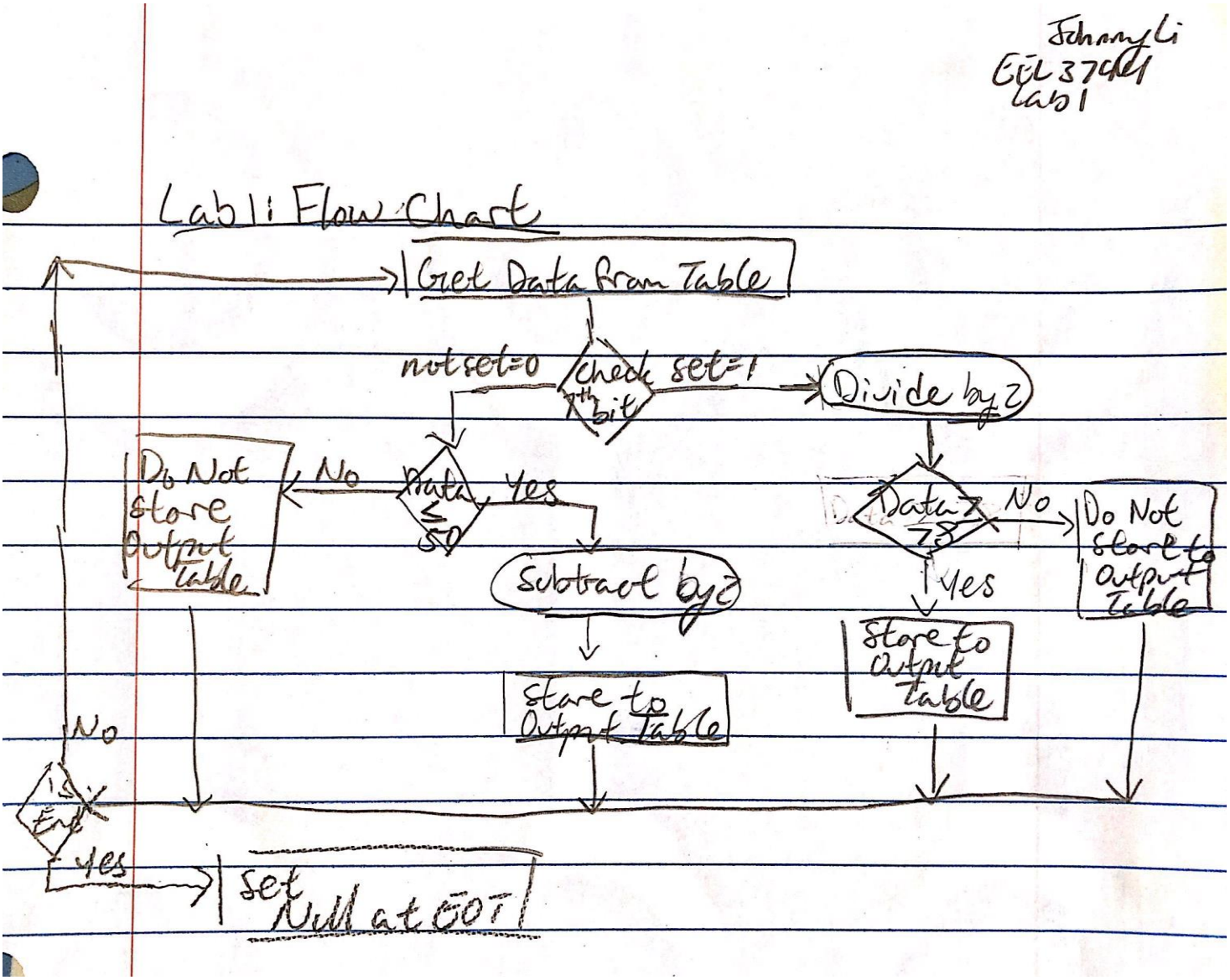
The assembly instruction to load data directly from the I/O memory is IN while to store data directly is OUT, utilizing the I/O port memory.

ix. Which assembly instructions can be used to load data indirectly from data memory within XMEGA AU microcontrollers? Which assembly instructions can be used to store data indirectly to data memory?

The assembly instruction that can be used to load data indirectly from the data memory is LD and LDD, loading with only registers Z and Y. The instructions that can be used to store data indirectly to data memory is ST and STD, also storing with registers Z and Y.

# PSEUDOCODE/FLOWCHARTS

## SECTION X (1, 2, etc.)



# PROGRAM CODE

## SECTION X (1, 2, etc.)

```
;*********************************MAIN PROGRAM*********************************
;Lab 1
;Section #: 1823
;Name: Johnny Li
;Class #: 12378
;PI Name: Jared Holley
;Description: Introduction to AVR and Assembly
	.cseg
	; configure the reset vector (ignore for now)
	.org 0x0
	rjmp MAIN

	; place main program after interrupt vectors (ignore for now)
	.org 0x100
MAIN:
	; point appropriate indices to input/output tables
	;Z points to input table.
	ldi ZL, low(IN_TABLE << 1) ; Shift due to being 16-bits in a 8-bit processor
	ldi ZH, high(IN_TABLE <<1) ; Z points to the table

	;Y points to output table
	ldi YL, low(OUT_TABLE)
	ldi YH, high(OUT_TABLE)

	; loop through input table, performing filtering and storing conditions
LOOP:
	; load value from input table into an appropriate register
	elpm r16,Z+

	; determine if the end of table has been reached (perform general check)
	CPI r16, NULL ;compare NULL and r16 to see if NULL is reached

	; if end of table (EOT) has been reached, i.e., the NULL character was
	; encountered, the program should branch to the relevant label used to
	; terminate the program (i.e., LOOP_END)
	BREQ LOOP_END

	; if EOT was not encountered, perform the first specified
	; overall conditional check on loaded value (CONDITION_1)
	BRNE CONDITION_1

CONDITION_1:
	; check if the first condition is met; if not, branch to the second
	; overall conditional check (CONDITION_2)
	; check if MSB is set
	SBRS r16,7
	;if set skip else jump to condition_2
	RJMP CONDITION_2

	; if the first condition is met, perform the relevant operation(s)
	; divide by 2
	LSR r16
	; if the resulting value is greater than or equal to 78, store it to the first available location
within the output table
	CPI r16,78
	BRSH STORE
```

```
        ; if the first overall condition was met, the second condition should not
        ; be performed and the next table value should be loaded, i.e.,
        ; the program should jump back to the beginning of the relevant loop
        ;rjmp LOOP    //<-------------

        ; if the first overall condition was not met, the second condition
        ; should be checked (CONDITION_2)
CONDITION_2:
        ; check if the second condition is met (if necessary, add any additional
        ; label[s], e.g., to handle the less than or equal condition)
        ; check value of the data is less than or equal to 50
        CPI r16, 50
        BRLO CONDITION_2P
        BREQ CONDITION_2P

        ; if the second condition is not met, the program should
        ; jump back to the beginning of the relevant loop and load the next
        ; value from the input table, i.e., the third specified condition
        rjmp LOOP

CONDITION_2P:
        ; if the second condition is met, perform the relevant operation(s)
        ; subtract 2 from the value
        SUBI r16, 2
        ; store this result to the first available location within the output table.
        rjmp STORE

        ; if the second condition was met, the program should
        ; unconditionally jump back to the beginning of the relevant loop and
        ; load the next value from the input table
        ;rjmp LOOP    //<--------

        ; end of program loop, perform specified cleanup actions
LOOP_END:
        ; store specified EOT value
        st Y+,r16
        rjmp DONE

;STORE value to output table
STORE:
        st Y+,r16
        rjmp LOOP

        ; end of program (infinite loop)
DONE:
        rjmp DONE
;*****************************END OF MAIN PROGRAM ****************************
```

# APPENDIX

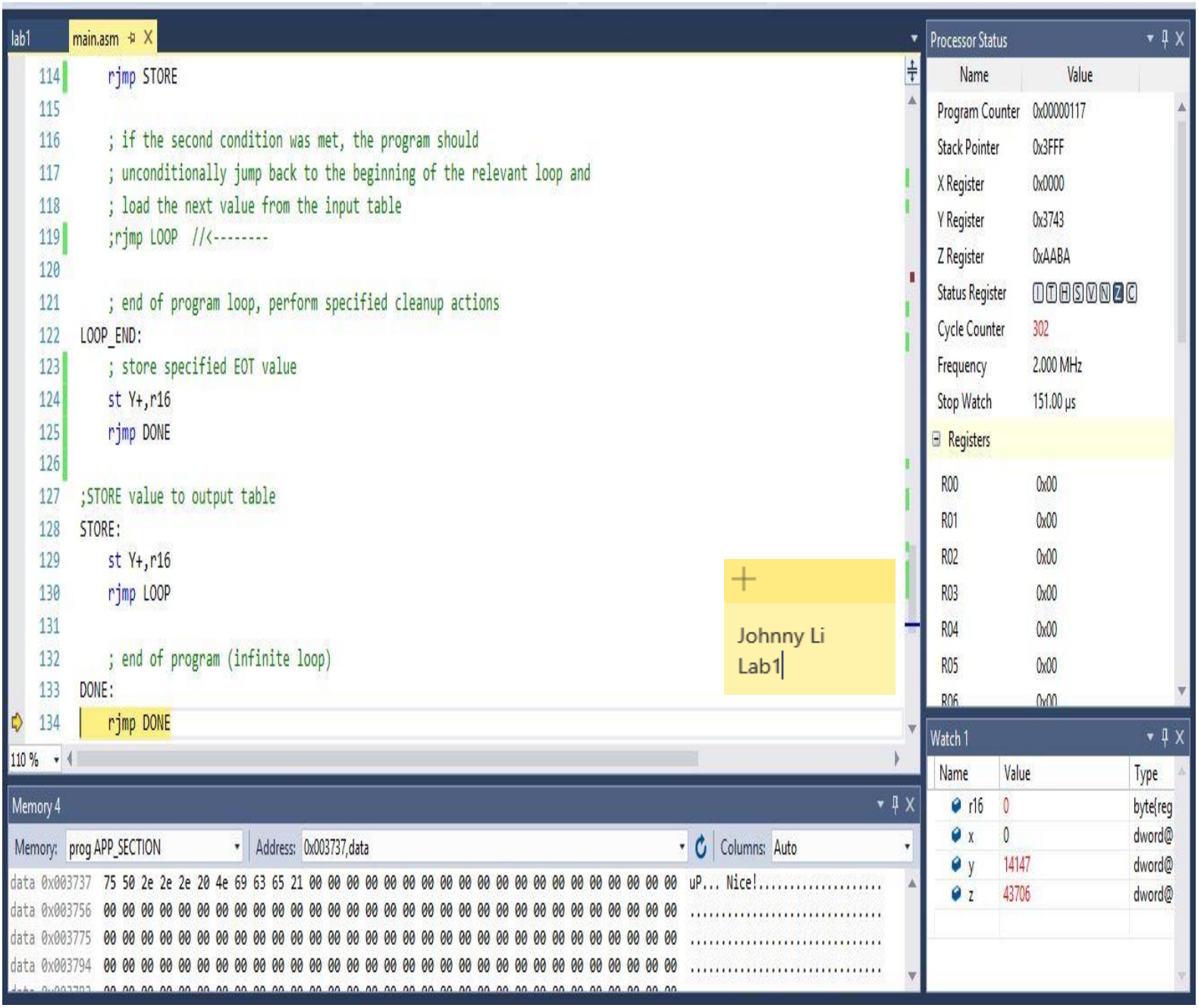
lab1 — main.asm

```
1   ;*****************************************************************************
2   ;  File name: lab1_skeleton.asm
3   ;  Author:  Christopher Crary
4   ;  Last Modified By: Dr. Schwartz
5   ;  Last Modified On: 29 August 2019
6   ;  Purpose: To filter data stored within a predefined input table based on a
7   ;           set of given conditions and store a subset of filtered values
8   ;           into an output table.
9   ;*****************************************************************************
10  ;***********************************INCLUDES**********************************
11      .include "ATxmega128a1udef.inc"
12  ;*********************************END OF INCLUDES*****************************
13  ;************************************EQUATES**********************************
14      ; potentially useful expressions
15      ; data to program address
16      .equ IN_TABLE_START_ADDR = 0xAAAA/2
17      .equ OUT_TABLE_START_ADDR = 0x3737
18      .equ NULL = 0
19  ;*********************************END OF EQUATES******************************
20  ;*****************************MEMORY CONFIGURATION*****************************
21      ; program memory constants (if necessary)
22      .cseg
23
24      .org IN_TABLE_START_ADDR
25
26  IN_TABLE: .db 234, 0xA0, '@', 060, 0b00110000, 93, 0x30, 042
27            .db 'J', 52, 0b10011100, 210, 0xC6, 0xCA, '#', 0
28      ; label used to calculate size of input table
29  IN_TABLE_END:
30
31      ; data memory allocation (if necessary)
32      .dseg
33
34      .org OUT_TABLE_START_ADDR
35  OUT_TABLE: .byte (IN_TABLE_END - IN_TABLE)
36  ;*************************END OF MEMORY CONFIGURATION***************************
```

Johnny Li
Lab1

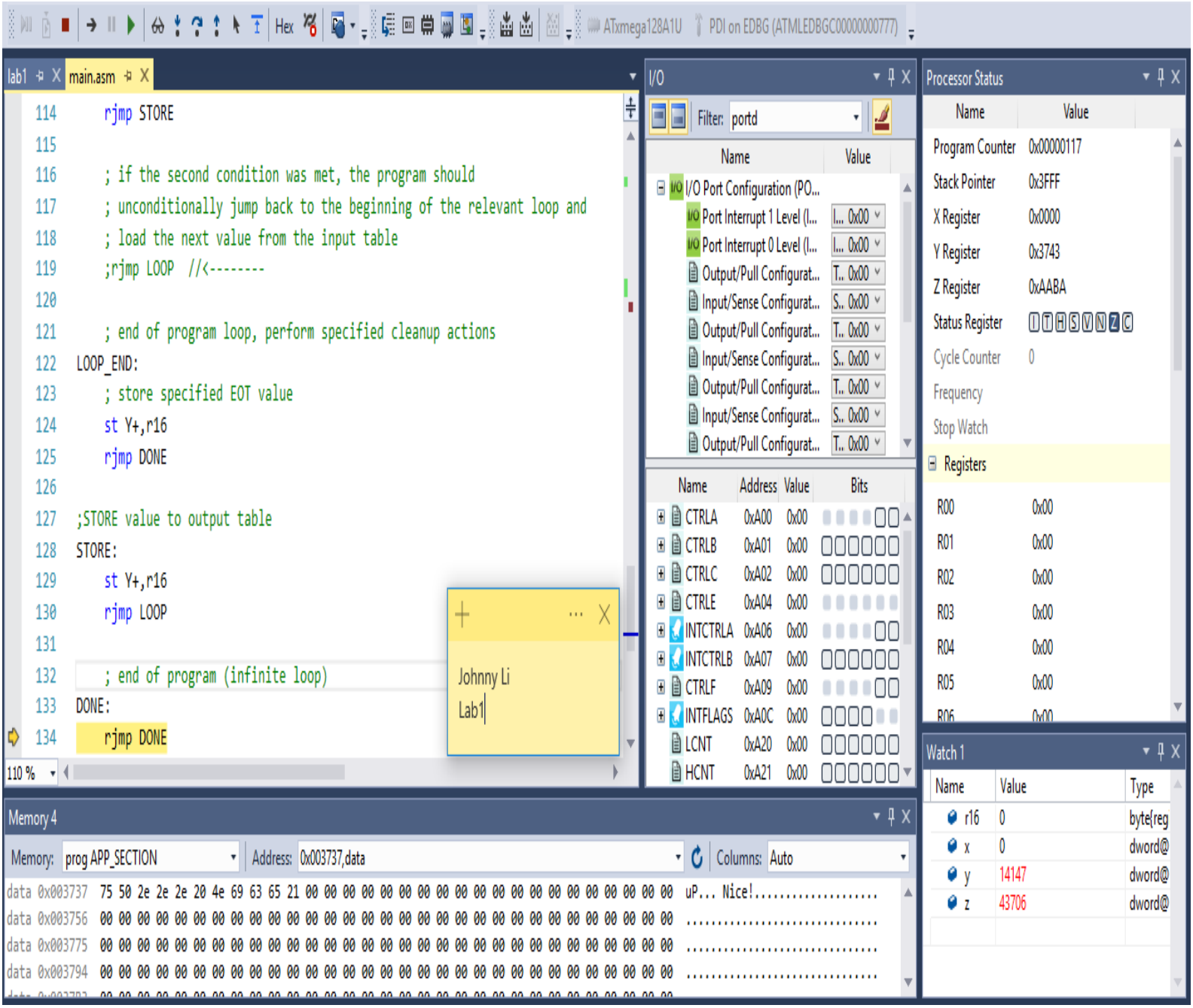
Screenshot 1: Memory Configuration

The first part of lab 1 skeleton code with the necessary changes made.




Screenshot 2: Message from simulation

The hidden message, “uP… Nice!”, displayed after running the code in simulation on Atmel Studio.




Screenshot 3: Message from emulation

The hidden message, “uP… Nice!”, displayed after running the code in emulation on the uPad.