REQUIREMENTS NOT MET

N/A

PROBLEMS ENCOUNTERED

I initially wrote each value in the input table with a separate .db directive, which caused the assembler to pad every entry with an extra 0x00 in program memory. This caused the program to stop early, as it encountered the padded zero as a false EOT. After realizing the issue, I rewrote the table using a single .db directive with values separated by commas, which removed the padding and allowed the loop to process all inputs correctly. Additionally, I was unsure how to use CPU_RAMPZ or ELPM, but I figured it out after reading from the class slides and the manual. I was also unaware of the built-in skip if bit cleared/set functions and created my own bit mask using AND and branching after comparison, but I revised these lines once I discovered these functions. Moreover, I am slightly confused by the lab format template and am not sure whether I formatted everything correctly.

FUTURE WORK/APPLICATIONS

This lab builds the foundation for upcoming work with GPIO inputs/outputs and the system clock. By learning how to structure data in memory, use pointers, and apply conditional logic, we gain the low-level control needed to toggle pins, read switches, and configure timing in future labs. Extending this program to interact with GPIO would demonstrate how filtering logic can be applied to real-time hardware inputs and outputs.

PRE-LAB EXERCISES

i. As specified in Lab 0, you should have, upon receiving your kit, verified that it contained all of the parts listed on µPAD v2.0 Parts List (Excel or PDF). If it did not, you should have immediately notified the PI if any components were missing. For documentation purposes (and before any assembly), you should have taken pictures of all of the parts in your kit (each of the PCBs, the chips, etc.). Include these images in your Lab 1 Pre-Lab Report. Your Lab 1 report should also include images of all that you were given in your lab kit.
   a. Included in the appendix.

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

   a. Word-alignment is used for program memory and byte-alignment is used for data memory in the ATxmega128A1U.

iii. 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?
   a. To load data indirectly from data memory within XMEGA AU microcontrollers you can use the LD/LDD instructions with the X, Y, or Z registers such as:
      i. LD Rd, X
      ii. LD Rd, Y+
      iii. LDD Rd, Z + q
   b. To store data indirectly to data memory, you can use the ST/STD instructions with the X, Y, or Z registers, such as:
      i. ST X, Rr
      ii. ST -Y, Rr
      iii. STD Y+q, Rr

iv. Which assembly instruction can be used to load data directly from any of the general purpose I/O memory of XMEGA AU microcontrollers? Which assembly instruction can be used to store data directly to any of the I/O memory?
   a. LDS Rd, k and IN Rd, A can be used to load data directly from any of the general purpose I/O memory of XMEGA AU microcontrollers.
   b. STS k, Rr and OUT A, Rr can be used to store data directly to any of the I/O memory.

v. 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?
   a. .DB is the assembler directive that places a byte of data in program memory (use under .CSEG).
   b. .BYTE is the assembler directive that allocates space within data memory (use under .DSEG)..

c. .EQU is the assembler directive that allows you to provide expressions (either constant or variable) with a meaningful name.

vi. Which assembly instructions can be used to read from (flash) program memory? For each instruction, list which registers can be used as an operand.
   a. LPM and ELPM can be used to read from program memory.
   b. For each instruction, you have to use the Z register pair as the pointer register (extended by RAMPZ when using ELPM for addresses larger than 64kb). You can use any general register (R0-31) as the destination, but R0 is the default. For example you could do .lpm r17, Z.

vii. Your Lab 1 report should also include images of all your OOTB PCBs.
   a. Included in the appendix.

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

   a. You would specify address 0x1F5BC within the debug window to view the address 0xFADE in program memory because you must shift the word address left one time to get the location of the corresponding bytes in program memory.

ix. When using the internal SRAM (not EEPROM), which memory locations can be utilized for the data segment (.dseg)? Why?
   a. When using the internal SRAM you can utilize the 0x2000 to 0x3FFF memory locations for the data segment because that is the memory range allocated for the SRAM, before that is reserved and after that is external memory.

x. Which is the first (i.e., lowest) program memory address (this is an address to the 16-bit wide program memory information) that would require the relevant RAMP register to be changed from its initial value of zero? Why?
   a. The first program memory address that would require the relevant RAMP register to be changed is the word address 0x8000 (just after 0x7FFF), because this corresponds to the byte address 0x10000. Since the Z register can only directly address up to 0xFFFF bytes, the RAMP register must be used once the byte address exceeds this limit.

xi. 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.
   a. In the context of pointing an index to a specific program memory address within an XMEGA AU architecture, the address value should be multiplied by two (shifted left once) since there are two bytes for every word and the program memory uses word addresses. There is no change to be

made in the context of pointing an index to a specific data memory address because the data memory addresses are byte addresses and correspond with the memory addresses.

PSEUDOCODE/FLOWCHARTS

SECTION 1

(I wrote pseudocode, but still used the provided skeleton. I altered my code based on the skeleton, but I used my pseudocode as a reference in some areas.)

```
;Define constants
        EOT = 0x00
        Input_start = 0xDADD
        Output_start = 0x3744

Org 0
        Jump to Main

Main
;load input table
        Org input_start
;Load low byte of input_start into ZL
        ZL = byte1( input_start << 1)
;Load highbyte of input start into ZH
        ZH = byte2  (input_start)
;Load data pointer in output table to Y
        YL = byte1 (output_start)
        YH = byte2 (output_start)
LOOP:
        ;Load value of Z pointer
Z = r16
Z++;
r17 = EOT
        ; check if EOT character
```

```
        ;IF EOT store a 0 in out table


        ; branch to end of program
        Compare R16 to R17
        Branch if equal to END


;check if bit 7 is clear (CONDITION_1)
;if bit 7 is clear multiply the original 8-bit value by 2 (shift left)
        Compare bit 7 of r16 to 0
        If equal (bit 7 clear), continue with multiply path
        Else branch to Set
        r16 shift left


;if the updated value is less than or equal to 110 add 12 to it and store it
        R20 = 110
        Compare r16, r20
        Branch if greater than to Set
        R16 = R16 + 12
        Store R16 in OUT
        Y++


;If bit 7 was set, divide by 2 and check if the result is greater than 70; if it is, then add 7
Set
        R16 shift right once
        R21 = 71
        Compare r16 to r21
        Branch if less to After
        R16 = r16 + 7
        Store in OUT
        Y++
```

After

Branch to LOOP

End

Branch end

//

PROGRAM CODE

SECTION 2

```
;
; Lab1.asm
;
; Created: 9/9/2025 4:18:13 PM
; Author : arist
;

;Lab 1, Section 22
;Name: Arion Stern
;Class #: 11303
;PI Name: Ian Santamauro
;Description: This program filters data stored in a predefined input table
;                     according to specified conditions and stores the resulting
;                     subset of values into an output table.


; Replace with your application code
;***********************************************************************
;  File name: lab1.asm
;  Author:  Christopher Crary
;  Last Modified By: Dr. Schwartz
;  Last Modified On: 27 Aug 2025
;  Description: To filter data stored within a predefined input table
;                           based on a set of given conditions and store
;                           a subset of filtered values into an output table.
;***********************************************************************
;*********************************INCLUDES*******************************
.include "ATxmega128a1udef.inc"
;***********END OF INCLUDES*****************************
;*********************************EQUATES********************************
; potentially useful expressions
.equ NULL = 0
.equ ThirtySeven = 3*7 + 37/3 - (3-7)  ; 21 + 12 + 4
.equ Inputs = 0xDADD
.equ outputs = 0x3744

.def Inpt_r16 = r16
.def EOT_r17 = r17
;.def bit7_r18 = r18
;.def temp_r19 = r19


;***********END OF EQUATES*****************************
;***********MEMORY CONFIGURATION************************
; program memory constants (if necessary)
.cseg
.org Inputs
IN_TABLE:
.db 33, 0xB4, 0b00110100, '8', 218, 0b11100100, 0xB8, 0b01000111, 'n', 118, '.', 0xD0, 0xCC, 062,
0b01111101, NULL
; label below is used to calculate size of input table
IN_TABLE_END:

; data memory allocation (if necessary)
.dseg
```

```
; initialize the output table starting address
.org outputs

OUT_TABLE:
.byte (IN_TABLE_END - IN_TABLE)
;***********END OF MEMORY CONFIGURATION**************
;***********MAIN PROGRAM*******************************
.cseg
; configure the reset vector
;      (ignore meaning of "reset vector" for now)
.org 0x0
       rjmp MAIN

; place main program after interrupt vectors
;      (ignore meaning of "interrupt vectors" for now)
.org 0x100
MAIN:
; point appropriate indices to input/output tables (is RAMP needed?)
ldi ZL, low(IN_TABLE<<1)
ldi ZH, high(IN_TABLE<<1)
ldi r16, byte3(IN_TABLE<<1)
out CPU_RAMPZ, r16

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 inpt_r16, Z+
       ldi EOT_R17, NULL

       ; determine if the end of table has been reached (perform general check)
       cp inpt_r16, eot_r17


       ; 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 (e.g., DONE)
       BREQ DONE

       ; if EOT was not encountered, perform the first specified
       ; overall conditional check on loaded value (CONDITION_1)
CHECK_1:
       ; check if the CONDITION_1 is met (bit 7 of # is cleared);
       ;   if not, branch to FAILED_CHECK1

       sbrc Inpt_r16, 7
       rjmp FAILED_CHECK1


       ; since the CONDITION_1 is met, perform the specified operation
       ;   (multiply # by 2, unsigned)
       LSL inpt_r16


       ; check if CONDITION_1a is met (result < 111); if so, then
       ;   jump to LESS_THAN_111; else store nothing and go back to LOOP
       ldi  r20, 111
       cp inpt_r16, r20
       BRLO LESS_THAN_111
```

```
	rjmp LOOP

LESS_THAN_111:
	; add 12 and store the result
	ldi r22, 12
	add inpt_r16, r22
	st Y+, inpt_r16

	rjmp LOOP


FAILED_CHECK1:
	; since the CONDITION_1 is NOT met (bit 7 of # is set),
	;    perform the second specified operation
	;    (divide by 2 [unsigned])
	LSR inpt_r16




	; check if CONDITION_2b is met (result >= 71); if so, jump to
	;    GREATER_EQUAL_71 (and do the next specified operation);
	;    else store nothing and go back to LOOP
	ldi r21, 71
	cp inpt_r16, r21
	BRSH GREATER_EQUAL_71

	rjmp LOOP

GREATER_EQUAL_71:
	; add 7 and store the result
	ldi r23, 7
	ADD inpt_r16, r23
	st Y+, inpt_r16


	;go back to LOOP
	rjmp LOOP

; end of program (infinite loop)
DONE:
	ldi r24, 0
	st Y, r24
	rjmp DONE
;***********END OF MAIN PROGRAM *********************
```

APPENDIX

Supporting ASM/C Code and Additional Information

- Included File:
  - lab1.asm (see Program Code section above)
- Referenced Header:
  - ATxmega128a1udef.inc (standard device definition file provided by Microchip)
- Additional Screenshots/Iimages:

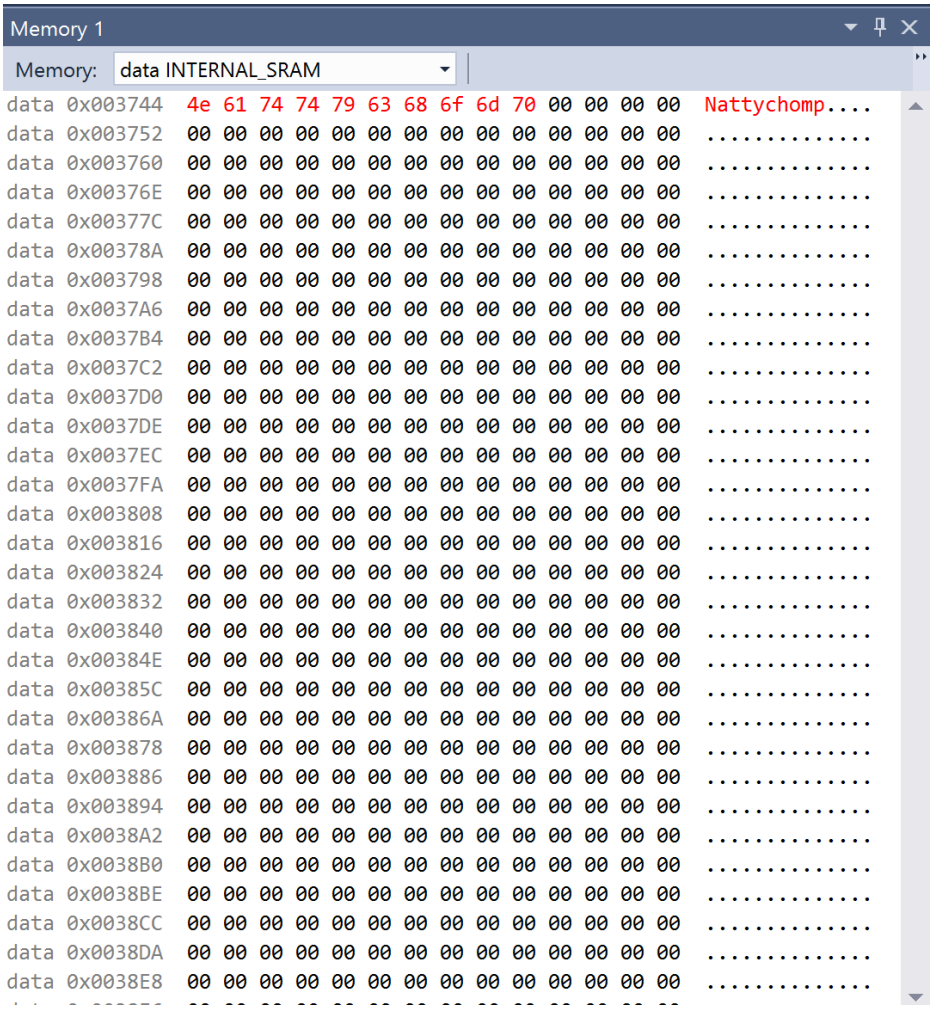

Figure 1: Screenshot of a Memory view window after executing the relevant program, showing Nattychomp string in ASCII at 0x3744.

Figure 2: Picture of all parts from the received lab kit and additional parts from OOTB