CECS-412-01
Summer 2014
Project #2

Name	Report (20 Points)	Demo (15 Points)	Quiz (5 Points)
Andrew Nguyen Vo			

Introduction to Modular Programming, Data Tables, and C Program Analysis

Andrew Nguyen Vo
CECS-412-01

ABSTRACT

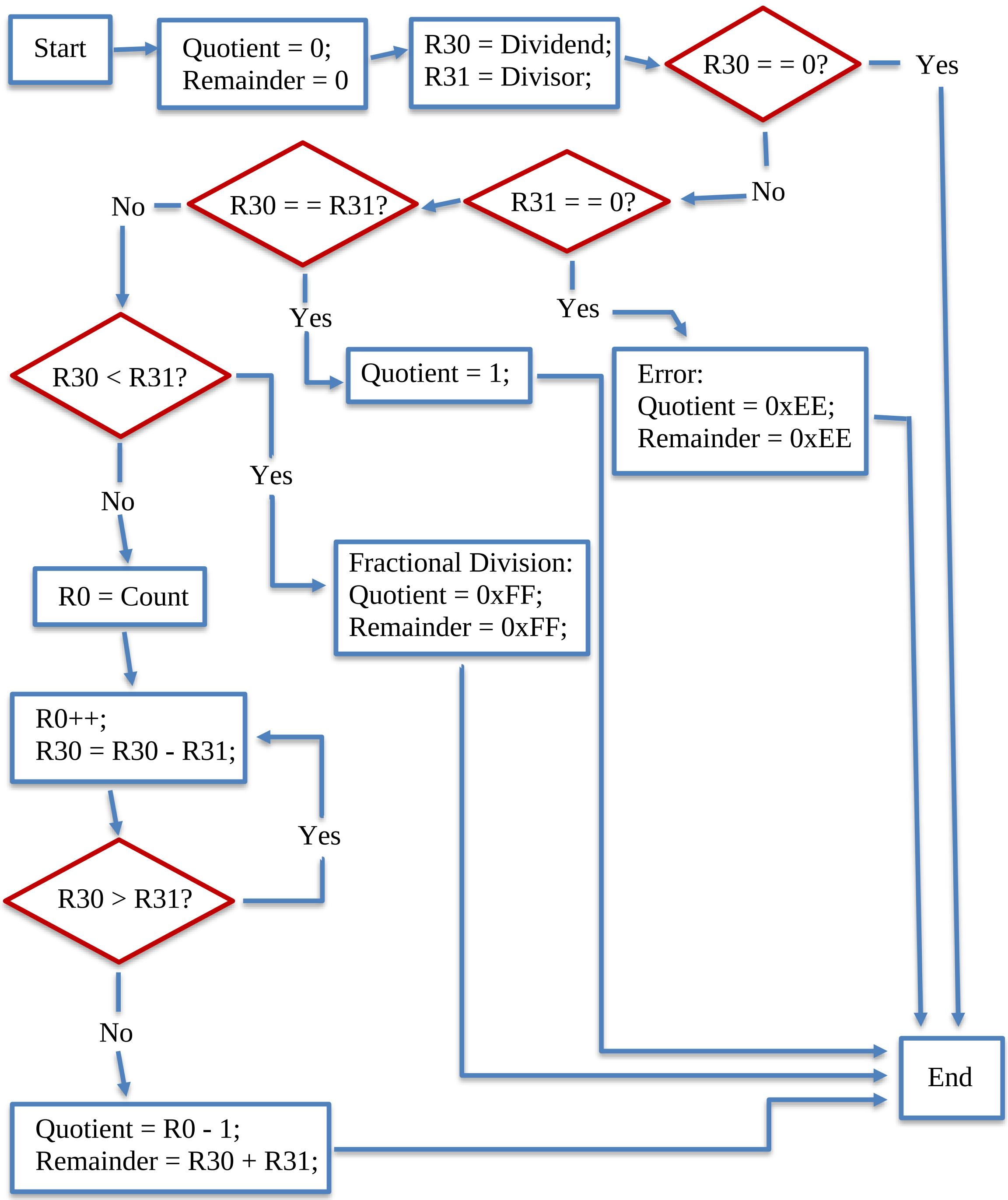
The simulator for the ATmega328P in Atmel Studio IDE 7.0 was used to perform all tests during this investigation. The purpose of this lab was to explore three distinct concepts related to Assembly language programming for embedded systems. The first topic, modular programming, was explored using calls to subroutines and analyzing the effects on the memory, specifically the stack pointer. The next topic, data tables, was studied using a simple Celsius to Fahrenheit lookup table to store a range of 20 corresponding values. The method of indexing in the memory and loading the results in SRAM were analyzed. The last topic of study, C programming, was investigated using a simple C program consisting of a single-line main function dividing two stored preset global variables and storing the result in a third variable. The contents of the .lss output file was analyzed to compare the Assembly instructions that produce the result of the C program. The investigations during this lab were carried out successfully as conclusions could clearly be drawn from the data collected.

Body

All of the following lab investigations were performed using the Windows version of Atmel Studio IDE 7.0. The first investigation involved the use of modular subroutines that carried out a simple division calculation in assembly. The actual values of the dividend and divisor were stored in r30 and r31 respectively. The variables for dividend and divisor were preset using the .equ directive, which performed preprocessor replacements in the program, swapping out each instance of the variable names “dividend” and “divisor” with the values specified at the top of the code. Similarly, the count variable was stored in r0, however it was predefined with the .set directive earlier in the code. In this case, the .set directive served the same purpose as the .equ, although these two directives have an important distinction. The variables set with .equ are static and cannot be changed, whereas any variable with .set can be overwritten by a later .set. The quotient and remainder are stored in SRAM 0x100 and 0x101 respectively. At the top of the code, the .byte directive reserved 1 byte of uninitialized space for each of these values in SRAM. The following is a description of the processes performed in the program.

The program first initializes the memory locations, preprocessor variables, and byte spaces in SRAM for the dividend, divisor, count, quotient, and remainder. The first subroutine called by the main: is init:, which sets the values in r0 and the addresses in SRAM for the quotient and remainder to 0. The program returns from the subroutine and calls getnums:, which sets r30 and r31 to the dividend and divisor respectively, which were declared as preprocessor variables earlier in the code. The program returns from this subroutine, and then calls test:. The next test subroutines are used to ensure that legal numbers are used to perform this non-fractional division operation. The first test checks to see if the dividend is equal to 0 and if so, the program jumps to loop infinitely at test1:, which effectively halts the program and leaves the quotient and remainder to be 0. If the dividend is not 0, the negative flag is set, the test is passed, and the program branches to test2:. test2: checks if the divisor is equal to 0, which would result in an illegal division by 0. If this is the case, the quotient and remainder are set to 0xEE to indicate an error, and the program infinitely loops at test3:. Else, the program branches to test4: which checks if the dividend and divisor are equal. If so, the quotient is set to 1 and the remainder is left to be 0 – the program is infinitely looped at test5: to halt the program. If this is not the case, the program branches to test6:. Using the same comparison as earlier, the program tests to see if the dividend is greater than the divisor. If it isn’t, then both the quotient and remainder are set to 0xFF, to indicate the resulting quotient is fractional and cannot be solved with this program – test7: is used to halt the program infinitely. Otherwise, if the dividend is greater than the divisor, the program branches to test8:, which effectively returns back to the main. The next subroutine to be called is divide:, which performs that actual operation of division intended by the program. A counter is initialized in r0 to keep track of the quotient as the subroutine is looped through. Every time it is looped, the counter at r0 is incremented by 1. Each loop performs a subtraction of the divisor from the dividend and continues looping until this subtraction results in a negative number.

The program then backtracks a single step, re-adding the divisor to the dividend register to obtain the remainder, and the quotient register is decremented by 1 to obtain the actual quotient. The values for the quotient and remainder are then stored in the addresses located at 0x100 and 0x101, respectively, and the program returns back to the main, which continues to infinitely loop and end the program at endmain:. The following is a flowchart demonstrating this process.



The following is pseudocode for the division program.

```
Quotient;
Remainder;
Dividend = 13;
Divisor = 3;
Count = 0;

main() {
        init();
        getnums();
        test();
        divide();
        END PROGRAM;
}

init() {
        Quotient = 0;
        Remainder = 0;
        return;
}

getnums() {
        r30 = Dividend;
        r31 = Divisor;
        return;
}

test() {
        if (r30 == 0) {
                END PROGRAM;
        } else if (r31 == 0) {
                Quotient = 0xEE;
                Remainder = 0xEE;
                END PROGRAM;
        } else if (r30 == r31) {
                Quotient = 1;
                END PROGRAM;
        } else if (r30 < r31) {
                Quotient = 0xFF;
                Remainder = 0xFF;
                END PROGRAM;
        } else {
                return;
        }
```

```
}

divide() {
	r0 = Count;
	do {
		r0++;
		r30 = r30 – r31;
	} while (r30 > r31);
	r0--;
	r30 = r30 + r31;
	Quotient = r0;
	Remainder = r30;
	return;
}
```

During this program, the STACK is observed to be changing as the Program Counter steps through the code during the CALLs and RETs to and from the subroutines in the program. Each time a CALL instruction is executed, the STACK in the SRAM is changed to record the Program Counter address where the subroutine was called. When the RET instruction is executed, the Program Counter is set to the most recently set value in the STACK in order to return from the function back to where it was originally called. When the program was rewritten using nested subroutines, an additional Program Counter address is stored in the STACK, which grows outward by one byte every time the CALL instruction is executed within a nested subroutine. When the RET instruction is executed, the Program Counter is set to the outermost STACK address, and the Stack Pointer shifts one byte inwards. The Program Counter is set to the address pointed to by the Stack Pointer for each subsequent RET instruction.

In the investigation of the relationship between the C program and the resulting assembly code, the INT and CHAR data types are handled differently in the resulting Assembly code. The INT global variables are simply loaded directly into the registers, while the CHAR global variables requires additional operations to be done on the r0 register to set a carry bit before performing a subtraction with carry instruction on r25 and r23. This is due to the CHAR variable being of lesser size than an INT, and thus the result set in r25 and r23 will be a subtraction. Because the CHAR is inherently 2 bytes and an INT is 1 byte, the signed forms of these data types can span a range in one direction of lesser magnitude than their unsigned counterparts, which can only be a positive value but can hold values of twice the magnitude than the signed version.

Source Code (Software)

```
;
; NestedSubroutines_L2.asm
;
; Created: 6/13/2018 12:45:57 AM
; Author : Andrew Nguyen Vo
;
;******************************
;* Declare Variables
;******************************
           .dseg           ;directive to specify data segment in
SRAM
           .org  0x100     ;originate data storage at address
                           ;0x100
quotient:  .byte 1         ;uninitialized quotient variable
                           ;stored in SRAM
                           ;aka data segment
remainder: .byte 1         ;uninitialized remainder variable
                           ;stored in SRAM
           .set  count = 0 ;initialized count variable stored in
                           ;SRAM
;******************************
           .cseg           ;Declare and Initialize Constants
                           ;(modify them for different results)
           .equ  dividend = 13    ;8-bit dividend constant
                           ;(positive integer)
                           ;stored in FLASH memory aka code
                           ;segment
           .equ  divisor = 3      ;8-bit divisor constant
                           ;(positive integer)
                           ;stored in FLASH memory
;******************************
;* Vector Table (partial)
;******************************
           .org  0x0       ;sets origin of code segment in FLASH
                           ;memory at
                           ;0x0
reset:          jmp        main  ;RESET Vector at address 0x0 in
                           ;FLASH memory
                           ;(handled by MAIN)
int0v:          jmp        int0h ;External interrupt vector at
                           ;address 0x2 in
                           ;Flash memory (handled by int0)
;******************************
```

```
;* MAIN entry point to program*
;*******************************
            .org  0x100      ;originate MAIN at address 0x100 in
                             ;FLASH memory
                             ;(step through the code)
main:       call  init       ;initialize variables subroutine, set
                             ;break
                             ;point here, check the STACK,SP,PC
                             ;STACK = ?? ??, SP 0x08FF, PC =
                             ;0x00000100
            call  getnums    ;Check the STACK,SP,PC here
                             ;STACK = ?? ??, SP = 0x08FF, PC =
                             ;0x00000102
            call  test       ;Check the STACK,SP,PC here
                             ;STACK = ?? ??, SP = 0x08FF, PC =
                             ;0x00000104
            call  divide     ;Check the STACK,SP,PC here
                             ;STACK = ?? ??, SP = 0x08FF, PC =
                             ;0x00000106
endmain:    jmp   endmain
init:       lds   r0,count   ;get initial count, set break point
                             ;here and
                             ;check the STACK,SP,PC
            sts   quotient,r0     ;use the same r0 value to clear
                             ;the quotient-
            sts   remainder,r0    ;and the remainder storage
                             ;locations
            ret              ;return from subroutine, check the
                             ;STACK,SP,PC
                             ;here.
                             ;STACK = 01 02, SP = 0x08FD, PC =
                             ;0x00000110
getnums:    ldi   r30,dividend    ;Check the STACK,SP,PC here.
                             ;STACK = 01 04, SP = 0x08FD, PC =
                             ;0x00000111
            ldi   r31,divisor     ;loads r31 with the value
                             ;specified by the
                             ;symbol divisor
            ret              ;Check the STACK,SP,PC here.
                             ;STACK = 01 04, SP = 0x08FD, PC =
                             ;0x00000113
test:       cpi   r30,0      ;is dividend == 0 ?
            brne  test2      ;if the value in r30 is not 0, branch
                             ;to test2:
test1:      jmp   test1      ;halt program, output = 0 quotient and
                             ;remainder = 0
```

```
test2:      cpi   r31,0       ;is divisor == 0 ?
            brne  test4       ;if the value in r31 is not 0, branch
                              ;to test4:
            ldi   r30,0xEE    ;set output to all EE's = Error
                              ;division by 0
            sts   quotient,r30     ;stores the value in r30 into
                              ;the variable
                              ;specified by quotient
            sts   remainder,r30    ;stores the value of r30 into
                              ;the variable
                              ;specified by remainder
test3:      jmp   test3       ;halt program, look at output
test4:      cp    r30,r31     ;is dividend == divisor ?
            brne  test6       ;if the values in r30 and r31 are not
                              ;the same,
                              ;branch to test6:
            ldi   r30,1       ;then set output accordingly
            sts   quotient,r30     ;stores the value of r30 in the
                              ;variable
                              ;specified by quotient
test5:      jmp   test5       ;halt program, look at output
test6:      brpl  test8       ;is dividend < divisor ?
            ser   r30         ;sets all the bits in r30
            sts   quotient,r30     ;stores the value of r30 in the
                              ;variable
                              ;specified by quotient
            sts   remainder,r30    ;set output to all FF's = not
                              ;solving Fractions
                              ;in this program
test7:      jmp   test7       ;halt program look at output
test8:      ret               ;otherwise, return to do positive
                              ;integer
                              ;division
divide:     lds   r0,count    ;loads r0 with the initial value 0
                              ;specified by
                              ;the count symbol
divide1:    inc   r0          ;increments the value of 30 by 1 for
                              ;every time
                              ;this subroutine is entered/looped.
                              ;This is
                              ;equal to the quotient
            sub   r30,r31     ;subtracts the value in r30 by the
                              ;value in r31
                              ;and stores in r30. This is the
                              ;standard process
                              ;of division.
```

```
            brpl  divide1    ;if negative flag is not set by the
                             ;subtraction,
                             ;loop back to divide1: in order to get
                             ;the
                             ;maximum number of times the value in
                             ;r31 goes into the value in r30
            dec   r0         ;else decrement r0 by 1 because this
                             ;implies
                             ;that the quotient is one less than
                             the current ;number stored in r0
            add   r30,r31    ;add the value in r30 with the value
                             ;in r31.
                             ;This should be less than the value in
                             ;the
                             ;divisor stored in r31, which is
                             ;otherwise known as the remainder
            sts   quotient,r0     ;stores the value of r0 in the
                             ;variable
                             ;specified by quotient
            sts   remainder,r30   ;stores the value of r30 in the
                             ;variable
                             ;specified by remainder
divide2:    ret              ;return from this subroutine back to
                             ;the main
int0h:           jmp   int0h      ;interrupt 0 handler goes here
            .exit


;
; NestedSubroutines_L2.asm
;
; Created: 6/13/2018 12:45:57 AM
; Author : Andrew Nguyen Vo
;
;******************************
;* Declare Variables
;******************************
            .dseg                 ;directive to specify data
                                  ;segment in SRAM
            .org  0x100           ;originate data storage at
                                  ;address 0x100
quotient:   .byte 1               ;uninitialized quotient variable
                                  ;stored in SRAM aka data segment
remainder:  .byte 1               ;uninitialized remainder
                                  ;variable stored in SRAM
```

```
          .set  count = 0         ;initialized count variable
                                  ;stored in SRAM
;******************************
          .cseg                   ;Declare and Initialize
                                  ;Constants (modify them for
                                  ;different results)
          .equ  dividend = 13     ;8-bit dividend constant
                                  ;(positive integer) stored in
                                  ;FLASH memory aka code segment
          .equ  divisor = 3       ;8-bit divisor constant
                                  ;(positive integer) stored in
                                  ;FLASH memory
;******************************
;* Vector Table (partial)
;******************************
          .org  0x0               ;sets origin of code segment in
                                  ;FLASH memory at 0x0
reset:    jmp   main              ;RESET Vector at address 0x0 in
                                  ;FLASH memory (handled by MAIN)
int0v:    jmp   int0h             ;External interrupt vector at
                                  ;address 0x2 in Flash memory
                                  ;(handled by int0)
;******************************
;* MAIN entry point to program*
;******************************
          .org  0x100             ;originate MAIN at address 0x100
                                  ;in FLASH memory (step through
                                  ;the code)
main:     call  init              ;initialize variables
                                  ;subroutine, set break point
                                  ;here, check the STACK,SP,PC
endmain:  jmp   endmain
init:     lds   r0,count          ;get initial count, set break
                                  ;point here and check the
                                  ;STACK,SP,PC
          sts   quotient,r0       ;use the same r0 value to clear
                                  ;the quotient-
          sts   remainder,r0      ;and the remainder storage
                                  ;locations
          call  getnums           ;calls the subroutine getnums:
          ret                     ;return from subroutine, check
                                  ;the STACK,SP,PC here.
getnums:  ldi   r30,dividend      ;Check the STACK,SP,PC here.
          ldi   r31,divisor       ;loads r31 with the value
                                  ;specified by the symbol divisor
          call  test              ;calls the subroutine test
```

```
            ret                     ;Check the STACK,SP,PC here.
test:       cpi   r30,0             ; is dividend == 0 ?
            brne  test2             ;if the value in r30 is not 0,
                                    ;branch to test2:
test1:      jmp   test1             ; halt program, output = 0
                                    ;quotient and 0 remainder
test2:      cpir  31,0              ; is divisor == 0 ?
            brne  test4             ;if the value in r31 is not 0,
                                    ;branch to test4:
            ldi   r30,0xEE          ; set output to all EE's = Error
                                    ;division by 0
            sts   quotient,r30      ;stores the value in r30 into
                                    ;the variable specified by
                                    ;quotient
            sts   remainder,r30     ;stores the value of r30 into
                                    ;the variable specified by
                                    ;remainder
test3:      jmp   test3             ; halt program, look at output
test4:      cp    r30,r31           ; is dividend == divisor ?
            brne  test6             ;if the values in r30 and r31
                                    ;are not the same, branch to
                                    ;test6:
            ldi   r30,1             ;then set output accordingly
            sts   quotient,r30      ;stores the value of r30 in the
                                    ;variable specified by quotient
test5:      jmp   test5             ; halt program, look at output
test6:      brpl  test8             ; is dividend < divisor ?
            ser   r30               ; sets all the bits in r30
            sts   quotient,r30      ; stores the value of r30 in the
                                    ;variable specified by quotient
            sts   remainder,r30     ; set output to all FF's = not
                                    ;solving Fractions in this
                                    ;program
test7:      jmp   test7             ; halt program look at output
test8:      call  divide            ;calls the subroutine divide:
            ret                     ; otherwise, return to do
                                    ;positive integer division
divide:     lds   r0,count          ;loads r0 with the value
                                    ;specified by the symbol count
divide1:    inc   r0                ;increments the value of the
                                    ;quotient by 1 for each loop
            sub   r30,r31           ;subtracts the value in r30 by
                                    ;the value in r31 and stores in
                                    ;r30
            brpl  divide1           ;if r30 still greater than r31
                                    ;then loop back to divide1:
```

```
            dec   r0                  ;else decrement quotient by 1
            add  r30,r31              ;add the value in r30 with the
                                      ;value in r31 to get the
                                      ;remainder
            sts   quotient,r0         ;stores the value of r0 in the
                                      ;variable specified by quotient
            sts   remainder,r30       ;stores the value of r30 in the
                                      ;variable specified by remainder
divide2:    ret                       ;return from this nested
                                      ;subroutine
int0h:      jmp   int0h               ; interrupt 0 handler goes here
            .exit


/*
* lab2p2.asm
* Celsius to Fahrenheit Look-Up Table
*  Created: 6/2/2014 10:17:31 AM  *   Author: Eugene Rockey  */
            .dseg
            .org       0x100
output:     .byte      1               ;reserves one byte of space
                                       ;for the output Fahrenheit
                                       ;value
            .cseg
            .org       0x0
            jmp        main            ;partial vector table at
                                       ;address 0x0
            .org       0x100           ;MAIN entry point at
                                       ;address 0x200 (step
                                       ;through the code)
main:       ldi        ZL,low(2*table) ;sets the low byte of Z-
                                       ;register to the low byte
                                       ;address of the data table
            ldi        ZH,high(2*table);sets the high byte of the
                                       ;Z-register to the high
                                       ;byte address of the data
                                       ;table
            ldi        r16,celsius     ;stores the input celsius
                                       ;value into r16
            add        ZL,r16          ;adds the value of r16 to
                                       ;Z, which moves the index
                                       ;of the data table that
                                       ;many values
            ldi        r16,0           ;clears r16
            adc        ZH,r16          ;sets the high byte of the
                                       ;Z-register to 0
```

```
          lpm                          ;lpm = lpm r0,Z in reality,
                                       ;what does this mean?
                                       ;loads into the default
                                       ;register r0, the value
                                       ;stored at the address now
                                       ;pointed to by
                                       ;the Z-register, which
                                       ;corresponds to an index in
                                       the lookup
table
          sts         output,r0        ;store look-up result to
                                       ;SRAM
          ret                          ;consider MAIN as a
                                       ;subroutine to return from
                                       ;- but back to where??
                                       ;returns the program
                                       ;counter to the beginning
                                       ;of the code segment at
                                       ;0x00
                                       ;Fahrenheit look-up table
table:    .db         32, 34, 36, 37, 39, 41, 43, 45, 46, 48, 50,
                      52, 54, 55, 57, 59, 61, 63, 64, 66
          .equ        celsius = 5      ;modify Celsius from
                                       ;0 to 19 degrees for
                                       ;different results
          .exit



;
; SortTable_L2.asm
;
; Created: 6/13/2018 12:48:03 AM
; Author : Andrew Nguyen Vo
;
          .dseg                   ; directive for specifying data
                                  ;segment SRAM
          .org  0x100             ; sets origin of SRAM to be at
                                  ;0x100
          .cseg                   ; directive for specifying code
                                  ;segment in FLASH memory
          .org  0x0               ; sets origin of FLASH memory to
                                  ;be at 0x0
          jmp   main              ; MAIN entry point at address
                                  ;0x200
main:     ldi   XL,0x00           ; Sets low byte of X-register to
```

```
                                  ;0x00
            ldi   XH,0x01         ; Sets high byte of X-register
                                  ;to 0x01
            ldi   ZL,low(2*table) ; Sets the low byte of the Z-
                                  ;register to the low byte of the
                                  ;byte address of the lookup
                                  ;table
            ldi   ZH,high(2*table); Sets the high byte of the Z-
                                  ;register to the high byte of
                                  ;the byte address of the lookup
                                  ;table
            ldi   r20,0           ; Sets the value of r20 to 0
                                  ; Store: used to copy all the
                                  ;lookup table values to SRAM
store:      inc   r20             ; increments r20 by 1
            lpm   r16,Z+          ; loads into r16 the value
                                  ;pointed to by the Z-register
                                  ;and post-increments Z by 1
            st    X+,r16          ; loads into address pointed to
                                  ;by X-register the value of r16
                                  ;and post-increments the value
                                  ;of X
                                  ;by 1
            cpi   r20,20          ; compares the value in r20 with
                                  ;the number 20
            brlo  store           ; if the value in r20 is lower
                                  ;than 20, loop back to store:
                                  ; repeat loop 20 times until all
                                  ;20 numbers from lookup table
                                  ;are copied over to SRAM
            ldi   r20,0           ; clears any value in r20
            ldi   XL,0x00         ; resets low byte of X-register
                                  ;to 0x00
            ldi   XH,0X01         ; resets high byte of X-register
                                  ;to 0x01
            movw  ZL,XL           ; copies the byte pair of the X-
                                  ;register to the byte pair of
                                  ;the Z-register
            jmp   sort            ; unconditionally jumps to sort:
                                  ; sort: acts as the outermost
                                  ;for-loop, iterating through all
                                  ;the values from the table
sort:       ld    r16,X+          ; loads into r16 the value
                                  ;pointed to by the X-register
                                  ;and post-increments X by 1
            mov   r21,r20         ; copies the value in r20 to r21
```

```
            movw  ZL,XL             ; copies the byte pair of the X-
                                    ;register to the Z-register
            inc   r20               ; increments r20 by 1
            cpi   r20,20            ; compares the value in r20 to
                                    ;the number 20
            brge  done              ; if the value in r20 is greater
                                    ;than or equal to 20, then
                                    ;branch to done:
            jmp   compare           ; else, unconditionally jump to
                                    ;compare:
                                    ; compare: acts as a nested for-
                                    ;loop to compare each value with
                                    ;every value to its right using
                                    ;min-max insertion sort
compare:    cpi   r21,19            ; compare the value in r21 with
                                    ;19
            brge  sort              ; if the value in r21 is greater
                                    ;than or equal to 19, branch to
                                    ;sort:
            inc   r21               ; increment the value of r21 by
                                    ;1
            ld    r17,Z+            ; load into r17 the value
                                    ;pointed to by the Z-register
            cp    r17,r16           ; compare the values of r17 and
                                    ;r16
            brlo  switch            ; if r17 is lower than r16, then
                                    ;branch to switch:
            jmp   compare           ; else, loop back to compare:
                                    ; switch: is used to swap the
                                    ;values at different addresses
                                    ;if the right one is lower than
                                    ;the
                                    ;left one
switch:     st    -X,r17            ; pre-decrements X to point to
                                    ;the address right before and
                                    ;stores in it the value of r17
            st    -Z,r16            ; pre-decrements Z to point to
                                    ;the address right before and
                                    ;stores in it the value of r16
            ld    r16,X+            ; loads into r16 the new value
                                    ;pointed to by the X-register
                                    ;and post-increments X by 1
            ld    r17,Z+            ; loads into r17 the new value
                                    ;pointed to by the Z-register
                                    ;and post-increments Z by 1
            jmp   compare           ; loops back to compare:
```

```
table:     .db
                52,12,85,45,9,62,3,4,88,1,20,4,13,02,51,02,79,6,5
                ,19

done:      jmp   done
           .exit
```

```
/*
* L2P3_C.c
* Example relationship between C and Assembly
* Created: 1/26/2018 4:56:28 PM * Author : Eugene Rockey */

//Compile and examine the .lss file beginning with main. Comment
all the lines the compiler left empty.
int Global_A;
int Global_B = 1;
int Global_C = 2;
void main(void)
{
Global_A = Global_C / Global_B;
}

/*
 a0:  80 91 00 01      lds   r24, 0x0100     ; 0x800100 <Global_C>
 a4:  90 91 01 01      lds   r25, 0x0101     ; 0x800101
<Global_C+0x1>
 a8:  60 91 02 01      lds   r22, 0x0102     ; 0x800102 <Global_B>
 ac:  70 91 03 01      lds   r23, 0x0103     ; 0x800103
<Global_B+0x1>
 b0:  05 d0            rcall .+10       ; 0xbc <__divmodhi4>
 b2:  70 93 05 01      sts   0x0105, r23     ; 0x800105
<__data_end+0x1>
 b6:  60 93 04 01      sts   0x0104, r22     ; 0x800104
<__data_end>
 ba:  08 95            ret
*/

unsigned int Global_A;
unsigned int Global_B = 1;
unsigned int Global_C = 2;
void main(void)
{
     Global_A = Global_C / Global_B;
```

```
}

/*

  a0: 80 91 00 01      lds   r24, 0x0100      ; 0x800100 <Global_C>
  a4: 90 91 01 01      lds   r25, 0x0101      ; 0x800101
<Global_C+0x1>
  a8: 60 91 02 01      lds   r22, 0x0102      ; 0x800102 <Global_B>
  ac: 70 91 03 01      lds   r23, 0x0103      ; 0x800103
<Global_B+0x1>
  b0: 05 d0            rcall .+10        ; 0xbc <__udivmodhi4>
  b2: 70 93 05 01      sts   0x0105, r23      ; 0x800105
<__data_end+0x1>
  b6: 60 93 04 01      sts   0x0104, r22      ; 0x800104
<__data_end>
  ba: 08 95            ret
*/

//Compile and examine the .lss file beginning with main, comment
all the lines the compiler left empty.
char Global_A;
char Global_B = 1;
char Global_C = 2;
void main(void)
{
     Global_A = Global_C / Global_B;
}
*/
/*
 a0:  80 91 00 01      lds   r24, 0x0100      ; 0x800100 <Global_C>
 a4:  08 2e            mov   r0, r24          ; the value of
                                              ;Global_C is copied
                                              ;to r0
 a6:  00 0c            add   r0, r0           ; r0 is doubled to
                                              ;test for a carry bit
 a8:  99 0b            sbc   r25, r25         ; r25 is subtracted
                                              ;from itself minus
                                              ;the carry bit if
                                              ;applicable to set
                                              ;the low byte
 aa:  60 91 01 01      lds   r22, 0x0101      ; 0x800101 <Global_B>
 ae:  06 2e            mov   r0, r22          ; the value of
                                              ;Global _B is copied
                                              ;to r0
 b0:  00 0c            add   r0, r0           ; ro is doubled to
                                              ;test for a carry bit
```

```
 b2:  77 0b           sbc   r23, r23        ;r23 is subtracted
                                            ;from itself minus
                                            ;the carry bit if
                                            ;applicable to set
                                            ;the low byte
 b4:  03 d0           rcall .+6             ; 0xbc <__divmodhi4>
 b6:  60 93 02 01     sts   0x0102, r22     ; 0x800102
<__data_end>
 ba:  08 95           ret
*/

unsigned char Global_A;
unsigned char Global_B = 1;
unsigned char Global_C = 2;
void main(void)
{
     Global_A = Global_C / Global_B;
}
/*
  a0: 80 91 00 01     lds   r24, 0x0100     ; 0x800100 <Global_C>
  a4: 60 91 01 01     lds   r22, 0x0101     ; 0x800101 <Global_B>
  a8: 03 d0           rcall .+6        ; 0xb0 <__udivmodqi4>
  aa: 80 93 02 01     sts   0x0102, r24     ; 0x800102
<__data_end>
  ae: 08 95           ret
*/
```

Schematics (Hardware)

None

Analysis

The concepts learned during this investigation pertain to the efficiency and processes regarding different design patterns for Assembly and C programming. The process of subroutines revealed how the STACK is manipulated using nested vs. un-nested subroutines. The growing STACK in the memory reveal how programs can cause a stack overflow if subroutines are not properly terminated. This is useful in writing valid code. The data table provides a method to quickly generate data when the data is known and expected. It can improve efficiency by eliminating unnecessary calculations, which adds extra time to the overall program. The limitation of using a data table is the size of memory needed to be reserved for the table alone. The C programming and analysis of the generated lss output file provides insight into how code in C translates to Assembly, which can be of much use for debugging code.

Conclusion

The purpose of these investigations was to gain new knowledge regarding specific concepts related to Assembly programming. The concepts explored were basic, yet vital to a fundamental understanding of how to more efficiently structure programs. They provided insight into the underlying processes for higher level operations, and revealed methods in which programs could be made more efficient at the Assembly level.

References

None