CECS/ECE-412
Spring 2022
Lab #2

Drew Branum

Report

(Points 80)

Demo
(15 Points)

Quiz
(5 Points)

Lab 2 Branum

Andrew Branum

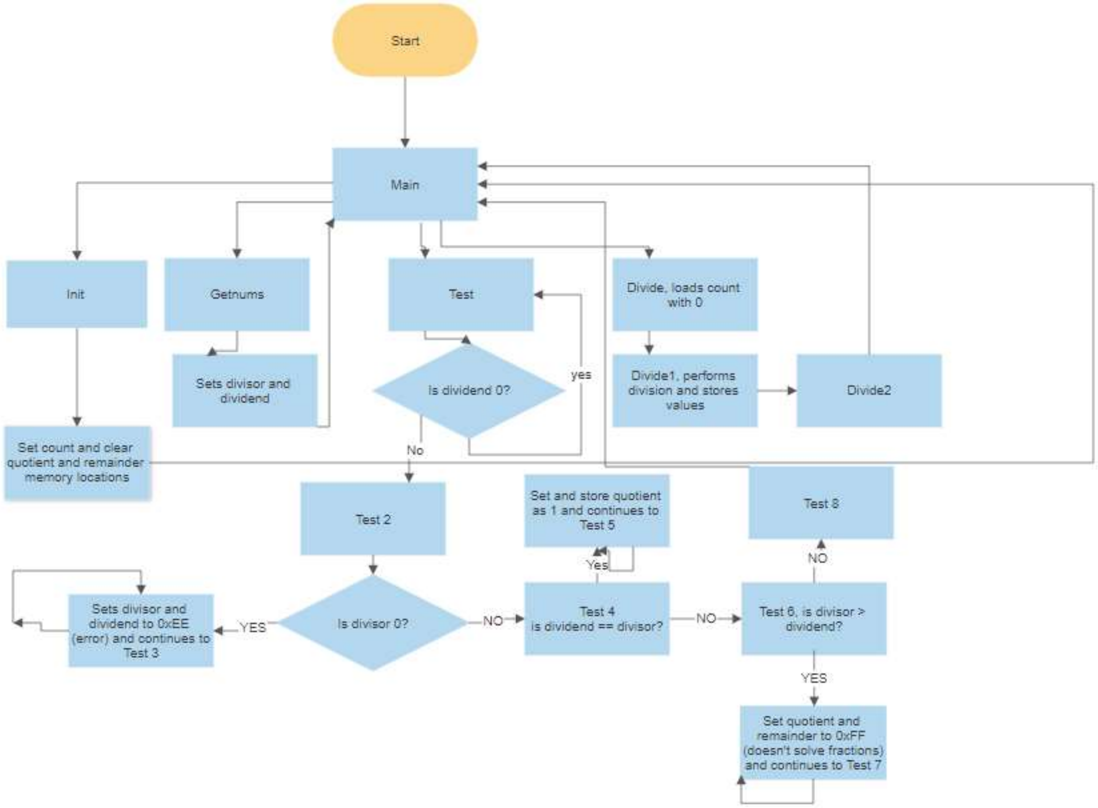
ECE 412

Abstract

Assembly programming and C language is researched further and implemented. Specifically, the AVR assembly language using Atmel 328P(B) MCU. Modular programming and subroutines are also examined and researched. Assembly language code is implemented to create a division program in Microchip Studio that divides two 8-bit numbers. The program's Stack, Stack Pointer, Program Counter, and memory locations are evaluated while debugging the program. A data table written in assembly programming language that converts its values from Celsius to Fahrenheit, is evaluated on how it is indexed and is re-written to sort 20 random values stored in the table. The table values are stored in FLASH memory and after being sorted, are transferred to SRAM. A new C Project for the ATMega328PB is created in Microchip Studio to compare C and assembly language. C code that divides two integers, signed and unsigned, is entered into Microchip Studio and the .lss file is examined to evaluate the equivalent assembly code. The C code is then changed to divide two characters, signed and unsigned, and the same process occurs. The data, results, and similarities are recorded and analyzed. Conclusions are made about the relationship between assembly language division and C language division, low-level modular programming, and relationships between software and hardware within Atmel Studio based off the debugger and .lss files. The conclusion agrees with Atmel documentation and research after the examination of assembly language code and .lss files.

Body

The latest Atmel Microchip Studio is utilized on a Microsoft Windows PC. Assembly language code that divides two 8-bit numbers is inserted into a new 8-bit assembler Atmel Studio Project using the ATmega328PB named Lab2. The main.asm code is generated automatically but is replaced by the assembly division code. At the beginning of the program, the quotient and remainder variables are allocated 1 byte of memory and the count is set to 0 in SRAM at 0x100. Directly underneath in the code segment, the dividend and divisor (8-bit constants) are set to their respective values at 0x100 within the FLASH memory. The code implemented then goes on to reset the vector at address 0x0 in the FLASH memory and sets the main address to 0x100 in FLASH. Within the main, the subroutine init is called first. The init subroutine initially gets the count that is stored in SRAM and loads r0 with the value which is 0. Then the quotient and remainder in SRAM are set with the value in r0 and the subroutine returns to main. Next, the getnums subroutine is called. This subroutine loads r30 with the dividend set in FLASH memory and r31 with the divisor and then returns to main. The test subroutine is called next. Initially, this subroutine checks to see if r30, or the dividend, is 0. If so, the program iterates to test1 and creates an infinite loop to halt the program since the quotient and remainder would both equal 0. If r30 isn't 0, then the program breaks to test2 to check if r31, or the divisor, is 0. When r31 equates to 0, the program sets r30, the quotient, and the remainder to 0xEE to symbolize an error that would occur from dividing by 0. It then iterates to test 3, creating an infinite loop to halt the program. When r31 is not 0, it breaks to test4 and checks to see if r30 and r31 are equal. If so, the program sets r30 and the quotient to 1 and iterates to test5 which creates an infinite loop. When r30 and r31 are not equal, the program breaks to test6 to see if the r30 is less than r31 by checking to see if r31 – r30 is a negative value. Negative values result in the program setting r30, the quotient, and remainder to 0xFF then iterating to test7 to create an infinite loop. Positive values cause the program to break to test8 which returns to main. The divide subroutine is called last and sets r0 with count. It then iterates to divide1 which increments r0 by 1. Divide1 then subtracts r31 from r30 and stores the value in r30. If r30 is positive after this, it branches back to divide1 and continues until r30 is negative. When r30 is negative, r0 is decremented by 1, and r30 and r31 are added together with the sum being placed back into r30. The quotient is then set with the value in r0, and the remainder is set with the value in r30. Divide2 is then hit and returns to main. The program then iterates to endmain and infinitely loops. A flow chart of the program and pseudocode is provided below:



Pseudocode:

```
Program Start
Allocate memory for quotient and remainder
Set Dividend and Divisor
Jump to Main
Start Main:
        Call init:
                Load count
                Clear quotient and remainder
                Return to main
        Call getnums:
                Load r30 with dividend
                Load r31 with divisor
                Return to main
        Call test:
                Test 1: Check is dividend = 0
                        Break to test2 if no
                        Infinite loop if yes
                Test 2: Check is divisor = 0
                        Break to test4 if no
                        Set r30, quotient, and remainder to 0xEE for error if yes
                        Infinite loop if yes
                Test 4: Check is dividend = divisor
                        Break to test6 if no
                        Set quotient to 1 if yes
```

Infinite loop if yes

Test 6: Check is dividend < divisor

Break to test8 if no

Set r30, quotient, and remainder to 0xFF for error if yes

Inifite loop if yes

Test 8: Return to main

Call divide:

Loads count into r0

Increments r0 by 1

Subtracts r31 from r30

Loops back if value is positive

Breaks if value is negativee and decreases r0 by 1

Adds r30 and r31

Sets quotient with r0

Sets remainder with r30

Return to main

Infinite Loop

Throughout the program the STACK, Stack Pointer, and Program Counter are all modified. At the beginning of the program, the Program Counter (PC) is 0x00000000, the Stack Pointer (SP) is 0x08FF, and the value in the stack at 0x08FF is 00 00. As the program jumps to the main, the PC changes to 0x00000100. When init is called within the main, the SP is changed to 0x08FD because 01 02 is inserted into the stack at 0x08FF and the PC changes to 0x0000010A. After init returns to main, the SP changes back to 0x08FF and the PC is 0x00000102. Getnums is then called, and the PC is changed to 0x00000111, the SP is changed to 0x08FD, and the stack changes from 01 02 to 01 04 because each call stores the return address of the instruction after the call so the program doesn't forget. Once again, when the subroutine returns to main the SP is back to 0x08FF and PC is 0x00000104. The test subroutine is then called and the SP changes to 0x08FD, the PC changes to 0x00000114, and the stack stores the value 01 06. As it returns, the PC is 0x00000106, and the SP reverts to 0x08FF. Divide is then called, and the stack contains 01 08, the PC is 0x00000131, and the SP is 0x08FD. Finally, when the subroutine returns, the PC sis 0x00000108 and the SP is 0x08FF again.

When the program is re-written with only one call function in the main, the SP, PC, and stack change accordingly. The PC remains the same at 0x00000100 when the program jumps too main. When init is called, the PC is 0x00000104, the SP is 0x08FD, and the stack holds 01 02. As getnums is called within init, the PC is 0x0000010D, SP is 0x08FB, and the stack holds 01 0c at 0x08FD. The subroutine test is then called within getnums, and the PC is 0x00000112, the SP is 0x08F9, and the stack still holds 01 11 at 0x08FB. When the divide subroutine is called in test, the PC is 0x00000131, the SP is 0x08F7, and the stack contains 01 30 at 0x08F9. When the program hits the return in divide2, it returns to the ret in test8 and the PC is 0x00000130 and the SP is 0x08F9. It then returns to the ret in getnums and the PC is 0x00000111 and the SP is 0x08FB. Finally, the ret in init is hit, and the PC is 0x0000010C and the SP is 0x08FD. After the final ret executes, the PC is 0x00000102, and the SP is 0x08FF. These values end up being different from the previous division example because the SP points to the value in the stack that hold the return address to the line after the call address. The PC is different because the first division example has 4 lines of code executed in the main while the revised example only had one and each line of code uses 2 bytes therefore leaving the respective PCs 0x00000102 and 0x00000108.

Next, a program that provides a simple look-up table that pertains to converting Celsius to Fahrenheit is implemented within Atmel Studio. The table contains 20 values of Fahrenheit values from 32-66. The variable Celsius is set to equal 5 and iterating through the program results in r0 and the output, which is a variable set at 0x100, to be set with the equivalent value in Fahrenheit (41 or 0x29).

A new C project for the ATMega328PB is then created to evaluate the relationship between C code and the hardware's assembly code. C code that divides two integers is given and implemented. At first, 2 unsigned characters are divided and the .lss file is examined. After that, two signed characters are divided and the .lss file is examined. The two yield differences that can be found in the .lss file. The main difference is that the unsigned just loads the values to registers, does the division operation, and then loads the answer to memory. The signed characters must check for the sign before doing the division. Then, unsigned and signed division of integers is evaluated. The unsigned division assigned 4 registers values rather than 2 as seen within character division because integers are double the size of characters. The unsigned division then carries out the division and stores the answer in two memory spaces since it is 4 bytes long rather than 2. The signed division on the other hand, is the exact same as the unsigned division.

Source Code (Software)

Division:

```
            .dseg
            .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 IRAM
        ;******************************
        ;* Vector Table (partial)
        ;******************************
            .org    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
            call    getnums         ;Check the STACK,SP,PC here. Stack: 02, SP: 0x08FF, PC: 0x00000102
            call    test            ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FF, PC: 0x00000104
            call    divide          ;Check the STACK,SP,PC here. Stack: 06, SP: 0x08Ff, PC: 0x00000106
endmain:    jmp     endmain
init:       lds     r0,count        ;get initial count, set break point here and check the STACK,SP,PC; Stack: 02, SP: 0x08FD, PC: 0x0000010A
            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: 02, SP: 0x08FD, PC: 0x00000110
getnums:    ldi     r30,dividend    ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FD, PC: 0x00000111
            ldi     r31,divisor
            ret                     ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FD, PC: 0x00000113
test:       cpi     r30,0           ; is dividend == 0 ?
            brne    test2
test1:      jmp     test1           ; halt program, output = 0 quotient and 0 remainder
test2:      cpi     r31,0           ; is divisor == 0 ?
            brne    test4
            ldi     r30,0xEE        ; set output to all EE's = Error division by 0
```

```
            sts     quotient,r30
            sts     remainder,r30
test3:      jmp     test3           ; halt program, look at output
test4:      cp      r30,r31         ; is dividend == divisor ?
            brne    test6
            ldi     r30,1           ;then set output accordingly
            sts     quotient,r30
test5:      jmp     test5           ; halt program, look at output
test6:      brpl    test8           ; is dividend < divisor ?
            ser     r30
            sts     quotient,r30
            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 count which is 0 since it is set in the SRAM in the code above
divide1:    inc     r0              ;increments r0 by 1
            sub     r30,r31         ;subtracts r31 from r30 and places the value back into r30
            brpl    divide1         ;branches to divide1 if the value above is positive
            dec     r0              ;decreases r0 by 1 to find the quotient
            add     r30,r31         ;adds r31 and r30 and places the sum in r30
            sts     quotient,r0     ;places the value in r0 in the quotient location in IRAM: 0x0100
            sts     remainder,r30   ;places the value in r30 in the remainder location in IRAM: 0x0101
divide2:    ret                     ;returns to the main
int0h:      jmp     int0h           ; interrupt 0 handler goes here
            .exit
```

Revised Division:

```
            .dseg
            .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 IRAM
        ;*****************************
        ;* Vector Table (partial)
        ;*****************************
            .org    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; Stack: 02, SP: 0x08FD, PC: 0x0000010A
            sts     quotient,r0     ;use the same r0 value to clear the quotient-
            sts     remainder,r0    ;and the remainder storage locations
            call    getnums         ;Check the STACK,SP,PC here. Stack: 02, SP: 0x08FF, PC: 0x00000102
            ret                     ;return from subroutine, check the STACK,SP,PC here. Stack: 02, SP: 0x08FD, PC: 0x00000110
getnums:    ldi     r30,dividend    ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FD, PC: 0x00000111
            ldi     r31,divisor
            call    test            ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FF, PC: 0x00000104
            ret                     ;Check the STACK,SP,PC here. Stack: 04, SP: 0x08FD, PC: 0x00000113
test:       cpi     r30,0           ; is dividend == 0 ?
            brne    test2
test1:      jmp     test1           ; halt program, output = 0 quotient and 0 remainder
test2:      cpi     r31,0           ; is divisor == 0 ?
            brne    test4
            ldi     r30,0xEE        ; set output to all EE's = Error division by 0
            sts     quotient,r30
            sts     remainder,r30
test3:      jmp     test3           ; halt program, look at output
test4:      cp      r30,r31         ; is dividend == divisor ?
            brne    test6
            ldi     r30,1           ;then set output accordingly
            sts     quotient,r30
test5:      jmp     test5           ; halt program, look at output
test6:      brpl    test8           ; is dividend < divisor ?
            ser     r30
            sts     quotient,r30
            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          ;Check the STACK,SP,PC here. Stack: 06, SP: 0x08Ff, PC: 0x00000106
            ret                     ; otherwise, return to do positive integer division
divide:     lds     r0,count        ;loads r0 with count which is 0 since it is set in the SRAM in the code above
divide1:    inc     r0              ;increments r0 by 1
            sub     r30,r31         ;subtracts r31 from r30 and places the value back into r30
            brpl    divide1         ;branches to divide1 if the value above is positive
            dec     r0              ;decreases r0 by 1 to find the quotient
            add     r30,r31         ;adds r31 and r30 and places the sum in r30
            sts     quotient,r0     ;places the value in r0 in the quotient location in IRAM: 0x0100
            sts     remainder,r30   ;places the value in r30 in the remainder location in IRAM: 0x0101
divide2:    ret                     ;returns to the main
int0h:      jmp     int0h           ; interrupt 0 handler goes here
            .exit
```

Table Comments:

```
            .dseg
            .org    0x100
output:     .byte   1                   ;allocates one byte for the table
            .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)     ;initializes the Z low pointer (r30) with 0x14 (20)
            ldi     ZH,high(2*table)    ;initializes the Z high pointer (r31) with 0x02 (2)
            ldi     r16,celsius         ;loads r16 with the value stored in celsius: 5
            add     ZL,r16              ;adds ZL (r30) and r16 and puts the sum in ZL
            ldi     r16,0               ;loads r16 with 0
            adc     ZH,r16              ;adds ZH (r31) and r16 with carry and puts the result in ZH
            lpm                         ;stores 41 (0x29)in r0 which is found at 0x0219 in Flash memory because thats where the Z pointer points to 0x0219 which stores the value 0x29
            sts     output,r0           ;store look-up result to SRAM, stores 29 in SRAM at 0x0100
            ret                         ;ret returns to the first part of the .cseg or if calling subroutines, it reutrns to the line after the call
        ; 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
```

Table Sort:

```
            .dseg
            .org    0x100
            .equ    byteNumber = 20
            .def    loopCount = r17

tbl:        .byte   byteNumber

output:     .byte   1 ;sets
            .cseg
            .org    0x0
            jmp     main ;partial vector table at address 0x0
            .org    0x100 ;MAIN entry point at address 0x100
main:       ldi     ZL,low(2*table) ;initializes the Z low pointer (r30) with 0x44
            ldi     ZH,high(2*table) ;initializes the Z high pointer (r31) with 0x02

            ldi     XL, low(tbl) ;initializes the X low pointer r(26) 00
            ldi     XH, high(tbl) ;initializes the X high pointer r(27) 01

            ldi     r16, celsius ;nothing?
            add     ZL, r16 ;adds ZL and r16 and the sum is put in ZL = 0
            ldi     r16,0   ;loads r16 with 0
            adc     ZH,r16 ;adds ZH and r16 and the sum is put in ZH =
            ldi     loopCount, byteNumber ;loads loopCOunt with Number = 20
tblLoop:    lpm     r16,Z+ ;loads the first value of the table into r16 because that is the first place the Z pointer points 
                        ;increments the Z pointer position by one to find the next value
            st      X+, r16 ; stores r16 in the first space pointed to by the X pointer and increments the pointer position by
            dec     loopCount
            brne    tblLoop

            ldi     r19, $14 ; outer loop counter, loads r19 with $14
outer:      ldi     r26, 00 ;loads r26 with 00
            ldi     r27, 01 ;loads r27 with 01
            ldi     r18, $14 ;loads r18 with $14
inner:      ld      r16,x+ ; Load r16 with data space loc. 0x0103
            ld      r17,x ; Load r17 with data space loc. 0x0104(X post inc)
            cp      r17,r16 ;compare
            brlt    swp ;breaks to swp if r17 is less than r16
back:       dec     r18 ;decreases r18
            brne    inner ;breaks to inner if r16 and r17 aren't equal
            dec     r19 ;decreases r19
            brne    outer ;breaks to outer if not even
end:        jmp     end
swp:        st      x,r16 ;stores r16 in current position of x pointer
            st      -x,r17 ;stores r17 in current position of x pointer (decresed by one)
            inc     r26 ;increments r26
            rjmp    back ;returns to back

            sts     output,r0 ;stores r0 in SRAM at output(0x0102)
            ret
table:      .db     1, 3, 4, 7, 5, 8, 6, 2, 12, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21
            .equ    celsius = 0 ;modify Celsius from 0 to 19 degrees for different results
```

Copyable code:

```
.dseg
```

```
                    .org   0x100
                    .equ   byteNumber = 20
                    .def   loopCount = r17

tbl:                .byte  byteNumber

output:             .byte  1 ;sets
                    .cseg
                    .org   0x0
                    jmp           main ;partial vector table at address 0x0
                    .org   0x100 ;MAIN entry point at address 0x100
main:               ldi           ZL,low(2*table) ;initializes the Z low pointer
                                  (r30) with 0x44
                    ldi           ZH,high(2*table) ;initializes the Z high pointer
                                  (r31) with 0x02

                    ldi           XL, low(tbl) ;initializes the X low pointer
                                  r(26) 00
                    ldi           XH, high(tbl) ;initializes the X high pointer
                                  r(27) 01

                    ldi           r16, celsius ;nothing?
                    add           ZL, r16 ;adds ZL and r16 and the sum is put in
                                                ZL = 0
                    ldi           r16,0  ;loads r16 with 0
                    adc           ZH,r16 ;adds ZH and r16 and the sum is put in ZH
=
                    ldi           loopCount, byteNumber ;loads loopCOunt with Num
                                                            ber = 20

tblLoop:            lpm           r16,Z+ ;loads the first value of the table into
                                  r16 because that is the first place the Z
                                  pointer points to
                                                ;increments the Z pointer position
                                  by one to find the next value
                    st            X+, r16 ; stores r16 in the first space pointed
                                  to by the X pointer and increments the pointer
                                  position by one
                    dec    loopCount
                    brne   tblLoop

                    ldi    r19, $14 ; outer loop counter, loads r19 with
$14
outer:        ldi          r26, 00       ;loads r26 with 00
                    ldi           r27, 01       ;loads r27 with 01
                    ldi           r18, $14 ;loads r18 with $14
inner:        ld           r16,x+ ; Load r16 with data space loc. 0x0103
                    ld            r17,x ; Load r17 with data space loc. 0x0104(X
post inc)
                    cp            r17,r16 ;compare
                    brlt   swp ;breaks to swp if r17 is less than r16
back:         dec          r18 ;decreases r18
                    brne   inner ;breaks to inner if r16 and r17 aren't equal
                    dec           r19 ;decreases r19
                    brne   outer ;breaks to outer if not even
end:          jmp          end
swp:          st           x,r16 ;stores r16 in current position of x pointer
                    st            -x,r17 ;stores r17 in current position of x
```

```
pointer (decresed by one)
                    inc          r26 ;increments r26
                    rjmp   back ;returns to back

                    sts          output,r0 ;stores r0 in SRAM at output(0x0102)
                    ret
table:        .db          1, 3, 4, 7, 5, 8, 6, 2, 12, 11, 13, 14, 15, 16, 17, 18,
19, 20, 21
                    .equ   celsius = 0 ;modify Celsius from 0 to 19 degrees for
different results
                    .org   0x100
                    .equ   byteNumber = 20
                    .def   loopCount = r17

tbl:                .byte  byteNumber

output:             .byte  1             ;sets
                    .cseg
                    .org   0x0
                    jmp          main ;partial vector table at address 0x0
                    .org   0x100 ;MAIN entry point at address 0x100
main:               ldi          ZL,low(2*table) ;initializes the Z low
                                       pointer (r30)
                                       with 0x44
                    ldi          ZH,high(2*table) ;initializes the Z high pointer
                                       (r31) with 0x02

                    ldi          XL, low(tbl) ;initializes the X low pointer
                                 r(26) 00
                    ldi          XH, high(tbl) ;initializes the X high pointer
                                       r(27) 01

                    ldi          r16, celsius
                    add          ZL, r16 ;adds ZL and r16 and the sum is put in
                                             ZL = 0
                    ldi          r16,0  ;loads r16 with 0
                    adc          ZH,r16 ;adds ZH and r16 and the sum is put in ZH
                                             =
                    ldi          loopCount, byteNumber ;loads loopCOunt with Num
                                       ber = 20
tblLoop:            lpm          r16,Z+ ;loads the first value of the table into
                                 r16 be
                                 cause that is the first place the Z
                          pointer points to
                                       ;increments the Z pointer position
                                 by one to find the next value
                    st     X+, r16 ; stores r16 in the first space pointed
                                             to by the X pointer and in-
                                       crements the pointer position by
                                       one
                    dec    loopCount
                    brne   tblLoop

                    ldi    r19, $14 ; outer loop counter, loads r19 with
                                 $14
outer:        ldi          r26, 00       ;loads r26 with 00
                    ldi    r27, 01       ;loads r27 with 01
                    ldi    r18, $14 ;loads r18 with $14
```

```
inner:      ld          r16,x+ ; Load r16 with data space loc. 0x0103
                  ld    r17,x ; Load r17 with data space loc. 0x0104(X
                                      post inc)
                  cp    r17,r16 ;compare
                  brlt  swp ;breaks to swp if r17 is less than r16
back:       dec         r18 ;decreases r18
                  brne  inner ;breaks to inner if r16 and r17 aren't equal
                  dec   r19 ;decreases r19
                  brne  outer ;breaks to outer if not even
end:        jmp         end
swp:        st          x,r16 ;stores r16 in current position of x pointer
                  st    -x,r17 ;stores r17 in current position of x
                                      pointer (decresed by one)
                  inc   r26 ;increments r26
                  rjmp  back ;returns to back

                  sts   output,r0 ;stores r0 in SRAM at output(0x0102)
                  ret
table:            .db         1, 3, 4, 7, 5, 8, 6, 2, 12, 11, 13, 14, 15, 16,
                              17, 18, 19, 20, 21
                  .equ  celsius = 0 ;modify Celsius from 0 to 19 degrees for
                        different results
```

Unsigned Char Division C:

```
000000f2 <main>:
unsigned char Global_B = 1;
unsigned char Global_C = 2;

void main(void){

      Global_A = Global_C / Global_B;
  f2:  80 91 00 01   lds    r24, 0x0100   ; loads r24 with the value at 0x0100
                                          (0x02)
  f6:  60 91 01 01   lds    r22, 0x0101   ; loads r22 with the value at 0x101
                                          (0x01)
  fa:  0e 94 82 00   call   0x104         ; calls 0x104
  fe:  80 93 02 01   sts    0x0102, r24   ; stores the value in r24 at 0x0102
 102:  08 95         ret
```

Signed Char Division C:

```
000000f2 <main>:
signed char Global_B = 1;
signed char Global_C = 2;

void main(void){

      Global_A = Global_C / Global_B;
  f2:  80 91 00 01   lds    r24, 0x0100   ;loads r24 with value at 0x0100 in SRAM
                                          (02)
  f6:  08 2e         mov    r0, r24              ;copies r24 to r0
  f8:  00 0c         add    r0, r0        ;adds r0 and r0 and puts sum in r0
  fa:  99 0b         sbc    r25, r25      ;subtract with carry high byte
  fc:  60 91 01 01   lds    r22, 0x0101   ;loads r22 with the value at 0x0101 (01)
 100:  06 2e         mov    r0, r22              ;copies r22 to r0
 102:  00 0c         add    r0, r0        ;adds r0 and r0
```

```
 104:  77 0b         sbc    r23, r23      ;subtract with carry high byte
 106:  0e 94 88 00   call   0x110         ; calls 0x110
 10a:  60 93 02 01   sts    0x0102, r22   ; stores r22 in 0x0102
 10e:  08 95         ret
```

Unsigned Int Division C:

```
000000f2 <main>:
unsigned int Global_B = 1;
unsigned int Global_C = 2;

void main(void){

       Global_A = Global_C / Global_B;
  f2:  80 91 00 01   lds    r24, 0x0100   ; loads r24 with values at 0x0100 (02)
  f6:  90 91 01 01   lds    r25, 0x0101   ; loads r25 with value at 0x0101 (0)
  fa:  60 91 02 01   lds    r22, 0x0102   ; loads r26 with value at 0x0102 (01)
  fe:  70 91 03 01   lds    r23, 0x0103   ; loads r23 with value at 0x0103 (00)
 102:  0e 94 88 00   call   0x110  ; calls 0x110
 106:  70 93 05 01   sts    0x0105, r23   ; stores r23 in data space at 0x0105 (0)
 10a:  60 93 04 01   sts    0x0104, r22   ; stores r22 in data space at 0x0104 (2)
                                                        answer
 10e:  08 95         ret                  ;return
```

Signed Int Division C:

```
000000f2 <main>:
signed int Global_B = 1;
signed int Global_C = 2;

void main(void){

       Global_A = Global_C / Global_B;
  f2:  80 91 00 01   lds    r24, 0x0100   ;load r24 with value at 0x0100
  f6:  90 91 01 01   lds    r25, 0x0101   ; load r25 with value at 0x0101
  fa:  60 91 02 01   lds    r22, 0x0102   ; load r22 with value at 0x0102
  fe:  70 91 03 01   lds    r23, 0x0103   ; load r23 with value at 0x0103
 102:  0e 94 88 00   call   0x110         ; call 0x110
 106:  70 93 05 01   sts    0x0105, r23   ; store r23 in data space at 0x0105
 10a:  60 93 04 01   sts    0x0104, r22   ; store r22 in data space at 0x0104
 10e:  08 95         ret                  ;return
```

Schematics (Hardware)

Include schematic(s) of circuits relevant to the project.
If there are none then type 'none'

none

Analysis

Overall, the laboratory process revealed information about the relationship between C code and assembly language code. Along with this, information about tables and sorting algorithms within assembly language was analyzed and implemented. Information about division within assembly language is learned through implementation, research, and analyzation of .lss files within C projects. The ATMEL datasheet for the 328P(B), User's manual for Atmel, Atmel Studio 7, and Google were used to investigate the topics and devices at hand as well. After all the research was completed and the code examples were implemented, analysis took place to discover how assembly language performs division in comparison with C code language. Analysis of table and sorting also took place with the table example. This allowed for a general overlook as to how assembly language stores values in the various memory locations. All the examples provided information about how assembly language perform various operations, and they were all validated by data sheets and research.

The implementation and research of these code examples were performed in order to gain a deeper knowledge on how embedded systems in micro-computers store memory and perform specific operations that most people don't give a second thought to because modern code language does it easily. This information is important to understand as assembly language is what modern day programming language is converted to at compile time. Understanding the code under the surface level code allows for developers to gain a deeper understanding on how exactly computers work. It also allows for faster programs, and easier debugging within assembly code. The practice of assembly language code allows developers to be even more specific when creating their programs. It takes away a level of obscurity between the developer and computer, allowing for clear and precise instructions.

Conclusion

Assembly programming language, tables, and division methods were researched and implemented within Atmel Studio 7 to gain a better understanding on how they work. The code examples provided were implemented and executed within Atmel Studio, and the various memory locations and .lss files were examined to see if they matched up with the datasheets. All the information that was analyzed matched up with the datasheets and research. Sorting algorithms and character division were also implemented and the information reveal also matched with the datasheets and research.

References

Cite all sources you researched and/or used to perform this lab
If no references were used then type 'none'

ATmega328PB Instruction Set Summary. (2021). *ATmega328PB Instruction Set Summary*.

Atmel START User's Guide. (2020). *Atmel START User's Guide*. https://www.microchip.com/content/dam/mchp/documents/atmel-start/Atmel-START-User-Guide-DS50002793B.pdf