ECE-412-01

Fall 2017

Project # 2

|  |
| --- |
| Name |
| Spencer Goff |

Project 2

Spencer Goff

ECE-412-01

Abstract

In this lab, many topics will be explored, such as storing values in SRAM, the benefits of lookup tables, the relationships between C and Assembly code, writing pseudocode and creating flow charts for Assembly code, using a counter to divide integers in Assembly, and how the stack and stack pointers vary based on nested versus consecutive calls.

Body

Program Description:

This program accepts two integers, a dividend and a divisor, and divides them. It does so by subtracting the divisor from the dividend repeatedly until the dividend reaches 0 or less. Each time this process occurs, a count variable is incremented. If the dividend reaches 0 exactly, the quotient is set equal to count. If the dividend reaches less than 0, count is decremented by 1 and then remainder is set equal to dividend.

After running the program and searching the .lss file, I found the following: the dividend is stored in FLASH memory (code segment); the divisor is store in FLASH memory; the count and remainder are stored in SRAM (address 002001); the quotient is also stored in SRAM (address 002000).

The Stack Pointer stores the memory address of the most recently executed command in the main function. Whenever the ret (“return”) command is called in a non-main section, the program resumes at the next command in main that hasn’t been executed yet. It knows where this command is stored based on the address stored in the Stack Pointer. When the program is executing in the main section, the address stored by Stack Pointer is 0x5FFF. When the program is executing in a non-main section, the address stored by Stack Pointer is 0x5FFC.

Pseudocode:

var quotient, remainder, count = 0, dividend = 13, divisor = 3

if(error)

{

begin oscillator failure interrupt handler

}

quotient = count

remainder = count

if(dividend != 0)

{

if(divisor != 0)

{

if(dividend != divisor)

{

if(dividend < divisor)

{

break all if

}

else

{

dividend = 0xFFF

quotient = dividend

remainder = dividend

}

}

else

{

dividend = 1

quotient = dividend

}

}

else

{

dividend = 0xEE

quotient = dividend

remainder = dividend

}

}

r0 = count

increment: r0 = r0 + 1

if((dividend - divisor) > 0)

{

go to increment

}

else

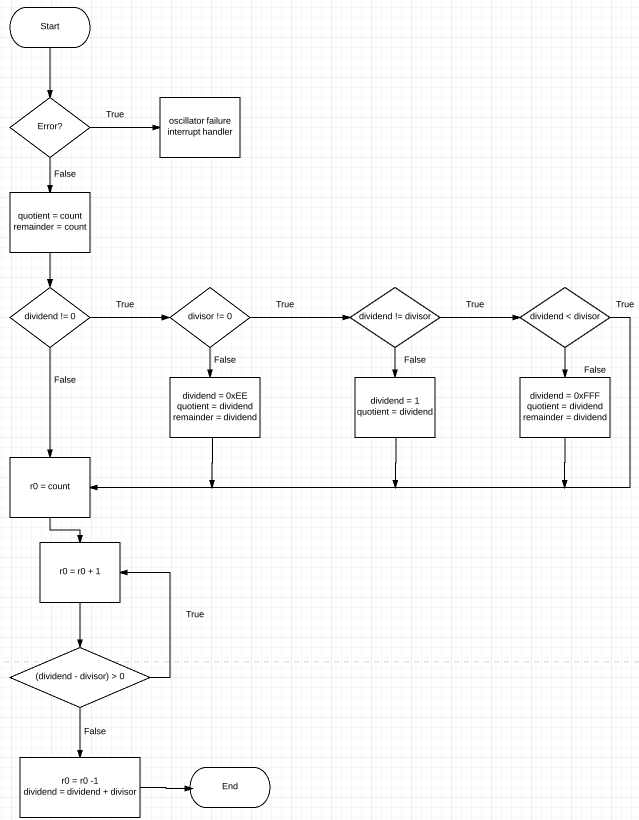
{

r0 = r0 - 1

dividend = dividend + divisor

}

Pseudocode Discussion: The above pseudocode is a high-level overview of the logic that the assembly program is carrying out. It isn’t any specific language, but is similar to modern languages such as Python and Swift. It should be simple to follow and understand the essentials of how the corresponding AVR assembly program (below) works.

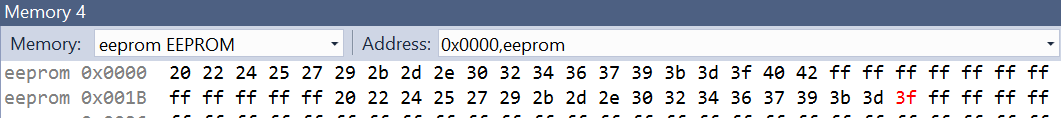
Flowchart:

Flowchart Description: The above flowchart diagrams, at a high level, the logic of the below AVR assembly code (and, by extension, the above pseudocode). The program begins at “Start” and finishes at “End”. Diamonds represent decisions made based on the contained statement, while rectangles represent processes to be executed.

When re-writing the program to have nested CALLs, the stack was different than when consecutive calls were used. When consecutive calls were used, the stack values consisted of 0x5FFC’s and 0x5FFF’s. When nested calls were used, however, 0x5FFC, 0x5FF9, 0x5FF6, and 0x5FF9 were all used.

In part 2, the Fahrenheit lookup table is indexed so that the 1st index is the equivalent Fahrenheit value to C = 0, the second index is the Fahrenheit equivalent of C = 1, and so on through C = 19 degrees. The Fahrenheit values are sometimes rounded (for example, C = 19 should convert to F = 66.2, but the lookup table rounds to 66).

In Part 3, the C program, the EEPROM locations at 0x0 were full of ff’s. After executing the table initialization, these changed to the hexadecimal values corresponding to the decimal values stored in the Fahrenheit table. This table is stored in EEPROM, as opposed to the table in Part 2, which was stored in SRAM. This means that the values stored in this table are not erased after stopping a program, unlike the Part 2 table that was stored in SRAM.



The eeprom.h header file provides methods allowing us to interact with EEPROM, such as reading, writing, and updating words, floats, bytes, and blocks.

Software

Part 1 Code – Original (commented)

/\*

\*

\* lab2p1.asm

\* Positive 8 bit Integer Division

\* Author: Eugene Rockey

\*/

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* Declare Variables

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.dseg

quotient: .byte 1 ;uninitialized quotient variable in SRAM (aka data segment)

remainder: .byte 1 ;uninitialized remainder variable stored in SRAM

.set count = 0 ;sets a “ count” variable in SRAM that can be changed later

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.cseg ; Declare, Initialize Constants (modify for different results)

.equ dividend = 13 ;8-bit dividend constant stored in FLASH memory aka code segment

.equ divisor = 3 ;8-bit divisor constant stored in FLASH memory

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* Vector Table (partial)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reset: jmp main ;RESET Vector at address 0x0 in FLASH memory (handled by MAIN)

ofail: jmp oscf ;Oscillator Failure Vector at address 0x2 in FLASH memory ()

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* MAIN entry point to program\*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.org 0x200 ;originate MAIN at address 0x200 in FLASH memory (step through the code)

main: call init ;initialize variables subroutine, set break point here, check the STACK: 0x5FFF, SP: 0x5FFF, PC:0x00000200

call getnums ;Check the STACK: 0x5FFF 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFF, PC: 0x00000202

call test ;Check the STACK: 0x5FFF 0x5FFC 0x5FFC 0x5FFF 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFF, PC: 0x00000204

call divide ;Check the STACK: 0x5FFF 0x5FFF 0x5FFC 0x5FFC 0x5FFF 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFF,PC: 0x00000206

endmain: jmp endmain

init: lds r0,count ;get initial count, set break point here and check the STACK: 0x5FFF 0x5FFC, SP: 0x5FFC, PC:0x0000020A

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: 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFC, PC: 0x00000210

getnums: ldi r30,dividend ;Check the STACK: 0x5FFC 0x5FFF 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFC,PC: 0x00000211 here.

ldi r31,divisor

ret ;Check the STACK: 0x5FFC 0x5FFC 0x5FFF 0x5FFC 0x5FFF 0x5FFC, SP: 0x5FFC, PC: 0x00000213 here.

test: cpi r30,0 ; is dividend == 0 ?

brne test2 ;branch if not equal (i.e. if r30 the dividend != 0, then go to 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

test7: jmp test7 ; halt program look at output

test8: ret ; otherwise, return to do positive integer division

divide: lds r0,count ;student comment goes here: load contents of r0 with the value of count

divide1: inc r0 ;increments the value of r0 (which was set to the value of count) by 1

sub r30,r31 ; subtracts the divisor from the dividend

brpl divide1 ;branch to divide1 if (dividend - divisor) > 0

dec r0 ;decrement value in r0

add r30,r31 ;add dividend and divisor

sts quotient,r0 ;set quotient = value in r0

sts remainder,r30 ;set remainder = dividend

divide2: ret ;goes back to main section

oscf: jmp oscf ; oscillator failure interrupt handler goes here.

.exit

Part 1 Code - Nested Version

/\*

\*

\* lab2p1.asm

\* Positive 8 bit Integer Division

\* Author: Eugene Rockey

\*/

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* Declare Variables

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.dseg

quotient: .byte 1 ;uninitialized quotient variable in SRAM (aka data segment)

remainder: .byte 1 ;uninitialized remainder variable stored in SRAM

.set count = 0 ;sets a “ count” variable in SRAM that can be changed later

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.cseg ; Declare, Initialize Constants (modify for different results)

.equ dividend = 13 ;8-bit dividend constant stored in FLASH memory aka code segment

.equ divisor = 3 ;8-bit divisor constant stored in FLASH memory

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* Vector Table (partial)

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reset: jmp main ;RESET Vector at address 0x0 in FLASH memory (handled by MAIN)

ofail: jmp oscf ;Oscillator Failure Vector at address 0x2 in FLASH memory ()

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

;\* MAIN entry point to program\*

;\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.org 0x200 ;originate MAIN at address 0x200 in FLASH memory (step through the code)

main: call init ;initialize variables subroutine, set break point here, check the STACK: 0x5FFF, SP: 0x5FFF, PC: 0x00000200

endmain: jmp endmain

init: lds r0,count ;get initial count, set break point here and check the STACK: 0x5FFC 0x5FFF, SP: 0x5FFC, PC:0x00000204

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: 0x5FFC 0x5FFF, SP: 0x5FFC, PC: 0x0000020A

ret ;return from subroutine, STACK: 0x5FFC 0x5FF9 0x5FF6 0x5FF9 0x5FF9 0x5FFC 0x5FFF, SP: 0x5FFC, PC: 0x0000020C

getnums: ldi r30,dividend ;Check the STACK: 0x5FF9 0x5FFC 0x5FFF, SP: 0x5FF9, PC: 0x0000020D

ldi r31,divisor

call test ;Check the STACK: 0x5FF9 0x5FF9 0x5FFC 0x5FFF, SP: 0x5FF9, PC: 0x0000020F

ret ;Check the STACK: 0x5FF9 0x5FF6 0x5FF9 0x5FF9 0x5FFC 0x5FFF, SP: 0x5FF9, PC: 0x00000213 here.

test: cpi r30,0 ; is dividend == 0 ?

brne test2 ;branch if not equal (i.e. if r30 the dividend != 0, then go to 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

test7: jmp test7 ; halt program look at output

test8: call divide ;Check the STACK: 0x5FF6 0x5FF9 0x5FF9 0x5FFC 0x5FFF,SP: 0x5FF6,PC: 0x00000206

ret ; otherwise, return to do positive integer division

divide: lds r0,count ;load contents of r0 with the value of count

divide1: inc r0 ;increments the value of r0 (which was set to the value of count) by 1

sub r30,r31 ; subtracts the divisor from the dividend

brpl divide1 ;branch to divide1 if (dividend - divisor) > 0

dec r0 ;decrement value in r0

add r30,r31 ;add dividend and divisor

sts quotient,r0 ;set quotient = value in r0

sts remainder,r30 ;set remainder = dividend

divide2: ret ;returns to main section

oscf: jmp oscf ; oscillator failure interrupt handler goes here.

.exit

Part 2 code (commented)

/\*

\* lab2p2.asm

\* Celsius to Fahrenheit Look-Up Table

\* Created: 6/2/2014 10:17:31 AM

\* Author: Eugene Rockey

\*/

.dseg

.org 0x2000

output: .byte 1 ;student comment goes here

.cseg

.org 0x0

jmp main ;partial vector table at address 0x0

.org 0x200 ;MAIN entry point at address 0x200 (step through the code)

main: ldi ZL,low(2\*table) ;stores the lower half of the table (57 to 66) into the lower half of the Z register (ZL)

ldi ZH,high(2\*table) ;stores the upper half of the table (32 to 55) into the upper half of the Z register (ZL)

ldi r16,celsius ;stores the celsius variable (originally = 5) into register 16

add ZL,r16 ;adds celsius value to the lower half of the Z register

ldi r16,0 ;sets the celcius variable equal to 0

adc ZH,r16 ;converts the upper half of the Z register from analog to digital

lpm ;lpm = lpm r0,Z in reality, what does this mean?: load the constant from memory that is pointed to by Z into register 0

sts output,r0 ;store look-up result to SRAM

ret ;consider MAIN as a subroutine to return from, but back where?: Back to the "jmp main" command and loop main

; \*\*\*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 = 19 ;modify Celsius from 0 to 19 degrees for different results

.exit

Part 3 code (C program) - original

/\*

\* lab2p3.c

\* Program EEPROM with Temperature Table

\* Author: Eugene Rockey

\*/

#include <avr/io.h>

#include <avr/eeprom.h>u

int main(void)

{

*uint8\_t* fahrenheit[20] =

{32,34,36,37,39,41,43,45,46,48,50,52,54,55,57,59,61,63,64,66};

for(int i = 0; i < 20; i++)

{

eeprom\_write\_byte((*uint8\_t* \*)i, fahrenheit[i]);

//writing to internal EEPROM via NVM Controller

} //look at eeprom EEPROM Memory Window, address 0x0000

}

Part 3 code (C program) – begin writing at space 0x20

/\*

\* lab2p3.c

\* Program EEPROM with Temperature Table

\* Author: Eugene Rockey

\*/

#include <avr/io.h>

#include <avr/eeprom.h>u

int main(void)

{

*uint8\_t* fahrenheit[20] =

{32,34,36,37,39,41,43,45,46,48,50,52,54,55,57,59,61,63,64,66};

for(int i = 0; i < 20; i++)

{

eeprom\_write\_byte((*uint8\_t* \*)i+32, fahrenheit[i]);

//writing to internal EEPROM via NVM Controller

} //look at eeprom EEPROM Memory Window, address 0x0000

}

Part 3 code (C program) – populate Celsius table

/\*

\* lab2p3.c

\* Program EEPROM with Temperature Table

\* Author: Eugene Rockey

\*/

#include <avr/io.h>

#include <avr/eeprom.h>

int main(void)

{

*uint8\_t* fahrenheit[20] = {32,34,36,37,39,41,43,45,46,48,50,52,54,55,57,59,61,63,64,66};

*uint8\_t* celcius[20];

for(int i = 0; i < 20; i++)

{

eeprom\_write\_byte((*uint8\_t* \*)i, fahrenheit[i]);

//writing to internal EEPROM via NVM Controller

} //look at eeprom EEPROM Memory Window, address 0x0000

for(int i = 0; i < 20; i++)

{

celcius[i] = eeprom\_read\_byte((*uint8\_t* \*)i);

//reading from internal EEPROM via NVM Controller

} //look at eeprom EEPROM Memory Window, address 0x0000

}

Schematics (Hardware)

For part 3, I used an ICE-3 and an A3BU.

Application-Based Analysis

Not applicable.

Conclusion

In this lab, many topics were explored, such as storing values in SRAM, the benefits of lookup tables, the relationships between C and Assembly code, writing pseudocode and creating flow charts for Assembly code, using a counter to divide integers in Assembly, and how the stack and stack pointers vary based on nested versus consecutive calls. Overall, this lab was successful and provided a solid base to build on for future Assembly- and C-based projects.

References

<http://maxembedded.com/2011/06/the-adc-of-the-avr/>

<http://www.atmel.com/images/Atmel-8362-8-and-16bit-AVR-microcontroller-ATxmega256A3BU_datasheet.pdf>

<http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_directives.html>