ECE 375 Lab 2

Large Number Arithmetic

Lab Time: Friday 4-6

Aaron Vaughan

Bradley Heenk

TA Signature

1 Introduction

The purpose of this lab is to increase our understanding of the use of the AVR instruction set library to implement common arithmetic operations on 16 to 24-bit unsigned binary numbers. Performing arithmetic operations does not require any additional program includes other than the standard .include "m128def.inc" file. We were tasked with coding a 16-bit adder, 16-bit subtractor, a 24-bit multiplier, and a compound function that uses all three of the other subroutines. The stack pointer is used to jump in and out of subroutine calls. The syntax and organization are becoming second nature by now for my lab partner and I.

2 Program Overview

This program performs addition, subtraction, multiplication and a compound function that uses all three operations. The setup is basic, in that we define some register names and call instructions out of the AVR instruction set. The tricky part is when jumping in and out of the subroutine calls, we must push all the registers onto the stack to preserve their values for the main program. Then, when exiting we must pop them back off the stack in reverse order, thus preserving their values. We must allocate some SRAM space to hold our input and output variables for each function. After hard coding the values into the SRAM during simulation, all of the functions can be called one after the other. The data space allocated to the output will be updated upon completion of each of the subroutines and displayed in the memory window in the debugger simulation.

3 Internal Register Definitions and Constants

Unlike the last lab, we do not have a driver include that sucks up all of the register resources. We had complete freedom to use as many registers as we wanted to implement the subroutines. We defined 6 registers that would be used for performing arithmetic operations on our variables. This configuration allowed us to hold entire 24-bit values in an easily recognizable form. For the use of multiplication function, we needed to use R0, and R1, but in the end, we never actually use the "MUL" instruction. There are two loop counters used. There is a constant value register named "zero" that may have been useful to clear a register value or variable in the SRAM data memory.

4 Interrupt Vectors

There is one interrupt vector within the skeleton code for this lab. It sets up the initial starting point of our main program. Beginning at memory address $0000 it simply jumps to the initialization routine.

5 Program Initialization

The zero register is set to zero in this section. Also the stack pointer must be set up in order to use the push & pop instructions.

The stack pointer must be initialized to use the stack in the algorithm. This is typically done, as implemented in the our code, to the end of ram. It takes two cycles to perform this because the stack pointer is 16-bits but the register used to communicate with it is only 8-bits wide. Without the use of the stack pointer and use of that type of data structure we would lose track of the main program memory location.

6 Main Program

The main program runs through each of the subroutine calls with a "NOP" instruction on either side of it to halt the simulation for use by the graders to check the otput/input values. When each of the subroutines are called, the main program runs in an infinite loop.

Before going into each of the subroutine calls we set up the values of the operands to be used by that subroutine by setting the Z pointer to the location in program memory that holds the value. We then use the LPM instruction to move it to a register and then store it out to the data space memory.

7 Subroutines

In the beginning of each of the subroutines, we use the stack to store the state of the program and at the end, before the return call, we restore the program by popping the values off the stack back to where they go. Also at the beginning of each of the subroutines, we clear the value stored into the solution in the SRAM location.

Function Name: ADD16
Description: Adds two 16-bit numbers and generates a 24-bit number where the high byte of the result contains the carry out bit.

Function Name: SUB16
Description: Subtracts two 16-bit numbers and generates a 16-bit result.

Function Name: MUL24
Description: Multiplies two 24-bit numbers and generates a 48-bit result using the shift and add technique.

Function Name: COMPOUND
Description: Computes the compound expression ((D - E) + F)$\hat{2}$ by making use of SUB16, ADD16, and MUL24.

D, E, and F are declared in program memory, and must be moved into data memory for use as input operands.

The ADD16 and SUB16 subroutines were cut and paste from previous labs.

8 Stored Program Data

We allocate program memory to store the values of our operands.

9 Additional Program Includes

There were no additional program includes for this lab.

10 Additional Questions

1. Although we dealt with unsigned numbers in this lab, the ATmega128 microcontroller also has some features which are important for performing signed arithmetic. What does the V flag in the status register indicate? Give an example (in binary) of two 8-bit values that will cause the V flag to be set when they are added together.

   The V flag is the overflow flag. It gets set when the result of an addition of two numbers result in an incorrect signed value (in two's compliment) If I add 0b01111111 plus 0b00000001 the sum should be decimal 128. In binary, the number is 0b10000000. When read as two's compliment of that number it looks like -128. The value therefore looks like we added two positive numbers and our solution was negative. This is illogical and to signify this, the overflow bit (V flag) is set.

2. In the skeleton file for this lab, the .BYTE directive was used to allocate some data memory locations for MUL16's input operands and result. What are some benefits of using this directive to organize your data memory, rather than just declaring some address constants using the .EQU directive?

   When researching this question at
   https://www.microchip.com/webdoc/avrassembler/
   avrassembler.wb_directives.html#avrassembler.wb_directives.EQU,
   I found the following statement describing the .equ directive:
   "The EQU directive assigns a value to a label. This label can then be used in later expressions. A label assigned to a value by the EQU directive is a constant and can not be changed or redefined." Not having the ability to redefine the values of these operands puts us at a disadvantage if and only if we want to change these values later and reuse the expressions. Using the .byte directive allows us to reuse the same portion of the data space for later calculations and change the values as we see fit.

11 Difficulties

Implementation of the shift then add multiplication algorithm took a bit of thinking. I realized a few ways to shorten the code after the midterm exam by using the ROR instruction. While writing MUL24 we had some extended jumps that needed to be called and finding a way arround the relative branch jumps was a bit tricky. I ended up setting and clearing the T bit in SREG to indicate when to use my long jump calls.

When we implemented the shift and add function, it worked fine for $ffff but when we used a 2byte operand there were zeros padded into the ost significant byte and the algorithm broke. We ended up using the traditional multiply function to take care of this issue.

12 Conclusion

In this lab, we were required to implement some basic arithmetic functions using AVR assembly code. The lab was largely uninteresting and tedious. We coded the project, compiled, then debugged for a bit, then set up the break points that will be used by the TA's to check the functionality of our subroutines. One useful thing that I learned was how to hard code values into the data memory from program memory.

13 Source Code And Challenge Code

Skip to content
Using Gmail with screen readers

Conversations
1.12 GB (7%) of 15 GB used
Manage
Terms · Privacy · Program Policies
Last account activity: 49 minutes ago
Details

```
;***********************************************************
;*
;* Bradley_Heenk_and_Aaron_vaughan_Lab5_challengecode.asm
;*
;* Enter the description of the program here
;*
;* This is the skeleton file for Lab 5 of ECE 375
;*
;***********************************************************
;*
;*  Author: Bradley Heenk and Aaron Vaughan
;*    Date: 11/1/2019
;*
;***********************************************************

.include "m128def.inc" ; Include definition file

;***********************************************************
;* Internal Register Definitions and Constants
;***********************************************************


.def mpr = r16 ; Multipurpose register
.def rlo = r0 ; Low byte of MUL result
.def rhi = r1 ; High byte of MUL result
.def A = r17 ; A variable low byte
.def Amd = r18 ; A middle byte
.def Ahi = r19 ; A high byte
.def B = r20 ; B variable low byte
.def Bmd = r21 ; B middle byte
.def Bhi = r22 ; B high byte
.def iloop = r23
.def oloop = r24
.def zero = r9 ; Zero register, set to zero in INIT, useful for calculations


;***********************************************************
;* Start of Code Segment
;***********************************************************
.cseg ; Beginning of code segment
```

```
;-------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------
.org $0000 ; Beginning of IVs
rjmp  INIT ; Reset interrupt

.org $0046 ; End of Interrupt Vectors

;-------------------------------------------------------------
; Program Initialization
;-------------------------------------------------------------
INIT: ; The initialization routine
; Initialize Stack Pointer
ldi mpr, low(RAMEND)
out SPL, mpr ; Load SPL with low byte of RAMEND
ldi mpr, high(RAMEND)
out SPH, mpr ; Load SPH with high byte of RAMEND

clr zero ; Set the zero register to zero, maintain
; these semantics, meaning, don't
; load anything else into it.

;-------------------------------------------------------------
; Main Program
;-------------------------------------------------------------
MAIN: ; The Main program
; Setup the ADD16 function direct test

; Move values 0xFCBA and 0xFFFF in program memory to data memory
; memory locations where ADD16 will get its inputs from
; (see "Data Memory Allocation" section below)
ldi XL, low(ADD16_OP1) ; Load in operanads
ldi XH, high(ADD16_OP1)
ldi ZL, low(OperandD<<1)
ldi ZH, high(OperandD<<1)

lpm mpr, Z+ ; Load program memory into mpr from Z and post-inc
st X+, mpr ; Store indirect from mpr into X
lpm mpr, Z ; Load program memory into mpr from Z
```

```
st X, mpr ; Store indirect from mpr into X

ldi YL, low(ADD16_OP2) ; Load in operanads
ldi YH, high(ADD16_OP2)
ldi ZL, low(OperandA<<1)
ldi ZH, high(OperandA<<1)

lpm mpr, Z+ ; Load program memory into mpr from Z and post-inc
st Y+, mpr ; Store indirect from mpr into Y and post inc
lpm mpr, Z ; Load program memory into mpr from Z
st Y, mpr ; Store indirect from mpr into Y

                nop ; Check load ADD16 operands (Set Break point here #1)
call    ADD16; Call ADD16 function to test its correctness
; (calculate FCBA + FFFF)

                nop ; Check ADD16 result (Set Break point here #2)
; Observe result in Memory window

; Setup the SUB16 function direct test

; Move values 0xFCB9 and 0xE420 in program memory to data memory
; memory locations where SUB16 will get its inputs from
ldi XL, low(SUB16_OP1) ; Load in operanads
ldi XH, high(SUB16_OP1)
ldi ZL, low(OperandB<<1)
ldi ZH, high(OperandB<<1)

lpm mpr, Z+ ; Load program memory into mpr from Z and post-inc
st X+, mpr ; Store indirect mpr into X and post inc
lpm mpr, Z ; Load Z from program memory into mpr
st X, mpr ; Store indirect from mpr into X

ldi YL, low(SUB16_OP2) ; Load in operanads
ldi YH, high(SUB16_OP2)
ldi ZL, low(OperandC<<1)
ldi ZH, high(OperandC<<1)

lpm mpr, Z+ ; Load program memory into mpr from Z and post-inc
st Y+, mpr ; Store indirect mpr into Y and post increment
```

```
lpm mpr, Z ; Load program memory from Z into mpr
st Y, mpr ; Store indirect mpr into Y

                nop ; Check load SUB16 operands (Set Break point here #3)
call SUB16; Call SUB16 function to test its correctness
; (calculate FCB9 - E420)

                nop ; Check SUB16 result (Set Break point here #4)
; Observe result in Memory window

; Setup the MUL24 function direct test

; Move values 0xFFFFFF and 0xFFFFFF in program memory to data memory
; memory locations where MUL24 will get its inputs from

ldi XL, low(addrA_24) ; Load in operanads
ldi XH, high(addrA_24)
ldi YL, low(addrB_24)
ldi YH, high(addrB_24)
ldi ZL, low(OperandG<<1)
ldi ZH, high(OperandG<<1)

lpm mpr, Z ; Load program memory into mpr from Z
st X+, mpr ; Store indirect from mpr into X and post inc
st Y+, mpr ; Store indirect from mpr into Y and post inc
lpm mpr, Z ; Load program memory into mpr from Z
st X+, mpr ; Store indirect from mpr into X and post inc
st Y+, mpr ; Store indirect from mpr into Y and post inc
lpm mpr, Z ; Load program memory into mpr from Z
st X, mpr ; Store indirect from mpr into X
st Y, mpr ; Store indirect from mpr into X

                nop ; Check load MUL24 operands (Set Break point here #5)

call MUL24 ; Call MUL24 function to test its correctness
; (calculate FFFFFF * FFFFFF)

                nop ; Check MUL24 result (Set Break point here #6)
; Observe result in Memory window
```

```
                nop ; Check load COMPOUND operands (Set Break point here #7)
call COMPOUND ; Call the COMPOUND function

                nop ; Check COMPUND result (Set Break point here #8)
; Observe final result in Memory window

DONE: rjmp DONE ; Create an infinite while loop to signify the
; end of the program.

;***********************************************************
;* Functions and Subroutines
;***********************************************************

;-----------------------------------------------------------
; Func: ADD16
; Desc: Adds two 16-bit numbers and generates a 24-bit number
; where the high byte of the result contains the carry
; out bit.
;-----------------------------------------------------------
ADD16:
; Execute the function here
; Load beginning address of first operand into X
ldi XL, low(ADD16_OP1) ; Load low byte of address
ldi XH, high(ADD16_OP1) ; Load high byte of address

; Load beginning address of second operand into Y
ldi YL, low(ADD16_OP2) ; Load low byte of address
ldi YH, high(ADD16_OP2) ; Load high byte of address

; Load beginning address of result into Z
ldi ZL, low(ADD16_Result) ; Load low byte of address
ldi ZH, high(ADD16_Result) ; Load high byte of address

ld A, X+ ; Load X into A and post inc X
ld B, Y+ ; Load Y into B and post inc Y
add B, A ; Add A and B together and store in A
st Z+, B ; Store indirect B into Z and post inc
ld A, X ; Load X into A
ld B, Y ; Load Y into B
adc B, A ; Add A and B together with carry and store into B
```

```
st Z+, B ; Store B into Z and post inc Z

brcc ADD_EXIT ; Branch if the carry bit is cleared.
st Z, XH

ADD_EXIT:
ret ; End a function with RET

;-----------------------------------------------------------
; Func: SUB16
; Desc: Subtracts two 16-bit numbers and generates a 16-bit
; result.
;-----------------------------------------------------------
SUB16:
; Execute the function here
; Load beginning address of first operand into X
ldi XL, low(SUB16_OP1) ; Load low byte of address
ldi XH, high(SUB16_OP1) ; Load high byte of address

; Load beginning address of second operand into Y
ldi YL, low(SUB16_OP2) ; Load low byte of address
ldi YH, high(SUB16_OP2) ; Load high byte of address

; Load beginning address of result into Z
ldi ZL, low(SUB16_Result) ; Load low byte of address
ldi ZH, high(SUB16_Result) ; Load high byte of address

ld A, X+ ; Load X into A and post inc X
ld B, Y+ ; Load Y into B and post inc Y
sub A, B ; Subtract A - B and store into A
st Z+, A ; Store A into Z and post inc z
ld A, X ; Load X into A
ld B, Y ; Load Y into B
sbc A, B ; Subtract with carry A - B - C and store into A
st Z+, A ; Store indirect A into Z and post inc Z

brcc SUB16_EXIT ; Branch if carry is cleared
st Z, XH

SUB16_EXIT:
```

```
ret ; End a function with RET

;-----------------------------------------------------------
; Func: MUL24
; Desc: Multiplies two 24-bit numbers and generates a 48-bit
; result.
;-----------------------------------------------------------
MUL24:
; Execute the function here
push XH ; Save X-ptr
push XL
push YH ; Save Y-ptr
push YL
push ZH ; Save Z-ptr
push ZL
push oloop ; Save counters
push iloop


clr zero ; Maintain zero semantics

; Set X to beginning address of A
ldi XL, low(addrA_24) ; Load low byte of address
ldi XH, high(addrA_24) ; Load high byte of address

; Set Y to beginning address of B
ldi YL, low(addrB_24) ; Load low byte of address
ldi YH, high(addrB_24) ; Load high byte of address

; Set Z to begginning address of resulting Product
ldi ZL, low(LAddrP_48) ; Load low byte of address
ldi ZH, high(LAddrP_48) ; Load high byte of address

ld A, X+ ; Load in all of our 24-bit operands into individual 8-bit registers piecewise
ld Amd, X+
ld Ahi, X
ld B, Y+
ld Bmd, Y+
ld Bhi, Y
```

```
ldi iloop, $00 ; Initialize the loop counter to zero
ldi ZL, $5C ; Put the Z pointer to the high bits

st -Z, B ; Put B out on the low side of the product
st -Z, Bmd
st -Z, Bhi
ldi ZL, $5C

MAYBE_ROLL:
cpi B, $01 ; Check to see if my Overflow-bit is set
breq ROLL_IT ; If it is, then go roll it in, Else, keep going

MAYBE_ADD:
brts ADD_A ; If the carry bit from shifting the first byte was set
; Then we need to add A<<24

; Shifting the first byte we keep track of the carry by setting the T-flag
FIRST_BYTE:
ld mpr, -Z ; Load get the value Z is pointing to
lsr mpr ; Shift the whole byte
st Z, mpr ; Update the byte value
brcs SET_T ; If there was a carry, we need to add A<<24

CARRY_BRANCH:
cpi ZL, $56 ; Make sure we dont change the operands
breq MAYBE_ROLL ; Roll in the carry bit to the previous byte
ld mpr, -Z ; Get the next byte
lsr mpr ; Shift it
brcc NO_CARRY ; If no carry bit rolled through
adiw Z, $01 ; Else point Z back at the previous byte
ld oloop, Z ; Load the previous byte
sbr oloop, $80 ; Roll in the Carry bit
st Z, oloop ; Update the byte value
sbiw Z, $01 ; Put Z back to the current data location
st Z, mpr ; Update the current shifted value
rjmp CARRY_BRANCH ; Shift the next byte (if there is one)

NO_CARRY:
st Z, mpr ; Just update the current data at Z address
rjmp CARRY_BRANCH ; Shift the next byte (if there is one)
```

```
ROLL_IT:
sbr mpr, $80 ; Roll in the carry bit
st Z, mpr ; Update the value
cpi iloop, $18 ; Check to see if we have finished
breq RETURN ; We_Done
jmp MAYBE_ADD ; Go check to see if we should jump into ADD_A

ADD_A:
inc iloop ; Increment iloop
ldi ZL, $59 ; Load immediate $59 into ZL
ld mpr, -Z ; Post inc Z and store into mpr
add mpr, A ; Add A and mpr and store into mpr
st Z, mpr ; Store indirect from mpr into Z
ld mpr, -Z ; Pre decrement Z and store into mpr
adc mpr, Amd ; Add with carry take mpr + Amd + C
st Z, mpr ; Store indirect mpr and store into Z
ld mpr, -Z ; Post decrement Z and store into mpr
adc mpr, Ahi ; Add with carry add mpr and Ahi together with carry
st Z, mpr ; Store indirect mpr with Z and store into Z
ldi ZL, $5C ; Load immiedate value of $5C into ZL
clt ; Clear the T bit in sreg
clr B ; Clear all the bits in B by preforming the XOR operation
brcs OVERFLOW ; Branch if the carry bit in sreg is set to OVERFLOW
jmp FIRST_BYTE ; Jump to the FIRST_BYTE

SET_T:
SET
jmp CARRY_BRANCH ; Jump to the CARRY_BRANCH
OVERFLOW:
ldi B, $01 ; Load immediate $01 and store into B
jmp FIRST_BYTE



RETURN:

/*********************************************************
DO NOT FORGET TO SWITCH TO LITTLE ENDIAN FORMAT!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*********************************************************/
```

```
pop iloop ; Restore all registers in reverves order
pop oloop
pop ZL
pop ZH
pop YL
pop YH
pop XL
pop XH


ret ; End a function with RET

;-----------------------------------------------------------
; Func: COMPOUND
; Desc: Computes the compound expression ((D - E) + F)^2
; by making use of SUB16, ADD16, and MUL24.
;
; D, E, and F are declared in program memory, and must
; be moved into data memory for use as input operands.
;
; All result bytes should be cleared before beginning.
;-----------------------------------------------------------
COMPOUND:

; This operation clears all the data in Data IRAM to clear all out data
; Before preforming the Compound operation
push mpr
ldi mpr, 100
ldi XL, low($0100)
ldi XH, high($0100)
CLEARFUNC:
push mpr
ldi mpr, $00
st X+, mpr
pop mpr
dec mpr
brne CLEARFUNC
pop mpr
```

```
; Setup SUB16 with operands D and E
; Perform subtraction to calculate D - E
ldi XL, low(SUB16_OP1) ; Load operands
ldi XH, high(SUB16_OP1)
ldi ZL, low(OperandD<<1)
ldi ZH, high(OperandD<<1)

lpm mpr, Z+
st X+, mpr
lpm mpr, Z
st X, mpr

ldi YL, low(SUB16_OP2) ; Load operands
ldi YH, high(SUB16_OP2)
ldi ZL, low(OperandE<<1)
ldi ZH, high(OperandE<<1)

lpm mpr, Z+
st Y+, mpr
lpm mpr, Z
st Y, mpr

rcall SUB16

; Setup the ADD16 function with SUB16 result and operand F
; Perform addition next to calculate (D - E) + F
ldi XL, low(ADD16_OP1) ; Load operands
ldi XH, high(ADD16_OP1)
ldi ZL, low(SUB16_Result)
ldi ZH, high(SUB16_Result)

ld mpr, Z+
st X+, mpr
ld mpr, Z
st X, mpr

ldi YL, low(ADD16_OP2) ; Load operands
ldi YH, high(ADD16_OP2)
ldi ZL, low(OperandF<<1)
ldi ZH, high(OperandF<<1)

lpm mpr, Z+
st Y+, mpr
lpm mpr, Z
st Y, mpr

rcall ADD16

; Setup the MUL24 function with ADD16 result as both operands
; Perform multiplication to calculate ((D - E) + F)^2
ldi XL, low(addrA_24) ; Load operands
ldi XH, high(addrA_24)
ldi YL, low(addrB_24)
ldi YH, high(addrB_24)
ldi ZL, low(ADD16_Result)
ldi ZH, high(ADD16_Result)

ld mpr, Z+
st X+, mpr
st Y+, mpr
ld mpr, Z+
st X+, mpr
st Y+, mpr
ld mpr, Z
st X, mpr
st Y, mpr

rcall MUL24A


ret ; End a function with RET

;-----------------------------------------------------------
; Func: MUL24-A
; Desc: An example function that multiplies two 16-bit numbers
; A - Operand A is gathered from address $0101:$0100
; B - Operand B is gathered from address $0103:$0102
; Res - Result is stored in address
; $0107:$0106:$0105:$0104
; You will need to make sure that Res is cleared before
; calling this function.
;-----------------------------------------------------------
MUL24A:
push  A ; Save A register
push B ; Save B register
push rhi ; Save rhi register
push rlo ; Save rlo register
push zero ; Save zero register
push XH ; Save X-ptr
push XL
push YH ; Save Y-ptr
push YL
push ZH ; Save Z-ptr
push ZL
push oloop ; Save counters
push iloop

clr zero ; Maintain zero semantics

; Set Y to beginning address of B
ldi YL, low(addrA_24) ; Load low byte
ldi YH, high(addrA_24) ; Load high byte

; Set Z to begginning address of resulting Product
ldi ZL, low(LAddrP_48) ; Load low byte
ldi ZH, high(LAddrP_48); Load high byte

; Begin outer for loop
ldi oloop, 2 ; Load counter
MUL16_OLOOPA:
; Set X to beginning address of A
ldi XL, low(addrB_24) ; Load low byte
ldi XH, high(addrB_24) ; Load high byte

; Begin inner for loop
ldi iloop, 2 ; Load counter
MUL16_ILOOPA:
ld A, X+ ; Get byte of A operand
ld B, Y ; Get byte of B operand
mul A,B ; Multiply A and B
ld A, Z+ ; Get a result byte from memory
ld B, Z+ ; Get the next result byte from memory
add rlo, A ; rlo <= rlo + A
adc rhi, B ; rhi <= rhi + B + carry
ld A, Z ; Get a third byte from the result
adc A, zero ; Add carry to A
st Z, A ; Store third byte to memory
st -Z, rhi ; Store second byte to memory
st -Z, rlo ; Store first byte to memory
adiw ZH:ZL, 1 ; Z <= Z + 1
dec iloop ; Decrement counter
brne MUL16_ILOOPA ; Loop if iLoop != 0
; End inner for loop

sbiw ZH:ZL, 1 ; Z <= Z - 1
adiw YH:YL, 1 ; Y <= Y + 1
dec oloop ; Decrement counter
brne MUL16_OLOOPA ; Loop if oLoop != 0
; End outer for loop

pop iloop ; Restore all registers in reverves order
pop oloop
pop ZL
pop ZH
pop YL
pop YH
pop XL
pop XH
pop zero
pop rlo
pop rhi
pop B
pop A
ret ; End a function with RET

;-----------------------------------------------------------
; Func: MUL16
; Desc: An example function that multiplies two 16-bit numbers
; A - Operand A is gathered from address $0101:$0100
; B - Operand B is gathered from address $0103:$0102
; Res - Result is stored in address
; $0107:$0106:$0105:$0104
; You will need to make sure that Res is cleared before
; calling this function.
;-------------------------------------------------------------
MUL16:
push  A ; Save A register
push B ; Save B register
push rhi ; Save rhi register
push rlo ; Save rlo register
push zero ; Save zero register
push XH ; Save X-ptr
push XL
push YH ; Save Y-ptr
push YL
push ZH ; Save Z-ptr
push ZL
push oloop ; Save counters
push iloop

clr zero ; Maintain zero semantics

; Set Y to beginning address of B
ldi YL, low(addrB) ; Load low byte
ldi YH, high(addrB) ; Load high byte

; Set Z to begginning address of resulting Product
ldi ZL, low(LAddrP) ; Load low byte
ldi ZH, high(LAddrP); Load high byte

; Begin outer for loop
ldi oloop, 2 ; Load counter
MUL16_OLOOP:
; Set X to beginning address of A
ldi XL, low(addrA) ; Load low byte
ldi XH, high(addrA) ; Load high byte

; Begin inner for loop
```

```
ldi iloop, 2 ; Load counter
MUL16_ILOOP:
ld A, X+ ; Get byte of A operand
ld B, Y ; Get byte of B operand
mul A,B ; Multiply A and B
ld A, Z+ ; Get a result byte from memory
ld B, Z+ ; Get the next result byte from memory
add rlo, A ; rlo <= rlo + A
adc rhi, B ; rhi <= rhi + B + carry
ld A, Z ; Get a third byte from the result
adc A, zero ; Add carry to A
st Z, A ; Store third byte to memory
st -Z, rhi ; Store second byte to memory
st -Z, rlo ; Store first byte to memory
adiw ZH:ZL, 1 ; Z <= Z + 1
dec iloop ; Decrement counter
brne MUL16_ILOOP ; Loop if iLoop != 0
; End inner for loop

sbiw ZH:ZL, 1 ; Z <= Z - 1
adiw YH:YL, 1 ; Y <= Y + 1
dec oloop ; Decrement counter
brne MUL16_OLOOP ; Loop if oLoop != 0
; End outer for loop

pop iloop ; Restore all registers in reverves order
pop oloop
pop ZL
pop ZH
pop YL
pop YH
pop XL
pop XH
pop zero
pop rlo
pop rhi
pop B
pop A
ret ; End a function with RET
```

```
;-----------------------------------------------------------
; Func: Template function header
; Desc: Cut and paste this and fill in the info at the
; beginning of your functions
;-----------------------------------------------------------
FUNC: ; Begin a function with a label
; Save variable by pushing them to the stack

; Execute the function here

; Restore variable by popping them from the stack in reverse order
ret ; End a function with RET


;***********************************************************
;* Stored Program Data
;***********************************************************

; Enter any stored data you might need here

; ADD16 operands
OperandA:
.DW 0xFFFF ; test value for operand A

; SUB16 operands
OperandB:
.DW 0xFCB9 ; test value for operand B
OperandC:
.DW 0xE420 ; test value for operand C

; MUL24 operands
OperandG:
.DW 0xFF ; test value for operand G

; Compoud operands
OperandD:
.DW 0xFCBA ; test value for operand D
OperandE:
.DW 0x2019 ; test value for operand E
OperandF:
```

```
.DW 0x21BB ; test value for operand F

;***********************************************************
;* Data Memory Allocation
;***********************************************************

.dseg
.org $0100 ; data memory allocation for MUL16 example
addrA: .byte 2
addrB: .byte 2
LAddrP: .byte 4

; Below is an example of data memory allocation for ADD16.
; Consider using something similar for SUB16 and MUL24.

.org $0110 ; data memory allocation for operands
ADD16_OP1: .byte 2 ; allocate two bytes for first operand of ADD16
ADD16_OP2: .byte 2 ; allocate two bytes for second operand of ADD16

.org $0120 ; data memory allocation for results
ADD16_Result: .byte 3 ; allocate three bytes for ADD16 result

.org $0130 ; data memory allocation for operands
SUB16_OP1: .byte 2 ; allocate two bytes for first operand of ADD16
SUB16_OP2: .byte 2 ; allocate two bytes for second operand of ADD16

.org $0140 ; data memory allocation for results
SUB16_Result: .byte 2 ; allocate three bytes for ADD16 result

.org $0150 ; data memory allocation for MUL24
addrA_24: .byte 3 ; starts at address $0150
addrB_24: .byte 3 ; starts at address $0153
LAddrP_48: .byte 6 ; starts at address $0156

;***********************************************************
;* Additional Program Includes
;***********************************************************
; There are no additional file includes for this program
Bradley_Heenk_and_Aaron_Vaughan_Lab5_challengecode.asm
Displaying Bradley_Heenk_and_Aaron_Vaughan_Lab5_challengecode.asm.
```