ECE 375 LAB 5

Large Number Arithmetic

Lab Time: Thursday 1000-1200

Eric Prather

PRELAB QUESTIONS

1. *For this lab, you will be asked to perform arithmetic operations on numbers that are larger than 8 bits. To be successful at this, you will need to understand and utilize many of the various arithmetic operations supported by the AVR 8-bit instruction set. List and describe all of the addition, subtraction, and multiplication instructions (i.e. ADC, SUBI, FMUL, etc.) available in AVR's 8-bit instruction set.*

- ADC: Add with carry
- ADD: Add without carry
- ADIW: Add immediate to word
- INC: Increment
- FMUL: Fractional multiply unsigned
- FMULS: Fractional multiply signed
- FMULSU: Fractional multiply signed with unsigned
- MUL: Multiply Unsigned
- MULS: Multiply signed
- LSL: Multiply by 2 (logical shift left)
- MULSU: Multiply signed with unsigned
- SBC: Subtract with carry
- SBCI: Subtract immediate with carry
- SBIW: Subtract immediate from word
- SUB: Subtract without carry
- SUBI: Subtract immediate

Fractional multiplication uses any radix.

2. Write pseudocode for an 8-bit AVR function that will take two 16-bit numbers (from data memory addresses $0111:$0110 and $0121:$0120), add them together, and then store the 16-bit result (in data memory addresses $0101:$0100). (Note: The syntax "$0111:$0110" is meant to specify that the function will expect little-endian data, where the highest byte of a multi-byte value is stored in the highest address of its range of addresses.)

Here is the pseudocode:

```
ADD16_PREDEFINED_ADDRESSES: ;uses r0, r1, Z, and mpr
        Z = $0110
        R0 = *(Z++)
        R1 = *(Z)
        Z = $0120
        mpr = *(Z++)
        r31 = *(Z--) ; saving space
        r30 = mpr ; for clarity, since we aren't using z to read anymore
        add r1 r31 // We start with the least significant (highest) byte.
        adc r0 r30 // Also adds in the previous carry bit.
        Z = $0100
        *(Z++) = r0
        *(Z++) = r1
```

3. Write pseudocode for an 8-bit AVR function that will take the 16-bit number in $0111:$0110, subtract it from the 16-bit number in $0121:$0120, and then store the 16-bit result into $0101:$0100.

```
SUB16_PREDEFINED_ADDRESSES: ;uses r0, r1, Z, and mpr
        Z = $0120
        R0 = *(Z++)
```

```
R1 = *(Z)
Z = $0110
mpr = *(Z++)
r31 = *(Z--) ; saving space
r30 = mpr ; for clarity, since we aren't using z to read anymore
sub r1 r31 // We start with the least significant (highest) byte.
subc r0 r30 // Also adds in the previous carry bit.
Z = $0100
*(Z++) = r0
*(Z++) = r1
```

INTRODUCTION

This lab was a detailed exercise in the more verbose, complex usage of AVR arithmetic instructions. There were several simple multi-byte register arithmetic operations to be implemented, which requires a creative use of registers. These were subtraction, addition, and multiplication. This lab coalesces in the combined execution of all three operations in sequence, piping output from some to inputs of others. Overall, this lab tests a firm understanding of number systems just as much as it tests a robust comprehension of AVR assembly instructions.

Just like in the previous lab, this one emphasizes memory organization and data management between the various spaces AVR constants live in. However, this time students are allowed to define their own space in data, a unique conventional exercise.

This lab is special because it will never be deployed to the ATMega128 boards. Instead, it will be run entirely in the simulator and evaluated by hand using manual inspections of the data window output.

PROGRAM OVERVIEW

This person first runs the initialization routine, then the main routine. After that, it loops infinitely at the end. It does not perform any I/O, and is meant to be run entirely in the simulator.

All sub-methods, SUB16, ADD16, MUL24, and COMP, all perform their operations on the data memory (IRAM) stored at constant-defined addresses assigned by the assembler. They do not preserve the original values of the registers, thereby saving clock cycles.

INITIALIZATION ROUTINE (INIT)

Nothing particularly interesting happens during the initialization routine of this program. Two important behaviors occur:

1. The stack pointer is initialized. Doing this is important for the correct operation of the "rcall" instruction.
2. The "zero" register is cleared. While this shouldn't explicitly be necessary, as registers should not have a value by default, it is done here for the sake of best practice.

MAIN ROUTINE

The main routine calls each of the arithmetical subroutines, in order. Throughout it are scattered a variety of "nop" instructions decorated with comments instructing the grader to add breakpoints to them. As such, the main function of the "main" routine is to provide a variety of "finished" states for various points in the system that can be observed in the data memory (IRAM) window of the debugger.

The main routine additionally provides some test operands automatically for these methods to accelerate the grading process (therefore, they do not have to be inputted manually by the grader for each run of the simulation).

SUB16

Subtracts two 16 bit numbers from each other. Specifically, the second provided number is subtracted from the first.

Input: *SUB16_OP1, *SUB16_OP2

Output: *SUB16_RESULT

ADD16

Adds two 16 bit numbers together.

Input: *ADD16_OP1, *ADD16_OP2

Output: *ADD16_RESULT

MUL24

Multiplies two 24-bit numbers together using the **shift and add** strategy, relying especially on the "rotate right through carry" command. This conforms to the challenge requirements.

IMPORTANT: Unlike other operations, the operand addresses represent **three and four** byte contiguous segments of memory, rather than **two and three**.

Input: *MUL24_OP1, *MUL24_OP2

Output: *MUL24_RESULT

COMP

Performs the operation ((D-E)+F)^2 in the following manner:

1. Subtracts E from D
2. Adds F to the difference
3. Performs MUL24 with MUL24_OP1 and MUL24_OP2 both set to
4. Sets output COMP_OUT

Input: COMP_D, COMP_E, and COMP_F

Output: COMP_RESULT

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 known as the “two’s compliment overflow indicator”. It is set whenever an operation between two numbers would result in an overflow **if both numbers being added were being stored in two’s compliment form**. Sometimes, therefore, this flag may be set when a regular overflow does not actually occur, such as when adding high unsigned numbers. Here is a binary example:

0b0111 1111 + 0b0000 0001 = 0b1000 0000 ; This is an overflow from + 127 to -128, because the addition overflew into the sign bit.

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?

The .BYTE directive is very useful from an organizational perspective because it assigns a very specific location in memory to be reserved specifically for a certain purpose by the assembler. In combination with .org, a consecutive series of different .BYTEs can be written and then all appear flushly in distinct, yet contiguous, memory space as addresses. It is less work on the programmer this way as it is nicely automated. This is also useful for future-proofing the assembly code- using this directive reduces the chance that you will accidentally declare two “data spaces” on top of each other when you do not mean to.

DIFFICULTIES

Because of external responsibilities, I could only spend half of the regular lab time for this lab in the proper area, so I had reduced opportunities to interact with and derive support from the TA.

I had difficulty remembering to use the load program memory instruction instead of the load immediate instruction when I defined my operand default values in program memory instead of as constants. Using ldi with an address in program memory threw a very interesting bug which took me a while to figure out; it didn’t crash the program, it just made the operand of the addition *the address* of the program memory the operand was stored in, rather than the operand itself. This was an easy fix once I figured out what was going on.

I spent a lot of time doing the same “load from memory” and “store to memory” code over and over with only slight tweaks. This was tedious and left me room for error because I was typing out more manually. In hindsight, I should have used a macro to load data at predefined addresses from memory to other places in memory- and other stuff like that.

The description of the efficient shift and multiply in the lab directions was very poor and did not cover edge cases. I had to do a lot of trial and error across a variety of edge cases to get it to work. As you can see here, adhering to the provided instructions for 0b0010 * 0b0010 gave me the wrong answer for some time:

```
     0010
     0010
     ----
0000 0010
0000 0001
0000
```

```
---------
0000 0010
0000 0001
0000
---------
0000 0001
0000 0000 c
0010
---------
0010 0001
0001 0000 c
0010
---------
0011 0000
0001 1000

After furhter consideration, I believe I did the manual calculation wrong; here is the correct version:
     0010
     0010
     ----
0000 0010
0000 0001
0000
---------
0000 0001
0000 0000 c
0010
---------
0010 0000
0001 0000
0000
---------
0001 0000
0000 1000
0000
---------
0000 1000
0000 0100
```

After reaching this point, I couldn't find what I was doing wrong. I followed this implementation, but increased the number of bytes in the result to 4 (from 1) and increased the number of loops.

What I ended up doing as a solution, although it was wildly inefficient, was append two further registers to the most significant side of my results registers and shifted my addition two registers towards more significant.

FOR THE LONGEST TIME I didn't realize my carry bit was being cleared by my dec / cpi combination, and it caused me a lot of grief during my looping multiplication (since the carry bit was super critical to what I was doing).

Conclusion

This lab presented our second instance of manually writing assembly code in ECE 375. As such, the operations we were required to implement were even more complex, andfewer support methods were defined. The main focus of this lab was on memory organization, which was left up to us (the students) rather than provided to us. Furthermore, all of our methods are now to be built to work with memory as parameters, rather than with numerical values stored in registers. The specific operations we were required to create, on the other hand, were relatively simple- just arithmetic. Another major milestone achieved in the completion of this lab was the working with automated testing. Since I did the challenge code, I was also taught a very valuable lesson about troubleshooting AVR code and making efficient use of registers.

Source Code

```
;***********************************************************
;*
;*      Prather_Eric_Lab5_sourcecode.asm
;*
;*      This is the program for lab 5 and fulfills the exact
;*  requirements specified by the lab handout pdf. consult
;*  this PDF for further information.
;*
;***********************************************************
;*
;*       Author: Eric Prather (prathere@oregonstate.edu)
;*            (932580666)
;*         Date: February 6th, 2020
;*
;***********************************************************

.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    zero = r2                       ;  Zero  register,  set  to  zero  in  INIT,  useful  for
calculations
.def    A = r3                          ; A variable
.def    B = r4                          ; Another variable

.def    oloop = r17                     ; Outer Loop Counter
.def    iloop = r18                     ; Inner Loop Counter


;***********************************************************
;*      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
                ; Init the 2 stack pointer registers
                ; Initialize Stack Pointer (Code from Lab 1)
                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
                ; Oh, by "two stack pointer registers", you meant the one 16
                ; byte register subdivided into 2 8bit I/O registers for stack.

                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)

                ; Operand Storage Operation
                ldi     XL, low(ADD16_OP1)     ; Load low byte of address
                ldi     XH, high(ADD16_OP1)    ; Load high byte of address
                ldi YL, low(ADD16_OP2)
                ldi YH, high(ADD16_OP2)

                ldi ZL, low(AddLeft<<1)
                ldi ZH, high(AddLeft<<1)
                lpm r17, Z+
                lpm r18, Z
                st X+, r17
                st X, r18

                ldi ZL, low(AddRight<<1)
                ldi ZH, high(AddRight<<1)
                lpm r17, Z+
                lpm r18, Z
                st Y+, r17
                st Y, r18

 nop ; Check load ADD16 operands (Set Break point here #1)
                rcall 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
                ; Operand Storage Operation
                ldi     XL, low(SUB16_OP1)     ; Load low byte of address
                ldi     XH, high(SUB16_OP1)    ; Load high byte of address
                ldi YL, low(SUB16_OP2)
                ldi YH, high(SUB16_OP2)

                ldi ZL, low(SubLeft<<1)
                ldi ZH, high(SubLeft<<1)
                lpm r17, Z+
                lpm r18, Z
                st X+, r17
                st X, r18

                ldi ZL, low(SubRight<<1)
                ldi ZH, high(SubRight<<1)
                lpm r17, Z+
                lpm r18, Z
                st Y+, r17
                st Y, r18

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

 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


                ; Unique behavior required for MUL 24 because we did not use the
.DW
```

```
                    ; assembler directive

                    ; OPERAND 1
                    ldi     XL, low(MUL24_OP1)     ; Load low byte of address
                    ldi     XH, high(MUL24_OP1)    ; Load high byte of address

                    ldi ZL, low(MulLeft<<1)
                    ldi ZH, high(MulLeft<<1)
                    lpm r17, Z+
                    lpm r18, Z+
                    lpm r19, Z

                    st X+, r17
                    st X+, r18
                    st X, r19

                    ; OPERAND 2
                    ldi XL, low(MUL24_OP2)
                    ldi XH, high(MUL24_OP2)

                    ldi ZL, low(MulRight<<1)
                    ldi ZH, high(MulRight<<1)
                    lpm r17, Z+
                    lpm r18, Z+
                    lpm r19, Z

                    st X+, r17
                    st X+, r18
                    st X, r19


      nop ; Check load MUL24 operands (Set Break point here #5)
                    ; Call MUL24 function to test its correctness
                    ; (calculate FFFFFF * FFFFFF)
                    rcall MUL24

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

                    ; Setting up the compound function
                    ;  You  know,  it'd  probably  be  easier  to  just...  Use  1  address
register
                    ldi XL, low(COMP_D)
                    ldi XH, high(COMP_D)
                    ldi ZL, low(OperandD<<1)
                    ldi ZH, high(OperandD<<1)
                    lpm r17, Z+
                    lpm r18, Z
                    st X+, r17
                    st X, r18

                    ldi XL, low(COMP_E)
                    ldi XH, high(COMP_E)
                    ldi ZL, low(OperandE<<1)
                    ldi ZH, high(OperandE<<1)
                    lpm r17, Z+
                    lpm r18, Z
                    st X+, r17
                    st X, r18

                    ldi XL, low(COMP_F)
                    ldi XH, high(COMP_F)
                    ldi ZL, low(OperandF<<1)
                    ldi ZH, high(OperandF<<1)
                    lpm r17, Z+
                    lpm r18, Z
                    st X+, r17
                    st X, r18

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

```
                            rcall COMPOUND

              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:
               ; 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)
               ldi YH, high(ADD16_OP2)

               ; Load beginning address of result into Z
               ldi ZL, low(ADD16_RESULT)
               ldi ZH, high(ADD16_RESULT)

               ; Execute the function
               ; Store operands in registers
               ld r17, X+
               ld r18, X
               ld r19, Y+
               ld r20, Y
               add r17, r19
               adc r18, r20
               ldi mpr, 0b0 ; use mpr for carry overflow
               brcc ADD16_NOCARRY
               ldi mpr, 0b1

ADD16_NOCARRY:
               ; Write value
               st Z+, r17;r0
               st Z+, r18;r1
               st Z, mpr
               ret                                            ; End a function with RET

;-----------------------------------------------------------
; Func: SUB16
; Desc: Subtracts two 16-bit numbers and generates a 16-bit
;               result.
;-----------------------------------------------------------
SUB16:
               ; Loading arguments and result addresses from memory
               ; will be the same for all arithmetic methods
               ; Load beginning address of first operand into X
                              ; 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)
               ldi YH, high(SUB16_OP2)

               ; Load beginning address of result into Z
               ldi ZL, low(SUB16_RESULT)
               ldi ZH, high(SUB16_RESULT)
```

```
                ; Execute the function
                ; Store operands in registers
                ld r17, X+
                ld r18, X
                ld r19, Y+
                ld r20, Y
                sub r17, r19
                sbc r18, r20
                ldi mpr, 0b0 ; use mpr for carry overflow
                brcc SUB16_NOCARRY
                ldi mpr, 0b1

SUB16_NOCARRY:
                ; Write value
                st Z+, r17;r0
                st Z+, r18;r1
                st Z, mpr
                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
                ; IMPORTANT: For challenge code, use shift (chal.)
                ; I am doing the challenge code, so I will use shift.

                ; PART 1: Load all of the operands into registers
                ldi     XL, low(MUL24_OP1)      ; Load low byte of address
                ldi     XH, high(MUL24_OP1)     ; Load high byte of address

                ; Load beginning address of second operand into Y
                ldi YL, low(MUL24_OP2)
                ldi YH, high(MUL24_OP2)

                ; Load beginning address of result into Z
                ldi ZL, low(MUL24_RESULT)
                ldi ZH, high(MUL24_RESULT)

                ; Execute the function
                ; Store operands in registers
                ld r17, X+
                ld r18, X+
                ld r19, X
                ld r20, Y+
                ld r21, Y+
                ld r22, Y

                ; PART 2:
                ; r19:r18:r17 x r22:r21:r20
                ; Here is my reasoning...
                ; Multiply by 2:
                ;     rd = rd << 1
                ; Multiply by 3:
                ;     rd = rd + (rd << 1)
                ; Multiply by 4:
                ;     rd = rd << 2
                ; Multiply by 5:
                ;     rd = rd + (rd << 2)
                ; So on and so forth. So in general, multiply is:
                ; rd = (rd << rr/2) + (rr%2 == 0 ? rd : 0)
                ; So now the trick now is to use loops and logical shifts to get this effect.
                ; However, after reading the lab report for the challenge code, I recognize
                ; that there is a much better way utilizing right-rotate. So I'll try that
                ; instead.

                ldi r23, 25 ; Loops 24 times
                ;clr r24 ; zero
                clr r24
```

```
		clr r25
		clr mpr
		clc ; Clear carry before we go into the main loop

SHIFT_ADD_LOOP:
		; r19:r18:r17 rotate right through carry
		ror mpr
		ror r25
		ror r24 ; Add carry from last addition if appropriate
		ror r19
		ror r18
		ror r17
		brcc SHIFT_ADD_NOCARRY; Branch if carry cleared
		; Add high half of r22:r21:r20

		; No addition on final step
;		cpi r23, 0x01
;		breq NO_ADD_FINAL_STEP
		clc
		add r24, r20
		adc r25, r21
		adc mpr, r22 ; If this results in a carry, it will be shifted in.
		brcc SHIFT_ADD_NOCARRY
		; Special code if carry needs to preserve to ror mpr
		dec r23
		cpi r23, 0x00
		breq MUL24_WRITE_OUTPUT
		bset 0 ; Set carry flag
		brcs SHIFT_ADD_LOOP; branch if carry set (always)

SHIFT_ADD_NOCARRY:
		; Loop guard
		dec r23
		cpi r23, 0x00
		brne SHIFT_ADD_LOOP


		;FINAL STEP

;		brcc MUL24_WRITE_OUTPUT

MUL24_WRITE_OUTPUT:
		st Z+, r17
		st Z+, r18
		st Z+, r19
		st Z+, r24
		st Z+, r25
		st Z, mpr
		ret					; End a function with RET

; SPECIAL CODE AREA for preserving carry bit


;-----------------------------------------------------------
; 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:
		; Arguments: *COMP_D, *COMP_E, and *COMP_F
		; Setup SUB16 with operands D and E
		; Perform subtraction to calculate D - E
		ldi YL, low(SUB16_OP1)
		ldi YH, high(SUB16_OP1)
		ldi XL, low(COMP_D)
		ldi XH, high(COMP_D)
```

```
ld r17, X+
ld r18, X
st Y+, r17
st Y+, r18 ; Y = &SUB16_OP2

ldi XL, low(COMP_E)
ldi XH, high(COMP_E)
ld r17, X+
ld r18, X
st Y+, r17
st Y, r18

rcall SUB16


; Setup the ADD16 function with SUB16 result and operand F
; Perform addition next to calculate (D - E) + F

; ADD16_OP1 = (D-E)
ldi YL, low(ADD16_OP1)
ldi YH, high(ADD16_OP1)
ldi XL, low(SUB16_RESULT)
ldi XH, high(SUB16_RESULT)
ld r17, X+
ld r18, X
st Y+, r17
st Y+, r18 ; Y = &ADD16_OP2

; ADD16_OP2 = F
ldi XL, low(COMP_F)
ldi XH, high(COMP_F)
ld r17, X+
ld r18, X
st Y+, r17
st Y, r18

rcall ADD16

; Setup the MUL24 function with ADD16 result as both operands
; Perform multiplication to calculate ((D - E) + F)^2
ldi YL, low(MUL24_OP1)
ldi YH, high(MUL24_OP1)
ldi XL, low(ADD16_RESULT)
ldi XH, high(ADD16_RESULT)
ld r17, X+
ld r18, X+
ld r19, X

st Y+, r17
st Y+, r18
st Y+, r19 ;   Y = &MUL24_OP2
st Y+, r17
st Y+, r18
st Y, r19

rcall MUL24

; Store output
ldi YL, low(COMP_RESULT)
ldi YH, high(COMP_RESULT)
ldi XL, low(MUL24_RESULT)
ldi XH, high(MUL24_RESULT)
ld r17, X+
ld r18, X+
ld r19, X+
ld r20, X+
ld r21, X+
ld r22, X
st Y+, r17
st Y+, r18
st Y+, r19
```

```
            st Y+, r20
            st Y+, r21
            st Y, r22

            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
; Using the .DW directive instead of .DB enforces endian-ness
; based on the supplied constant.

; ADD16 operands
AddLeft:
	.DW 0xFCBA
AddRight:
	.DW 0xFFFF
; Expected output: 0x0001 FCB9

; SUB16 operands
SubLeft:
	.DW 0xFCB9
SubRight:
	.DW 0xE420
; Expected output: 0x1899

; MUL24 operands
; Note: 3 byte DB padded by 1 byte.
; 0xFFFFFF and 0xFFFFFF are final operands
MulLeft:
	;.DB 0x02, 0x00, 0x00, 0x00
	.DB 0xFF, 0xFF, 0xFF, 0x00 ; least significant: left
MulRight:
    ;.DB 0x02, 0x00, 0x00, 0x00
	.DB 0xFF, 0xFF, 0xFF, 0x00 ; least significant: left

; 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

; SUB16 Memory, like above
.org $0130
SUB16_OP1:
        .byte 2
SUB16_OP2:
        .byte 2
.org $0140
SUB16_RESULT:
        .byte 3

; MUL 24
.org $0150
MUL24_OP1:
        .byte 3
MUL24_OP2:
        .byte 3
.org $0160
MUL24_RESULT:
        .byte 6


; COMP Memory (operands predefined)
.org $0170
COMP_D:
        .byte 2
COMP_E:
        .byte 2
COMP_F:
        .byte 2
.org $0180
COMP_RESULT:
        .byte 8

;***********************************************************
;*      Additional Program Includes
;***********************************************************
; There are no additional file includes for this program
```