ECE 375 LAB 3 - CHALLENGE

AVL Simulation

Lab Time: Monday 12-1:50

Khuong Luu

Trevor Swope

ANSWERS TO CHALLENGE QUESTIONS

1. The FUNCTION subroutine adds its two 16-bit inputs together and return a 16-bit result. I can tell this by reading and stepping (in debugging/simulation mode) through the operations in this subroutine. Specifically, I see this subroutine (1) adds two low byte of its two input and stores the result into the low byte of the result; (2) adds two high bytes of its two inputs along with carry bit of the last add operation and stores the result into the high byte of the (same) result; (3) if there is carry from the adding of two high bytes, then byte right on the left of the high byte (which is higher than the high byte of the result because of AVR little-endian format) is set to $01 and exit, otherwise it does nothing more and exit. These steps convey exactly the way we add to 16 bits input together.

   Detailed description of the operations being performed by the FUNCTION subroutine:

Firstly, the subroutine loads constant values, which are the addresses of low bytes of the 2-bytes inputs and the result, to register X, Y, and Z, respectively.

```
ldi         XL, $00                 ;
ldi         XH, $01                 ; X = $0100
ldi         YL, $02                 ;
ldi         YH, $01                 ; Y = $0102
ldi         ZL, $04                 ;
ldi         ZH, $01                 ; Z = $0104
```

Then, the subroutine loads (indirectly) the actual low bytes of the two inputs to register A and B. The X+ and Y+ are post-increment to prepare for the next load (indirect) operation later on.

```
ld          A, X+                   ; A = M[X] -> A = M[$0100]     ; X=X+1 -> X = $0101
ld          B, Y+                   ; B = M[Y] -> B = M[$0102]     ; Y=Y+1 -> Y = $0103
```

Then, the subroutine adds A to B and accumulates the result to B, then store the result in B to the memory word whose address is the value of register Z:

```
add         B, A                    ; B = B + A
st          Z+, B                   ; M[Z] = B -> M[$0104] = B     ; Z=Z+1 -> Z = $0105
```

Then, the subroutine loads (indirectly) the actual high bytes of the two inputs to register A and B. (Here the current content of A and B are overwritten). There is no post-increment because we don't need to load any more value to A and B:

```
ld          A, X                    ; A = M[X] -> A = M[$0101]
ld          B, Y                    ; B = M[Y] -> B = M[$0103]
```

Then, the subroutine add A, B, and the carry bit of the last add operation together and accumulates the result to B. Afterward, it stores B into the high byte of Z, then post-increment Z to prepare for setting carry bit (if there is any)

```
adc         B, A                    ; B = B + A + (_carry_)
st          Z+, B                   ; M[Z] = B -> M[$0105] = B     ; Z=Z+1 -> Z = $0106
```

Finally, the subroutine checks if the carry bit flag has been set after the last add operation. If the carry bit is zero, then the subroutine exists. Otherwise, the subroutine sets the byte to the left of the high byte (which is higher than the high byte) to $01 to indicate the carry bit and then exists:

```
brcc    EXIT                              ; if (carry=0), branch to EXIT
st              Z, XH                     ; if (carry=1), M[$0106] = $01
```

2. Two 16-bits inputs that cause the "brcc EXIT" branch NOT to be taken and therefore the "st Z, XH" instruction to be executed before the subroutine returns is: $ffff and $ffff

3. The purpose of "st Z, XH" is to set the byte right on the left of the high byte of the result to $01 (at this moment, XH = $01) to reflect the carry bit resulted from the "adc B, A" instruction on two high bytes of the two inputs. We conserve this carry bit so that we can use it later when we need.

SOURCE CODE

```
;***********************************************************
;*
;*      Lab3Sample.asm
;*
;*      This is a sample ASM program, meant to be run only via
;*      simulation. First, four registers are loaded with certain
;*      values. Then, while the simulation is paused, the user
;*      must copy these values into the data memory. Finally, a
;*      function is called, which performs an operation, using
;*      the previously-entered values in memory as input.
;*
;***********************************************************
;*
;*       Author: Taylor Johnson
;*         Date: January 15th, 2016
;*
;***********************************************************

.include "m128def.inc"                  ; Include definition file

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

.def    mpr = r16
.def    i = r17
.def    A = r18
.def    B = r19

;***********************************************************
;*      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
                ldi             mpr, low(RAMEND)        ; initialize Stack Pointer
                out             SPL, mpr
                ldi             mpr, high(RAMEND)
```

```
		out		SPH, mpr

;***********************************************************
;*	Main Program
;***********************************************************
MAIN:
		clr		r0			; *** SET BREAKPOINT HERE *** (#1)
		dec		r0			; initialize r0 value


		clr		r1			; *** SET BREAKPOINT HERE *** (#2)
		ldi		i, $04
LOOP:	lsl		r1				; initialize r1 value
		inc		r1
		lsl		r1
		dec		i
		brne	LOOP				; *** SET BREAKPOINT HERE *** (#3)


		clr		r2			; *** SET BREAKPOINT HERE *** (#4)
		ldi		i, $0F
LOOP2:	inc		r2				; initialize r2 value
		cp		r2, i
		brne	LOOP2				; *** SET BREAKPOINT HERE *** (#5)

							; initialize r3 value
		mov		r3, r2			; *** SET BREAKPOINT HERE *** (#6)

		;		Note: At this point, you need to enter several values
		;		directly into the Data Memory. FUNCTION is written to
		;		expect memory locations $0101:$0100 and $0103:$0102
		;		to represent two 16-bit operands.
		;
		;		So at this point, the contents of r0, r1, r2, and r3
		;		MUST be manually typed into Data Memory locations
		;		$0100, $0101, $0102, and $0103 respectively.

							; call FUNCTION
		rcall	FUNCTION			; *** SET BREAKPOINT HERE *** (#7)

							; infinite loop at end of MAIN
 DONE:	rjmp	DONE

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

;-----------------------------------------------------------
; Func: FUNCTION
; Desc: ???
;-----------------------------------------------------------
FUNCTION:
		ldi		XL, $00				;
		ldi		XH, $01				; X = $0100
		ldi		YL, $02				;
		ldi		YH, $01				; Y = $0102
		ldi		ZL, $04				;
		ldi		ZH, $01				; Z = $0104
		ld		A, X+				; A = M[X] ; X=X+1
		ld		B, Y+				; B = M[Y] ; Y=Y+1
		add		B, A				; B = B + A
		st		Z+, B				; M[Z] = B ; Z=Z+1
		ld		A, X				; A = M[X]
		ld		B, Y				; B = M[Y]
		adc		B, A				; B = B + A + carry
		st		Z+, B				; M[Z] = B ; Z=Z+1
		brcc	EXIT					; if (carry=0), branch to EXIT
		st		Z, XH				; if (carry=1), M[Z] = HIGH(Z) = $01
EXIT:
		ret						; return from rcall
```