**ECE-3226-50:** Lab #3

Control Flow, Arrays, and Strings

Robert Campbell

**Objective:**

The purpose of this experiment is to gain experience using control flow statements in assembly program design with the use of branch and compare instructions. One type of example is using the control flow statements to iterate through arrays and strings.

**Equipment:**

AVR Studio 7.0

**Procedure:**

Part 1a)

A program that first sums four 8-bit numbers, then sums two 16-bit numbers, and finally iterates through an ASCII String.

jmp Init

Init:

Start:

; Summing four 8-bit words

ldi zl, LOW(2\*Byte\_Array)

ldi zh, HIGH(2\*Byte\_Array)

clr r10

clr r11

clr r1

clr r16

Loop1:

lpm r0, z+

add r10, r0

adc r11, r1

subi r16, -1

cpi r16, 4

brlo Loop1

; Summing two 16-bit words

ldi zl, LOW(2\*Word\_Array)

ldi zh, HIGH(2\*Word\_Array)

lpm r24, z+

lpm r25, z+

lpm r0, z+

lpm r1, z+

add r24, r0

adc r25, r1

; Iterating through an ASCII string

ldi zl, LOW(2\*String1)

ldi zh, HIGH(2\*String1)

Loop2:

lpm r16, z+

cpi r16, 0

brne Loop2

End:

rjmp End

;=============

; Declarations

Byte\_Array:

.DB 123,45,67,89 ; a list of four bytes

Word\_Array:

.DW 2137,984, 0 ; a wordwise list of labels

String1:

.DB "This is a text.", 0 ; a list of byte characters

Results: After 8-bit addition: 0x0144, after 16-bit addition: 0x0c31, and the program iterated through the string and stopped.

**Question1\_1:** In this program, what is the purpose of the LPM instruction? For this instruction the second operand is always Z+. Which register(s) does Z constitute, and what is the + for? Why is this program using register Z instead of registers X or Y with the LPM instruction?

The purpose of lpm is to load data from program memory. Z is the only valid index registers for this operator and contains the memory address of the specific program memory. The + indicates that the index will be incremented by one at the end of the instruction.

**Question1\_2:** When using the .DB and .DW (and similar) directives, it is best to place such constant declarations at the end of the program code (after the rjmp End instruction). What happens if you place (cut & paste) them before the rjmp Init instruction? Does the program still work? If so, will it necessarily work with any constant declarations?

It does still work, but it loads values into program memory and it runs the values as machine code. It only worked because it did eventually reach the location of the first program instruction, but only after significantly more clock cycles.

**Question1\_3:** In the first the first part of the program, we are iterating (looping) over an array of four 8-bit unsigned numbers, and saving the sum over all the numbers as a 16-bit value in the register pair R10:R11. When we load an element of the array from program memory, what register is it being stored in?

It is loaded into R0.

**Question1\_4:** What is the purpose of register R1?

It is an empty register so that the carry value can be loaded into the high-byte result.

**Question1\_5:** Using this method, would it be possible to take the sum of an array of signed numbers? Why or why not?

Not as is. It would have to be rewritten to account for sign extension.

**Question1\_6:** This part of the program uses a loop that iterates (loops) four times. What register(s) and instruction(s) are used to enable the loop to iterate (loop) the desired number of times?

subi r16, -1

cpi r16, 4

brlo Loop1

subi r16, -1 is used to increment a counter and cpi r16, 4 is used to compare if the counter has reached the requisite number of iterations. The brlo Loop1 instruction keeps looping until the previous comparison become equal to or greater.

Part 1c)

A program that sums four 8-bit numbers in a different way than the first program.

jmp Init

Init:

Start:

ldi zl, LOW(2\*Byte\_Array)

ldi zh, HIGH(2\*Byte\_Array)

clr r10

clr r11

clr r1

ldi r16, 4

Loop1:

lpm r0, z+

add r10, r0

adc r11, r1

dec r16

brne Loop1

End:

rjmp End

;=============

; Declarations

Byte\_Array:

.DB 123,45,67,89 ; a list of four bytes

Results: The same sum of 0x0144, however this program ran in 40 cycles instead of the previous program’s 44.

Part 1d)

Modify the previous 16-bit adder to add using a loop.

jmp Init

Init:

Start:

; Summing two 16-bit words

ldi zl, LOW(2\*Word\_Array)

ldi zh, HIGH(2\*Word\_Array)

clr r0

clr r1

ldi r16, 2

loop2: lpm r0, z+

lpm r1, z+

add r24, r0

adc r25, r1

dec r16

brne loop2

End:

rjmp End

;=============

; Declarations

Word\_Array:

.DW 2137,984, 0 ; a wordwise list of labels

Results: The same result as the first 16-bit adder, 0xc31, but this program ran slower, using 29 cycles instead of the first 16-bit adder using 16.

**Question1\_7:** The third part of the program iterates over an ASCII string. Which register shows the value of the current ASCII character that in the string being read?

R17

**Question1\_8:** Strings of ASCII characters commonly end in a terminator, which signifies the end of the string. What is the hexadecimal value of the terminator used in this ASCII string? Why is it desirable to use an unprintable character as the string terminator?

0x00. By using an unprintable character, it makes it less likely that the loop will be terminated accidentally.

**Question1\_9:** This part of the program uses a loop to iterate over the string, but the loop iteration condition here is different than used previously. What test is being performed to determine when the loop should stop iterating?

This comparison is testing the register value to "null" or 0x00, and continuing the loop until encountering a null value.

Part 1f)

A program that iterates through all of the characters in a string in a different way than the first program.

jmp Init

Init:

Start:

; Iterating through an ASCII string

ldi zl, LOW(2\*String1)

ldi zh, HIGH(2\*String1)

ldi r17, 0;

Loop2:

Inc r17;

lpm r16, z+

cpi r16, 0

brne Loop2

subi r17, 1;

End:

rjmp End

;=============

; Declarations

String1:

.DB "This is a text.", 0 ; a list of byte characters

Results: The program successfully iterated through the string and ended.

Part 2a)

A program that iterates through the capital letters of the alphabet in order, A through Z, and outputs them to PortB.

jmp Init

Init:

Start:

SER R17;

Out DDRB, r17;

ldi zl, low(2\*char1)

ldi zh, high(2\*char1)

lpm r16, z;

Out Portb, r16

loop1:

inc r16

out portb, r16

cpi r16, 0x5A ; compares with 'Z'

brne loop1

end: rjmp end

Char1:

.DB 'A'

Results: Successfully iterated through all of the characters and output them in the appropriate I/O register.

**Question2\_1:** There are various ways (checks) you could use to terminate your loop for this program. What is an alternate method for terminating the loop for this program?

BRLO would also work, as all capital values prior to 'Z' are numerically lower than 'Z' or 0x5A.

Part 2b)

A program that, given a set of values, sums all of the numbers larger than 100.

jmp Init

Init:

Start:

ldi zl, low(2\*Array)

ldi zh, high(2\*Array)

ldi r18, 17;

clr r0

clr r1

clr r2 ; null register for cpc

loop1:

lpm r16, z+

lpm r17, z+

cpi r16, 101

cpc r17, r2

brlo skipadd; jump to skipadd if less than 100

add r0, r16;

adc r1, r17;

skipadd:

dec r18

cpi r18, 0

brne loop1;

end: rjmp end

Array:

.DW 573, 16, 8, 39, 8192, 483, 1602, 198, 2607, 215, 101, 33, 598, 63, 882, 100, 120

Results: Successfully added only the specified values with a result of 0x3CD3.

**Question2\_2:** Is your final sum what you would expect? State the result.

Yes. 0x3cd3 or decimal 15571.

**Question2\_3:** How would your program need to be different if the values were signed values, such that you ignore values between -100 and 100, otherwise the number is added to the sum? Write the modified program.

Add a second conditional statement. If it fails either being greater than -100 or less than 100, skip adding.

Part 2c)

A program that, given a base value (base), calculates the value of that base raised to a given power (expon).

jmp Init

Init:

Start:

Ldi zl,LOW(2\*base)

Ldi zh, HIGH(2\*base)

lpm r16, z ; base r16

ldi zl, LOW(2\*expon)

ldi zh, HIGH(2\*expon)

lpm r17, z ; expon r17

clr r0

clr r1

clr r18 ; interim low

clr r19 ; interim high

ldi r20, 1 ; result low byte

clr r21 ; result high byte

cpi r17, 0

breq end ; If the exponent is 0, jump to end

loop:

mul r20, r16;

movw r19:r18, r1:r0 ; interim storage for later addition

mul r21, r16

add r19, r0;

movw r21:r20,r19:r18 ; Move to the result bytes

dec r17;

brne loop

end: rjmp end

; Declarations

base: .DB 2; the base

expon: .DB 15; the exponent

Results: All test vectors evaluated correctly.

Part 2d)

A program that tests if two strings are identical. The program sets R0 to 0x01 if they are identical and to 0x00 if they are not.

jmp Init

Init:

Start:

clr r0 ; output register

clr r18 ; Character from string 1

clr r16 ; String 1 index low

clr r17 ; String 1 index high

clr r19 ; Character from String 2

clr r20 ; String 2 index low

clr r21 ; String 2 index high

ldi r16, low(2\*string1)

ldi r17, high(2\*string1)

ldi r20, low(2\*string2)

ldi r21, high(2\*string2)

loop:

movw zh:zl, r17:r16;

lpm R18, z+

movw r17:r16, zh:zl;

movw zh:zl,r21:r20

lpm r19,z+

movw r21:r20, zh:zl;

cp r19,r18

brne end;

cpi r18, 0

brne loop;

inc r0

end: rjmp end

string1: .DB "This is a good book.",0

string2: .DB "This is a good book.",0

Results: The program correctly evaluated the test vectors.

**Discussion/Conclusion:**

The purpose of this experiment was to gain familiarity with flow control instructions, strings, and arrays. Since AVR only supports greater-than-or-equal-to and less-than branch instructions but not less-than-or-equal-to and greater-than, and their equivalent unsigned instructions, I needed to restructure some of the ways that values were compared to fit those instructions. For example, in Part 2b I originally was comparing values against 100 for the branch instruction, however that would add any values equal to 100 into the sum when the requirement was for any values greater than 100. The solution was to compare against 101 instead, since the values were full integers.

I also got more practice and familiarity with the movw instruction, since the power function could not easily work without it. I originally as trying to design the power function as a series of nested for loops with addition to avoid copying values around as much as the final version does, but it didn’t return correct values for all test vectors and took much longer to run.