Control Flow, Arrays, and Strings
Microprocessors Lab
ECE 3226-37

Charlie Coleman
Lab Partner: Amy Guo

October 9, 2017

Objective:

The purpose of this lab is to experiment with control flow in assembly. For example, we used different instructions to compare and branch. Then we used new types like array and string. We also experimented with constants from program memory.

Equipment:

AVR Studio 7

Procedure:

The program was given as part of the hand out. One part of the program adds the elements in an array for four 8-bit unsigned numbers. Then the result is saved as a 16-bit word in the R10: R11 register pair. The second part adds two 16-bit numbers, and stores the result as a 16-bit word. The last part loops over an ASCII string.

```
Start:
    ldi zl, LOW(2*Byte_Array)
    ldi zh, HIGH(2*Byte_Array)

    ; Summing four 8-bit words
    clr r10
    clr r11
    clr r1
    ldi r16, 0x04
Loop1:
    lpm r0, z+

    add r10, r0
    adc r11, r1

    subi r16, 1
    brne 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,  ; a wordwise list of labels

String1:
.DB "This is a text.", 0; a list of byte characters
```

Results for Part 1:

Part	Result
Default	0x0144
Part 1c	0x0144
Part 1d	0x0C31
Part 1f	0x0F

1a)

Question 1.1: 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?

Answer: The LPM instruction is used to load a byte from the program memory. Z is R31:R30, and the + is a post-increment, so after the command executes and the Z register is updated with Z+1. Z is used because the LPM instruction only uses Z.

Question 1.2: 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?

Answer: If the .DB and .DW are placed at the beginning of the program code, the output is the same. If the constant is being saved in the data segment instead of the code segment (.DSEG), you must remember to place a .CSEG directive before the program code starts.

1b)

Question 1.3: 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?

Answer: The element is being stored in R0.

Question 1.4: What is the purpose of register R1?

Answer: In this program, R1 is used for the add with carry command (ADC). ADC is not an immediate command, so both operands must be registers.

Question 1.5: Using this method, would it be possible to take the sum of an array of signed numbers? Why or why not?

Answer: Yes, ADD and ADC both work with signed numbers.

Question 1.6: 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?

Answer: R16 holds the number of loops that the program will execute. R16 is decremented in each loop, and the command BRNE is used to branch until the Z bit in SREG is 1.

1c) In 1c, the given code was modified so that it would iterate in a different fashion.

```
start:
    ldi r16, 0x04
loop1:
    subi r16, 1
    brne Loop1
```

1d) In 1d, the given program was modified so that it used a loop for the 16-bit addition.

```
ldi zl, LOW(2*Word_array)
ldi zh, HIGH(2*Word_array)

clc
clr r0
clr r1
ldi r17, 2
loop2:
    lpm r24, z+
    lpm r25, z+

    add r0, r24
    adc r1, r25

    subi r17, 1
    brne loop2
```

1e)

Question 1.7: The third part of the program iterates over the ASCII string. Which register shows the value of the current ASCII character in the string being read?

Answer: R16 holds the value of the ASCII character.

Question 1.8: 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?

Answer: `0x0` is typically used for the terminator of an ASCII string. An unprintable character is used because if the code is written incorrectly, it will not display the terminator at the end of the string.

Question 1.9: 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?

Answer: This part of the program uses an immediate comparison between the loaded byte and 0. If the loaded byte is equal to zero, the program stops looping. This is accomplished using the `BRNE` command.

1f) In 1f, the last part of the given program was modified so that it found the length of the string.

```
ldi zl, LOW(2*String1)
ldi zh, HIGH(2*String1)

clr r0
loop3:
    lpm r16, z+
    cpi r16, 0
    breq end
    subi r0, -1
    cpi r16, 0
    brne loop3

end:    rjmp end
```

2a) We created a program to increment through the capital letters of the alphabet to output the corresponding ASCII value for each letter.

```
ldi r16, 0x41
ldi r17, 0x1A
ldi r18, 0xFF
out DDRB, r18
```

```
loop: out PORTB, r16
    subi r16, -1
    subi r17, 1
    brne loop
end: rjmp end
```

Question 2.1: 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?

Answer: In our code, we used a counter to terminate the loop, but alternatively you could use an immediate comparison between the register holding the current letter and the ASCII value of Z, stopping when the letter output is "Z".

2b) This program iterates over the given array of 16-bit unsigned numbers. Its job is to only add the number if the value is greater than 100.

```
clr r0
clr r1
clr r2

ldi zl, low(num_list << 1)
ldi zh, high(num_list << 1)

ldi r16, 0x64
ldi r17, 0x00
ldi r18, 0x10
clr r6

loop1:
    lpm r19, z+
    lpm r20, z+

    cp r16, r19
    cpc r17, r20
    brge pt2

    add r0, r19
    adc r1, r20
    adc r2, r6

pt2: dec r18
    brne loop1
```

```
end: rjmp end

.CSEG
num_list: .DW 573, 16, 8, 39, 9162, 483, 1602, 198,
              3507, 215, 33, 598, 63, 882, 100, 120
```

Question 2.2: Is your final sum what you would expect? State the result.
Answer: The final sum is as expected. We received an output of `0x43BC`.

Question 2.3: How would your program need to be different if the values were signed values, such that you ignore values between -100 and 100, and add the magnitude of the rest of the numbers to the sum? Write the modified program.
Answer: To modify the program, I would include two comparisons, one to check if the data was greater than 100, one to check if the data is less than -100. If neither are the case, skip to the end.

```
clr r0
clr r1
clr r2

ldi zl, low(num_list << 1)
ldi zh, high(num_list << 1)

ldi r16, 0x64
ldi r17, 0x00
ldi r21, 0x9C
ldi r22, 0xFF
ldi r18, 0x10
clr r6

loop1:
    lpm r19, z+
    lpm r20, z+

    cp r16, r19
    cpc r17, r20
    brlt addnum
    cp r21, r19
    cpc r22, r20
    brge addnum
    jmp pt2
```

```
addnum:
    add r0, r19
    adc r1, r20
    adc r2, r6

pt2: dec r18
    brne loop1

end: rjmp end

.CSEG
num_list: .DW 573, 16, 8, 39, 9162, 483, 1602, 198,
              3507, 215, 33, 598, 63, 882, 100, 120
```

2c) This program calculates the powers of unsigned integers. We used the .DB directive to create two 8-bit unsigned integers. Then we loaded them from program memory to calculate the exponent. The answer was sent to R0:R1 to be stored as a 16-bit value.

```
LDI ZL, LOW(Base<<1)
LDI ZH, HIGH(Base<<1)
LPM R16, Z
LDI ZL, LOW(Expon<<1)
LDI ZH, HIGH(Expon<<1)
LPM R17, Z

CLR R19
CLR R20
INC R19
CLR R21
CLR R22
INC R21

LOOP:
    CPI R17, 0
    BREQ END
    MUL R21, R16; result goes to R0:R1
    MOV R19, R0
    MOV R20, R1
    MUL R22, R16
    ADD R20, R0
    MOV R21, R19
    MOV R22, R20 ; Answer stored in R22:R21
```

```
    DEC R17
    BRNE LOOP

END: RJMP END

.CSEG
Base: .DB 0x02
Expon: .DB 0x0F
```

2d) This program compares two strings. When the strings are the same, R0 will be one. If the strings are different, R0 will be zero.

```
.CSEG

CLR R5
CLR R6

Loop:
    LDI ZL, LOW(String1<<1)
    LDI ZH, HIGH(String1<<1)
    ADD ZL, R5
    ADC ZH, R6
    LPM R16, Z+

    LDI ZL, LOW(String2<<1)
    LDI ZH, HIGH(String2<<1)
    ADD ZL, R5
    ADC ZH, R6
    LPM R17, Z+

    INC R5
    CP R16, R17
    BREQ endCheck
    BRNE notEqual

endCheck:
    CPI R16, 0
    BREQ EQUAL
    BRNE Loop

EQUAL:
    CLR R0
    INC R0
    JMP end

notEqual:
```

```
    CLR R0

end:
RJMP end

.CSEG
String1: .DB "This is a good book.", 0
String2: .DB "This is a good book. The other one is better.", 0
```

Discussion/Conclusion:

In this experiment, we learned how to utilize branch functions in AVR Assembly programs. These commands can be used to implement functionality similar to conditional statements in a high level programming language (if, while, for). We also learned to utilize the program memory as a storage space for information. This data can then be loaded into a register within the program. Lastly, we learned to use the `MUL` command to multiply two values, which stores the value to `R0:R1`.

We encountered errors in the lab when our comparisons and branch commands were implemented incorrectly in 2c. This caused the result to be one degree higher than calculated. After changing the code, we were able to correct the error and receive the calculated answer. Another problem we encounter was using the `.DB` directive after a `.DSEG` directive, which does not work. The strings had to be moved after `.CSEG`, and the `LDI` commands had to be edited, multiplying the address of the strings by two.

Overall, this lab was a success as all parts output the calculated values.