**ECE-3226-50:** Lab #4

Data Memory and Procedures

Robert Campbell

**Objective:**

To introduce the use of procedures and manipulating data memory in assembly. Primarily, the use of the stack to preserve information during procedure calls and manipulating data stored in data memory.

**Equipment:**

AVR Studio 7.0

**Procedure:**

**Part 1: Introducing Data Memory**

Description: This program implements bubble sort on data that is loaded from program memory into data memory.

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

; Start of Program

jmp Init ; first line executed

;=========

Init:

; <insert code here to initialize ports, as needed>

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

; Main body of program

Start:

; Copy elements of array (initialized in program memory) to data memory

; Pseudocode:

;

; n = Array\_Size

;

; for (i = 0; i < 6; i++)

; Byte\_Array[i] = Array\_Decl[i];

;

ldi zl, LOW(2\*Array\_Size) ; Z = address of Array\_Size

ldi zh, HIGH(2\*Array\_Size)

lpm r10, z ; r10 = size of array (6 elements/bytes)

ldi zl, LOW(2\*Array\_Decl) ; Z = address of Array\_Decl array

ldi zh, HIGH(2\*Array\_Decl)

ldi yl, LOW(Byte\_Array) ; Y = address of Byte\_Array array

ldi yh, HIGH(Byte\_Array)

mov r16, r10 ; i = n (r16 holds i)

CopyLoop:

lpm r5, z+ ; r5 = Array\_Decl[i]

st y+, r5 ; Byte\_Array[i] = r5

subi r16, 1 ; i--

brne CopyLoop ; if (i != 0) repeat Copy\_Loop

; Perform bubble sort

; Pseudocode:

;

; n = Array\_Size

;

; for (i = n-1; i > 0; i--)

; {

; for (j = 0; j < i; j++)

; {

; if (Byte\_Array[j] > Byte\_Array[j+1])

; {

; // swap bytes

; temp = Byte\_Array[j];

; Byte\_Array[j] = Byte\_Array[j+1];

; Byte\_Array[j+1] = temp;

; }

; }

; }

mov r16, r10 ; i = n (r16 holds i)

subi r16, 1 ; i = i-1

OuterLoop:

cpi r16, 0

breq ExitOuter ; if (i == 0) exit OuterLoop

ldi yl, LOW(Byte\_Array) ; Y = address of Byte\_Array array

ldi yh, HIGH(Byte\_Array)

clr r17 ; j = 0 (r17 holds j)

InnerLoop:

cp r17, r16

breq ExitInner ; if (j >= i) exit InnerLoop

ld r5, y ; r5 = Byte\_Array[j]

ldd r6, y+1 ; r6 = Byte\_Array[j+1]

cp r5, r6

brlo SkipSwap ; if (Byte\_Array[j] < Byte\_Array[j+1])goto

; SkipSwap

st y, r6 ; Byte\_Array[j] = r6 (Byte\_Array[j+1])

std y+1, r5 ; Byte\_Array[j+1] = r5 (Byte\_Array[j])

SkipSwap:

adiw y, 1 ; y++

inc r17 ; j++

rjmp InnerLoop ; repeat InnerLoop

ExitInner:

dec r16 ; i--

rjmp OuterLoop ; repeat OuterLoop

ExitOuter:

End: rjmp End

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

; Declarations

; Constants in Program Memory (can't write/store to Program Memory)

Array\_Decl:

.DB 0, 192, 13, 4, 163, 209

Array\_Size:

.DB 2 \* (Array\_Size - Array\_Decl)

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

; Data Memory

.DSEG

Byte\_Array:

;.DB 0, 192, 13, 4, 163, 209

.BYTE 6

1. **Question1\_1:** What does the .BYTE directive do? How many bytes does this code allocate for the variable denoted by "Byte\_Array:"? What are the initial values of those bytes?

*It reserves 6 bytes for the byte array, initialized with 0x00. It copy's the array values over. 00 c0 0d 04 a3 d1.*

**Question 1\_2:** What value is stored under the “Array\_Size” label? Explain why the  equation computing that value works. If the had more (or less) elements in the “Array\_Decl” array, would the code still work? Would it work for any number of elements?

*The Array\_Size label stores the value 6. The computation works because the labels both store the address values of the stored data, in terms of 2-byte words. The expression simply subtracts the starting point of the array from the endpoint of the array, and then multiplies by two the convert from word values to byte values.*

**Question1\_3:** Recall from the last lab that when accessing data values in program memory, it is necessary in the LDI instruction to multiply the address (of the label) by 2 (e.g. ldi zl, LOW(2\*Array\_Decl)). In contrast, we do NOT need to do that for addresses of variables in data memory (e.g. ldi yl, LOW(Byte\_Array)). Why is that?

*Because Data memory addresses by the byte instead of by the word.*

**Question1\_4**: Just as we did in the last lab, in the first part of this program we're using the LPM instruction to read from program memory. But now we're immediately writing the data to the data memory space. What instruction is being used to store the data into data memory? And more importantly, why don't we simply initialize the values in data memory?

*ST is being used to store the values in data\_memory. Data memory is not initialized in the assembler. Even using the .db function, no values are written to the addresses.*

1. **Question1\_5:** In the second part of this program the bubble sort is implemented in AVR assembly. Whereas only the LPM instruction is available for loading data from program memory, we now see a variety of instructions being used for reading and writing data from/to data memory. What are the two instructions being used to read from data memory? What are the two instructions being used to write to data memory?

*ld & ldd. st & std*

**Question1\_6:** Consider the following instruction sequence from the inner loop of the program:

ld r5, y  
 ldd r6, y+1

For this sequence, why didn't the program use the following sequence instead? What's the difference between the two sequences?

ld r5, y   
 ld r6, y+

*Ld expects a register and an index register with, at most, pre-decrement or post-increment. Ldd allows the for the register and an index register with a numeric shift. In the example with two ld instructions, the same value is being loaded in registers r5 and r6 with the index y only incrementing after loading into r6.*

1. **Question1\_7:** What result(s) are generated if the brlo SkipSwap instruction is replaced by the brlt SkipSwap instruction? Why do these results make sense?

*Replacing BRLO with BRLT will still work, because the result of comparing or subtracting two unsigned numbers will give a negative result. If we were comparing signed numbers, however, the two instructions would not be interchangeable.*

**Part 2: Introducing Procedure Calls and Returns**

Description: A program that determines if a provided value is even or odd through a procedure calls and a loop.

PC:       *CODE:*

|  |  |  |
| --- | --- | --- |
|  | *.nolist* |  |
|  | *.list* |  |
| *0x0000* | *jmp Init* |  |
|  | *Init:* |  |
|  | *SP\_Init:* |  |
| *0x0002* |  | *ldi r16,LOW(RAMEND)* |
| *0x0003* |  | *out spl,r16* |
| *0x0004* |  | *ldi r16,HIGH(RAMEND)* |
| *0x0005* |  | *out sph,r16* |
|  | *Start:* |  |
| *0x0006* |  | *ldi zl, LOW(2\*Array\_Size) ; Z = address of Array\_Size* |
| *0x0007* |  | *ldi zh, HIGH(2\*Array\_Size)* |
| *0x0008* |  | *lpm r10, z ; r10 = size of array (6 elements/bytes)* |
| *0x0009* |  | *ldi zl, LOW(2\*Byte\_Array) ; Z = address of Array\_Decl array* |
| *0x000a* |  | *ldi zh, HIGH(2\*Byte\_Array)* |
| *0x000b* |  | *ldi yl, LOW(Even\_Results) ; Y = address of Byte\_Array array* |
| *0x000c* |  | *ldi yh, HIGH(Even\_Results)* |
| *0x000d* |  | *mov r16, r10 ; i = n (r16 holds i)* |
|  | *CheckLoop:* | |
| *0x000e* |  | *lpm r1, z+ ; r5 = Array\_Decl[i]* |
| *0x000f* |  | *rcall Even\_Odd* |
| *0x0010* |  | *st y+, r0 ; Byte\_Array[i] = r5* |
| *0x0011* |  | *subi r16, 1 ; i--* |
| *0x0012* |  | *brne CheckLoop ; if (i != 0) repeat Copy\_Loop* |
|  | *End:* |  |
| *0x0013* |  | *rjmp End* |
|  | *Even\_Odd:* | |
| *0x0014* |  | *mov r20, r1* |
| *0x0015* |  | *andi r20, 0x01* |
| *0x0016* |  | *dec r20* |
| *0x0017* |  | *andi r20, 0x01* |
| *0x0018* |  | *mov r0, r20* |
| *0x0019* |  | *Ret* |
|  | *;=============* | |
|  | *; Declarations* | |
|  | *; Constants in Program Memory (can't write/store to Program Memory)* | |
|  | *Byte\_Array:* | |
|  |  | *.DB 5, 29, 126, 1, 93, 0, 68, 33* |
|  | *Array\_Size:* | |
|  |  | *.DB 2 \* (Array\_Size - Byte\_Array)* |
|  | *;============* | |
|  | *; Data Memory* | |
|  | *.DSEG* |  |
|  | *Even\_Results:* | |
|  |  | *.BYTE 8* |

Table - Even\_Odd Code and Program Counter

1. **Question2\_1:** Step through the even program using the <F11> key (don't use the <F10> key at present). Monitor the program counter as you are stepping through the code. Print out a copy of this program, and next to each instruction indicate the instruction address for that instruction.

*See Table 1.*

**Question 2\_2:** In particular, what is the address of the first instruction in the "Even\_Odd" procedure? What is the value of the Stack Pointer (SP) register before you

execute the RCALL instruction? Scroll down to the bottom of data memory (Stack memory area) and look at the contents of the last 10 memory locations from the

highest memory address to the next lower memory locations. What is the value of the

Stack Pointer (SP) register after you execute RCALL? Scroll down to the bottom of data memory and look at the contents of the last 10 memory locations from the highest memory address to the next lower memory locations. What has been saved in the stack area? What does it represent?

*The address of the first instruction in Even\_Odd is 0x0014. The stack pointer before executing RCALL is 0x085F, after executing is 0x085D. It is the memory address of the instruction to return to once the process has finished executing.*

**Question2\_3:** Similar to the last question, what is the value of the Stack Pointer (SP) register before you execute the RET instruction? What is it's value after you execute RET? What is the value of the program counter? How does it correspond to the

contents of memory at the address pointed to by the Stack Pointer after you executed RET?

*Before executing RET the value at the stack pointer is 0x085D, and after, the value is 0x085F. The PC is 0x0010, which is equal to  value that was stored on the stack.*

**Question2\_4:** Add the following two lines to the program right after the lpm r10, z instruction:

mov r1, r10

rcall Even\_Odd

Now what value is stored in the top of the stack while you are executing the Even\_Odd procedure?

*The stack contains 0x000B*

1. **Question2\_5:** Reset the program and start executing from the beginning of the program again (the original program, so delete the two lines you added above), but this time use the <F10> key while stepping through the program. How does stepping through program using the <F10> key differ from stepping through the program using the <F11> key?

*F10 executes “step-over” instead of “step-into”, so it executes the entire process at once and shows the user the next line after the process.*

1. **Question2\_6:** When calling a procedure in assembly, you need to know how input arguments must be passed into the procedure, and how return arguments are passed back to the caller by the procedure. In the case of the Even\_Odd procedure, how are the input arguments and return arguments based between the caller and the procedure?

*The input parameters are passed through r1, and the output parameters are passed through r0.*

**Question2\_7:** Another issue that is important with regards to procedures is what registers they use. Change all instances of r20 in the Even\_Odd procedure to r16. Does the code work? Why not? After trying that, add two lines to the Even\_Odd procedure. Add the following line before the first instruction in the procedure:

push r16

And add the following line right before the RET instruction:

pop r16

Now try the code again. It should work now. Why does it work now? Look up the PUSH and POP instruction in the instruction set, and figure out how we're using them to resolve the problem.

*The program will not work if r20 is changed to r16 because r16 is the counter, so it would mess up the loop.*

*It would work after adding the  push and pop instructions because they would preserve the value  of r16 in the stack.*

1. **Question2\_8:** Explain how the Even\_Odd procedure determines whether a number is even or odd. Describe how you could modify the program to return 1 if the number is odd and 0 if the number is even (i.e. reverse of the current method).

*It uses the andi instruction to bitwise And 0x01 with the number. This is done to find out if the lat bit is a 0 (even) or a 1(odd). In order to change the output to be 0 for even numbers and 1 for odd numbers, the lines “dec r20” and “andi r20, 0x01” should be omitted.*

**Part 3: Creating  Procedures.**

Description: A brute force division procedure that takes a dividend and divisor, then iterates through integer values for the quotient until the remainder is less than the provided divisor.

start:

ldi r16, low(RAMEND)

out SPL, r16

Ldi r16, high(RAMEND)

out SPH, r16

clr r16

clr r17

clr r18

clr r19

clr r20 ; remainder

clr r21 ; remainder

clr r22 ; incremental quotient

ldi r16, low(2389) ; numerator lowbyte

ldi r17, high(2389) ; numerator highbyte

ldi r18, 0x1a ; denominator lowbyte

call division

clr r22 ; incremental quotient

ldi r16, low(215) ; numerator lowbyte

ldi r17, high(215) ; numerator highbyte

ldi r18, 5 ; denominator lowbyte

call division

clr r22 ; incremental quotient

ldi r16, low(0) ; numerator lowbyte

ldi r17, high(0) ; numerator highbyte

ldi r18, 9 ; denominator lowbyte

call division

clr r22 ; incremental quotient

ldi r16, low(63) ; numerator lowbyte

ldi r17, high(63) ; numerator highbyte

ldi r18, 68 ; denominator lowbyte

call division

clr r22 ; incremental quotient

ldi r16, low(136) ; numerator lowbyte

ldi r17, high(136) ; numerator highbyte

ldi r18, 52 ; denominator lowbyte

call division

clr r22 ; incremental quotient

ldi r16, low(152) ; numerator lowbyte

ldi r17, high(152) ; numerator highbyte

ldi r18, 1 ; denominator lowbyte

call division

end:

rjmp end

division:

inc r22

mul r22, r18 ; multiplies denom by quot

movw r20, r16

sub r20, r0

sbc r21, r1

cp r20, r18

cpc r21, r19

brsh division

ret



Table - Test Vectors and Cycles to Complete

**Question3\_1:** For each of the calls to your Division procedure, how many cycles did it take to compute the quotient and remainder?

*See table 2.*

**Discussion/Conclusion:**

This experiment focused on the use of procedure calls and use of data memory. The programs themselves functioned as expected, with the division program only needing some tweaking to get to work as needed. One of the things that took extra time in the division program was understanding and deciding which conditional branching to use. It was very easy to test and branch in ways that lead to the code branching in more places than necessary, though I feel we ended up minimizing the number of branching points in the end.

The other portion that took some getting used to was monitoring the specific portion of data memory that contained the stack. It took iterating through a few breakpoints a handful of times to get used to seeing where and when the changes occur, such as with a call, but once we became familiar with it, it became easy to follow.