ECE-3226/CSCI-3451 - Lab #4: Data Memory and Procedures

Contents:

Overview

Purpose: The lab introduces you to using procedures and data memory in assembly. While the last lab took you the first step in this direction, introducing control flow and symbolic constants (data initialized in program memory), this lab provides you with the final tools you need to create full program implementations in assembly (excluding I/O).
In particular, you will learn:

- the purpose of the *program counter*
- the similarities and differences between branches & jumps and procedure calls & returns
- the similarities and differences between accessing data in data memory vs. program memory
- how to read (load) and write (store) values from/to data memory
- how to allocate bytes in the data section
- how to automatically determine array sizes by performing arithmetic on labels
- the purpose of the stack pointer (SP) register and the use of the stack in procedure calls & returns
- how to compute array sizes using labels
- some new assembly instructions:
  - procedure call (RCALL) and return (RET) instructions
  - data memory load (LD, LDD, and LDS) and store (ST, STD, and STS) instructions
- some new assembly directives:
  - the declare data section directive (.DSEG)
  - the allocate bytes directive (.BYTE)

Pre_Lab:

Part3 (Division procedure)

Lab Assignment :

- **Part 1: Introducing Data Memory**

For the first part of the lab, you are given the assembly program, bbl_sort.asm. This is the first program you have encountered that uses the data memory space; all prior programs have exclusively utilized program memory. Create a project for this program in AVR Studio, and begin stepping through to understand what it is doing.

As indicated by the comments, there are essential two parts to this program. This first part copies the values initialized in program memory to data memory. The second part use the Bubble Sort algorithm to sort the elements of the array.

When stepping through this program, you will find it useful to watch the changing state of the data memory. To watch the data memory you have to be in the debug mode, so start debugging. Next, select **Debug → Windows → Memory → Memory1** from the menu bar, and then when the memory window opens, select "**data IRAM**" in the pull-down menu in the upper-left hand corner of the window.

Set breakpoints on the following two instructions and try this out:

```
breq ExitInner
rjmp End
```

Now, answer the following questions about this program:

a. **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? *Note 1: For information on AVR assembler directives, consult the AVR Assembler User Guide. Note 2: You may want to execute the first part of the code at least twice before answering the last question.*

Question1_2: What value is stored under the "Array_Size:" label? Explain why the equation computing that value works. If we had more (or less) elements in the "Array_Decl" array, would the code still work? Would it work for any number of elements?

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?

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? *Note: You may want to try using the .DB and .DW directives in the data memory section (i.e. under .DSEG).*

b. **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?

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+
```

c. **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?

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

In the second part of the lab, you are given the assembly program, Even.asm. This is program will introduce you to procedure calls and returns, and give you additional experience with data memory. Create a project for this program in AVR Studio, and begin stepping through to understand what it is doing.

a. **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.

Question2_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?

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?

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?

b. 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?

c. 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?

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.

d. 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).

- **Part 3: Creating Procedures**

  Problem: *Division* Procedure

  In the final part of the lab, you are to create a program that contains a procedure that performs division. You will use a simple brute-force approach to compute the quotient and remainder. Since division is simply multiplication in reverse, we can take a brute-force approach to division that iteratively tries a number of quotients until it finds the correct one. In other words, consider the linear equation:

  y = q * x + r

Division, and in particular, integer division, corresponds to the case when we are given values for `y` an `x` and want to determine `q` and `r` The value of `y` is our initial value, the **dividend** or **numerator**, and the value of `x` is the **divisor** or **denominator**, the value by which we will be dividing `y`. The result includes `q`, the **quotient**, and `r`, the **remainder**.

We'll assume unsigned division for simplicity. In this case, the remainder, `r`, must (by definition), be less than the value of `x`, the divisor. Consequently, we can determine the quotient and remainder of division of `y` by `x` by taking `x` and iteratively testing it (multiplying it) by possible quotients, starting from 0. In other words, start with `q = 0` and iterate through `q = 0, 1, 2, 3, ...` until we find the smallest `q` such that `r` < `x`, where `r = y - (q * x).`

Create an assembly program with a procedure that performs division of a 16-bit unsigned dividend by an 8-bit unsigned divisor. The result will be an 8-bit unsigned quotient and an 8-bit unsigned remainder. The procedure will need to receive (from the caller) the input values for the dividend and divisor, and return (to the caller) the quotient and remainder.

Have the main body of your program contain a few calls (at least three) to your Division procedure, each called with different values for the dividend and divisor, to appropriately test your procedure.

Question3_1: For each of the calls to your Division procedure, how many cycles did it take to compute the quotient and remainder?
Note: The easiest way to get these cycle counts is to set breakpoints at each procedure call, and then step over the procedure, noting the difference in cycle count before and after stepping over the procedure.

bbl_sort.asm

```
;**********************************************
; written by:
; Updated by: Armineh Khalili           8/27/2014
; date:
; file saved as:
; for AVR:                            ATmega32
; clock frequency:
;**********************************************
; Program Function:   <describe purpose of program here>


; .device      ATmega32               ; don't need because it's in .inc file
below
.nolist
.include       "C:\Program Files\Atmel\AVR
Tools\AvrAssembler2\Appnotes\m32def.inc"
.list
;=================
; 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:
        .BYTE   6
```

Even.asm

```
;**************************************************
; written by:
; Updated by: Armineh Khalili           8/27/2014
; date:
; file saved as:
; for AVR:                            ATmega32
; clock frequency:
;**************************************************
; Program Function:   <describe purpose of program here>


; .device      ATmega32               ; don't need because it's in .inc file
below
.nolist
.include       "C:\Program Files\Atmel\AVR
Tools\AvrAssembler2\Appnotes\m32def.inc"
.list

;=================
; Start of Program

        jmp     Init                          ; first line executed

;=========
Init:
        ; initialize stack pointer
SP_Init:
        ldi            r16,LOW(RAMEND)
        out            spl,r16
        ldi            r16,HIGH(RAMEND)
        out            sph,r16
;=====================
; 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*Byte_Array)  ; Z = address of Array_Decl array
        ldi     zh, HIGH(2*Byte_Array)
        ldi     yl, LOW(Even_Results)  ; Y = address of Byte_Array array
        ldi     yh, HIGH(Even_Results)
        mov     r16, r10               ; i = n  (r16 holds i)
CheckLoop:
        lpm     r1, z+                 ; r5 = Array_Decl[i]
        rcall   Even_Odd
        st      y+, r0               ; Byte_Array[i] = r5
        subi    r16, 1                 ; i--
        brne    CheckLoop              ; if (i != 0) repeat Copy_Loop
End:
        rjmp End


; Procedure:  Even
;    Function:  returns a 1 if number is even, otherwise a 0
;    Inputs:
;               r1    <-  first 8-bit unsigned value
;    Outputs:
;               r0       <-  1 if even, 0 if odd
                ; Pseudocode:
                ;
                ;   if (a & 0x01 == 0x00)
                ;       return 1;
                ;       else
                ;               return 0;
Even_Odd:
        mov     r20, r1
        andi    r20, 0x01
        dec     r20
        andi    r20, 0x01
        mov     r0, r20
        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
```