ECE 375 LAB 4

Writing Assembly Code

Lab Time: Thursday 1000-1200

Eric Prather

PRELAB QUESTIONS

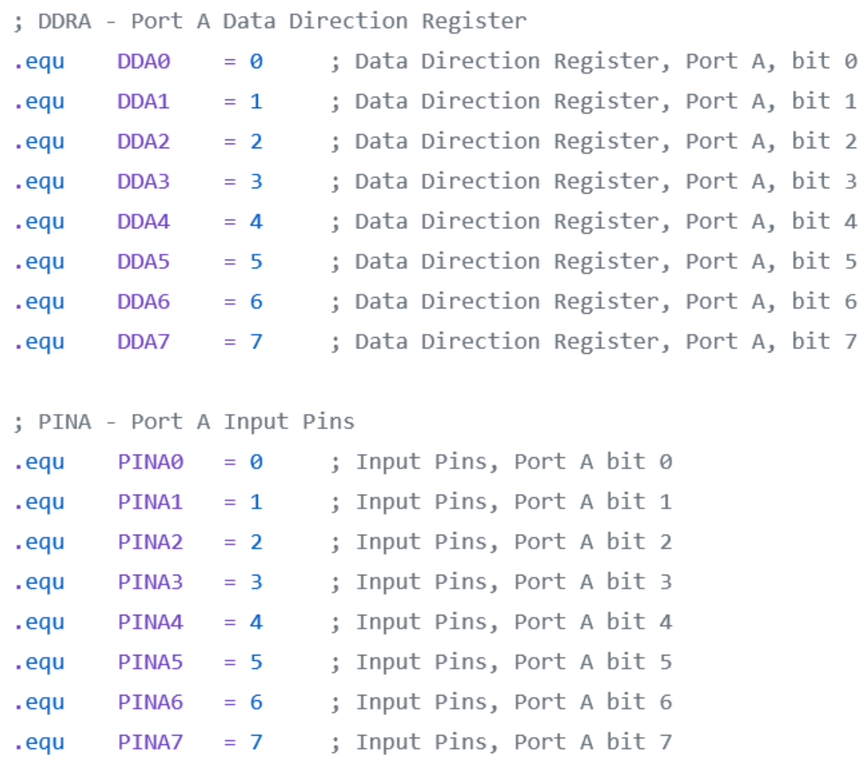
1. The stack pointer is a register that stores a memory address pointing to the location on a first-in first-out (FIFO) region of memory called the "stack". It can be used to keep track of a call stack or for data- although in this lab, its main purpose should be as an augment to the program counter (PC). It is initialized in the SPI_INIT: section of our AVR code to the manual-prescribed memory address (see manual page 13).

   Pseudocode:
   ```
   Initialization Region:
           R16 = High byte of RAMEND // const
           Stack Pointer 1 = R16
           R16 = low byte of RAMEND  // const
           Stack Pointer high = r16
           //Cannot combine instructions
   ```
2. The AVR instruction LPM is short for "Load Program Memory". It loads the value pointed to by Z to the specified register. It is different from the regular load command, which loads from data memory, not program memory. Program memory is organized in 2B words. Here is some pseudocode to demonstrate its operation:

   Pseudocode:

   ```
   r17:16 = loadProgramMemory(Z) // r17:16 = programMemory[Z]
   ```

3. This assembly definition file provides a series of useful macros, similar to the atmel c headers (Discussed in Prather_Eric_Lab2_Report.pdf). For example, the I/O pins are all given names and values here via .equ commands. Some other important features, such as the special registers and special data pointers, are also defined here. This question requires examples of definitions to be provided, so here is a screenshot:



```
; DDRA - Port A Data Direction Register
.equ    DDA0    = 0     ; Data Direction Register, Port A, bit 0
.equ    DDA1    = 1     ; Data Direction Register, Port A, bit 1
.equ    DDA2    = 2     ; Data Direction Register, Port A, bit 2
.equ    DDA3    = 3     ; Data Direction Register, Port A, bit 3
.equ    DDA4    = 4     ; Data Direction Register, Port A, bit 4
.equ    DDA5    = 5     ; Data Direction Register, Port A, bit 5
.equ    DDA6    = 6     ; Data Direction Register, Port A, bit 6
.equ    DDA7    = 7     ; Data Direction Register, Port A, bit 7

; PINA - Port A Input Pins
.equ    PINA0   = 0     ; Input Pins, Port A bit 0
.equ    PINA1   = 1     ; Input Pins, Port A bit 1
.equ    PINA2   = 2     ; Input Pins, Port A bit 2
.equ    PINA3   = 3     ; Input Pins, Port A bit 3
.equ    PINA4   = 4     ; Input Pins, Port A bit 4
.equ    PINA5   = 5     ; Input Pins, Port A bit 5
.equ    PINA6   = 6     ; Input Pins, Port A bit 6
.equ    PINA7   = 7     ; Input Pins, Port A bit 7
```

INTRODUCTION

This project is the first AVR programming task in this course. It requires foundational knowledge regarding AVR instructions.

PROGRAM OVERVIEW

After running one-time initialization, an infinite loop executes that writes “strings” (i.e. bytes[]) to the LCD display. It involves a detailed discussion of memory access, which was not especially pertinent to prior labs. It also provides a stack pointer for function alls.

INIT

The initialization routine provides a one-time initialization of key registers that allow the BumpBot program to execute correctly. First the Stack Pointer is initialized, allowing the proper use of function and subroutine calls. Port B was initialized to all outputs and will be used to direct the motors. Port D was initialized to inputs and will receive the whisker input. Finally, the Move Forward command was sent to Port B to get the TekBot moving forward.

MAIN

Performs three looping branching conditionals, augmented by the subroutines described later:

1. Button 0 pressed: Write “Hello world” to one line and “Eric Prather” to the other of the LCD

2. Button 1 pressed: Inverse of 0.

3. Button 3 pressed: Clear LCD

WRITE_Z_L

Transfers the program data at address Z to the memory at address Y. Uses X as an intermediary data buffer.

READ_D

Rudimentary “read” function for IO pins which was stripped down following a more efficient bitmask representation being discovered.

LCDDRIVER.ASM

The LCDDriver.ASM file contains several subroutines which have already been written and provided as part of lab 4. As such, it is not described in great detail here. Documentation for each question is provided individually. It is also available in this handy set of one-line references in the lab4 pdf:

- `LCDInit` – Initialize the LCD (call once from within `INIT`)
- `LCDWrite` – Update all characters of both lines
- `LCDWrLn1` – Update all characters of line 1 only
- `LCDWrLn2` – Update all characters of line 2 only
- `LCDClr` – Clear (write “space” to) all characters of both lines
- `LCDClrLn1` – Clear all characters of line 1 only
- `LCDClrLn2` – Clear all characters of line 2 only
- `LCDWriteByte` – Write directly to a single character of the LCD
- `Bin2ASCII` – Convert an 8-bit unsigned value into an ASCII string

ADDITIONAL QUESTIONS

1. In this lab, you were required to move data between two memory types: program memory and data memory. Explain the intended uses and key differences of these two memory types.

The ATMega128 has two different main types of memory available on the board with different intrinsic properties, plus a third persistent type that serves as a bootstrapper that can be flashed to by an external programmer. The program memory is organized into two-byte words and has several addresses of importance (such as 0000, the address the program counter begins at). Fetch instructions will usually relate to program memory. Address indexing in program memory is especially unique, as a special operation is required to get the “high” and “low” bytes of various parts of program memory. Usually, the contents of program memory are **processor instructions**. Data memory, on the other hand, has very few rules and is organized intuitively into one-byte chunks. Usually, the content of data memory is **defined and maintained by the program** according to the program instructions (hence data in data memory has no intrinsically assigned meaning). Data *can* be placed in program memory, such as for constant literal values- in fact, we do this with strings in our lab. However, data can only be written to program memory during runtime on “self-programmable” devices.

2. You also learned how to make function calls. Explain how making a function call works (including its connection to the stack), and explain why a RET instruction must be used to return from a function.

A function call simply stores the current instruction address on the stack and moves the program counter to point at a certain address. Then, the “ret” command restores the program counter back to its original position by “popping” this address from the stack. Often times, the values in the registers are also pushed and popped to the stack so that they can be unloaded/reloaded alongside the “subroutine”, but this is not always the case. The system breaks down if the stack pointer is not pointing to the original address at the end of the subroutine, so that is the main thing to keep in mind when fiddling with this.

The specific function of the stack is described in the prelab. However, very briefly, it is a first-in first-out (FIFO) data structure with a special register pair associated with it (the “stack pointer”), generally growing from RAMEND downwards.

3. To help you understand why the stack pointer is important, comment out the stack pointer initialization at the beginning of your program, and then try running the program on your mega128 board and also in the simulator. What behavior do you observe when the stack pointer is never initialized? In detail, explain what happens (or no longer happens) and why it happens.

When the stack pointer is never initialized, its default value is undefined. In this specific hardware, all registers are set to 0 in order to prevent “frying” the board, so that means this will be the original address pointed to by the stack pointer as well. This means that the stack originally points to the **beginning of the program**, and because the stack grows **downwards**, the first time it is decremented the board will cause what is known as a “stack overflow error”. The handling thereof varies by implementation, but the AVR128Mega board will either reboot or wrap to an inaccessible pointer (outside of the range of valid memory addresses, $FFFF) and cause a cascading issue.

DIFFICULTIES

- I accidentally included LCDDriver.asm at the beginning of my main file instead of at the end.
- I tried to use the port B lamp outputs for debugging purposes, but it turns out that a property of the LCD driver prohibited access to Port B. Spent some time with the TA troubleshooting this, but it took up all of my lab section and set me behind somewhat
- I couldn’t quite figure out how to move from program memory to data memory right. I tried to use the debug utility to fix this to no avail. After spending a lot of time on this on my own, I ended up figuring out that if I just ignored the fact that program memory was organized into 16 bit words, it worked just fine. All of the bytes were contirguous anyway, as long as I bitshifted after my first assignment I could increment freely for some reason. Maybe that is an intrinsic part of the LPM command (it bitshifts right, increments, then bitshifts left if it sees Z+?). Actually, I bet that’s the case.
- I was doing a really complicated thing with bit-shifting to try to apply a bit mask when it was much simpler to write andi 0b00000001 and cpi 0b00000001, for example.

CONCLUSION

This lab introduced a variety of fundamental assembly programming language concepts which will be essential in future assignments. It also served to integrate many of the discussions of AVR principles from the lecture of the class with the lab. Understanding the architecture of the ATMega128 board was tremendously helpful in outlining which specific functionalities needed to be implemented in the code. For example, the requirement of the stack pointer initialization or of the interoperability of program and data memory would not have been intuitive with a higher-level programming background alone, where such items are handled by a compiler.

This lab was unique in that it was my first proper assembly program, so I had to put a wide variety of different skills all to the test at the same time.

SOURCE CODE

```
; Eric Prather (prathere 932580666)
; January 27, 2020
;***********************************************************
;*
;*	Prather_Eric_Lab4_sourcecode.asm
;*
;*	This is Lab 4 of ECE 375
;*
;***********************************************************
;*
```

```
;*       Author: Eric Prather (prathere@oregonstate.edu)
;*            (932580666)
;*         Date: January 27, 2020
;*
;***********************************************************

.include "m128def.inc"                ; Include definition file
;.include "LCDDriver.asm"        ; Include Driver

;***********************************************************
;*      Internal Register Definitions and Constants
;***********************************************************
.def    mpr = r16                           ; Multipurpose register is
                                                          ; required for LCD Driver
.def    mpr2 = r23                          ; I need one too!
.def    loop_count = r24                ;
.def    init_loop_count = r25
;***********************************************************
;*      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
               ; Initialize Stack Pointer (Code from Lab 1)
               ldi            mpr, low(RAMEND)
               out            SPL, mpr                ; Load SPL with low byte of RAMEND
               ldi            mpr, high(RAMEND)
               out            SPH, mpr                ; Load SPH with high byte of RAMEND

               ; Initialize LCD Display;
               rcall LCDInit ; In LCDDriver.asm

               ; Initialize Port D for input (Code from Lab 1)
               ; Unclear if this is handled by LCDInit as well,
               ; but I think it's best to be safe and double-check
               ; it all here.
               ldi            mpr, $00                ; Set Port D Data Direction Register
               out            DDRD, mpr               ; for input
               ldi            mpr, $FF                ; Initialize Port D Data Register
               out            PORTD, mpr              ; so all Port D inputs are Tri-State

               ; Move strings from Program Memory to Data Memory
               ; STRING_NAME - STRING_NAME_END
               ; STRING_HW - STRING_HW_END
               ; Haven't figured out how to do this yet.


               ; Do not use the LED outputs because they are reserved
               ; by the LED drivers.

               ; rjump MAIN
               ; NOTE that there is no RET or RJMP from INIT, this
               ; is because the next instruction executed is the
               ; first instruction of the main program

;***********************************************************
;*      Main Program
; $0100 = line 1
; $0110 = line 2
; Y register points to data memory
; Z register points to program memory
;***********************************************************
MAIN:                                               ; The Main program
               rcall LCDClr
               ldi init_loop_count, 6
MAIN_LOOP:
               rcall READ_D ; Used to be more in this function but now it's pretty desolate
               andi mpr2, 0b00000001
               cpi mpr2, 0b00000001 ; only care abt one bit for conditional
               breq SKIP1

               ; Case 1: Pressed 0
               ldi YL, low($0100)
               ldi YH, high($0100)
               ldi ZL, low(STRING_NAME << 1)
               ldi ZH, high(STRING_NAME << 1)
               rcall WRITE_Z_Y
               ldi YL, low($0110)
               ldi YH, high($0110)
               ldi ZL, low(STRING_HW << 1)
               ldi ZH, high(STRING_HW << 1)
               rcall WRITE_Z_Y
               rcall LCDWrite
SKIP1:
               rcall READ_D
               andi mpr2, 0b00000010
               cpi mpr2, 0b00000010
               breq SKIP2

               ; Case 1: Pressed 0
               ldi YL, low($0110) ; note this is swapped from case 1
               ldi YH, high($0110)
               ldi ZL, low(STRING_NAME << 1)
               ldi ZH, high(STRING_NAME << 1)
               rcall WRITE_Z_Y
               ldi YL, low($0100) ; note this is swapped from case 1
               ldi YH, high($0100)
               ldi ZL, low(STRING_HW << 1)
               ldi ZH, high(STRING_HW << 1)
               rcall WRITE_Z_Y
               rcall LCDWrite
SKIP2:
               rcall READ_D
               andi mpr2, 0b10000000
               cpi mpr2, 0b10000000
               breq MAIN_LOOP

               ; Case 3: Pressed 7 (clear)
               rcall LCDClr

               rjmp    MAIN_LOOP             ; jump back to main and create an infinite
                                                            ; while loop.  Generally, every main
program is an
                                                            ; infinite while loop, never let the
main program
                                                            ; just run off

               rjmp    MAIN

;***********************************************************
;*      Functions and Subroutines
;***********************************************************
;-----------------------------------------------------------
; Func: WRITE_Y_Z
; Desc: Writes the contents of Y (in data memory) from Z
; (in program memory). Uses X as a buffer.
;-----------------------------------------------------------
WRITE_Z_Y:
       mov loop_count, init_loop_count
LOOP_WRITE_Z_Y:
       ;lpm r27, Z+ ; XL = r27
```

```
        ;lpm r26, Z+ ; XH = r26
        ;st Y+, r27
        ;st Y+, r26
        ;inc ZL

        ; IMPORTANT: Multiply ALL PROGRAM MEMORY ADRESSES BY TWO when assigning.
        ; Following four lines are ARGUMENTS: they are set by the caller
        ; before responsibility is passed to this function.
        ; ldi start_of_string_low addr ZL
        ; ldi start_of_string_high addr ZH
        ; ldi data_memory_target_low YL
        ; ldi data_memory_target_high YH

        ; Following lines depend on Y and Z being initalized as above
        ; Read 2 byte from program memory, write 2 byte to data memory.
        lpm r26, Z+;
        lpm r27, Z+;
        st Y+, r26;
        st Y+, r27;

        ;inc ZL ; Point to next byte in program memory
        ;inc YL ; Point to next byte in data memory


        ; Loop until word is done.
        cpi loop_count, 0
        dec loop_count
        brne LOOP_WRITE_Z_Y
        ret

;-----------------------------------------------------------
; Func: Read D
; Desc: Puts PIND values as required into mpr2
;-----------------------------------------------------------
READ_D:
               in mpr2, PIND
               ;andi mpr2, (1<<0|1<<1|1<<7) ; set all unused bits to 0?
               ret
;-----------------------------------------------------------
; Func: Template function header
; Desc: Cut and paste this and fill in the info at the
;               beginning of your functions
;-----------------------------------------------------------
FUNC:                                              ; Begin a function with a label
               ; Save variables by pushing them to the stack

               ; Execute the function here

               ; Restore variables by popping them from the stack,
               ; in reverse order

               ret                                        ; End a function with RET

;***********************************************************
;*     Stored Program Data
;***********************************************************

;-----------------------------------------------------------
; An example of storing a string. Note the labels before and
; after the .DB directive; these can help to access the data
;-----------------------------------------------------------
STRING_BEG:
.DB            "My Test String"              ; Declaring data in ProgMem
STRING_END:

STRING_NAME:
.DB            "Eric Prather"
STRING_NAME_END:

STRING_HW:
.DB            "Hello World!"
```

```
WTRING_HW_END:

;***********************************************************
;*      Additional Program Includes
;***********************************************************
.include "LCDDriver.asm"             ; Include the LCD Driver
```