CPE 325: Intro to Embedded Computer Systems

Lab04

Introduction to MSP430 Assembly Language Programming

Submitted by: Michael Agnew

Date of Experiment: 2/2/2024

Report Deadline: 2/6/2024

Demonstration Deadline: 2/12/2024

Introduction:

The main topic behind this lab was programming in Code Composer Studio in the assembly programming language. The main theory behind this lab was Assembly Directives and the various Addressing modes that exist in the assembly programming language. There were two different problems that had to be solved, as well as a bonus problem that was also made available. The main idea behind this lab other than simply programming in assembly, was pulling in characters one at a time from a given string, running tests on them and using them according to the instructions made available in the lab assignment.

Theory:

Assembler Directives: Assembler directives are parts of assembly that are used to help define a segment of code. For instance, when the assembler sees one using the (.string) directive, it allocates room in memory with the correct ASCII characters as well as a NULL character that will allow the assembler to tell when it has reached the end of the string. Another thing that directives are useful for are header files. In the example code that was provided for this lab tutorial, there is a line where a header directive is made which contains macro definitions such as the code that ends the watchdog timer. This is also important for declaring strings where the processor, like explained earlier, will allocate memory for the string to be stored.

Addressing Modes: There are many different addressing modes that are used in assembly. All of these addressing modes are used for a specific purpose and help provide the user with many useful tools for making assembly programs. The different types of addressing modes are: register, indexed, symbolic, absolute, indirect, immediate, and indirect with autoincrement.

Register: This is considered the fastest and shortest addressing mode. It specifies whatever operand is being used as well as what registers are being operated on. The syntax for this addressing mode is simply (Rn).

Indexed: The indexed addressing mode is used when the operand being used is located in memory. Not only this, but its address is being calculated as a sum of the displacement (X) and a specified address register. This displacement of X is made clear in the next instruction word. The syntax for this is [X(Rn)].

Symbolic: This addressing mode is considered to be a subset of the previously discussed indexed addressing mode. The difference between these two modes is that for symbolic, the address register is the program counter (PC) which means that the address of the operand is specified relative to the current value of the program counter (PC). The syntax for this mode is (ADDR).

Absolute Mode: Absolute mode is used when specifying the direct address of a given operand that is located in memory. This instruction is associated with a word which is used to identify the address. The instruction identifies the memory address of the given operand directly, which is combined with the status register that is being used by the first instruction as the address register. The syntax for this mode is (&ADDR).

Indirect Register: This mode of addressing is solely used for source operands. The instruction associated with it identifies the address register (Rn). The syntax for this mode is (@Rn).

Indirect Autoincrement: Indirect autoincrement addressing is used where the address of the given operand that is in memory is used for the identified address register (Rn). The change is when whatever is in register (Rn) is incremented by 2 for word operations but only 1 for byte operations. Syntax for this addressing mode is (@Rn+). This addressing mode could be used when one wants (Rn) to automatically update where it points to the next operand after its operation. This is useful when wanting to pull characters from a string like what was done for the given assignment for this lab. Indirect Autoincrementing is used when storing the next character of the string in the register before checking that character for the desired reasons. After this operation, the autoincrementing moves over 1, (because it is a byte operation) to the next character.

Immediate Mode: Immediate addressing mode is used when addressing an immediate (a constant) which is used as an operand. Syntax for this is (#N).

Problem 01:

The first problem that was assigned was to make an assembly code meant to count the number of digits and the number of capital letters that are in a given string. This was done by checking the range of each character to determine whether or not it was in the ASCII range of capital letters, or digits. If it was not, then it moved on to the next character. After the code executes, the register window is used to view P1OUT and P2OUT which are the output of the number of digits and capital letters respectively. The string used was: "Welcome To MSP430F5529 Assembly Programming!".

P1IN	0xFE	Port 1 Input [Memory Mapped]
P1OUT	0x07	Port 1 Output [Memory Mapped]
P1DIR	0x00	Port 1 Direction [Memory Mappe
P1REN	0x00	Port 1 Resistor Enable [Memory M
P1DS	0x00	Port 1 Drive Strenght [Memory M
P1SEL	0x00	Port 1 Selection [Memory Mappe
P1IV	0x0000	Port 1 Interrupt Vector Word [Me.
P1IES	0x00	Port 1 Interrupt Edge Select [Mem
P1IE	0x00	Port 1 Interrupt Enable [Memory .
P1IFG	0x00	Port 1 Interrupt Flag [Memory Ma
P2IN	0xFF	Port 2 Input [Memory Mapped]
P2OUT	0x08	Port 2 Output [Memory Mapped]

Figure 1. View of Register Window Showing Value of P1IN, P2IN, P1OUT and P2OUT.

The first problem also required a flowchart be made, this was done and is as such:

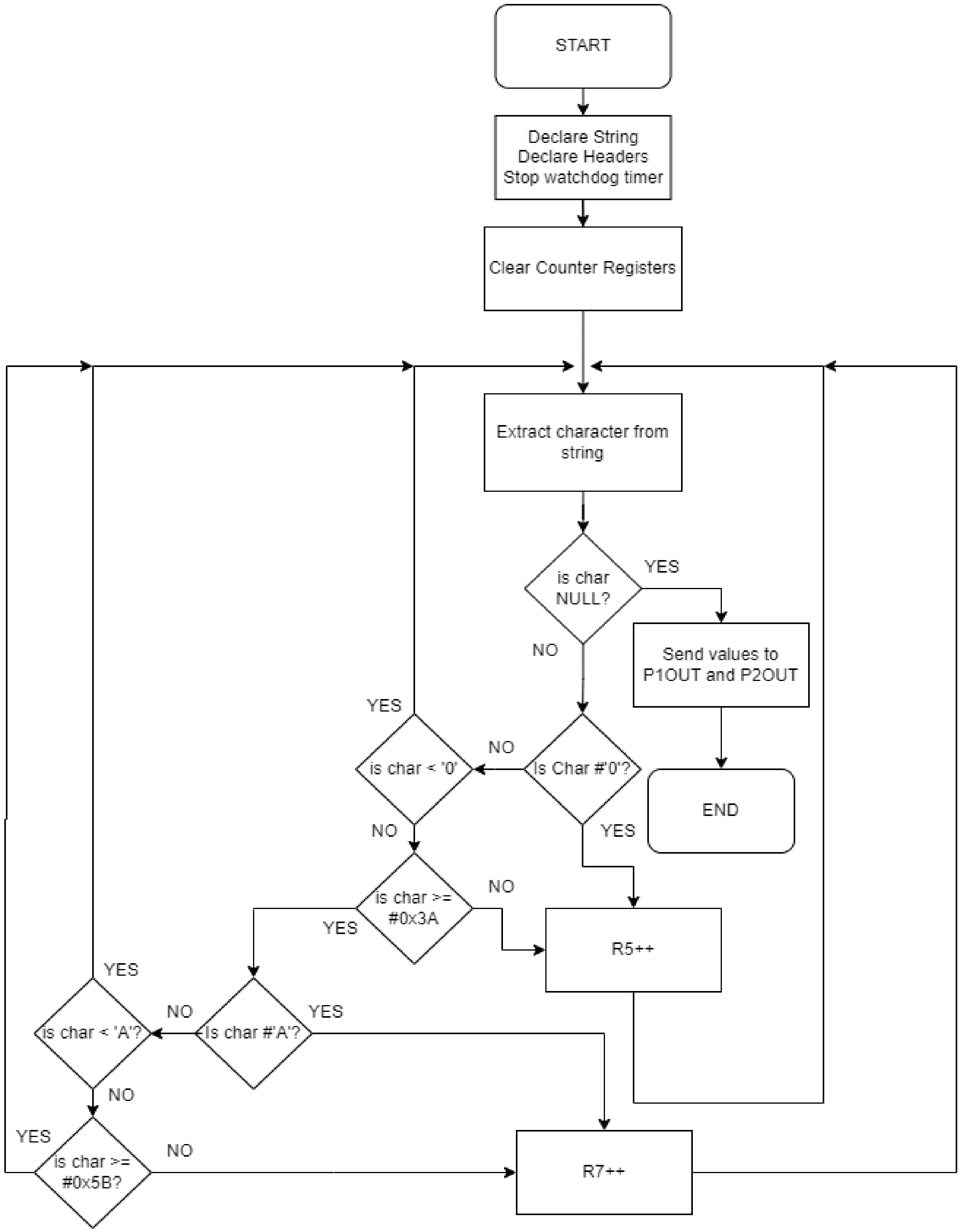


Figure 2. Flowchart for Problem 01.

Problem 02:

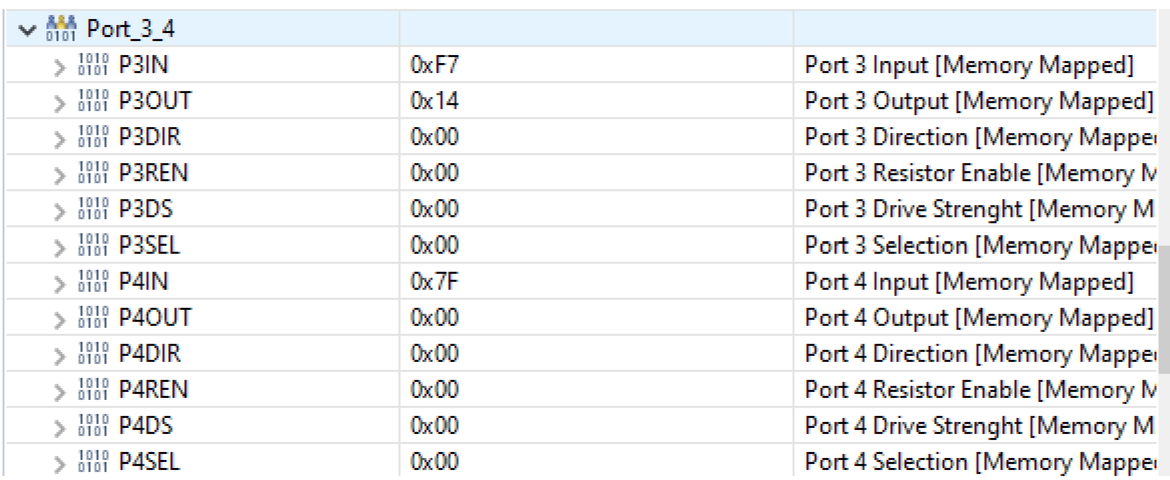
The second problem that was assigned was where the user was meant to take a mathematical expression in string form, and evaluate it to the correct answer. For example, the mathematical expression that was used for this code was “7-2+2”. The code takes in the character 7, changes it to its decimal value, and then assigns it to a register. The code then pulls in the next character, the operator and stores it in a register as well. After this, the code pulls in the next digit, 2, which it converts to its decimal version before also storing it in a register. The code then checks the operator to determine whether to add or subtract the two numbers. After the program performs the proper operation, it moves onto the next operator and next digit. This is repeated until the program reaches the NULL ASCII character which represents the end of the string. The result of the mathematical expression is then sent to P2OUT which is displayed in the register window:

Name	Value	Description
P1OUT	0x07	Port 1 Output [Memory Mapped]
P1DIR	0x00	Port 1 Direction [Memory Mappe
P1REN	0x00	Port 1 Resistor Enable [Memory M
P1DS	0x00	Port 1 Drive Strenght [Memory M
P1SEL	0x00	Port 1 Selection [Memory Mappe
P1IV	0x0000	Port 1 Interrupt Vector Word [Me.
P1IES	0x00	Port 1 Interrupt Edge Select [Mem
P1IE	0x00	Port 1 Interrupt Enable [Memory .
P1IFG	0x00	Port 1 Interrupt Flag [Memory Ma
P2IN	0xFF	Port 2 Input [Memory Mapped]
P2OUT	0x07	Port 2 Output [Memory Mapped]
P2DIR	0x00	Port 2 Direction [Memory Mappe
P2REN	0x00	Port 2 Resistor Enable [Memory M
P2DS	0x00	Port 2 Drive Strenght [Memory M
P2SEL	0x00	Port 2 Selection [Memory Mappe
P2IV	0x0000	Port 2 Interrupt Vector Word [Me.
P2IES	0x00	Port 2 Interrupt Edge Select [Mem
P2IE	0x00	Port 2 Interrupt Enable [Memory .

Figure 3. Result of “7-2+2” Stored in P2OUT, Input P2IN in the Register Window.

Bonus Problem:

The bonus problem that was assigned for this lab was meant to be similar to the rest. The code was meant to take in the string as input and pull in one character at a time. This code was to then change each lowercase letter in the string to an uppercase letter. The code was to keep track of each change that it made and store it in a register and send the result to P3OUT. Also, the code was meant to output the changed string to the memory browser where it can be viewed as a series of characters. This code was partially completed with it only completing half of the desired results. It changes the string to capital and then logs the amount of changes. The string used for this code is: "I enjoy learning MSP430 Assembly!". These changes are shown here (in HEX, 14 in HEX is 20 in decimal):



Port_3_4		
P3IN	0xF7	Port 3 Input [Memory Mapped]
P3OUT	0x14	Port 3 Output [Memory Mapped]
P3DIR	0x00	Port 3 Direction [Memory Mappe
P3REN	0x00	Port 3 Resistor Enable [Memory M
P3DS	0x00	Port 3 Drive Strenght [Memory M
P3SEL	0x00	Port 3 Selection [Memory Mappe
P4IN	0x7F	Port 4 Input [Memory Mapped]
P4OUT	0x00	Port 4 Output [Memory Mapped]
P4DIR	0x00	Port 4 Direction [Memory Mappe
P4REN	0x00	Port 4 Resistor Enable [Memory M
P4DS	0x00	Port 4 Drive Strenght [Memory M
P4SEL	0x00	Port 4 Selection [Memory Mappe

Figure 4. Port Input and Output Register View of Bonus Problem.

Conclusion:

Fortunately, no problems were encountered over the course of this lab. Everything that was set out to be completed was done so in the allotted time and according to the assignment instructions. For part one, the program successfully takes in the string as input one character by a time, and checks each character for being a digit or a capital letter. It then increments the required counters which are then output to the corresponding ports. For problem 2, the program successfully evaluates the string to its correct answer based on the numbers and which operations are to be performed. The correct answer is then output to the correct port. This lab teaches one the ability to perform all of these actions in the assembly language. It teaches one to take a string apart character by character and perform tests and evaluations on it in order to complete the problems that were laid out. Overall, the lab was a success and the desired results were obtained.

Appendix:

Problem 01:

```
; -------------------------------------------------------------------------------
; File:        Lab4P1Final.asm
; Function:    Counts the number of Digits and Capital Letters in a String and Displays the
number to P1OUT and P2OUT Respectively
; Description:  Program traverses an input array of characters
;              to detect capital letters or digits; exits when a NULL is detected
; Input:       The input string specified in myStr
; Output:      The port P1OUT and P2OUT display the number of digits and capital letters in the
string
; Author(s):   Michael Agnew, ma0133@uah.edu
; Date:          February 2nd, 2024
; -------------------------------------------------------------------------------
        .cdecls C, LIST, "msp430.h"     ; Include device header file
;-------------------------------------------------------------------------------
```

```
        .def    RESET                   ; Export program entry-point to
                                        ; make it known to linker.
;-------------------------------------------------------------------------------
myStr:  .string "Welcome To MSP430F5529 Assembly Programming!", ''
        ; .string does not add NULL at the end of the string;
        ; '' ensures that a NULL follows the string.
        ; You can alternatively use .cstring "HELLO WORLD, I AM THE MSP430!"
        ; that adds a NULL character at the end of the string automatically.
;-------------------------------------------------------------------------------
        .text                           ; Assemble into program memory.
        .retain                         ; Override ELF conditional linking
                                        ; and retain current section.
        .retainrefs                     ; And retain any sections that have
                                        ; references to current section.
;-------------------------------------------------------------------------------
RESET:  mov.w   #__STACK_END,SP         ; Initialize stack pointer
        mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
main:   ; bis.b   #0FFh, &P1DIR         ; Do not output the result on port pins
        mov.w   #myStr, R4              ; Load the starting address of the string into R4
        clr.b   R5                      ; Register R5 will serve as a counter
```

```
        clr.b   R7                          ; Register R7 will serve as a counter
gnext:  mov.b   @R4+, R6        ; Get a new character
        cmp     #0, R6          ; Is it a null character
        jeq     lend            ; If yes, go to the end


        cmp.b   #'0', R6        ; Is the character 0?
        jl      gnext                       ; if it's ascii value is lower, move on
        cmp.b   #0x3A, R6                   ; compare it to the ascii value above 9
        jge         capital                     ; if it's greater than ascii value of 9,
then it must be a letter
        inc.w   R5                              ; if it is in this range, increment the
counter
        jmp     gnext                       ; at the end, move to the next character


capital:cmp.b   #'A', R6                    ; compare character to ascii value A
                jl          gnext                   ; if it's less, then jump to next
character
                cmp.b   #0x5B, R6                   ; compare to ascii value above Z
                jge     gnext                       ; if it's greater, jump to next character
                inc.w   R7                              ; if it is in the range, then
increment capital letter counter
```

```
                jmp     gnext                           ; afterwards, move to the next
character

lend:   mov.b   R5,&P1OUT           ; Write result in P1OUT (not visible on port pins)
                mov.b   R7,&P2OUT                       ; Write result in P2OUT (not visible
on port pins)
        bis.w   #LPM4, SR           ; LPM4
        nop                         ; Required only for debugger
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
        .global __STACK_END
        .sect   .stack
;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
        .sect   ".reset"            ; MSP430 RESET Vector
        .short  RESET
        .end
```

Problem 02:

```
; ------------------------------------------------------------------------------
; File:       Lab4P2CodeFinal.asm
; Function:   Evalutes a string as a math operation
```

```
; Description:  Program traverses an input array of characters
;               to detect operands and operators, then evalutes the mathematical expression; exits
when a NULL is detected
; Input:        The input string specified in myStr
; Output:       The port P2OUT displays the answer to the expression
; Author(s):    Michael Agnew, ma0133@uah.edu
; Date:         February 2nd, 2024
; Revised:      February 6th, 2024
; -----------------------------------------------------------------------------
        .cdecls C, LIST, "msp430.h"     ; Include device header file
;------------------------------------------------------------------------------
        .def    RESET                   ; Export program entry-point to
                                        ; make it known to linker.
;------------------------------------------------------------------------------
myStr:  .string "7-2+2", ''
        ; .string does not add NULL at the end of the string;
        ; '' ensures that a NULL follows the string.
        ; You can alternatively use .cstring "HELLO WORLD, I AM THE MSP430!"
        ; that adds a NULL character at the end of the string automatically.
;------------------------------------------------------------------------------
        .text                           ; Assemble into program memory.
        .retain                         ; Override ELF conditional linking
                                        ; and retain current section.
```

```
        .retainrefs                     ; And retain any sections that have
                                        ; references to current section.
;-------------------------------------------------------------------------------
RESET:  mov.w   #__STACK_END,SP         ; Initialize stack pointer
        mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
main:   ; bis.b   #0FFh, &P1DIR         ; Do not output the result on port pins
        mov.w   #myStr, R4              ; Load the starting address of the string into R4
        clr.b   R5                      ; Register R5 will serve as a counter
gnext:  mov.b   @R4+, R6                ; Get a new character
        cmp     #0, R6                  ; Is it a null character
        jeq     lend                    ; If yes, go to the end
 ;
==========================================================================
==


                cmp.b   #'0', R6                ; Check if 0
        jne     check1                  ; if not, move to check if 1
                mov.b   #0, R6                                  ; if so, assign decimal value 0 to R6
        jmp     op                                              ; afterwards, jump to op branch
```

```
; ========================================= REPEAT STEPS FOR REST OF
INTEGERS ===============
check1: cmp.b   #'1', R6
                jne     check2
                mov.b   #1, R6
                mov.b  R6, R9
        jmp     op

check2: cmp.b   #'2', R6
                jne     check3
                mov.b   #2, R6
                mov.b  R6, R9
        jmp     op

check3: cmp.b   #'3', R6
                jne     check4
                mov.b   #3, R6
                mov.b  R6, R9
        jmp     op

check4: cmp.b   #'4', R6
                jne     check5
                mov.b   #4, R6
```

```
            mov.b  R6, R9
        jmp         op

check5: cmp.b   #'5', R6
            jne     check6
            mov.b   #5, R6
            mov.b  R6, R9
        jmp         op

check6: cmp.b   #'6', R6
            jne     check7
            mov.b   #6, R6
            mov.b  R6, R9
        jmp         op

check7: cmp.b   #'7', R6
            jne     check8
            mov.b   #7, R6
            mov.b  R6, R9
        jmp         op

check8: cmp.b   #'8', R6
            jne     check9
```

```
                mov.b   #8, R6
                mov.b  R6, R9
        jmp         op

check9: cmp.b   #'9', R6
                jne     op
                mov.b   #9, R6
                mov.b  R6, R9
                jmp      op

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

op:             mov.b   @R4+, R7                        ; pull in next char from string
                cmp     #0, R7              ; Is it a null character
        jeq     lend                ; If yes, go to the end
        jmp         src                                 ; afterwards, jump to src branch

;
====================================================================
==
```

```
src:    mov.b   @R4+, R8
                cmp     #0, R8              ; Is it a null character
        jeq     lend                ; If yes, go to the end

            cmp.b   #'0', R8            ; Is it 0 character
        jne     check11             ; If not, go to the next check
                mov.b   #0, R8                              ; if it is, assign decimal value 0 to R8
                jmp     subbr                               ; afterwards, jump to subbr branch

; ========================================== REPEAT STEPS FOR REST OF
INTEGERS ===============

check11:cmp.b   #'1', R8
                jne     check22
                mov.b   #1, R8
                jmp     subbr

check22:cmp.b   #'2', R8
                jne     check33
                mov.b   #2, R8
                jmp     subbr

check33:cmp.b   #'3', R8
```

```
            jne     check44
            mov.b   #3, R8
            jmp     subbr

check44:cmp.b   #'4', R8
            jne     check55
            mov.b   #4, R8
            jmp     subbr

check55:cmp.b   #'5', R8
            jne     check66
            mov.b   #5, R8
            jmp     subbr

check66:cmp.b   #'6', R8
            jne     check77
            mov.b   #6, R8
            jmp     subbr

check77:cmp.b   #'7', R8
            jne     check88
            mov.b   #7, R8
            jmp     subbr
```

```
check88:cmp.b   #'8', R8
        jne     check99
        mov.b   #8, R8
        jmp     subbr

check99:cmp.b   #'9', R8
        mov.b   #9, R8
        jmp     subbr
;
========================================================================
==

subbr:  cmp.b  #'-', R7 ; check to see if minus sign
        jne      addbr    ; if not, jump to add branch
        sub.b    R8, R9   ; if so, subtract R8 from R9 and store in R9
        jmp            op            ; aftewards, get the next operator

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

addbr:  cmp.b   #'+', R7 ; check to see if plus sign
```

```
                add.b   R8, R9  ; if so, add the two numbers and store in R9
                jmp             op                  ; afterwards, get the next operator


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



lend:   mov.b   R9,&P2OUT               ; Write result in P2OUT (not visible on port pins)
        bis.w   #LPM4, SR               ; LPM4
        nop                             ; Required only for debugger
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
        .global __STACK_END
        .sect   .stack
;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
        .sect   ".reset"                ; MSP430 RESET Vector
        .short  RESET
        .end
```

Bonus Problem:

```
; -----------------------------------------------------------------------------
; File:         Lab04_D1.asm
; Function:     Counts the number of Digits and Capital Letters in a String and Displays the
number to P1OUT and P2OUT Respectively
; Description:  Program traverses an input array of characters
;               to detect capital letters or digits; exits when a NULL is detected
; Input:        The input string specified in myStr
; Output:       The port P1OUT and P2OUT display the number of digits and capital letters in the
string
; Author(s):    Michael Agnew, ma0133@uah.edu
; Date:         February 2nd, 2024
; -----------------------------------------------------------------------------
        .cdecls C, LIST, "msp430.h"     ; Include device header file
;-----------------------------------------------------------------------------
        .def    RESET                   ; Export program entry-point to
                                        ; make it known to linker.
;-----------------------------------------------------------------------------
myStr:  .string "I enjoy learning MSP430 Assembly!", ''
        ; .string does not add NULL at the end of the string;
        ; '' ensures that a NULL follows the string.
        ; You can alternatively use .cstring "HELLO WORLD, I AM THE MSP430!"
        ; that adds a NULL character at the end of the string automatically.
```

```
;-------------------------------------------------------------------------------
        .text                           ; Assemble into program memory.
        .retain                         ; Override ELF conditional linking
                                        ; and retain current section.
        .retainrefs                     ; And retain any sections that have
                                        ; references to current section.
;-------------------------------------------------------------------------------
RESET:  mov.w   #__STACK_END,SP         ; Initialize stack pointer
        mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
main:   ; bis.b   #0FFh, &P1DIR         ; Do not output the result on port pins
        mov.w   #myStr, R4              ; Load the starting address of the string into R4
        clr.b   R5                      ; Register R5 will serve as a counter
        clr.b   R7                                              ; Register R7 will serve as a counter
gnext:  mov.b   @R4+, R6                ; Get a new character
        cmp     #0, R6                  ; Is it a null character
        jeq     lend                    ; If yes, go to the end

checkA: cmp.b   #'a', R6
                jne     checkB
                mov.b  #0x41, R6
```

```
        inc.w   R7                  ; If yes, increment counter
        jmp     gnext               ; Go to the next character

checkB: cmp.b   #'b', R6
                jne     checkC
                mov.b  #0x42, R6
        inc.w   R7                  ; If yes, increment counter
        jmp     gnext               ; Go to the next character

checkC: cmp.b   #'c', R6
                jne     checkD
                mov.b  #0x43, R6
        inc.w   R7                  ; If yes, increment counter
        jmp     gnext               ; Go to the next character

checkD: cmp.b   #'d', R6
                jne     checkE
                mov.b  #0x44, R6
        inc.w   R7                  ; If yes, increment counter
        jmp     gnext               ; Go to the next character

checkE: cmp.b   #'e', R6
                jne     checkF
```

```
                mov.b  #0x45, R6
      inc.w   R7                    ; If yes, increment counter
      jmp     gnext                 ; Go to the next character

checkF: cmp.b   #'f', R6
                jne     checkG
                mov.b  #0x46, R6
      inc.w   R7                    ; If yes, increment counter
      jmp     gnext                 ; Go to the next character

checkG: cmp.b   #'g', R6
                jne     checkH
                mov.b  #0x47, R6
      inc.w   R7                    ; If yes, increment counter
      jmp     gnext                 ; Go to the next character

checkH: cmp.b   #'h', R6
                jne     checkI
                mov.b  #0x48, R6
      inc.w   R7                    ; If yes, increment counter
      jmp     gnext                 ; Go to the next character

checkI: cmp.b   #'i', R6
```

```
            jne     checkJ
            mov.b  #0x49, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkJ: cmp.b   #'j', R6
            jne     checkK
            mov.b  #0x4A, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkK: cmp.b   #'k', R6
            jne     checkL
            mov.b  #0x4B, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkL: cmp.b   #'l', R6
            jne     checkM
            mov.b  #0x4C, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character
```

```
checkM: cmp.b   #'m', R6
        jne     checkN
        mov.b  #0x4D, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkN: cmp.b   #'n', R6
        jne     checkO
        mov.b  #0x4E, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkO: cmp.b   #'o', R6
        jne     checkP
        mov.b  #0x4F, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character

checkP: cmp.b   #'p', R6
        jne     checkQ
        mov.b  #0x50, R6
    inc.w   R7                  ; If yes, increment counter
    jmp     gnext               ; Go to the next character
```

```
checkQ: cmp.b   #'q', R6
               jne     checkR
               mov.b  #0x51, R6
     inc.w   R7                 ; If yes, increment counter
     jmp     gnext              ; Go to the next character

checkR: cmp.b   #'r', R6
               jne     checkS
               mov.b  #0x52, R6
     inc.w   R7                 ; If yes, increment counter
     jmp     gnext              ; Go to the next character

checkS: cmp.b   #'s', R6
               jne     checkT
               mov.b  #0x53, R6
     inc.w   R7                 ; If yes, increment counter
     jmp     gnext              ; Go to the next character

checkT: cmp.b   #'t', R6
               jne     checkU
               mov.b  #0x54, R6
```

```
       inc.w   R7                     ; If yes, increment counter
       jmp     gnext                  ; Go to the next character

checkU: cmp.b   #'u', R6
                jne     checkV
                mov.b  #0x55, R6
       inc.w   R7                     ; If yes, increment counter
       jmp     gnext                  ; Go to the next character

checkV: cmp.b   #'v', R6
                jne     checkW
                mov.b  #0x56, R6
       inc.w   R7                     ; If yes, increment counter
       jmp     gnext                  ; Go to the next character

checkW: cmp.b   #'w', R6
                jne     checkX
                mov.b  #0x57, R6
       inc.w   R7                     ; If yes, increment counter
       jmp     gnext                  ; Go to the next character

checkX: cmp.b   #'x', R6
                jne     checkY
```

```
                mov.b  #0x58, R6
      inc.w   R7                 ; If yes, increment counter
      jmp     gnext              ; Go to the next character

checkY: cmp.b   #'y', R6
                jne     checkZ
                mov.b  #0x59, R6
      inc.w   R7                 ; If yes, increment counter
      jmp     gnext              ; Go to the next character

checkZ: cmp.b   #'z', R6
                jne     gnext
                mov.b  #0x5A, R6
      inc.w   R7                 ; If yes, increment counter
      jmp     gnext              ; Go to the next character

lend:   mov.b   R5,&P1OUT          ; Write result in P1OUT (not visible on port pins)
                mov.b   R7,&P3OUT
      bis.w   #LPM4, SR          ; LPM4
      nop                        ; Required only for debugger
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
```

```
            .global __STACK_END
            .sect   .stack
;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET
            .end
```