CPE 323 MSP430
MSP430 Assembly Language Programming

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective:
Introduce MSP430 assembly language

Contents

1 Introduction

In this section we will introduce MSP430 Assembly Language Programing using several illustrative examples. Before we do that, let us first talk briefly about developing software for embedded systems in general.

In desktop/server computing systems we typically develop and debug software programs on the same or a similar platform the program is going to run on. However, software for embedded systems is typically developed on a workstation/desktop computer and then downloaded into the target platform (embedded system) through a dedicated interface. Debugging of embedded systems is made possible through either software emulation or dedicated debuggers that allow us to interact with a program running on the target platform.

Software for embedded systems is typically developed using modern software development environments (SDEs) that integrate editors, assembler, compiler, linker, stand-alone simulator, embedded emulator or debugger, and flash programmer. Examples of SDEs we can use are IAR for MSP430 and TI's Code Composer Studio for MSP430 (used in the laboratory). Below is a brief description of major components of modern SDEs.

- Editor: Allows you to enter source code (assembly, C, or C++). A good editor will have features to help you format your code nicely for improved readability, comply with syntax rules, easily locate definitions and symbols, and other useful features that make life of a software developer easier.
- Assembler: a program that translates source code written in assembly language into executable code.
- Compiler: a program that translates source code written in C or C++ into executable code.
- Linker: a program that combines multiple files with executable code and routines from libraries and arranges them into memory that complies with rules for a specific microcontroller.
- Simulator: a program that simulates operation of the microcontroller on a desktop computer, thus alleviating the need to have actual hardware when testing software. Simulators vary in functionality – some include support only for the processor, whereas others can also simulate behavior of peripheral devices.
- Flash Programmer: a program that downloads the embedded software into flash memory of the microcontroller.
- Embedded emulator/debugger: a program running on the desktop computer that allows software under development to run on the target platform and controls its execution (i.e., allowing the program under development to run one instruction before returning control to the debugger or to run until a breakpoint is reached). It controls running of the program on the target through a special interface, e.g., JTAG for MSP430.

Figure 1 shows a typical development flow that starts from assembly code residing in one or more input files (with extension .asm or .s43 for MSP430 assembly programs). These files are translated into object files using assembler. The object files together with libraries are tied together by linker that produces an executable file. The executable file is then loaded into the simulator or downloaded into the target platform using flash programmer.

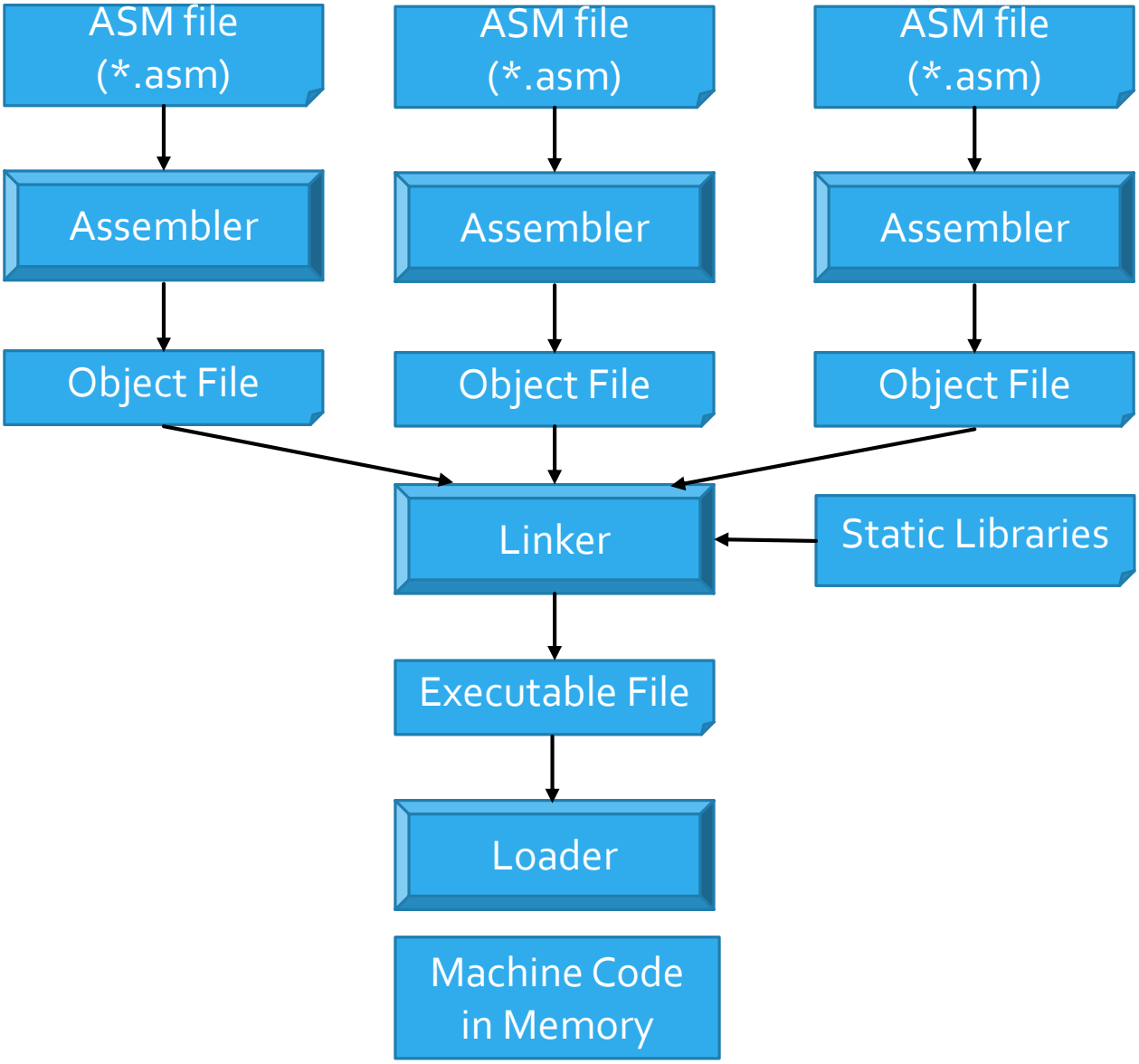

Figure 1. Design flow from assembly programs to machine code.

2 Assembly Language Directives

Assembly language directives tell the assembler to set the data and program at particular addresses, allocate space in memory for variables, allocate space in memory and initialize constants, define synonyms, or include additional files.

Asm430 (TI CCStudio MSP430 assembler) has predefined sections into which various parts of a program are assembled. Uninitialized data is assembled into the .bss section, initialized data into the .data section, and executable code into the .text section. A430 (IAR MSP430 assembler) also uses sections or segments, but there are no predefined segment names. However, it is convenient to adhere to the names used by C compiler: DATA_16_Z for uninitialized data, CONST for constant (initialized) data, and CODE for executable code. Table 1 lists main sections and section directives used by ASM430 (TI's CCS assembler) and A430 (IAR assembler).

Table 1. Sections and section directives in ASM430 and A430.

Description	ASM430 (CCS)	A430 (IAR)
Reserve size bytes in the uninitialized sect.	.bss	-
Assemble into the initialized data section	.data	RSEG const
Assemble into a named initialized data section	.sect	RSEG
Assemble into the executable code	.text	RSEG code
Reserve space in a named (uninitialized) section	.usect	-
Align on byte boundary	.align 1	-
Align on word boundary	.align 2	EVEN

Table 2 describes most frequently used assembly language directives for defining constants. The constants can be placed in either the .text section which resides in the Flash memory and then they cannot be changed or in the .data section that is in RAM memory and the data can be programmatically changed.

Table 2. Constant Initialization Directives

Description	ASM430 (CCS)	A430 (IAR)
Initialize one or more successive bytes or text strings	.byte or .string	DB/DC8
Initialize 32-bit IEEE floating-point	.float	DF
Initialize a variable-length field	.field	-
Reserve size bytes in the current location	.space	DS/DS8
Initialize one or more 16-bit integers	.word	DW/DC16
Initialize one or more 32-bit integers	.long	DL/DC32

The example below shows assembly language directives for allocating space in RAM memory for two variables in RAM memory using A430 (IAR) and ASM430 (CCS).

Example #1:

```
; IAR
        RSEG DAT16_N          ; switch to DATA segment
        EVEN                  ; make sure it starts at even address
MyWord: DS 2                  ; allocate 2 bytes / 1 word
MyByte: DS 1                  ; allocate 1 byte

; CCS Assembler (Example #1)
MyWord: .usect ".bss", 2, 2   ; allocate 1 word
MyByte: .usect ".bss", 1      ; allocate 1 byte

; CCS Assembler (Example #2)
        .bss  MyWord,2,2     ; allocate 1 word
        .bss  MyByte,1       ; allocate 1 byte
```

Example #2: Figure 2 shows a sequence of assembly language directives that populate Flash memory with 8-bit constants (.byte directive), 16-bit constants (.word directive), and 32-bit constants (.long directive). We can specify decimal constants (number without any prefix or suffix), binary numbers (suffix b), octal numbers (suffix q), and hexadecimal numbers (suffix h or prefix 0x). ASCII characters are specified using single quotes, whereas a string under double quotes is a series of ASCII characters. Two single quotes in line 21 represent a NULL character added at the end of the string "ABCD". Figure 3 illustrates the content of the flash memory after these directives are carried out. Please note that assembler decided to place these constants in the Flash memory starting at the address 0x3100. As a result of parsing this sequence, the assembler creates a table of symbols (synonyms) shown in Figure 4.

```
1  ;-------------------------------------------------------------------------------
2  ; define data section with constants
3
4  b1:          .byte   5          ; allocates a byte in memory and initialize it with 5
5  b2:          .byte   -122       ; allocates a byte with constant -122
6  b3:          .byte   10110111b  ; binary value of a constant
7  b4:          .byte   0xA0       ; hexadecimal value of a constant
8  b5:          .byte   123q       ; octal value of a constant
9  tf:          .equ 25
10              .align 2           ; move a location pointer to the first even address
11 w1:          .word   21         ; allocates a word constant in memory;
12 w2:          .word  -21
13 w3:          .word tf
14 dw1:         .long  100000      ; allocates a long word size constant in memory;
15                                 ; 100000 (0x0001_86A0)
16 dw2:         .long 0xFFFFFFEA
17              .align 2
18 s1:          .byte 'A', 'B', 'C', 'D'    ; allocates 4 bytes in memory with string ABCD
19 s2:          .byte "ABCD", ''   ; allocates 5 bytes in memory with string ABCD + NULL
```

Figure 2. Assembly Language Directives: An Example.

Label	Address	Memory[15:8]	Memory[7:0]
b1	0x3100	0x86	0x05
b3	0x3102	0xA0	0xB7
b5	0x3104	--	0x51
w1	0x3106	0x00	0x15
w2	0x3108	0xFF	0xEB
w3	0x310A	0x00	0x19
dw1	0x310C	0x86	0xA0
	0x310E	0x00	0x01
dw2	0x3110	0xFF	0xEA
	0x3112	0xFF	0xFF
s1	0x3114	0x42	0x41
	0x3116	0x44	0x43
s2	0x3118	0x42	0x41
	0x311A	0x44	0x43
	0x311C	--	0x00

Figure 3. Memory content (a word-view).

Symbol	Value [hex]
b1	0x3100
b2	0x3101
b3	0x3102
b4	0x3103
b5	0x3104
tf	0x0019
w1	0x3106
w2	0x3108
w3	0x310A
dw1	0x310C
dw2	0x3110
s1	0x3114
s2	0x3118

Figure 4. Table of symbols (maintained by assembler).

To allocate space in RAM memory we use directives as shown below. Figure 5 shows a sequence of directives for allocating space in memory. Figure 6 shows the content of the RAM after allocation (it is not initialized) and Figure 7 shows the table of symbols created by the assembler upon parsing these directives. Note #1. Assembler placed allocated space at the address of 0x1100, which belongs to RAM memory for the MSP430FG4618. Note #2: RAM memory is built out of SRAM cells; upon powering chip up these cells take a state of either logic 1 or logic 0 in a random fashion, but for us the memory cells do not have meaningful content, so we consider them uninitialized.

```
1           .bss v1b,1,1      ; allocates a byte in memory, equivalent to DS 1
2           .bss v2b,1,1      ; allocates a byte in memory
3           .bss v3w,2,2      ; allocates a word of 2 bytes in memory
4           .bss v4b,8,2      ; allocates a buffer of 8 bytes
5           .bss vx,
```

Figure 5. Assembly Language Directives: An Example.

Label	Address	Memory[15:8]	Memory[7:0]
v1b	0x1100	--	--
v3w	0x1102	--	--
v4b	0x1104	--	--
	0x1106	--	--
	0x1108	--	--
	0x110A	--	--
	0x110C		

Figure 6. Memory content (a word-view).

Symbol	Value [hex]
v1b	0x1100
v2b	0x1101
v3w	0x1102
v4b	0x1104
vx	0x110C

Figure 7. Table of symbols (maintained by assembler).

3 Decimal and Integer Addition of 32-bit Integers

In this section we will consider an assembly program that sums up two 32-bit integers (lint1, and lint2) producing two 32-bit results, one assuming these integers represent regular binary coded 32-bit integers (lsumi) and one assuming these integers represent packed binary coded decimal numbers (BCD). The assembler will place these directives at the first address that belong to the Flash memory (0x3100 in case for MSP430FG4618). Thus, lint1 is at 0x3100 and lint2 is at 0x3104 and that they are initialized as shown in Figure 8 (lines 27 and 28). The output variables are allocated in RAM (we cannot write into the Flash memory) at the starting address of RAM which is 0x1100 (lsumd) and 0x1104 (lsumi). As MSP430 performs only operations on 8-bit bytes and 16-bit words, to find decimal and binary sums in this example, we will need to perform the requested operations in two rounds – one to sum up lower words and one to sum up upper words of input variables. First, lower 16-bit of lint1 (address with label lint1) is loaded into register R8 (line 40). Please note the source operand is specified using the symbolic addressing mode, thus R8<=M[lint1]. Next, decimal addition DADD.W instruction is used to add the lower 16-bit of lint2 to R8, R8<=R8+M[lint2]+C (line 41). Note: DADD instruction performs the following operation: src+dst+C => dst (decimally). Now register R8 contains the lower 16-bit of the decimal sum and it is moved to lsumd, M[lsumd]<=R8 (line 42). In the next round, we reach to upper 16-bit of lint1 residing at lint1+2, as well as upper 16-bit of lint2 and store the result to lsumd+2. Please note that the DADD.W instruction in line 41 produces a carry bit that needs to be used by the DADD.W instruction in line 44 to have correct sum. Luckily, we have two move instructions in between that do not affect the carry flag between lines 41 and 44, so the carry bit produced in line 41 is used by DADD.W in line 44. Consequently, to make this code work properly, we clear carry flag before we start computation (line 39).

A similar sequence of steps is performed for binary addition. Here, we use ADD.W instruction in the first round and ADDC.W instruction in the second round instead DADD.W instructions. Also, note that carry generated by the ADD.W instruction in line 47 is used by the ADDC in line 50.

```
1  ;-------------------------------------------------------------------------------
2  ; File       : LongIntAddition.asm
3  ; Function   : Sums up two long integers represented in binary and BCD
4  ; Description: Program demonstrates addition of two operands lint1 and lint2.
5  ;              Operands are first interpreted as 32-bit decimal numbers and
6  ;              and their sum is stored into lsumd;
7  ;              Next, the operands are interpreted as 32-bit signed integers
8  ;              in two's complement and their sum is stored into lsumi.
9  ; Input      : Input integers are lint1 and lint2 (constants in flash)
10 ; Output     : Results are stored in lsumd (decimal sum) and lsumi (int sum)
11 ; Author     : A. Milenkovic, milenkovic@computer.org
12 ; Date       : August 24, 2018
13 ;-------------------------------------------------------------------------------
14             .cdecls C,LIST,"msp430.h"       ; Include device header file
15
16 ;-------------------------------------------------------------------------------
17             .def    RESET                   ; Export program entry-point to
18                                             ; make it known to linker.
19 ;-------------------------------------------------------------------------------
20             .text                           ; Assemble into program memory.
```

```
21              .retain                         ; Override ELF conditional linking
22                                              ; and retain current section.
23              .retainrefs                     ; And retain any sections that have
24                                              ; references to current section.
25  ;-------------------------------------------------------------------------------
26  ;-------------------------------------------------------------------------------
27  lint1:          .long 0x45678923
28  lint2:          .long 0x23456789
29  ;-------------------------------------------------------------------------------
30  ;-------------------------------------------------------------------------------
31  lsumd:          .usect ".bss", 4,2       ; allocate 4 bytes for decimal result
32  lsumi:          .usect ".bss", 4,2       ; allocate 4 bytes for integer result
33  ;-------------------------------------------------------------------------------
34  RESET:       mov.w   #__STACK_END,SP         ; Initialize stack pointer
35  StopWDT:     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
36  ;-------------------------------------------------------------------------------
37  ; Main code here
38  ;-------------------------------------------------------------------------------
39                clr.w   R2              ; clear status register
40                mov.w   lint1, R8       ; get lower 16 bits from lint1 to R8
41                dadd.w  lint2, R8       ; decimal addition, R8 + lower 16-bit of lint2
42                mov.w   R8, lsumd       ; store the result (lower 16-bit)
43                mov.w   lint1+2, R8     ; get upper 16 bits of lint1 to R8
44                dadd.w  lint2+2, R8     ; decimal addition
45                mov.w   R8, lsumd+2     ; store the result (upper 16-bit)
46                mov.w   lint1, R8       ; get lower 16 bite from lint1 to R8
47                add.w   lint2, R8       ; integer addition
48                mov.w   R8, lsumi       ; store the result (lower 16 bits)
49                mov.w   lint1+2, R8     ; get upper 16 bits from lint1 to R8
50                addc.w  lint2+2, R8     ; add upper words, plus carry
51                mov.w   R8, lsumi+2     ; store upper 16 bits of the result
52
53                jmp $                   ; jump to current location '$'
54                                        ; (endless loop)
55
56
57  ;-------------------------------------------------------------------------------
58  ; Stack Pointer definition
59  ;-------------------------------------------------------------------------------
60              .global __STACK_END
61              .sect   .stack
62
63  ;-------------------------------------------------------------------------------
64  ; Interrupt Vectors
65  ;-------------------------------------------------------------------------------
66              .sect   ".reset"                ; MSP430 RESET Vector
67              .short  RESET
68
69
```

Figure 8. Decimal and integer addition (first implementation).

Figure 9 shows a program ready to run with all necessary directives and an alternative implementation from the one shown in Figure 8. First, let us describe how the program execution starts. Upon powering up, so-called PUC (power-up clear) signal in hardware is generated. The first thing MSP430 does is to fetch a word from location 0xFFFE (the top word address in the 64 KB address space). This location is known as ***reset interrupt vector***. Note: the top 32 words of the 64KB address space are reserved for the interrupt vector table (IVT) and the top most address is reserved for the reset vector, which is the highest priority interrupt

request in MSP430. The content of this location is moved into Program Counter (PC <= M[0xFFFE]). Thus, note that our entry point in the program (address of the first instruction) has label RESET (line 37). The value of the symbol RESET is used to initialize the reset vector in the interrupt vector table (lines 82 and 83).

```
1  ;-------------------------------------------------------------------------------
2  ; File       : LongIntAdditionv2.asm
3  ; Function   : Sums up two long integers represented in binary and BCD
4  ; Description: Program demonstrates addition of two operands lint1 and lint2.
5  ;              Operands are first interpreted as 32-bit decimal numbers and
6  ;              and their sum is stored into lsumd;
7  ;              Next, the operands are interpreted as 32-bit signed integers
8  ;              in two's complement and their sum is stored into lsumi.
9  ;              This version uses loops.
10 ; Input      : Input integers are lint1 and lint2 (constants in flash)
11 ; Output     : Results are stored in lsumd (decimal sum) and lsumi (int sum)
12 ; Written by : A. Milenkovic, milenkovic@computer.org
13 ; Date       : August 24, 2018
14 ;-------------------------------------------------------------------------------
15             .cdecls C,LIST,"msp430.h"       ; Include device header file
16
17 ;-------------------------------------------------------------------------------
18             .def    RESET                   ; Export program entry-point to
19                                             ; make it known to linker.
20 ;-------------------------------------------------------------------------------
21             .text                           ; Assemble into program memory.
22             .retain                         ; Override ELF conditional linking
23                                             ; and retain current section.
24             .retainrefs                     ; And retain any sections that have
25                                             ; references to current section.
26
27 ;-------------------------------------------------------------------------------
28 ;-------------------------------------------------------------------------------
29 lint1:       .long 0x45678923
30 lint2:       .long 0x23456789
31 ;-------------------------------------------------------------------------------
32 ;-------------------------------------------------------------------------------
33 lsumd:       .usect ".bss", 4,2            ; allocate 4 bytes for decimal result
34 lsumi:       .usect ".bss", 4,2            ; allocate 4 bytes for integer result
35 ;-------------------------------------------------------------------------------
36
37 RESET:      mov.w   #__STACK_END,SP         ; Initialize stackpointer
38 StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
39
40 ;-------------------------------------------------------------------------------
41 ; Main loop here
42 ;-------------------------------------------------------------------------------
43 ; Decimal addition
44             mov.w   #lint1, R4              ; pointer to lint1
45             mov.w   #lsumd, R8              ; pointer to lsumd
46             mov.w   #2, R5                  ; R5 is a counter (32-bit=2x16-bit)
47             clr.w   R10                     ; clear R10
48 lda:        mov.w   4(R4), R7               ; load lint2
49             mov.w   R10, R2                 ; load original SR
```

```
50              dadd.w  @R4+, R7                ; decimal add lint1 (with carry)
51              mov.w   R2, R10                 ; backup R2 in R10
52              mov.w   R7, 0(R8)               ; store result (@R8+0)
53              add.w   #2, R8                  ; update R8
54              dec.w   R5                      ; decrement R5
55              jnz     lda                     ; jump if not zero to lda
56  ; Integer addition
57              mov.w   #lint1, R4              ; pointer to lint1
58              mov.w   #lsumi, R8              ; pointer to lsumi
59              mov.w   #2, R5                  ; R5 is a counter (32-bit=2x16-bit)
60              clr.w   R10                     ; clear R10
61  lia:        mov.w   4(R4), R7               ; load lint2
62              mov.w   R10, R2                 ; load original SR
63              addc.w  @R4+, R7                ; decimal add lint1 (with carry)
64              mov.w   R2, R10                 ; backup R2 in R10
65              mov.w   R7, 0(R8)               ; store result (@R8+0)
66              add.w   #2, R8                  ; update R8
67              dec.w   R5                      ; decrement R5
68              jnz     lia                     ; jump if not zero to lia
69
70              jmp     $                       ; jump to current location '$'
71                                              ; (endless loop)
72
73  ;-------------------------------------------------------------------------------
74  ; Stack Pointer definition
75  ;-------------------------------------------------------------------------------
76              .global __STACK_END
77              .sect   .stack
78
79  ;-------------------------------------------------------------------------------
80  ; Interrupt Vectors
81  ;-------------------------------------------------------------------------------
82              .sect   ".reset"                ; MSP430 RESET Vector
83              .short  RESET
84
```

Figure 9. Decimal and integer 32-bit addition.

4 Counting Characters ‘E’ in a String

This section defines the problem that will be solved by the *Count Characters* program using the MSP430 assembly language. Our task is to develop an assembly program that will scan a given string of characters, for example, “`HELLO WORLD, I AM THE MSP430!`”, and find the number of appearances of the character ‘E’ in the string. A counter that records the number of characters ‘E’ is then written to the parallel port P1. The port should be configured as an output port, and the binary value of the port will correspond to the counter value.

To solve this assignment, let us first analyze the problem statement. Your task is to write an assembly program that will count the number of characters ‘E’ in a string. First, the problem implies that we need to allocate space in memory that will keep the string “`HELLO WORLD, I AM THE MSP430!`”. The string has 29 characters and they are encoded using the ASCII table. To allocate and initialize a string in memory we can use an assembly language directive, .byte or

.string: `.string "HELLO WORLD, I AM THE MSP430!"`. We can also put a label to mark the beginning of this string in memory, for example, mystr: `mystr .string "HELLO WORLD, I AM THE MSP430!"`.

When assembler sees the .string directive, it will allocate the space in memory required for the string that follows and initialize the allocated space with corresponding ASCII characters. We will also specify an additional NULL ASCII character to terminate the string (`ascii(NULL)=0x00`). So, the total number of bytes occupied by this string terminated by the NULL character is 30.
Our task is now to write a program that will scan the string, character by character, check whether the current character is equal to the character 'E', and if yes, increment a counter. The string scan is done in a program loop. The program ends when we reach the end of the string, which is detected when the current character is equal to the NULL character (0x00).
To scan the string we will use a register to point to the current character in the string. This pointer register is initialized at the beginning of the program to point to the first character in the string. The pointer will be incremented in each iteration of the program loop. Another register, initialized to zero at the beginning, will serve as the counter, and it is incremented every time the current character is 'E'.
After we exit the program loop, the current value of the counter will be written to the port P1, which should be initialized as an output port.
Note: It is required that you are familiar with the MSP430 instruction set and addressing modes to be able to solve this problem. Also, we will assume that the string is no longer than 255 characters, so the result can be displayed on an 8-bit port.

4.1 Count Characters Assembly Code

Figure 10 shows the assembly code for this program. Here is a short description of the assembly code.
The comments in a single line start with a column character (;).
Line 11, `.cdecls C,LIST,"msp430.h"`, is a C-style pre-processor directive that specifies a header file to be included in the source code. The header file includes all macro definitions, for example, special function register addresses (WDTCTL), and control bits (WDTPW+WDTHOLD).

Next, in line 17 we allocate the string `myStr` using `.string` directive: `myStr .string "HELLO WORLD, I AM THE MSP430!", ''`. As explained above, this directive will allocate 30 bytes in memory starting at the address 0x3100 and initialize it with the string content, placing the ASCII codes for the string characters in the memory. The hexadecimal content in memory will be as follows: 48 45 4c 4c 4f 20 57 4f 52 4c 44 2c 20 49 20 41 4d 20 54 48 45 20 4d 53 50 34 33 30 21 00 (ascii('H')=0x48, ascii('H')=0x45, … ascii('!')=0x21, ascii(NULL)=x00).

.text is a section control assembler directive that controls how code and data are located in memory. `.text` is used to mark the beginning of a relocatable code or data segment. This directive is resolved by the linker.

The first instruction in line 26 initializes the stack pointer register (`mov.w #__STACK_END,SP`). Our program does not use the stack, so we could have omitted lines 51 and 52 that define the stack section as well as this instruction.
The instruction `mov.w #WDTPW|WDTHOLD,&WDTCTL` sets certain control bits of the watchdog timer control register (`WDTCTL`) to disable it. The watchdog timer by default is active upon reset, generating interrupt requests periodically. As this functionality is not needed in our program, we simply need to disable it.

Parallel ports in the MSP430 microcontroller can be configured as either input or output ports. A control register PxDIR determines whether the port x is an input or an output port (we can configure each individual port pin). Our program drives all eight pins of the port P1, so it should be configured as an output port by setting each individual pin to 1 (`P1DIR=0xFF`). Register R4 is loaded to point to the first character in the string. Register R5, the counter, is cleared before starting the main program loop.

The main loop starts at the next label. We use the autoincrement addressing mode to read a new character (one byte) from the string (`mov.b @R4+, R6`). The current character is kept in register R6. We then compare the current character with the NULL character (`cmp.b #0,R6`). If it is the NULL character, the end of the string has been reached and we exit the loop (JEQ lend). Pay attention that we used JEQ instruction? Why is this instruction used? Which flag is inspected?
If it is not the end of the string, we compare the current character with 'E'. If there is no match we go back to the first instruction in the loop. Otherwise, we increase the value of the counter (register R5). Finally, once the end of the string has been reached, we move the lower byte from R5 to the parallel port 1, `P1OUT=R5[7:0]`.

```
1   ;-------------------------------------------------------------------------------
2   ; File       : Lab4_D1.asm (CPE 325 Lab4 Demo code)
3   ; Function   : Counts the number of characters E in a given string
4   ; Description: Program traverses an input array of characters
5   ;              to detect a character 'E'; exits when a NULL is detected
6   ; Input      : The input string is specified in myStr
7   ; Output     : The port P1OUT displays the number of E's in the string
8   ; Author     : A. Milenkovic, milenkovic@computer.org
9   ; Date      : August 14, 2008
10  ;-------------------------------------------------------------------------------
11          .cdecls C,LIST,"msp430.h"       ; Include device header file
12
13  ;-------------------------------------------------------------------------------
14          .def    RESET                   ; Export program entry-point to
15                                          ; make it known to linker.
16
17  myStr:  .string "HELLO WORLD, I AM THE MSP430!", ''
18  ;-------------------------------------------------------------------------------
19          .text                           ; Assemble into program memory.
20          .retain                         ; Override ELF conditional linking
21                                          ; and retain current section.
22          .retainrefs                     ; And retain any sections that have
23                                          ; references to current section.
24
25  ;-------------------------------------------------------------------------------
```

```
26  RESET:  mov.w   #__STACK_END,SP         ; Initialize stack pointer
27          mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
28
29  ;-------------------------------------------------------------------------------
30  ; Main loop here
31  ;-------------------------------------------------------------------------------
32  main:   bis.b   #0FFh,&P1DIR            ; configure P1.x output
33          mov.w   #myStr, R4              ; load the starting address of the string into R4
34          clr.b   R5                      ; register R5 will serve as a counter
35  gnext:  mov.b   @R4+, R6                ; get a new character
36          cmp     #0,R6                   ; is it a null character
37          jeq     lend                    ; if yes, go to the end
38          cmp.b   #'E',R6                 ; is it an 'E' character
39          jne     gnext                   ; if not, go to the next
40          inc.w   R5                      ; if yes, increment counter
41          jmp     gnext                   ; go to the next character
42
43  lend:   mov.b   R5,&P1OUT               ; set all P1 pins (output)
44          bis.w   #LPM4,SR                ; LPM4
45          nop                             ; required only for Debugger
46
47
48  ;-------------------------------------------------------------------------------
49  ; Stack Pointer definition
50  ;-------------------------------------------------------------------------------
51          .global __STACK_END
52          .sect   .stack
53
54  ;-------------------------------------------------------------------------------
55  ; Interrupt Vectors
56  ;-------------------------------------------------------------------------------
57           .sect   ".reset"               ; MSP430 RESET Vector
58           .short  RESET
59           .end
```

Figure 10. MSP430 Assembly Code for Count Character Program.

5 Subroutines

In a given program, it is often needed to perform a particular sub-task many times on different data values. Such a subtask is usually called a subroutine. For example, a subroutine may sort numbers in an integer array or perform a complex mathematical operation on an input variable (e.g., calculate sin(x)). It should be noted, that the block of instructions that constitute a subroutine can be included at every point in the main program when that task is needed. However, this would be an unnecessary waste of memory space. Rather, only one copy of the instructions that constitute the subroutine is placed in memory and any program that requires the use of the subroutine simply branches to its starting location in memory. The instruction that performs this branch is named a CALL instruction. The calling program is called CALLER and the subroutine invoked is called CALLEE.

The instruction that is executed right after the CALL instruction is the first instruction of the subroutine. The last instruction in the subroutine is a return instruction (ret), and we say that the subroutine returns to the program that called it. Since a subroutine can be called from different places in a calling program, we must have a mechanism to return to the appropriate

location (the first instruction that follows the CALL instruction in the calling program). At the time of executing a CALL instruction we know the program location of the instruction that follows the CALL (the program counter or PC is pointing to the next instruction). Hence, we should save the return address at the time the CALL instruction is executed. The way in which a machine makes it possible to call and return from subroutines is referred to as its *subroutine linkage method*.

The simplest subroutine linkage method is to save the return address in a specific location. This location may be a register dedicated to this function, often referred to as the *link register*. When the subroutine completes its task, the return instruction returns to the calling program by branching indirectly through the link register.
The CALL instruction is a special branch instruction and performs the following operations:

- Store the content of the PC in the link register;
- Branch to the target address specified by the instruction.

The RETURN instruction is a special branch instruction that performs the following operations:

- Branch to the address contained in the link register.

5.1 Subroutine Nesting

A common programming practice, called subroutine nesting, is to have one subroutine call another. In this case, the return address of the second call is also stored in the link register destroying the previous contents. Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Subroutine nesting can be carried out to any depth. For example, imagine the following sequence: subroutine A calls subroutine B, subroutine B calls subroutine C, and finally subroutine C calls subroutine D. In this case, the last subroutine D completes its computations and returns to the subroutine C that called it. Next, C completes its execution and returns to the subroutine B that called it and so on. The sequence of returns is as follows: D returns to C, C returns to B, and B returns to A. That is, the return addresses are generated and used in the last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. Many processors do this automatically. A particular register is designated as the stack pointer, or SP, that is implicitly used in this operation. The stack pointer points to a stack called the processor stack.
The CALL instruction is a special branch instruction and performs the following operations:

- Push the content of the PC onto the top of the stack
- Update the stack pointer (SP $\leftarrow$ SP – 2)
- Branch to the target address specified by the instruction

The RET instruction is a special branch instruction that performs the following operations:

- Pop the return address from the top of the stack into the PC
- Update the stack pointer (SP $\leftarrow$ SP + 2).

5.2 Parameter Passing

When calling a subroutine, a calling program needs a mechanism to provide to the subroutine the input parameters, the operands that will be used in computation in the subroutine or their addresses. Later, the subroutine needs a mechanism to return output parameters, the results

of the subroutine computation. This exchange of information between a calling program and a subroutine is referred to as *parameter passing*. Parameter passing may be accomplished in several ways. The parameters can be placed in registers or in memory locations, where they can be accessed by subroutine. Alternatively, the parameters may be placed on a processor stack.

Let us consider the following program shown in Figure 11. We have two integer arrays arr1 and arr2. The program finds the sum of the integers in arr1 and displays the result on the ports P1 and P2, and then finds the sum of the integers in arr2 and displays the result on the ports P3 and P4. It is obvious that we can have a single subroutine that will perform this operation and thus make our code more readable and reusable. The subroutine needs to get three input parameters: what is the starting address of the input array, how many elements the array has, and where to display the result. In this example, the subroutine does not return any output parameter to the calling program.

Let us first consider the main program (Figure 12) and the corresponding subroutine (suma_rp, Figure 13) if we pass the parameters through registers. Passing parameters through registers is straightforward and efficient. Three input parameters are placed in registers as follows: R12 keeps the starting address of the input array, R13 keeps the array length, and R14 defines the display identification (#0 for P1&P2 and #1 for P3&P4). The calling program places the parameters in these registers, and then calls the subroutine using `CALL #suma_rp` instruction. The subroutine uses the R7 register to hold the sum of the integers in the array. The register R7 may contain valid data that belongs to the calling program, so our first step should be to push the content of the R7 register onto the stack. The last instruction before the return from the subroutine is to restore the original content of R7. Generally, it is a good practice to save all general-purpose registers used as temporary storage in the subroutine as the first thing in the subroutine, and to restore their original contents (the contents pushed on the stack at the beginning of the subroutine) just before returning from the subroutine. This way, the calling program will find the original contents of the registers as they were before executing the CALL instruction. Other registers that our subroutine uses are R12, R13, and R14. These registers keep parameters, so we assume we can modify them (they do not need to preserve their original value once we are back in the calling program).

```
1  ;-------------------------------------------------------------------------------
2  ; File       : Lab5_D1.asm (CPE 325 Lab5 Demo code)
3  ; Function   : Finds a sum of two integer arrays
4  ; Description: The program initializes ports,
5  ;              sums up elements of two integer arrays and
6  ;              display sums on parallel ports
7  ; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
8  ; Output     : P1OUT&P2OU displays sum of arr1, P3OUT&P4OUT displays sum of arr2
9  ; Author     : A. Milenkovic, milenkovic@computer.org
10 ; Date       : September 14, 2008
11 ;-------------------------------------------------------------------------------
12             .cdecls C,LIST,"msp430.h"       ; Include device header file
13
14 ;-------------------------------------------------------------------------------
15             .def    RESET                   ; Export program entry-point to
16                                             ; make it known to linker.
```

```
17  ;-------------------------------------------------------------------------------
18              .text                           ; Assemble into program memory.
19              .retain                         ; Override ELF conditional linking
20                                              ; and retain current section.
21              .retainrefs                     ; And retain any sections that have
22                                              ; references to current section.
23
24  ;-------------------------------------------------------------------------------
25  RESET:       mov.w   #__STACK_END,SP         ; Initialize stack pointer
26  StopWDT:     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
27
28  ;-------------------------------------------------------------------------------
29  ; Main code here
30  ;-------------------------------------------------------------------------------
31  main:       bis.b   #0xFF,&P1DIR            ; configure P1.x as output
32              bis.b   #0xFF,&P2DIR            ; configure P2.x as output
33              bis.b   #0xFF,&P3DIR            ; configure P3.x as output
34              bis.b   #0xFF,&P4DIR            ; configure P4.x as output
35              ; load the starting address of the array1 into the register R4
36              mov.w   #arr1, R4
37              ; load the starting address of the array1 into the register R4
38              mov.w   #arr2, R5
39              ; Sum arr1 and display
40              clr.w   R7                      ; Holds the sum
41              mov.w   #8, R10                 ; number of elements in arr1
42  lnext1:     add.w   @R4+, R7                ; get next element
43              dec.w   R10
44              jnz     lnext1
45              mov.b   R7, P1OUT               ; display sum of arr1
46              swpb    R7
47              mov.b   R7, P2OUT
48              ; Sum arr2 and display
49              clr.w   R7                      ; Holds the sum
50              mov.w   #7, R10                 ; number of elements in arr2
51  lnext2:     add.w   @R5+, R7                ; get next element
52              dec.w   R10
53              jnz     lnext2
54              mov.b   R7, P3OUT               ; display sum of arr1
55              swpb    R7
56              mov.b   R7, P4OUT
57              jmp     $
58
59  arr1:        .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
60  arr2:        .int     1, 1, 1, 1, -1, -1, -1     ; the second array
61
62
63  ;-------------------------------------------------------------------------------
64  ; Stack Pointer definition
65  ;-------------------------------------------------------------------------------
66              .global __STACK_END
67              .sect   .stack
68
69  ;-------------------------------------------------------------------------------
70  ; Interrupt Vectors
71  ;-------------------------------------------------------------------------------
72              .sect   ".reset"                ; MSP430 RESET Vector
73              .short  RESET
74              .end
```

Figure 11. Assembly program for summing up two integer arrays.

```
1   ;-------------------------------------------------------------------------------
2   ; File       : Lab5_D2_main.asm (CPE 325 Lab5 Demo code)
3   ; Function   : Finds a sum of two integer arrays using subroutines
4   ; Description: The program initializes ports and
5   ;              calls suma_rp to sum up elements of integer arrays and
6   ;              display sums on parallel ports.
7   ;              Parameters to suma_rp are passed through registers, R12, R13, R14.
8   ; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
9   ; Output     : P1OUT&P2OU displays sum of arr1, P3OUT&P4OUT displays sum of arr2
10  ; Author     : A. Milenkovic, milenkovic@computer.org
11  ; Date       : September 14, 2008
12  ;-------------------------------------------------------------------------------
13              .cdecls C,LIST,"msp430.h"       ; Include device header file
14
15  ;-------------------------------------------------------------------------------
16              .def    RESET                   ; Export program entry-point to
17                                              ; make it known to linker.
18              .ref    suma_rp
19  ;-------------------------------------------------------------------------------
20              .text                           ; Assemble into program memory.
21              .retain                         ; Override ELF conditional linking
22                                              ; and retain current section.
23              .retainrefs                     ; And retain any sections that have
24                                              ; references to current section.
25
26  ;-------------------------------------------------------------------------------
27  RESET:      mov.w   #__STACK_END,SP         ; Initialize stackpointer
28  StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
29
30  ;-------------------------------------------------------------------------------
31  ; Main code here
32  ;-------------------------------------------------------------------------------
33  main:       bis.b   #0xFF,&P1DIR            ; configure P1.x as output
34              bis.b   #0xFF,&P2DIR            ; configure P2.x as output
35              bis.b   #0xFF,&P3DIR            ; configure P3.x as output
36              bis.b   #0xFF,&P4DIR            ; configure P4.x as output
37
38              mov.w   #arr1, R12              ; put address into R12
39              mov.w   #8, R13                 ; put array length into R13
40              mov.w   #0, R14                 ; display #0 (P1&P2)
41              call    #suma_rp
42
43              mov.w   #arr2, R12              ; put address into R12
44              mov.w   #7, R13                 ; put array length into R13
45              mov.w   #1, R14                 ; display #0 (P3&P4)
46              call    #suma_rp
47              jmp     $
48
49  arr1:        .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
50  arr2:        .int     1, 1, 1, 1, -1, -1, -1     ; the second array
51
52  ;-------------------------------------------------------------------------------
53  ; Stack Pointer definition
54  ;-------------------------------------------------------------------------------
55              .global __STACK_END
56              .sect   .stack
57
58  ;-------------------------------------------------------------------------------
59  ; Interrupt Vectors
60  ;-------------------------------------------------------------------------------
```

```
61              .sect   ".reset"                ; MSP430 RESET Vector
62              .short  RESET
63              .end
```

Figure 12. Main assembly program for summing up two integer arrays using a subroutine suma_rp.

```
 1   ;-------------------------------------------------------------------------------
 2   ; File       : Lab5_D2_RP.asm (CPE 325 Lab5 Demo code)
 3   ; Function   : Finds a sum of an input integer array
 4   ; Description: suma_rp is a subroutine that sums elements of an integer array
 5   ; Input      : The input parameters are:
 6   ;                   R12 -- array starting address
 7   ;                   R13 -- the number of elements (>= 1)
 8   ;                   R14 -- display ID (0 for P1&P2 and 1 for P3&P4)
 9   ; Output     : No output
10   ; Author     : A. Milenkovic, milenkovic@computer.org
11   ; Date       : September 14, 2008
12   ;-------------------------------------------------------------------------------
13              .cdecls C,LIST,"msp430.h"       ; Include device header file
14
15              .def suma_rp
16
17              .text
18
19   suma_rp:
20              push.w  R7              ; save the register R7 on the stack
21              clr.w   R7              ; clear register R7 (keeps the sum)
22   lnext:     add.w   @R12+, R7       ; add a new element
23              dec.w   R13             ; decrement step counter
24              jnz     lnext           ; jump if not finished
25              bit.w   #1, R14         ; test display ID
26              jnz     lp34            ; jump on lp34 if display ID=1
27              mov.b   R7, P1OUT       ; display lower 8-bits of the sum on P1OUT
28              swpb    R7              ; swap bytes
29              mov.b   R7, P2OUT       ; display upper 8-bits of the sum on P2OUT
30              jmp     lend            ; skip to end
31   lp34:      mov.b   R7, P3OUT       ; display lower 8-bits of the sum on P3OUT
32              swpb    R7              ; swap bytes
33              mov.b   R7, P4OUT       ; display upper 8-bits of the sum on P4OUT
34   lend:      pop     R7              ; restore R7
35              ret                     ; return from subroutine
36
37              .end
```

Figure 13. Subroutine for summing up an integer array (suma_rp).

If many parameters are passed, there may not be enough general-purpose register available for passing parameters into the subroutine. In this case we use the stack to pass parameters. Figure 14 shows the calling program and Figure 15 shows the subroutine. Before calling the subroutine we place parameters on the stack using PUSH instructions (the array starting address, array length, and display id – each parameter is 2 bytes long). The CALL instruction pushes the return address on the stack. The subroutine then stores the contents of the registers R7, R6, and R4 on the stack (another 6 bytes) to save their original content. The next step is to retrieve input parameters (array starting address and array length). They are on the stack, but to know exactly where, we need to know the current state of the stack and its organization (how does it grow, and where does SP point to). The total distance between the top of the stack and the location on the stack where we placed the starting address is 12 bytes.

So the instruction `mov.w 12(SP), R4` loads the register R4 with the first parameter (the array starting address). Similarly, the array length can be retrieved by `mov.w 10(SP), R6`. The register values are restored before returning from the subroutine (notice the reverse order of POP instructions). Once we are back in the calling program, we can free 6 bytes on the stack used to pass parameters.

```
1  ;-------------------------------------------------------------------------------
2  ; File       : Lab5_D3_main.asm (CPE 325 Lab5 Demo code)
3  ; Function   : Finds a sum of two integer arrays using a subroutine suma_sp
4  ; Description: The program initializes ports and
5  ;              calls suma_rp to sum up elements of integer arrays and
6  ;              display sums on parallel ports.
7  ;              Parameters to suma_sp are passed through the stack.
8  ; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
9  ; Output     : P1OUT&P2OU displays sum of arr1, P3OUT&P4OUT displays sum of arr2
10 ; Author     : A. Milenkovic, milenkovic@computer.org
11 ; Date       : September 14, 2008
12 ;-------------------------------------------------------------------------------
13             .cdecls C,LIST,"msp430.h"           ; Include device header file
14
15 ;-------------------------------------------------------------------------------
16             .def    RESET                       ; Export program entry-point to
17                                                 ; make it known to linker.
18             .ref    suma_sp
19 ;-------------------------------------------------------------------------------
20             .text                               ; Assemble into program memory.
21             .retain                             ; Override ELF conditional linking
22                                                 ; and retain current section.
23             .retainrefs                         ; And retain any sections that have
24                                                 ; references to current section.
25 ;-------------------------------------------------------------------------------
26 RESET:      mov.w   #__STACK_END,SP             ; Initialize stackpointer
27 StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL      ; Stop watchdog timer
28
29 ;-------------------------------------------------------------------------------
30 ; Main code here
31 ;-------------------------------------------------------------------------------
32 main:       bis.b   #0xFF,&P1DIR                ; configure P1.x as output
33             bis.b   #0xFF,&P2DIR                ; configure P2.x as output
34             bis.b   #0xFF,&P3DIR                ; configure P3.x as output
35             bis.b   #0xFF,&P4DIR                ; configure P4.x as output
36
37             push    #arr1                       ; push the address of arr1
38             push    #8                          ; push the number of elements
39             push    #0                          ; push display id
40             call    #suma_sp
41             add.w   #6,SP                       ; collapse the stack
42             push    #arr2                       ; push the address of arr1
43             push    #7                          ; push the number of elements
44             push    #1                          ; push display id
45             call    #suma_sp
46             add.w   #6,SP                       ; collapse the stack
47
48             jmp     $
49
50 arr1:       .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
51 arr2:       .int     1, 1, 1, 1, -1, -1, -1     ; the second array
52
53 ;-------------------------------------------------------------------------------
```

```
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

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

Figure 14 . Main program for summing up two integer arrays using a subroutine suma_sp.

```
;-------------------------------------------------------------------------------
; File       : Lab5_D3_SP.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_sp is a subroutine that sums elements of an integer array
; Input      : The input parameters are on the stack pushed as follows:
;                 starting addrress of the array
;                 array length
;                 display id
; Output     : No output
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

            .def    suma_sp

            .text
suma_sp:
                                            ; save the registers on the stack
            push    R7                      ; save R7, temporal sum
            push    R6                      ; save R6, array length
            push    R4                      ; save R5, pointer to array
            clr.w   R7                      ; clear R7
            mov.w   10(SP), R6              ; retrieve array length
            mov.w   12(SP), R4              ; retrieve starting address
lnext:      add.w   @R4+, R7                ; add next element
            dec.w   R6                      ; decrement array length
            jnz     lnext                   ; repeat if not done
            mov.w   8(SP), R4               ; get id from the stack
            bit.w   #1, R4                  ; test display id
            jnz     lp34                    ; jump to lp34 display id = 1
            mov.b   R7, P1OUT               ; lower 8 bits of the sum to P1OUT
            swpb    R7                      ; swap bytes
            mov.b   R7, P2OUT               ; upper 8 bits of the sum to P2OUT
            jmp     lend                    ; jump to lend
lp34:       mov.b   R7, P3OUT               ; lower 8 bits of ths sum to P3OUT
            swpb    R7                      ; swap bytes
            mov.b   R7, P4OUT               ; upper 8 bits of the sum to P4OUT
lend:       pop     R4                      ; restore R4
            pop     R6                      ; restore R6
            pop     R7                      ; restore R7
            ret                             ; return

            .end
```

Figure 15. Subroutine for summing up an integer array (parameters are passed through the stack).

6 Allocating Space for Local Variables

Subroutines often need local workspace. So far we have looked at relatively simple subroutines and managed to develop them by using only a subset of general-purpose registers to keep local variables. However, what are we going to do if we have arrays and matrices as local variables in subroutines? Allocating space in RAM memory is an obvious solution, but the question is how to manage such a space. Assigning a portion of RAM residing at a fixed address is not a good option. First, it will make our code tied to this particular address – different members of MSP430 family may have different sizes and placement of RAM memory, so the code may not be portable. In addition, subroutines may be called recursively, so that multiple instances of local variables need to be kept separately. Clearly, having a reserved portion of RAM at a fixed address is not amiable for relocatable, reentrant, and recursive subroutines. The solution is to use so-called dynamic allocation. Local variables in a subroutine (those that exist during the lifetime of subroutines) are allocated on the stack once we enter the subroutine and de-allocated (removed) form the stack before we exit the subroutine.
The storage allocated by a subroutine for local storage is called *stack frame*. To manage space on the stack frame we typically use a general purpose register that acts as stack frame pointer. Let us assume we want to use register R12 as the frame pointer and that we want to allocate local space for an integer array of 10 elements (20 bytes in MSP430). The first step done in the subroutine is to push the R12 onto the stack and then redirect R12 to point to the current top of the stack (where its original copy is kept on the stack). After that, allocating space is simply moving the stack pointer 20 bytes below the current top of the stack. The following sequence of instructions performs required operations.

```
mysub:  PUSH    R12     ; save R12
        MOVE.W  SP, R12 ; R12 points to TOS
        SUB.W   #20, SP ; allocate 20 bytes for local storage
```

This way register R12 can act as an anchor or address register through which we can reach input variables on the stack (residing on the stack above the anchor) or local variables (residing on the stack below the anchor in the stack frame). One advantage of this approach is that we do not have to use SP to reach local variables or input parameters. Using SP to reach input and local variables requires developers to track distance between the current SP and locations of interest on the stack, which could be a burden when the stack is growing or shrinking dynamically inside the subroutine. Register R12 is anchored and does not change its value during subroutine execution.

Before we exit the subroutine, we need to de-allocate the local stack frame and restore the original value of R12 as follows.

```
        MOV.W   R12, SP ; free the stack frame
        POP.W   R12     ; restore R12
        RET             ; retrieve the return address from the stack
```

To demonstrate practical use of the stack frame we will rewrite the subroutine for summing up elements of an integer array from Figure 15. This time we assume that the total array sum and the loop counter are not kept in general-purpose registers, but rather are local variables for the

subroutine kept on the stack frame. Figure 16 shows the subroutine sum_spsf that expects the input parameters passed through the stack, but allocates 4 bytes in the stack frame for the total sum (at the address SFP+4) and the counter (at the address SFP+2).

```
;-------------------------------------------------------------------------------
; File       : Lab5_D4_SPSF.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_spsf is a subroutine that sums elements of an integer array.
;              The subroutine allocates local variables on the stack:
;                  counter (SFP+2)
;                  sum (SFP+4)
; Input      : The input parameters are on the stack pushed as follows:
;                  starting address of the array
;                  array length
;                  display id
; Output     : No output
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

            .def    suma_spsf

             .text
suma_spsf:
        ; save the registers on the stack
        push    R12             ; save R12 - R12 is stack frame pointer
        mov.w   SP, R12         ; R12 points on the bottom of the stack frame
        sub.w   #4, SP          ; allocate 4 bytes for local variables
        push    R4              ; pointer register
        clr.w   -4(R12)         ; clear sum, sum=0
        mov.w    6(R12), -2(R12) ; get array length
        mov.w   8(R12), R4      ; R4 points to the array starting address
lnext:  add.w   @R4+, -4(R12)   ; add next element
        dec.w   -2(R12)         ; decrement counter
        jnz     lnext           ; repeat if not done
        bit.w   #1, 4(R12)      ; test display id
        jnz     lp34            ; jump to lp34 if display id = 1
        mov.b   -4(R12), P1OUT  ; lower 8 bits of the sum to P1OUT
        mov.b   -3(R12), P2OUT  ; upper 8 bits of the sume to P2OUT
        jmp     lend            ; skip to lend
lp34:   mov.b   -4(R12), P3OUT  ; lower 8 bits of the sum to P3OUT
        mov.b   -3(R12), P4OUT  ; upper 8 bits of the sume to P4OUT
lend:   pop     R4              ; restore R4
        add.w   #4, SP          ; collapse the stack frame
        pop     R12             ; restore stack frame pointer
        ret                     ; return
        .end
```

Figure 16. Subroutine for summing up an integer array that uses local variables sum and counter allocated on the stack.

7 To Learn More

1. MSP430 User Manual, http://www.ece.uah.edu/~milenka/npage/data/cpe323/Documents/slau056j-4xx-UG.pdf
2. Textbook, Chapter 5