CPE 323
MODULE 04
MSP430 ASSEMBLY LANGUAGE PROGRAMMING

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module introduces the MSP430 assembly language. The closest to the ISA a programmer can be is by writing assembly language programs. In this module you will learn about software developer flows for embedded systems, assembly language directives that help you allocate space in memory and initialize your constants, structure and organization of assembly language programs, subroutines, allocating space on the stack, and passing parameters to subroutines.

Objectives

Upon completion of this module learners will be able to:

- *Apply a proper design flow for embedded software using assembly language programming*
- *Read and comprehend functionality of programs written in MSP430 assembly language*
- *Design and write programs in MSP430 assembly language to solve specific tasks*

Contents

1 Introduction

This module introduces the MSP430 assembly language. The closest to the ISA a programmer can be is by writing assembly language programs that run on a bare-metal hardware platform. Bare-metal expression is used to indicate that the platform has no operating system running on it. In this module you will learn about software design flows for embedded systems, assembly language directives that help you allocate space in memory and initialize your constants, structure and organization of assembly language programs, subroutines, allocating space on the stack, and passing parameters to subroutines. These topics are introduced using several illustrative examples.

2 Embedded software development environment and design flow

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 that is used in this course. 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 to improve its 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 controls execution of the program on the target platform. This typically involves 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.

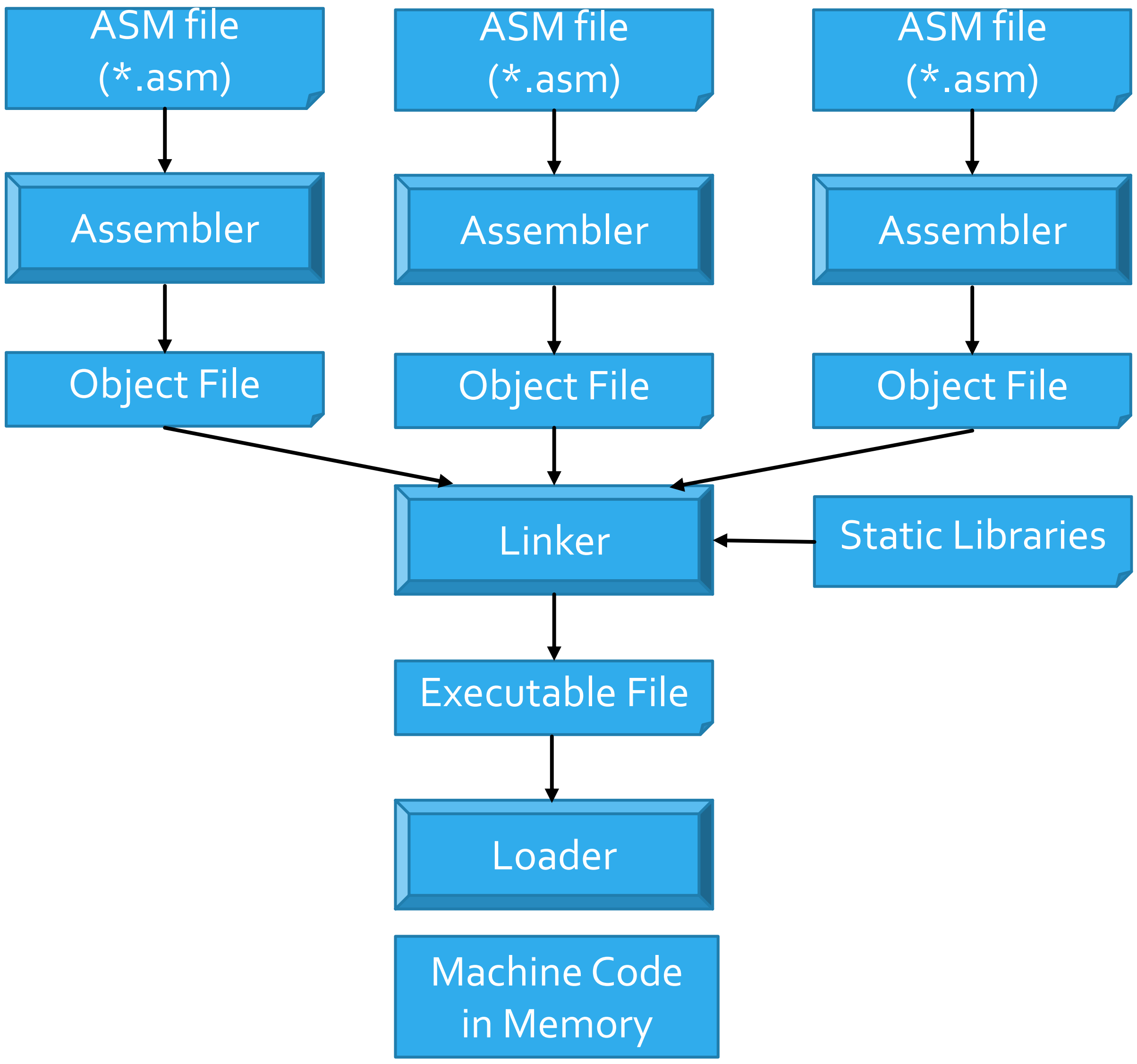


Figure 1. Design flow for embedded software: from assembly programs to machine code in memory.

Things to remember 2-1. Design flow.

Understand the design flow and the role of various components in a software development environment, namely assembler, linker, and loader.

3 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) similarly 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 and A430.

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 3-1. Allocating space in RAM memory using IAR and CCS assemblers.
; 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 2 bytes / 1 word
MyByte: .usect “.bss”, 1     ; allocate 1 byte

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

An example in Code 1 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 with ASCII code 0x00 that is added at the end of the string “ABCD”. Table 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 in case of MSP430FG4618 (the start address of the flash memory). As a result of parsing this sequence, the assembler creates a table of symbols (synonyms) shown in Table 4. The table of symbols is an internal structure of the assembler and it helps resolve symbolic names used in assembly language programs.

```
1   ;-------------------------------------------------------------------------------
2   ; define data section with constants
3
4   b1:         .byte   5         ; allocates a byte in memory and initialize it with 5
```

```
b2:         .byte   -122       ; allocates a byte with constant -122
b3:         .byte   10110111b  ; binary value of a constant
b4:         .byte   0xA0       ; hexadecimal value of a constant
b5:         .byte   123q       ; octal value of a constant
tf:         .equ 25
            .align 2           ; move a location pointer to the first even address
w1:         .word   21         ; allocates a word constant in memory;
w2:         .word  -21
w3:         .word tf
dw1:        .long  100000      ; allocates a long word size constant in memory;
                               ; 100000 (0x0001_86A0)
dw2:        .long 0xFFFFFFEA
            .align 2
s1:         .byte 'A', 'B', 'C', 'D'    ; allocates 4 bytes in memory with string ABCD
s2:         .byte "ABCD", ''   ; allocates 5 bytes in memory with string ABCD + NULL
```

Code 1. Assembly language directives for allocating constants (or initialized data section in RAM).

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

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

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

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

To allocate space in RAM memory we use directives as shown in Code 2. Table 5 shows the content of the RAM after allocation (it is not initialized) and Table 6 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 is the starting address of the RAM memory in 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 as they are still not initialized. Note #3: When using .data directive that defines initialized variables in RAM, the assembler will automatically generate machine instructions that are responsible for initializing these locations. These machine instructions are carried out before your program execution starts.

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

Code 2. Assembly language directives for allocating space in memory that is not initialized.

Table 5. Memory content (a word-view).

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

Table 6. Table of symbols (maintained by assembler).

Symbol	Value [hex]

v1b	0x1100
v2b	0x1101
v3w	0x1102
v4b	0x1104
vx	0x110C

`Things to remember 3-1. Assembly language directives.`

Assembly language directives help developers organize their software: they tell the assembler to set the data and program at particular addresses (code sections), allocate space in memory and initialize constants, allocate space in memory for uninitialized variables, define synonyms, or include additional files.

`Things to remember 3-2. Common assembly language directives for allocating and initializing constants.`

Commonly used assembly language directives for allocating memory and initializing are .byte, .word, .long, .float. To align current location pointer to a first even address we use .align 2.

4 Decimal and Integer Addition of 32-bit Integers

In this section we will design an assembly language 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). Our first step is to define the input variables lint1 and lint2 as shown in Code 3, lines 27 and 28. We use `.long` directive followed by their values given in the hexadecimal number system. They are defined as constants and the assembler will place these variables at the first address that belongs to the Flash memory (0x4400 in case of MSP430F5529 or 0x3100 in case of MSP430FG4618). Thus, lint1 is at the memory location with address 0x4400 and lint2 is at 0x0x4404. Assume they are initialized as shown in Code 3 (lines 27 and 28). Table 7 illustrates the content of relevant memory locations before the program execution starts. The output variables lsumd and lsumi are allocated in RAM as shown in lines 31 and 32. Please note that the results cannot be written into the Flash memory which is considered as read-only memory. These variables are stored at the starting address of RAM, which is 0x2400 (lsumd) and 0x2404 (lsumi) in case of MSP430F5529. Now, we have our variables taken care of, we can move on designing the program.

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), where C is the current value of the Carry bit from the program status register (R2). 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 a correct sum. Luckily, the two move instructions in between do not affect the Carry flag, so that the Carry flag produced in line 41 remains unchanged until it is used by DADD.W in line 44. Consequently, to make this code work properly, we clear carry status register R2 before we start computation (line 39).

Table 7. Memory content (a word-view) before the start of program execution.

Label	Address	Memory[15:0]
lsumd	0x2400	0x????
	0x2402	0x????
lsumi	0x2404	0x????
	0x2406	0x????
	. . .	
	. . .	
lint1	0x4400	0x8923
	0x4402	0x4567
lint2	0x4404	0x6789
	0x4406	0x2345

A similar sequence of steps is performed for binary addition. Here, we use the ADD.W instruction in the first round and the ADDC.W instruction in the second round instead of DADD.W instructions. Also, note that the Carry flag generated by the ADD.W instruction in line 47 is used by the ADDC in line 50. The upper 16-bits of the result is written back to lsumi+2 in line 51.

```
 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
```

```
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .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.
;-------------------------------------------------------------------------------
;-------------------------------------------------------------------------------
lint1:         .long 0x45678923
lint2:         .long 0x23456789
;-------------------------------------------------------------------------------
;-------------------------------------------------------------------------------
lsumd:         .usect ".bss", 4,2       ; allocate 4 bytes for decimal result
lsumi:         .usect ".bss", 4,2       ; allocate 4 bytes for integer result
;-------------------------------------------------------------------------------
RESET:      mov.w   #__STACK_END,SP         ; Initialize stack pointer
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
              clr.w   R2             ; clear status register
              mov.w   lint1, R8      ; get lower 16 bits from lint1 to R8
              dadd.w  lint2, R8      ; decimal addition, R8 + lower 16-bit of lint2
              mov.w   R8, lsumd      ; store the result (lower 16-bit)
              mov.w   lint1+2, R8    ; get upper 16 bits of lint1 to R8
              dadd.w  lint2+2, R8    ; decimal addition
              mov.w   R8, lsumd+2    ; store the result (upper 16-bit)
              mov.w   lint1, R8      ; get lower 16 bite from lint1 to R8
              add.w   lint2, R8      ; integer addition
              mov.w   R8, lsumi      ; store the result (lower 16 bits)
              mov.w   lint1+2, R8    ; get upper 16 bits from lint1 to R8
              addc.w  lint2+2, R8    ; add upper words, plus carry
              mov.w   R8, lsumi+2    ; store upper 16 bits of the result

              jmp $                  ; jump to current location '$'
                                     ; (endless loop)


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

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

```

```
69
```

Code 3. Decimal and integer addition (first implementation).

What happens when we are done with our computation? Normally, the program exits and returns control to the underlying shell program. However, in embedded systems our programs typically run forever – as long as power supply is provided. Also, typically there is no shell program to return to. Consequently, we add an extra instruction in line 53 that is basically an infinite loop (jump to itself). Once we compute the sums, our program remains stuck in this loop. Why is this necessary? To answer this question, ask yourself what will happen without this line of code. Our processor will continue to fetch and execute instructions sequentially, regardless of whether we have useful instructions in the flash memory or some random content. To prevent this uncontrolled behavior, we terminate a program by entering this infinite loop.

Now that we explained how we terminate program execution, let us describe how the program execution starts. Upon powering up, a so-called PUC (power-up clear) signal in hardware is generated. The first thing MSP430 does as a response to PUC is to fetch a word from location 0xFFFE (the top word address in the first 64 KiB of address space). This location is known as the ***Reset interrupt vector***. Note: the top 32 words of the 64 KiB 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 the Program Counter (PC <= M[0xFFFE]). Note that our entry point in the program (address of the first instruction) has label RESET (line 37), i.e., RESET is a symbolic name of the starting address of our program. The value of the symbol RESET is used to initialize the reset vector in the interrupt vector table (lines 82 and 83). Thus, when we power up the MSP430-based system, the PUC interrupt is generated and as a result of that, the starting address of the program is moved into the PC (R0), so the MSP430 starts execution from the first instruction in our program.

Code 4 shows a program ready to run with all necessary directives and an alternative implementation from the one shown in Code 3. In this implementation two rounds of adding up components of a long integer are carried out in a loop. In this implementation we initialize the register R4 to contain the starting address of the first integer lint1 (line 44) and the register R8 to contain the starting address of the decimal sum (lsumd, line 45). Please note what addressing mode we use to load the address of the operands lint1 and lsumd in registers R4 and R8, respectively. Register R5 acts as a step counter and it is initialized to value 2 (we have two rounds).

The first instruction of the loop in line 48 moves a word from the address 4+R4 into the register R7. At the address 4+R4, we will find lower 16 bits of the integer lint2. Decimal addition is carried out in line 50. Please note that we use the autoincrement addressing mode to read the lower 16 bit of the variable lint1, so the register R4 is automatically adjusted to point to the upper word of lint1 once this instruction is executed. The sum from the register R7 is moved to the memory location reserved for lsumd (line 52). You may wonder why do we have instructions in lines 49 and 51? They respectively restore and back up the content of the status register R2. As we now have a bit more complex sequence of instructions, the instruction DADD.W in line 50 of the first iteration produces the Carry bit that should be used by the same instruction in the second iteration of the loop. Unfortunately, our code increments the address register R8 (R8<=R8+2) and decrements the step counter (R5<=R5-1) and both of these instructions will overwrite the Carry bit set by DADD.W. That is the reason that we are backing up the content of the status register that contains the valid Carry bit in line 51 and restoring it in line 49, so that the DADD.W instruction in line 50 gets the correct value of the Carry bit. Similar reasoning is used for integer addition.

The added benefit of this implementation is that now you can solve a problem of summing up of any size integers. For example, if you have 128-bit long integers, you can just make a couple of changes and set the step counter to 128/16=8 instead of 2 in this implementation and this code will give you a correct sum.

```
 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 stack pointer
38  StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
```

```

;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------
; Decimal addition
            mov.w   #lint1, R4              ; pointer to lint1
            mov.w   #lsumd, R8              ; pointer to lsumd
            mov.w   #2, R5                  ; R5 is a counter (32-bit=2x16-bit)
            clr.w   R10                     ; clear R10
lda:        mov.w   4(R4), R7               ; load lint2
            mov.w   R10, R2                 ; load original SR
            dadd.w  @R4+, R7                ; decimal add lint1 (with carry)
            mov.w   R2, R10                 ; backup R2 in R10
            mov.w   R7, 0(R8)               ; store result (@R8+0)
            add.w   #2, R8                  ; update R8
            dec.w   R5                      ; decrement R5
            jnz     lda                     ; jump if not zero to lda
; Integer addition
            mov.w   #lint1, R4              ; pointer to lint1
            mov.w   #lsumi, R8              ; pointer to lsumi
            mov.w   #2, R5                  ; R5 is a counter (32-bit=2x16-bit)
            clr.w   R10                     ; clear R10
lia:        mov.w   4(R4), R7               ; load lint2
            mov.w   R10, R2                 ; load original SR
            addc.w  @R4+, R7                ; decimal add lint1 (with carry)
            mov.w   R2, R10                 ; backup R2 in R10
            mov.w   R7, 0(R8)               ; store result (@R8+0)
            add.w   #2, R8                  ; update R8
            dec.w   R5                      ; decrement R5
            jnz     lia                     ; jump if not zero to lia

            jmp     $                       ; jump to current location '$'
                                            ; (endless loop)

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

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

```

Code 4. Decimal and integer 32-bit addition.

Things to remember 4-1. Assembly program structure.

Understand structure of an assembly program. How do we allocate and initialize input variables? How do we define entry point in the program to be executed? How do we analyze a problem to be solved, solve it, and implement the solution using assembly language program?

5 Counting Characters 'E' in a String

In this section we will design a program that will count the number of characters 'E' in a string 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 (2 in this example). 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 problem, let us first analyze the problem statement. First, the problem implies that we need to allocate space in memory that will keep the string. 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, e.g.: `.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 the 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 next step is 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 matches 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 character 'E' is encountered.

After we exit the program loop, the current value of the counter will be written to the port P1, which should be initialized at the beginning of the program 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.

Code 5 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 `mySt`r using `.string` directive: `mySt`r `.string "HELLO WORLD, I AM THE MSP430!", ''`. As explained above, this directive will allocate 30 bytes in memory starting at the address 0x3100 (when using MSP430FG4618) 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('E')=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 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 program 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 the watchdog timer.

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 gnext label (line 35). The mov.b instruction reads a character from the string and moves it to the register R6 (lower 8 bits). We use the autoincrement addressing mode for the source operand to adjust the pointer to point to the next character in the string. As this is a byte instruction, **`mov.b`** `@R4+, R6`, the register R4<=R4+1 (we increment the address register for the size of the operand in bytes, it is 1 in this case). 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 found the character 'E' and we increase the value of the counter in 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)
```

```
; Function   : Counts the number of characters E in a given string
; Description: Program traverses an input array of characters
;              to detect a character 'E'; exits when a NULL is detected
; Input      : The input string is specified in myStr
; Output     : The port P1OUT displays the number of E's in the string
; Author     : A. Milenkovic, milenkovic@computer.org
; Date      : August 14, 2008
;-------------------------------------------------------------------------------
        .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
        .def    RESET                   ; Export program entry-point to
                                        ; make it known to linker.

myStr:  .string "HELLO WORLD, I AM THE MSP430!", ''
;-------------------------------------------------------------------------------
        .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            ; configure P1.x output
        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   #'E',R6                 ; is it an 'E' character
        jne     gnext                   ; if not, go to the next
        inc.w   R5                      ; if yes, increment counter
        jmp     gnext                   ; go to the next character

lend:   mov.b   R5,&P1OUT               ; set all P1 pins (output)
        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
```

Code 5. MSP430 Assembly Code for Count Character Program.

6 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 called 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, 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 the 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:

- *Stores the contents of the PC in the link register*
- *Branches to the target address specified by the instruction.*

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

- *Branches to the address contained in the link register.*

6.1 Subroutine Nesting

A common programming practice, called *subroutine nesting*, is to have one subroutine calls 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. Consider an example in Figure 2. The main program calls subroutine A (subA), the subA calls the subroutine B (subB), the subB calls the subroutine C. In this case, the sequence of instructions executed follows the arrows shown in Figure 2. The last subroutine C completes its computations and returns to the subroutine B that called it. Next, B completes its execution and returns to the subroutine A that called it. Finally, the subroutine A returns to the main. The sequence of return addresses are generated 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.

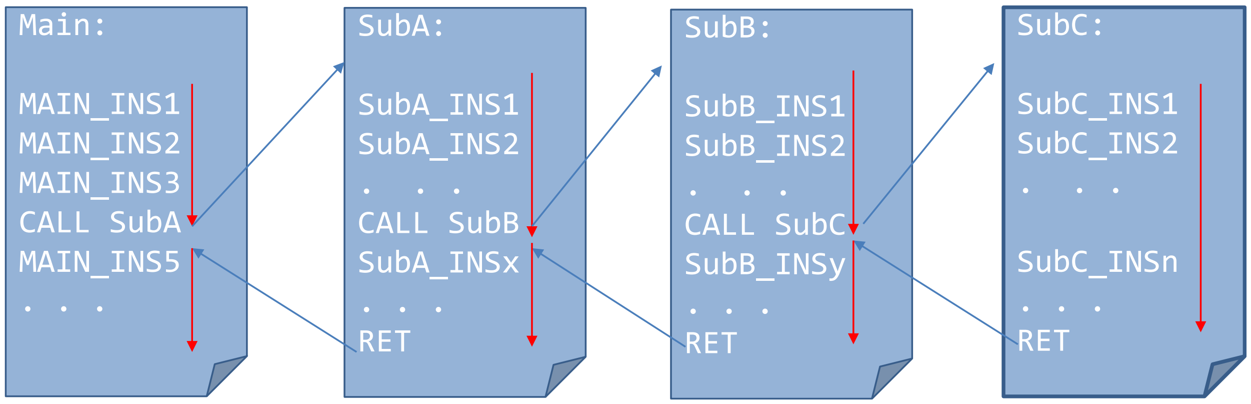


Figure 2. Illustration of subroutine nesting.

The CALL instruction is a special branch instruction and performs the following operations:

- *Pushes the contents of the PC on the top of the stack*
- *Updates the stack pointer*
- *Branches to the target address specified by the instruction*

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

- *Pops the return address from the top of the stack into the PC*
- *Updates the stack pointer.*

`Things to remember 6-1. CALL and RETURN instructions.`

The CALL instruction in MSP430 pushes the content of PC to the top of the stack, decrements register SP←SP – 2, and moves the target address specified by the CALL instruction to PC.

The RET instruction in MSP430 retrieves the content from the current top of the stack, and updates SP, SP←SP + 2. Please note that RET is an emulated instruction and it is equivalent to the following instruction: MOV @SP+, PC. Please note that RET instruction should always find the corresponding return address on the top of the stack that is pushed by the corresponding CALL instruction.

6.2 Parameter Passing

When calling a subroutine, a calling program needs a mechanism to provide the input parameters, the operands that will be used in computation in the subroutine or their addresses, to the subroutine. 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 Code 6. 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 two input parameters: what is the starting address of the input array and how many elements the array has. It should return the sum to the caller.

Let us next consider the main program (Code 7) where we pass the parameters through the registers. Passing parameters through the registers is straightforward and efficient. Two input parameters are placed in registers as follows: R12 keeps the starting address of the input array, R13 keeps the array length. The calling program places the parameters in these registers, and then calls the subroutine using the CALL #suma_rp instruction. The subroutine shown in Code 8 uses the register R14 to hold the sum of the array elements and to return the result back to the caller. We do not need any other registers and since all these registers are used in passing parameters, we do not need to push any register onto the stack. However, generally it is a good practice to save all the 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 the CALL instruction.

```
;-------------------------------------------------------------------------------
; File       : Lab5_D1.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of two integer arrays
; Description: The program initializes ports,
;              sums up elements of two integer arrays and
;              display sums on on parallel port output registers
; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
; Output     : P1OUT&P2OUT displays sum of arr1, P3OUT&P4OUT displays sum of arr2
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
```

```
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
            .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
StopWDT:     mov.w   #WDTPW|WDTHOLD,&WDTCTL ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
main:
            ; load the starting address of the array1 into the register R4
            mov.w   #arr1, R4
            ; load the starting address of the array2 into the register R5
            mov.w   #arr2, R5
            ; Sum arr1 and display
            clr.w   R7                      ; holds the sum
            mov.w   #8, R10                 ; number of elements in arr1
lnext1:     add.w   @R4+, R7                ; add the current element to sum
            dec.w   R10                     ; decrement arr1 length
            jnz     lnext1                  ; get next element
            mov.b   R7, P1OUT               ; display lower byte of sum of arr1
            swpb    R7                      ; swap bytes
            mov.b   R7, P2OUT               ; display upper byte of sum of arr1
            ; Sum arr2 and display
            clr.w   R7                      ; Holds the sum
            mov.w   #7, R10                 ; number of elements in arr2
lnext2:     add.w   @R5+, R7                ; get next element
            dec.w   R10                     ; decrement arr2 length
            jnz     lnext2                  ; get next element
            mov.b   R7, P3OUT               ; display lower byte of sum of arr2
            swpb    R7                      ; swap bytes
            mov.b   R7, P4OUT               ; display upper byte of sum of arr2
            jmp     $

arr1:       .int   1, 2, 3, 4, 1, 2, 3, 4   ; the first array
arr2:       .int   1, 1, 1, 1, -1, -1, -1   ; the second array

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

;-------------------------------------------------------------------------------
; Interrupt Vectors
```

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

```

Code 6. Assembly program for summing up two integer arrays.

```
;-------------------------------------------------------------------------------
; File       : Lab5_D2_main.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of two integer arrays using a subroutine.
; Description: The program calls suma_rp to sum up elements of integer arrays and
;              then displays the sum on parallel ports.
;              Parameters to suma_rp are passed through registers, R12, R13.
;              The subroutine suma_rp return the result in register R14.
; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
; Output     : P1OUT&P2OU displays sum of arr1, P3OUT&P4OUT displays sum of arr2
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008 (revised August 2020)
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
            .ref    suma_rp
;-------------------------------------------------------------------------------
            .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 stackpointer
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
main:
            mov.w   #arr1, R12              ; put address into R12
            mov.w   #8, R13                 ; put array length into R13
            call    #suma_rp
            ; P1OUT is at address 0x02, P2OUT is address 0x03
            ; we can write the 16-bit result to both at the same time
            ; P2OUT contains the upper byte and P1OUT the lower byte
            mov.w   R14, &P1OUT             ; result goes to P2OUT&P1OUT

            mov.w   #arr2, R12              ; put address into R12
            mov.w   #7, R13                 ; put array length into R13
            mov.w   #1, R14                 ; display #0 (P3&P4)
            call    #suma_rp
            mov.w   R14, &P3OUT             ; result goes to P4OUT&P3OUT
            jmp     $
```

```

arr1:        .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2:        .int     1, 1, 1, 1, -1, -1, -1     ; the second array

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

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

```

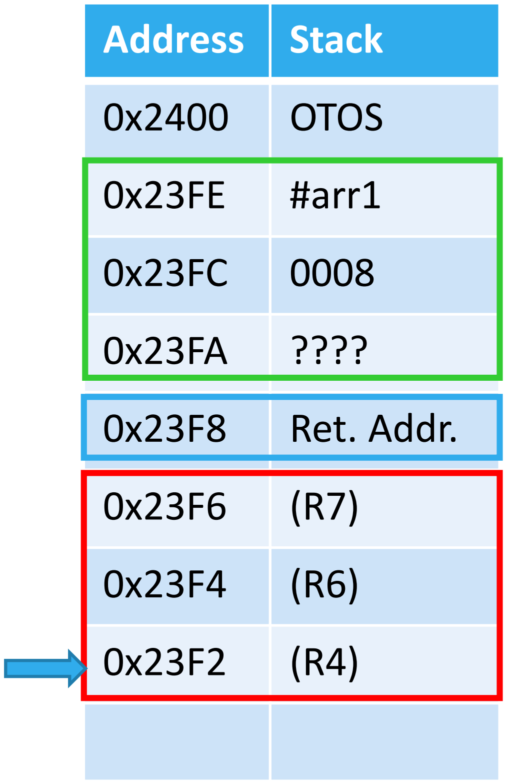
Code 7. Main assembly program for summing up two integer arrays using a subroutine suma_rp.

```
;-------------------------------------------------------------------------------
; File       : Lab5_D2_RP.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of an input integer array
; Description: suma_rp is a subroutine that sums elements of an integer array
; Input      : The input parameters are:
;                   R12 -- array starting address
;                   R13 -- the number of elements (>= 1)
; Output     : The output result is returned in register R14
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008 (revised on August 2020)
;------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"      ; Include device header file

            .def suma_rp

            .text

suma_rp:
            clr.w   R14             ; clear register R14 (keeps the sum)
lnext:      add.w   @R12+, R14      ; add a new element
            dec.w   R13             ; decrement step counter
            jnz     lnext           ; jump if not finished
lend:       ret                     ; return from subroutine
            .end

```

Code 8. Subroutine for summing up an integer array (suma_rp).

If many parameters are passed, there may not be enough general-purpose registers available for passing parameters into the subroutine. In this case we use the stack to pass parameters. Code 9 shows the calling program (Lab5_D3_main.asm) and Code 10 shows the subroutine suma_sp (Lab5_D3_SP.asm). Before calling the subroutine, we place parameters on the stack using PUSH instructions (the array starting address, array length), and allocate the space for the sum returned by the subroutine (lines 32, 33 and 34 in Code 9). Please note how we allocate space for the result on the stack. We refer to this code section as a prologué – steps we carry out in the caller to prepare input parameters for the callee. The CALL instruction pushes the return address on the stack.

The subroutine pushes the contents of the registers R7, R6, and R4 on the stack (another 6 bytes) to save their original content as these registers are used in the subroutine. 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 it grows, and where SP points to). Figure 3 illustrates the content of the stack once the processor finished execution of line 21 in the subroutine suma_sp (Code 10). The original values of the registers pushed onto the stack occupy 6 bytes, the return address pushed by the CALL instruction 2 bytes, the space for the result occupies 2 bytes, and the input parameters occupy 4 bytes. The total distance between the current top of the stack marked by a blue arrow and the location on the stack where we placed the starting address is 12 bytes. So the instruction `MOV 12(SP), R4` loads the register R4 with the first parameter (the array starting address). Similarly, the array length can be retrieved by `MOV 10(SP), R6`. Once the elements are summed up, the result is written into the reserved location on the stack: **mov.w** R7, 8(SP). 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 read the sum from the top of the stack and then we can free 6 bytes on the stack used in the prologué (code that proceeds the CALL instruction that prepares parameters). The instructions executed in the caller to free the stack up and retrieve the results are often referred to as epilogué.



Address	Stack
0x2400	OTOS
0x23FE	#arr1
0x23FC	0008
0x23FA	????
0x23F8	Ret. Addr.
0x23F6	(R7)
0x23F4	(R6)
0x23F2	(R4)

Figure 3. Illustration of the stack after instruction in line 21 of the subroutine suma_sp is executed. The green box illustrates the locations on the stack prepared by the caller (main program), blue box is the return address pushed by the CALL instruction, and the red box shows the registers backed up on the stack by the subroutine suma_sp. Note: we assume that initial SP=0x2400 (OTOS stands for the original top-of-the-stack).

```
;-------------------------------------------------------------------------------
; File       : Lab5_D3_main.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of two integer arrays using a subroutine suma_sp
; Description: The program calls suma_sp to sum up elements of integer arrays and
;              stores the respective sums in parallel ports' output registers.
;              Parameters to suma_sp are passed through the stack.
; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
; Output     : P2OUT&P1OUT stores the sum of arr1, P4OUT&P3OUT stores the sum of arr2
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008 (revised August 2020)
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"         ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                     ; Export program entry-point to
                                              ; make it known to linker.
            .ref    suma_sp
;-------------------------------------------------------------------------------
```

```
            .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
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main code here
;-------------------------------------------------------------------------------
main:
            push    #arr1                   ; push the address of arr1
            push    #8                      ; push the number of elements
            sub.w   #2, SP                  ; allocate space for the sum
            call    #suma_sp
            mov.w   @SP, &P1OUT             ; store the sum in P2OUT&P1OUT
            add.w   #6,SP                   ; collapse the stack

            push    #arr2                   ; push the address of arr1
            push    #7                      ; push the number of elements
            sub     #2, SP                  ; allocate space for the sum
            call    #suma_sp
            mov.w   @SP, &P3OUT             ; store the sume in P4OUT&P3OUT
            add.w   #6,SP                   ; collapse the stack

            jmp     $

arr1:       .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2:       .int     1, 1, 1, 1, -1, -1, -1     ; the second array

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

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


```

Code 9. 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 address of the array
;                  array length
; Output     : The result is returned through the stack
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008 (revised August 2020)
;-------------------------------------------------------------------------------
            .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   R7, 8(SP)               ; store the sum on the stack
lend:       pop     R4                      ; restore R4
            pop     R6                      ; restore R6
            pop     R7                      ; restore R7
            ret                             ; return
            .end
```

Code 10. Subroutine for summing up an integer array (parameters are passed through the stack).

Things to remember 6-2. Passing parameters.

Passing parameter defines the way the caller and callee communicate with each other – you can think about it as a contract between them. The parameters can be passed through registers (fastest approach) or through the stack (or combining the two approaches).

7 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 address mapping 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 (thus, saving its original content) 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
        MOV.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 Code 10. 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. Code 11 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). Figure 4 illustrates the content of the stack after the instruction in line 25 of Code 11 is executed.

Code 12 shows the implementation of the caller program – it is practically the same as the one in Code 9, except that suma_spsf is called instead of suma_sp.

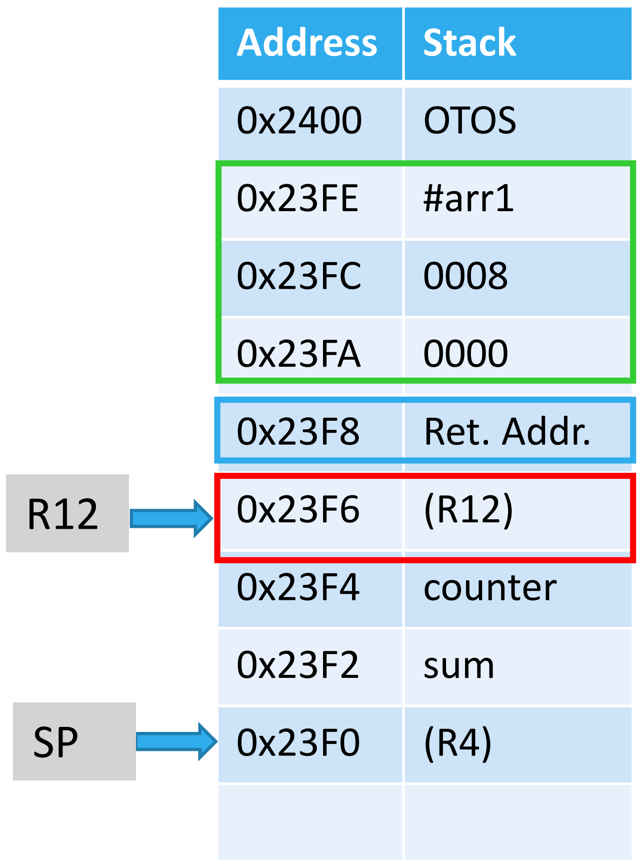


Figure 4. Illustration of the stack after instruction in line 25 of the subroutine suma_spsf is executed. The green box illustrates the locations on the stack prepared by the caller (main program), blue box is the return address pushed by the CALL instruction, and the red box shows the stack frame register (R12 in this example). The local variables of the subroutine counter and sum are allocated on the stack, right below the stack frame. We refer to them relative to the stack frame.

```
1   ;-------------------------------------------------------------------------------
2   ; File       : Lab5_D4_SPSF.asm (CPE 325 Lab5 Demo code)
3   ; Function   : Finds a sum of an input integer array
4   ; Description: suma_spsf is a subroutine that sums elements of an integer array.
5   ;              The subroutine allocates local variables on the stack:
6   ;                  counter (SFP-2)
7   ;                  sum (SFP-4)
8   ; Input      : The input parameters are on the stack pushed as follows:
9   ;                  starting address of the array
10  ;                  array length
11  ; Output     : The result is returned through the stack
12  ; Author     : A. Milenkovic, milenkovic@computer.org
13  ; Date       : September 14, 2008 (revised on August 2020)
14  ;-------------------------------------------------------------------------------
15            .cdecls C,LIST,"msp430.h"       ; Include device header file
16
17            .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
        mov.w   -4(R12), 4(R12)     ; write the result on the stack
        pop     R4                  ; restore R4
        add.w   #4, SP              ; collapse the stack frame
        pop     R12                 ; restore stack frame pointer
        ret                         ; return
        .end

```

Code 11. Subroutine for summing up an integer array that uses local variables sum and counter allocated on the stack.

```
;-------------------------------------------------------------------------------
; File       : Lab5_D4_main.asm (CPE 325 Lab5 Demo code)
; Function   : Finds a sum of two integer arrays using a subroutine suma_spsf
; Description: The program initializes ports and
;              calls suma_spsf to sum up elements of integer arrays and
;              display sums on parallel ports.
;              Parameters to suma_spsf are passed through the stack.
; Input      : The input arrays are signed 16-bit integers in arr1 and arr2
; Output     : P1OUT&P2OUT displays sum of arr1, P3OUT&P4OUT displays sum of arr2
; Author     : A. Milenkovic, milenkovic@computer.org
; Date       : September 14, 2008
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
            .ref    suma_spsf
;-------------------------------------------------------------------------------
            .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 stackpointer
StopWDT:    mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer

;-------------------------------------------------------------------------------
; Main code here
```

```
;-------------------------------------------------------------------------------
main:       push    #arr1                       ; push the address of arr1
            push    #8                          ; push the number of elements
            sub.w   #2, SP                      ; allocate space for the sum
            call    #suma_spsf
            mov.w   @SP, &P1OUT                 ; move result from the TOS to P2OUT&P1OUT
            add.w   #6,SP                       ; collapse the stack
            push    #arr2                       ; push the address of arr1
            push    #7                          ; push the number of elements
            sub.w   #2, SP                      ; allocate space for the sum
            call    #suma_spsf
            mov.w   @SP, &P3OUT                 ; move result from the TOS to P4OUT&P3OUT
            add.w   #6,SP                       ; collapse the stack

            jmp     $

arr1:       .int     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2:       .int     1, 1, 1, 1, -1, -1, -1     ; the second array

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

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

Code 12. Main program calling subroutine suma_spsf.

Things to remember 7-1. Local variables.

Local variables in subroutines are typically allocated dynamically on the stack. For that purpose we use a register as stack frame pointer that is pushed on the stack at the beginning of the subroutine. The local variables are allocated on the stack beneath the frame pointer and are removed from the stack before we exit the subroutine.

8 Performance, Execution Time

In this section we will briefly discuss the notion of performance. Computer engineers define performance as an ability of a computer to perform a task. There are two typical definitions of performance: user-centric or speed-centric that focuses on how long does it take to complete a task, and system-centric or throughput-centric that focuses on how many tasks can be completed in a unit of time. In this text we will focus on the speed flavor of performance, where we are interested in completing a task in minimum amount of time. Thus, performance a machine achieves while executing a program under test (let's call it PUT) is a reciprocal of the time needed to execute the program, also known as ET (Execution Time). Performance is a “higher-is-better” metric, whereas the execution time is a “lower-is-better” metric.

We can formally define performance as:

$$P(PUT) = \frac{1}{ET(PUT)}$$

The execution time can be calculated as a product of the number of processor clock cycles needed to execute the program PUT and the clock cycle time (CCT – clock cycle time), which is a reciprocal of the processor clock frequency (CF), CCT=1/CF. Next, the number of processor clock cycles to execute the program PUT is a product of the number of instructions executed in the program or Instruction Count (IC) and the average number of clock Cycles per Instruction executed, CPI. Consequently, we can write the following equation that is often referred to as the Iron Law of processor performance:

$$ET(PUT) = IC \cdot CPI \cdot CCT = \frac{IC \cdot CPI}{CF}$$

In units, the equation can be looked at as follows:

$$ET(PUT) = \frac{Instruction}{Program} \cdot \frac{Cycle}{Instruction} \cdot \frac{Second}{Cycle} = \frac{Second}{Program}$$

Generally, we are interested to complete tasks as soon as possible. This way, we can have more satisfied users, we can meet stringent time deadlines (e.g., not to miss important events, remember the story about Moon landing and control processor being over tasked), or we can spend more time in a low-power modes, thus saving energy and consequently extending the operating time of battery powered devices.

Determining of the number of clock cycles needed to execute a program is relatively easy in Code Composer. You have a cycle counter that is updated as you execute every instruction. The time to execute an instruction in MSP430 is deterministic and specified in the user manual. An instruction can take between 1 clock cycle to execute (instructions operating on operands in registers) to 6 clock cycles, depending on the instruction type (double-operand, single-operand, branch) and the addressing modes. Figure 5 shows instruction lengths and the number of clock cycles Type I (double operand) instructions take to execute. As you can see the number of clock cycles is solely a function of the addressing mode, e.g., MOV.W R5, R8 takes just one clock cycle, whereas MOV.W &EDE, &TONI takes 6 clock cycles. How many clock cycles does MOV.W @R5, TONI takes? Why? All MSP430 branch instructions are 1 instruction word long and take exactly 2 clock cycles to execute, regardless of the branch outcome (taken or not taken). Figure 6 shows instruction lengths and the number of clock cycles Type II instructions take to execute, whereas Figure 7 shows these for RET/RETI instructions.

The following example illustrates how you can apply this information to determine a program execution time and other metrics of interest. Assume that an input string has 20 ASCII characters (including the NULL that terminates it) and our task is to determine the execution time of this code snippet that copies the source string to a destination address contained in register R15.

```
; R14 points to the beginning of the source string (terminated by a NULL)
; R15 to the beginning of the destination string
lstrcopy:   mov.b   @R14+, 0(R15) ;
            tst.b   0(R15)        ;
            jz      lfin          ;
            inc.w   R15           ;
            jmp     lstrcopy      ;
lfin:
```

The first step is to determine how many instructions are executed in this code snippet. Based on the problem we have that the main loop will execute 19 times, each with 5 instructions. The last iteration will execute 3 instructions, plus the ret instruction. Thus,

$$IC = 19 * 5 + 3 = 95 + 3 + 1 = 99 \; instructions$$

Next, using information about execution time of individual instructions, we can write the following.

```
; R15 to the beginning of the destination string
lstrcopy:   mov.b   @R14+, 0(R15)  ; 4 cc (see note in the table)
            tst.b   0(R15)         ; 4 cc (an emulated instruction)
            jz      lfin          ;  2 cc
            inc.w   R15           ;  1 cc
            jmp     lstrcopy      ;  2 cc
lfin:       ret                   ;  4 cc
```

$$Cycles = 19 * (4 + 4 + 2 + 1 + 2) + 4 + 4 + 2 + 4 = 19 * 13 + 14 = 261 \; cycles$$

Assuming 1 μs clock cycle time, this code snippet takes:

$$ET = Cycles * CCT = 261 * 1 = 261 \text{ μs} = 0.261 \text{ ms} = 0.261 \cdot 10^{-3} \; s$$

Now we can determine CPI for this code snippet:

$$CPI = \frac{Cycles}{IC} = \frac{261}{99} = 2.64$$

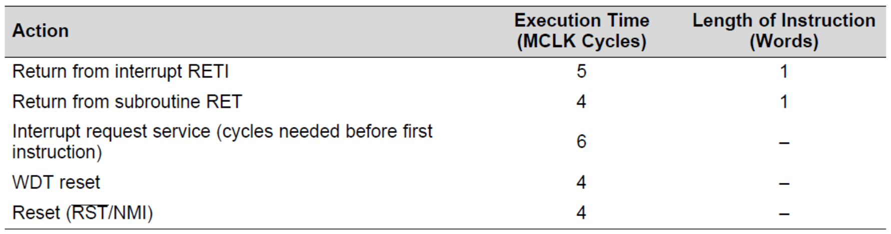
Addressing Mode		No. of Cycles	Length of Instruction	Example
Source	Destination			
Rn	Rm	1	1	`MOV R5,R8`
	PC	3	1	`BR R9`
	x(Rm)	4[1]	2	`ADD R5,4(R6)`
	EDE	4[1]	2	`XOR R8,EDE`
	&EDE	4[1]	2	`MOV R5,&EDE`
@Rn	Rm	2	1	`AND @R4,R5`
	PC	4	1	`BR @R8`
	x(Rm)	5[1]	2	`XOR @R5,8(R6)`
	EDE	5[1]	2	`MOV @R5,EDE`
	&EDE	5[1]	2	`XOR @R5,&EDE`
@Rn+	Rm	2	1	`ADD @R5+,R6`
	PC	4	1	`BR @R9+`
	x(Rm)	5[1]	2	`XOR @R5,8(R6)`
	EDE	5[1]	2	`MOV @R9+,EDE`
	&EDE	5[1]	2	`MOV @R9+,&EDE`
#N	Rm	2	2	`MOV #20,R9`
	PC	3	2	`BR #2AEh`
	x(Rm)	5[1]	3	`MOV #0300h,0(SP)`
	EDE	5[1]	3	`ADD #33,EDE`
	&EDE	5[1]	3	`ADD #33,&EDE`
x(Rn)	Rm	3	2	`MOV 2(R5),R7`
	PC	5	2	`BR 2(R6)`
	TONI	6[1]	3	`MOV 4(R7),TONI`
	x(Rm)	6[1]	3	`ADD 4(R4),6(R9)`
	&TONI	6[1]	3	`MOV 2(R4),&TONI`
EDE	Rm	3	2	`AND EDE,R6`
	PC	5	2	`BR EDE`
	TONI	6[1]	3	`CMP EDE,TONI`
	x(Rm)	6[1]	3	`MOV EDE,0(SP)`
	&TONI	6[1]	3	`MOV EDE,&TONI`
&EDE	Rm	3	2	`MOV &EDE,R8`
	PC	5	2	`BR &EDE`
	TONI	6[1]	3	`MOV &EDE,TONI`
	x(Rm)	6[1]	3	`MOV &EDE,0(SP)`
	&TONI	6[1]	3	`MOV &EDE,&TONI`

[1] MOV, BIT, and CMP instructions execute in one fewer cycle.

Figure 5. Type I (double operand) instruction lengths and cycles.

Addressing Mode	No. of Cycles			Length of Instruction	Example
	RRA, RRC SWPB, SXT	PUSH	CALL		
Rn	1	3	4	1	`SWPB R5`
@Rn	3	3	4	1	`RRC @R9`
@Rn+	3	3	4	1	`SWPB @R10+`
#N	N/A	3	4	2	`CALL #LABEL`
X(Rn)	4	4	5	2	`CALL 2(R7)`
EDE	4	4	5	2	`PUSH EDE`
&EDE	4	4	6	2	`SXT &EDE`

Figure 6. Type II (single operand) instruction lengths and cycles.



Action	Execution Time (MCLK Cycles)	Length of Instruction (Words)
Return from interrupt RETI	5	1
Return from subroutine RET	4	1
Interrupt request service (cycles needed before first instruction)	6	–
WDT reset	4	–
Reset ($\overline{\text{RST}}$/NMI)	4	–

Figure 7. RET/RETI instruction lengths and cycles.

`Things to remember 8-1. Performance (speed flavor).`

Performance captures how quickly a processor can execute a task. It is reciprocal of the program execution time. The program execution time, ET, is computed as follows:

$$ET = IC \cdot CPI \cdot CCT = \frac{IC \cdot CPI}{CF}$$

where IC is the number of executed instructions in a program, CPI is the average number of clock cycles per instruction, and CCT is the processor clock cycle time.

Sometimes you will see metrics such as MIPS used in reporting performance. MIPS stands for *Million of Instructions Per Second*, thus we can write:

$$MIPS = \frac{IC}{ET \cdot 10^6} = \frac{IC}{IC \cdot CPI \cdot CCT \cdot 10^6} = \frac{CF}{CPI \cdot 10^6} = \frac{CF\ [MHz]}{CPI}$$

Please note that MIPS could be quite misleading when comparing performance of a task executed on processors with different ISAs. The equation above suggests that processors with ISAs resulting in a small CPIs would always perform better than processor with more complex ISAs resulting in a larger CPI. This is simply not true.

Let's say you want to sum up two single-precision floats in MSP430 – it will take thousands of instructions to do so using MSP430 ISA (assume it takes 3,000 instructions), but the CPI would be relatively small, e.g., CPI ≈ 2.0. So the total time to complete this task is 3,000x2 = 6,000 clock cycles. Now, imagine we design an MSP430fp that adds a functional unit and corresponding instructions to perform addition on floats. Such an instruction may take for example 100 clock cycles to execute, inflating the program's CPI to a large number, but it will take just a single instruction to complete this operation. MIPS would misled you to say that the plain MSP430 would be a better choice. However, that is not true because it will just take you 100 clock cycles to do so on the improved MSP430fp (or 20x faster). Please note that we assume that the clock frequency remains constant, i.e., both the plain MSP430 and this hypothetical MSP430fn run at the same clock frequency.

A cousin of MIPS is FLOPS that stands for *Floating-Point Instructions Per Second* and is often used in scientific computing because it captures information on how many floating-point operations a machine can execute in one second.

9 To Learn More

1. MSP430x5xx and MSP430x6xx Family User's Guide, https://www.ti.com/lit/slau208
2. MSP430x4xx Family User's Guide, https://www.ti.com/lit/pdf/slau056

10 Exercises

Problem #1.

Consider the following code segment. Reverse engineer the code. What does this code do? Determine the total execution time in second for the code sequence concluding with line 16.

```
01              bis.b   #0xFF, &P1DIR
02              bis.b   #0xFF, &P2DIR
03              mov.w   #UIArr, R14
04              mov.w   #UIArrA, R13
05              sub.w   R14, R13
06              rra     R13
07              mov.w   #0x0, R7
08 LNext:       mov.w   @R14+, R15
09              cmp.w   R7, R15
10              jnc     Lskip
11              mov.w   R15, R7
12 Lskip:       dec.w   R13
13              jnz     LNext
14              mov.b   R7, P1OUT
15              swpb    R7
16              mov.b   R7, P2OUT
17              jmp     $
18 UIArr: .word 4, 5, 200, 500, 60000
19 UIArrA:
```

Problem #2.

Design and write an MSP430 assembly language subroutine *unsigned int asciihex2ui(char* mya)* that processes an ASCII array with 4 elements and converts its value into an unsigned integer. The ASCII array **ahex** contains a hexadecimal representation of the number in ASCII format – e.g., "AB34" is an array of 4 ASCII characters (0x41,0x42, 0x33,0x34) that should be converted into an unsigned integer with value 0xAB34 (1010.1011.0011.0100b).

What does the main program do with the returned value? What is the input parameter and how is it passed to the subroutine?

```
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
```

```
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
            bis.b   #0xFF, &P1DIR           ; set P1 as output port
            bis.b   #0xFF, &P2DIR           ; set P2 as output port
            push    #ahex                   ;
            call    #asciihex2ui            ;
            mov.b   R12, &P1OUT
            swpb    R12
            mov.b   R12, &P2OUT
            jmp     $

ahex:       .byte   'A', 'B', '3', '4'       ; ASCII array for input
```

Problem #3.

Design and write an MSP430 assembly language subroutine *int hex_num (int *myw)* that processes an integer to determine the number of numerical symbols (0-9) in its hexademical representation. For example, *hex_num(0xABBA)=0*, and *hex_num(0x345A)*=3. The main program that calls the subroutine is shown below.

What does the main program do with the returned value? How do we pass the input parameter?

```
RESET       mov.w   #__STACK_END,SP         ; Initialize stack pointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
            bis.b   #0xFF, &P1DIR           ; set P1 as output port
            push    #myw                    ;
            call    #hex_num                ;
            mov.b   R7, &P1OUT
            jmp     $

myw:       .word   0xAB3D                  ;
```