CPE 325: Embedded Systems Laboratory
Laboratory #5 Tutorial
MSP430 Assembly Language Programming
Subroutines, Passing Parameters, and Hardware Multiplier

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

Objective:

This tutorial will continue the introduction to assembly language programming with the MSP430 hardware. In this lab, you will learn the following topics:

Developing subroutines in assembly language
Passing parameters to subroutines using registers and the stack
Working with hardware multiplier on the MSP430

Notes:

All previous tutorials are required for successful completion of this lab, especially, the tutorials introducing the TI Experimenter's Board and the Code Composer Studio software development environment.

Contents:

1 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.

1.1 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:

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.

1.2 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 1. We have two integer arrays arr1 and arr2. The program finds the sum of the integers in arr1 and displays the result in P1OUT and P2OUT, and then finds the sum of the integers in arr2 and displays the result in P3OUT and P4OUT. 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, how many elements the array has, and to return one operand – the sum of the array elements.

```
;-------------------------------------------------------------------------------
; 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
```

Figure 1. Summing up arrays without a subroutine (Lab5_D1.asm)

Let us next consider the main program (Figure 2) 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 Figure 3 uses 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_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
```

Figure 2. Summing up arrays using suma_rp (Lab5_D2_main.asm)

```
;-------------------------------------------------------------------------------
; 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

```

Figure 3. Subroutine that adds up the elements of the input array using parameters passed through registers (Lab5_D2_RP.asm)

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. Figure 4 shows the calling program (Lab5_D3_main.asm) and Figure 5 shows the subroutine (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. Please note how we allocate space on the stack for the result. 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 it grows, and where SP points to). The original values of the registers pushed onto the stack occupy 6 bytes, the return address 2 bytes, the space for output occupies 2 bytes, and the input parameters occupy 4 bytes. 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 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. 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 prologue (code that proceeds the CALL instruction that prepares parameters).

```
;-------------------------------------------------------------------------------
; 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

```

Figure 4. Summing up arrays using suma_sp (Lab5_D3_main.asm)

```
;-------------------------------------------------------------------------------
; 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
```

Figure 5. Subroutine that adds up the elements of the input array using parameters passed through the stack (Lab5_D3_SP.asm)

2 Hardware Multiplier

The MSP430 contains an optional peripheral hardware multiplier that allows the user to quickly perform multiplication operations. Multiplication operations using the standard instruction set can be complex and consume a lot of processing time; however, the hardware multiplier is a specialized peripheral that can perform arithmetic operations using a few instructions. The multiplier can perform up to 16-bit by 16-bit multiplication and can perform signed or unsigned multiplication with or without an accumulator. Some MSP430 models have no multiplier, but some models have a 32-bit by 32-bit multiplier. It is important to check the datasheet for your particular device to understand the available peripherals. MSP430F5529 has 32-bit by 32-bit multiplier peripheral called MPY32 that is capable of operating on 8-bit, 16-bit, 24-bit and 32-bit operands.

To use the hardware multiplier, you simply move your first operand (multiplicand) into a register designed to accept the first operand. There are twelve registers which can accept the first operand, and the one you choose determines the type of multiplication that will be performed. The second operand is then moved to the OP2 (or OP2L and OP2H) register. The result of the multiplication is calculated and placed in four registers – RES0 through RES3. An additional result register, SUMEXT, can be used with RESLO and RESHI for 16x16 bit multiplication. These registers are present for compatibility with 8- and 16- bit multipliers. The MSP430 user's guide includes a list of examples for performing the different types of multiplication, and they are listed here for convenience.

```
; 32x32 Unsigned Multiply
MOV   #01234h,&MPY32L   ; Load low-word of first operand
MOV   #01234h,&MPY32H   ; Load high-word of first operand
MOV   #05678h,&OP2L     ; Load low-word of second operand
MOV   #05678h,&OP2H     ; Load high-word of second operand
; ...                   ; Process results

; 16x16 Unsigned Multiply
MOV   #01234h,&MPY      ; Load first operand
MOV   #05678h,&OP2      ; Load second operand
; ...                   ; Process results

; 8x8 Unsigned Multiply. Absolute addressing.
MOV  #012h,&MPY_B       ; Load first operand
MOV #034h,&OP2_B        ; Load 2nd operand
; ...                   ; Process results

; 32x32 Signed Multiply
MOV   #01234h,&MPYS32L  ; Load low-word of first operand
MOV   #01234h,&MPYS32H  ; Load high-word of first operand
MOV   #05678h,&OP2L     ; Load low-word of second operand
MOV   #05678h,&OP2H     ; Load high-word of second operand
; ...                   ; Process results

; 16x16 Signed Multiply
MOV   #01234h,&MPYS     ; Load first operand
MOV   #05678h,&OP2      ; Load 2nd operand
; ...                   ; Process results

; 8x8 Signed Multiply. Absolute addressing.
MOV.B #012h,&MPYS_B     ; Load first operand
SXT   &MPYS             ; Sign extend first operand
MOV.B #034h,&OP2_B      ; Load 2nd operand
SXT   &OP2              ; Sign extend 2nd operand
; ...                   ; Process results

; 16x16 Unsigned Multiply Accumulate
MOV   #01234h,&MAC      ; Load first operand
MOV   #05678h,&OP2      ; Load 2nd operand
; ...                   ; Process results

; 8x8 Unsigned Multiply Accumulate. Absolute addressing
MOV.B #012h,&MAC_B      ; Load first operand
MOV.B #034h,&OP2_B      ; Load 2nd operand
; ...                   ; Process results

; 16x16 Signed Multiply Accumulate
MOV   #01234h,&MACS     ; Load first operand
MOV   #05678h,&OP2      ; Load 2nd operand
; ...                   ; Process results

; 8x8 Signed Multiply Accumulate. Absolute addressing
```

```
MOV.B #012h,&MACS_B      ; Load first operand
MOV.B #034h,&OP2_B       ; Temp. location for 2nd operand
; ...                    ; Process results
```

```
/*****************************************************************************
 *  File:        Lab5_D4.asm
 *  Description: MSP430F5529 Demo: This program defines two integer variables
 *                         .. and performs multiplication of these two intergers and
 *                         .. and stores the result in another variable
 *
 *  Clocks:      ACLK = 32768Hz,
 *               MCLK = SMCLK = 1 Mhz
 *
 * Input:        None
 * Output:       None
 *
 *                MSP430F5529
 *             -----------------
 *        /|\|              XIN|-
 *         | |                 | 32kHz
 *         --|RST          XOUT|-
 *           |                 |
 *           |                 |
 *           |                 |
 *           |                 |
 *           |                 |
 *           |                 |
 *
 *  Modified:   Prawar Poudel
 *  Date:       August 2020
 *****************************************************************************/

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

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
;-------------------------------------------------------------------------------
;                   USER DATA
;-------------------------------------------------------------------------------
                    .data                     ; so that it goes to data section
val1:        .int   45                        ; val1 is an integer whose value is 45
val2:        .int   3                         ; val2 is an integer whose value is 3
val3:        .int   0                         ; val3 will hold result of val1*val2
                                              ; .. let us init it with 0 for now
;-------------------------------------------------------------------------------
            .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 loop here
;-------------------------------------------------------------------------------
main:
                   mov val1,&MPY                   ; moves val1 to R5
                   mov val2,&OP2                   ; moves val2 to R6

                   ; since we have both the numbers already, let us get the results
                   ; after three clock cycles (for 16X16 multiplication)
                   nop
                   nop
                   nop

                   mov  RESLO,&val3

                   jmp $                           ; infinite loop



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

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

3 References

You should read the following references to gain more familiarity with subroutines, passing parameters, and the hardware multiplier:

- MSP430 Assembly Language Programming
- Page 177-185 in Davies' *MSP430 Microcontroller Basics* (subroutines and passing parameters)
- Chapter 25 (32-bit Hardware Multiplier), pages 672-690, in the MSP430F5529 user's guide (http://www.ti.com/lit/ug/slau208q/slau208q.pdf)