Assembly Language Programming: Subroutines

by Alex Milenkovich, milenkovic@computer.org

Objectives: Introduce subroutines, subroutine nesting, processor stack, and passing the parameters to subroutines.

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 result in 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 next program counter or PC). Hence, we should save the return address at the time the CALL instruction is executed. The way in which a machine makes it possible to call and return from subroutines is referred to as its *subroutine linkage method*.

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

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

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

- Branch to the address contained in the link register.

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. 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 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, SP, that is implicitly used in this operation. The stack pointer point to a stack called the processor stack.

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

- Push the contents of the PC on the top of the stack
- Update the stack pointer
- Branch to the target address specified by the instruction

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

- Pop the return address from the top of the stack into the PC
- Update the stack pointer.

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 Figure 1. We have two integer arrays arr1 and arr2. The program finds the sum of the first one and displays the result on the ports P1 and P2, and then finds the sum of the second array and displays the result on the ports P3 and P4. It is obvious that we can have a single subroutine that will perform this operation and thus make our code more readable and reusable. The subroutine needs to get three input parameters: what is the starting address of the input array, how many parameters the array has, and where to display the result. In this example the subroutine does not return any output parameter to the calling program.

Let us first consider the main program (Figure 2) and corresponding subroutine (suma_rp.s43, Figure 3) if we pass parameters through registers. Passing parameters through registers is straightforward and efficient. Three input parameters are placed in registers R12 (starting address), R13 (array length), and R14 (display id, #0 for P1&P2 and #1 for P3&P4). The calling program places the parameters in these registers, and then calls the subroutine using CALL #suma_rp instruction. The subroutine uses register R7 as an accumulator. The register R7 may contain valid data that belong to the calling program, so our first step should be to push the content of the register R7 on the stack. The last instruction before the return from the subroutine is to restore the original content of R7. This way, the calling program will find the original contents of the registers as they were before the CALL instruction. Other registers that our subroutine uses are R12, R13, and R14 that keeps parameters and we assume we can modify them (they do not need to preserve their original value once we are back in the calling program).

```
/*------------------------------------------------------------------------------
* Program    : Find a sum of two integer arrays;
* Input      : The input arrays are signed 16-bit integers in arr1 and arr2
* Output     : Display sum of arr1 on P1OUT&P2OUT and sum of arr2 on P3OUT&P4OUT
* Modified by: A. Milenkovic, milenkovic@computer.org
* Date       : September 14, 2008
* Description: MSP430 IAR EW; Demonstation of the MSP430 assembler
*------------------------------------------------------------------------------*/
#include "msp430.h"                     ; #define controlled include file


        NAME    main                    ; module name

        PUBLIC  main                    ; make the main label vissible
                                        ; outside this module

        ORG     0FFFEh
        DC16    init                    ; set reset vector to 'init' label

        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        BIS.B   #0xFF,&P1DIR            ; configure P1.x as output
        BIS.B   #0xFF,&P2DIR            ; configure P2.x as output
        BIS.B   #0xFF,&P3DIR            ; configure P3.x as output
        BIS.B   #0xFF,&P4DIR            ; configure P4.x as output

        MOV.W   #arr1, R4               ; load the starting address of the array1 into the
register R4
        MOV.W   #arr2, R5               ; load the starting address of the array1 into the
register R4
;       Sum arr1 and display
        CLR     R7                      ; Holds the sum
        MOV     #8, R10                 ; number of elements in arr1
lnext1: ADD     @R4+, R7                ; get next element
        DEC     R10
        JNZ     lnext1
        MOV.B   R7, P1OUT               ; display sum of arr1
        SWPB    R7
        MOV.B   R7, P2OUT
;       Sum arr2 and display
        CLR     R7                      ; Holds the sum
        MOV     #7, R10                 ; number of elements in arr2
lnext2: ADD     @R5+, R7                ; get next element
        DEC     R10
        JNZ     lnext2
        MOV.B   R7, P3OUT               ; display sum of arr1
        SWPB    R7
        MOV.B   R7, P4OUT

        JMP     $

arr1    DC16     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2    DC16     1, 1, 1, 1, -1, -1, -1     ; the second array

        END
```

Figure 1. Assembly program for summing up two integer arrays (sumarray_main.s43).

```
/*----------------------------------------------------------------------------
* Program    : Find a sum of two integer arrays using a subroutine (suma_rp.s43)
* Input      : The input arrays are signed 16-bit integers in arr1 and arr2
* Output     : Display sum of arr1 on P1OUT&P2OUT and sum of arr2 on P3OUT&P4OUT
* Modified by: A. Milenkovic, milenkovic@computer.org
* Date       : September 14, 2008
* Description: MSP430 IAR EW; Demonstation of the MSP430 assembler
*----------------------------------------------------------------------------*/
#include "msp430.h"                     ; #define controlled include file


        NAME    main                    ; module name

        PUBLIC  main                    ; make the main label vissible
                                        ; outside this module

        EXTERN  suma_rp

        ORG     0FFFEh
        DC16    init                    ; set reset vector to 'init' label

        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        BIS.B   #0xFF,&P1DIR            ; configure P1.x as output
        BIS.B   #0xFF,&P2DIR            ; configure P2.x as output
        BIS.B   #0xFF,&P3DIR            ; configure P3.x as output
        BIS.B   #0xFF,&P4DIR            ; configure P4.x as output

        MOV     #arr1, R12            ; put address into R12
        MOV     #8, R13               ; put array length into R13
        MOV     #0, R14               ; display #0 (P1&P2)
        CALL    #suma_rp

        MOV     #arr2, R12            ; put address into R12
        MOV     #7, R13               ; put array length into R13
        MOV     #1, R14               ; display #0 (P3&P4)
        CALL    #suma_rp
        JMP     $

arr1    DC16     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2    DC16     1, 1, 1, 1, -1, -1, -1     ; the second array

        END
```

Figure 2. Assembly program for summing up two integer arrays using a subroutine suma_rp.s43 (sumarray_subrp.s43).

```
/*-----------------------------------------------------------------------------
* Program  : Subroutine for that multiplies two 8-bit signed integers using HW multiplier
* Input    : The input parameters are:
              R12 -- array starting address
              R13 -- the number of elements (assume it is =>1)
              R14 -- dispay (0 for P1&P2 and 1 for P3&P4)
* Output   : No output parameters
*------------------------------------------------------------------------------*/
#include "msp430.h"                     ; #define controlled include file

        PUBLIC suma_rp

        RSEG CODE

suma_rp:
        ; save the registers on the stack
        PUSH    R7                      ; temporal sum
        CLR     R7
lnext:  ADD     @R12+, R7
        DEC     R13
        JNZ     lnext
        BIT     #1, R14                 ; display on P1&P2
        JNZ     lp34                    ; it's P3&P4
        MOV.B   R7, P1OUT
        SWPB    R7
        MOV.B   R7, P2OUT
        JMP     lend
lp34:   MOV.B   R7, P3OUT
        SWPB    R7
        MOV.B   R7, P4OUT
lend:   POP     R7                      ; restore R7
        RET
        END
```

Figure 3. Assembly subroutine for summing up an integer array (suma_rp.s43).

If many parameters are passed, there may not be enough general-purpose register available for passing parameter into the subroutine. In this case we use the stack to pass parameters. Figure 4 shows the calling program (sumarray_subsp.s43) and Figure 5 illustrates the subroutine (suma_sp.s43). Before calling the subroutine we prepare parameters on the stack using PUSH instructions (the array starting address, array length, and display id – each parameter is 2 bytes long). The CALL instruction pushes the return address on the stack. The subroutine then stores the contents of the registers R7, R6, and R4 on the stack (another 8 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 on the stack and how stack is organized. The original values of registers occupy 6 bytes, the return address 2 bytes, the display id 2 bytes, and the array length 2 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 can free 6 bytes on the stack used to pass parameters.

```
/*----------------------------------------------------------------------------
* Program    : Find a sum of two integer arrays
* Input      : The input arrays are signed 16-bit integers in arr1 and arr2
* Output     : Display sum of arr1 on P1OUT&P2OUT and sum of arr2 on P3OUT&P4OUT
* Modified by: A. Milenkovic, milenkovic@computer.org
* Date       : September 14, 2008
* Description: MSP430 IAR EW; Demonstration of the MSP430 assembler
*----------------------------------------------------------------------------*/

#include "msp430.h"                     ; #define controlled include file


        NAME    main                    ; module name

        PUBLIC  main                    ; make the main label vissible
                                        ; outside this module

        EXTERN  suma_sp

        ORG     0FFFEh
        DC16    init                    ; set reset vector to 'init' label

        RSEG    CSTACK                  ; pre-declaration of segment
        RSEG    CODE                    ; place program in 'CODE' segment

init:   MOV     #SFE(CSTACK), SP        ; set up stack

main:   NOP                             ; main program
        MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
        BIS.B   #0xFF,&P1DIR            ; configure P1.x as output
        BIS.B   #0xFF,&P2DIR            ; configure P2.x as output
        BIS.B   #0xFF,&P3DIR            ; configure P3.x as output
        BIS.B   #0xFF,&P4DIR            ; configure P4.x as output

        PUSH    #arr1                   ; push the address of arr1
        PUSH    #8                      ; push the number of elements
        PUSH    #0                      ; push display id
        CALL    #suma_sp
        ADD     #6,SP                   ; collapse the stack

        PUSH    #arr2                   ; push the address of arr1
        PUSH    #7                      ; push the number of elements
        PUSH    #1                      ; push display id
        CALL    #suma_sp
        ADD     #6,SP                   ; collapse the stack

        JMP     $

arr1    DC16     1, 2, 3, 4, 1, 2, 3, 4     ; the first array
arr2    DC16     1, 1, 1, 1, -1, -1, -1     ; the second array

        END
```

Figure 4. Assembly program for summing up two integer arrays using a subroutine suma_sp.s43 (sumarray_subsp.s43).

```
/*--------------------------------------------------------------------------------
* Program  : Subroutine for that sums up elements of an interger array
* Input    : The input parameters are passed through the stack:
              starting address of the array
              array length
              display id
* Output   : No output parameters
*--------------------------------------------------------------------------------*/
#include "msp430.h"                    ; #define controlled include file

        PUBLIC suma_sp

       RSEG CODE

suma_sp:
        ; save the registers on the stack
        PUSH    R7                     ; temporal sum
        PUSH    R6                     ; array length
        PUSH    R4                     ; pointer to array
        CLR     R7
        MOV     10(SP), R6              ; retrieve array length
        MOV     12(SP), R4
lnext:  ADD     @R4+, R7
        DEC     R6
        JNZ     lnext
        MOV     8(SP), R4              ; get id from the stack
        BIT     #1, R4                ; display on P1&P2
        JNZ     lp34                   ; it's P3&P4
        MOV.B   R7, P1OUT
        SWPB    R7
        MOV.B   R7, P2OUT
        JMP     lend
lp34:   MOV.B   R7, P3OUT
        SWPB    R7
        MOV.B   R7, P4OUT
lend:   POP     R4                     ; restore R4
        POP     R6
        POP     R7
        RET
        END
```

Figure 5. Assembly subroutine for summing up an integer array (suma_sp.s43).

2. Assignments

1. Write a subroutine *m8x8sa* using the MSP430 assembly language to multiply two signed 8-bit integers and return back the 16-bit result. Use shift-and-add method to multiply two numbers. Pass parameters through general-purpose registers.
2. Write a subroutine *m8x8h* using the MSP430 assembly language to multiply two signed 8-bit integers and return back the 16-bit result. Use hardware multiplier of the MSP430 to perform multiplication.
3. Write the main program that that allocates two arrays of 8-bit signed numbers and calculates their scalar product $s = \sum_{i=0}^{N-1} a(i) \cdot b(i)$ using subroutines from 1) and 2).