CPE 325: Intro to Embedded Computer Systems

Lab05

MSP430 Assembly Language Programming Subroutines, Passing Parameters, and Hardware Multiplier

Submitted by: Michael Agnew

Date of Experiment: 2/12/2024

Report Deadline: 2/13/2024

Demonstration Deadline: 2/19/2024

Introduction:

The main objective of this lab is understanding and implementing subroutines with the MSP430 assembly language. The experiment at hand is to sum up the squares of an array using a main source file that utilizes subroutines as functions to complete the given assignment. This is done through the usage of passing parameters by means of the stack pointer, and using various tools such as the hardware multiplier. The tutorial goes over several things that are important to be understood for this lab, such as subroutines, nesting subroutines, passing parameters, and the hardware multiplier. These are all very important concepts that need to be understood as subroutines are a very important part of any program.

Theory:

Subroutines: Subroutines are a very important part of programming and are used extremely often. A subroutine, is a segment of code that is called by the caller function (typically the main) and is used for a specific purpose before returning to the caller function with whatever it was meant to accomplish. For example, one might want a subroutine that sums up two given values. With this task in mind, one could call a subroutine named “sum” and pass it two numbers (parameters) which it would then sum and return to the original program. In this lab, the assignment laid out a specific purpose that the subroutines were meant to uphold. This purpose was to take in two parameters as input and perform operations on them to achieve the desired result. These parameters were the starting address of an array and the length of said array. With this information the subroutines were meant to sum up the squares of the elements of the array. The first subroutine, named SMUL, utilized the shift-and-add algorithm to achieve this while the subroutine named HMUL used the hardware multiplier that is available from the MSP430.

Passing Parameters: In the context of subroutines, one of the most important parts of these blocks of code are the parameters that are being passed back and forth. As mentioned, the subroutines SMUL and HMUL took in the array starting address as well as the array length as parameters and used them to complete the given assignment. Parameters can be passed in several ways; the first being registers.

Registers: Registers are a resource that are absolutely necessary and vital to completing a program in assembly. There are 16 registers total in MSP430 assembly and some have special functions, such as being used for the program counter and stack pointer, while others are meant for any use. These registers take the place of variables which would be used in a higher level language like C. Registers are used to pass values and addresses back and forth between themselves as well as having various operations being performed upon them. It is with these attributes that registers are used a lot when passing parameters to subroutines as they help store data required for the subroutine to do its operations.

Stack: The stack is a very useful part of the assembly programming language as well and is vital for making more complex code. The stack is a data structure that follows LIFO (last in first out) and allows one to save register values onto it which can be accessed later. This action allows one to free up registers that can then be used for other data and operations as the amount of registers are very limited. This makes saving register values on the stack extremely important as it allows one to make much more complex programs. Saving values on the stack is another way of passing parameters to subroutines and is how the subroutines received the data in the program that was made for this lab. The starting address of the array as well as the size of the array were both pushed onto the stack and then pulled into the designated subroutines who used the data to complete the assignment before sending the data back by means of the stack pointer.

Memory: Another way of passing parameters to a subroutine is by saving them in memory. These memory locations can then be accessed by the subroutine and the data that was stored in them can then be used for their intended purpose. This is similar to storing data on the stack but instead it is stored in memory addresses which can be pulled out later by whatever subroutine needs it.

Problem 01:

The first and only problem that was presented in this lab saw the advent of multiplication using the MSP430 assembly language. The way that this lab was meant to be completed was for a program to be made that takes an integer array and finds the sum of the squares of the given elements. This was meant to be done with two separate subroutines called SMUL and HMUL. These subroutines were meant to receive the information needed through the stack where they would receive the starting address of the array and a constant dictating the size of the array as well. The intent of SMUL was that it would multiply the sum of the squares by using the shift-and-add algorithm. This is done by checking the LSB of the current number, and if it is 1, it adds that number to the current sum and then shifts that number and the multiplicand to the left and the right respectively. This is repeated until the LSB of the number is 0 which is when the loop exits and the next element of the array is pulled in to undergo the same process. The second subroutine, HMUL, is a much simpler process, where the current index of the array is pulled into a register, where it is then loaded into the hardware multiplier, made available by the MSP430, which is then multiplied by itself in order to square it and returned to the original program. This process then repeats until the end of the array. After the execution of these subroutines, the main program assigns the values they return to PxOUT to display the results.

The input for the length of arrays 1 and 3 is:

```
push    #8                      ; Push the number of elements
```

Figure 1. Input for Number of Elements in Array 1 and 3.

The input for the length of arrays 2 and 4 is:

```
push    #7                      ; Push the number of elements
```

Figure 2. Input for Number of Elements in Array 2 and 4.

The arrays that were made for this assignment were as follows for the figure below:

```
arr1:   .int    1, 7, -5, 4, 2, -3, 9, 6
arr2:   .int    2, 4, 1, 6, -1, -1, -1
arr3:   .int    1, 7, -5, 4, 2, -3, 9, 6
arr4:   .int    2, 4, 1, 6, -1, -1, -1
```

Figure 3. The Coded Arrays for HMUL SMUL.

After running the code, the output for SMUL with arr1 as input was as follows:

> P1OUT	0xDD	Port 1 Output [Memory Mapped]

Figure 4. Output of SMUL for arr1.

After running the code, the output for HMUL with arr3 (the same values as arr1) were:

> P3OUT	0xDD	Port 3 Output [Memory Mapped]

Figure 5. Output of HMUL for arr3.

After running the code, the output for SMUL with arr2 as input was as follows:

Figure 6. Output of SMUL for arr2.

After running the code, the output for HMUL with arr4 (the same values as arr2) were:

Figure 7. Output of HMUL for arr4.

The assignment that was given for this lab also required the creation of a flowchart that illustrates the flow of the program and how it works. The following three figures are the flowcharts that were made for each code.

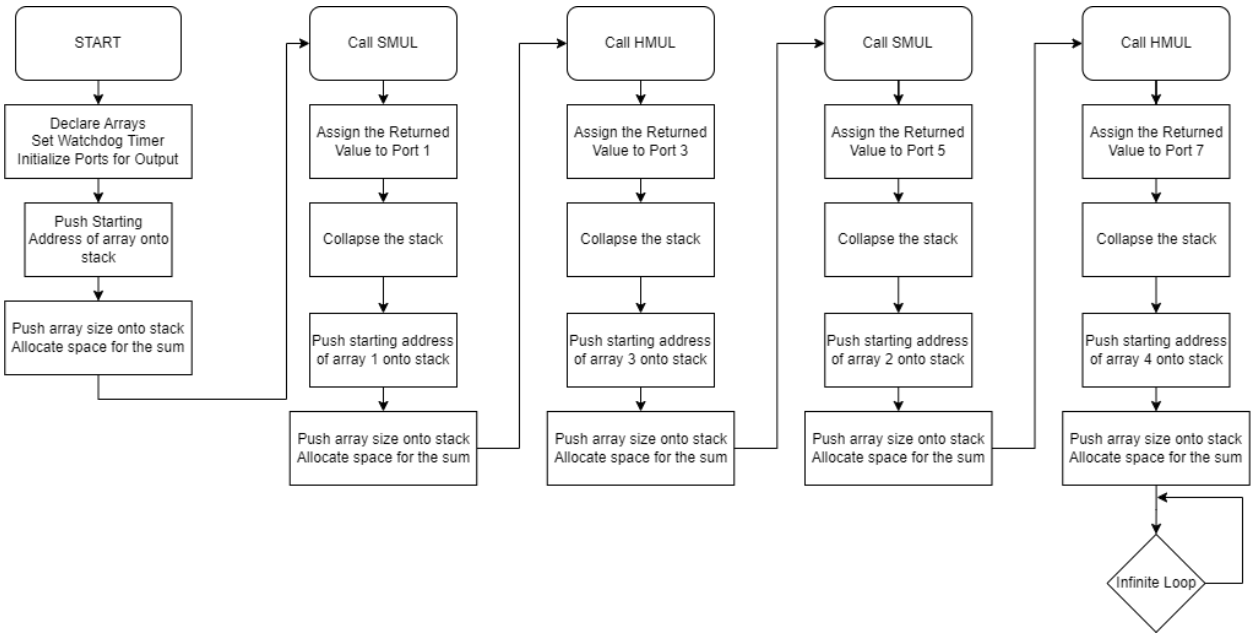

Figure 8. Flowchart for Main Code.

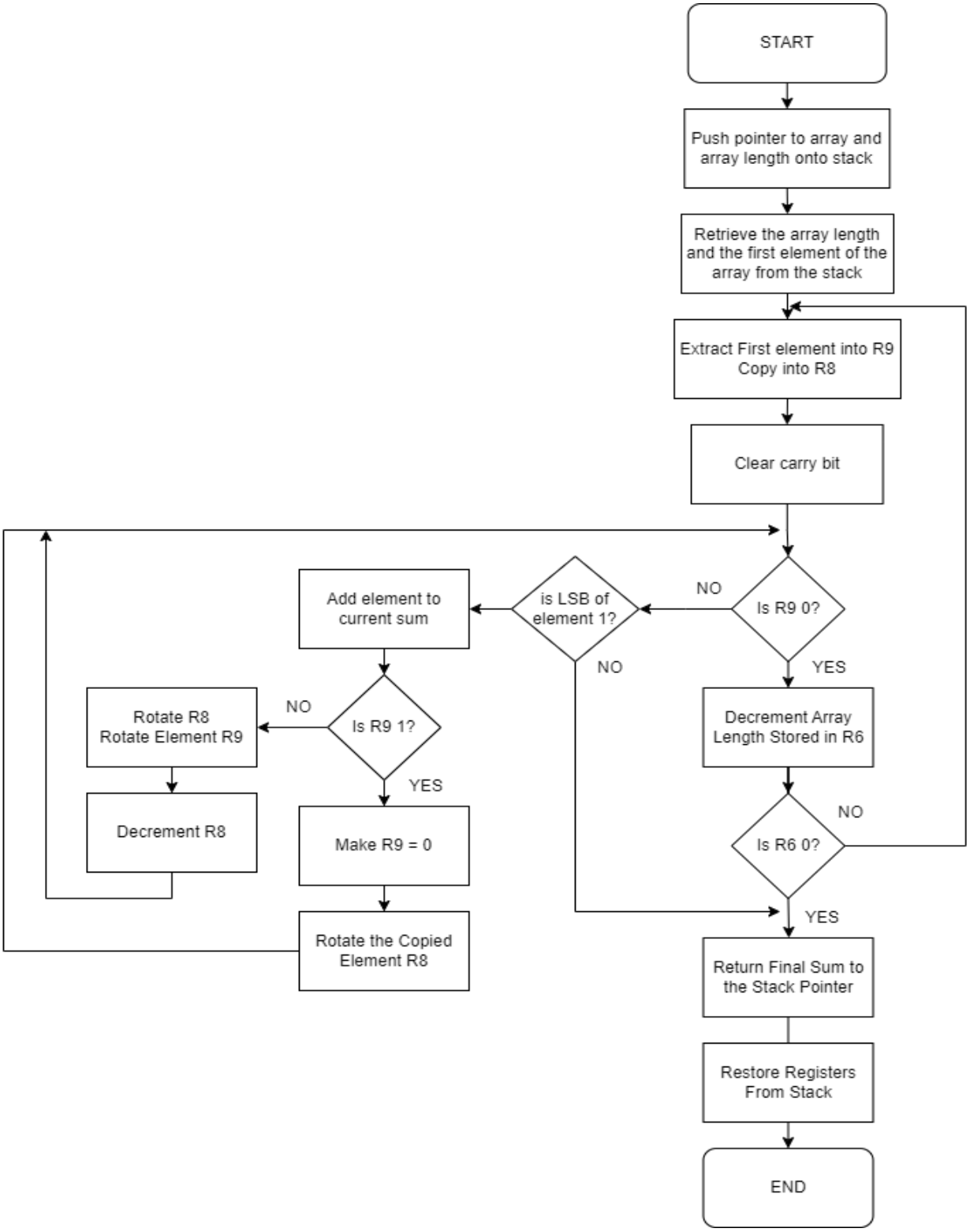

Figure 9. Flowchart for SMUL Subroutine.

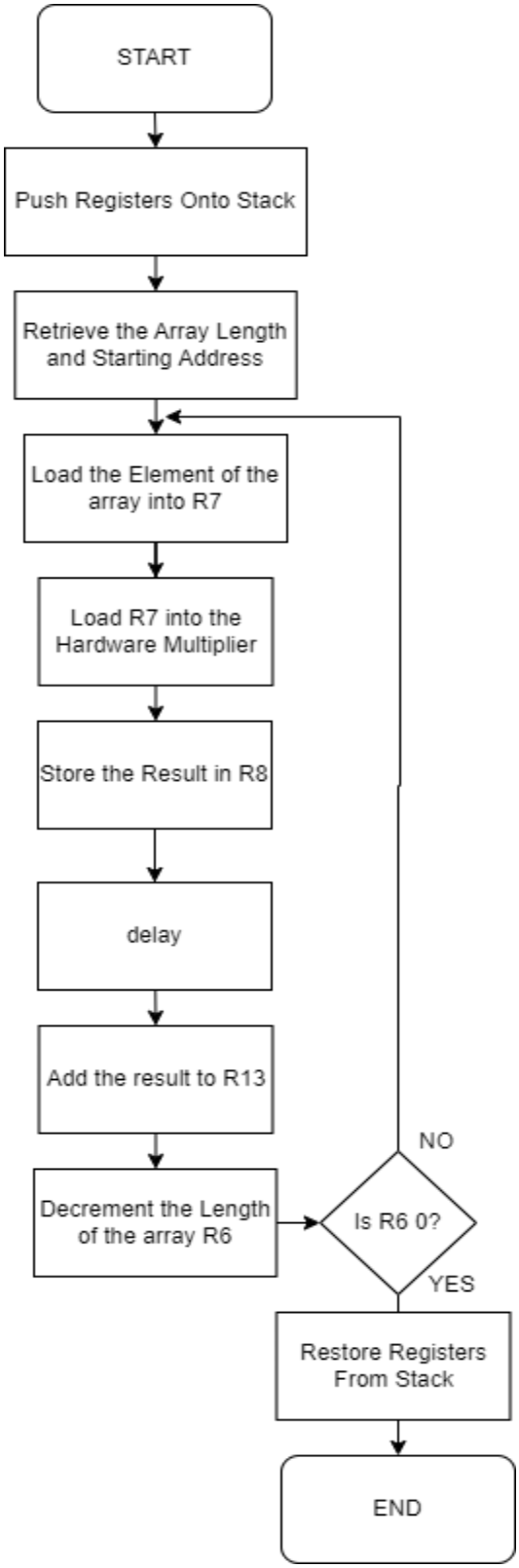

Figure 10. Flowchart For HMUL Subroutine.

Timing Measurements: After taking the timing measurements for each subroutine it is quite clear which one is faster. The timing measurements were taken by means of the clock cycle counter that is included in the tools for CCS. Utilizing this, the exact number of clock cycles were able to be found for each subroutine. The amount of clock cycles it took for the HMUL to execute was 188, while the amount for SMUL was 1055. With this information known, it is clear that one can conclude that the hardware multiplication is much more efficient and faster than the shift-and-add algorithm. This is because the shift-and-add algorithm performs a lot more operations and has to be coded in manually as opposed to the hardware multiplier that is already built into the MSP430. Based on previous labs, it is known that 500,000 clock cycles is half a second. With this being known, one can say that 1,000,000 clock cycles is 1 second. Therefore, at a rate of ~1000 clock cycles per 8 elements. The SMUL algorithm could do ~8000 elements per second. This would obviously be lower because it actually takes 1055 clock cycles for this subroutine to execute so the actual number of elements would be ~7,500. Using the same calculations, one can also deduce that the number of elements that the hardware multiplier could calculate per second would be ~42,500. This is a significant improvement and the answer is clear as to which is the faster method.

Conclusion:

Fortunately, there were no issues encountered over the course of this lab. Everything that was laid out by the tutorial and everything that was meant to be completed in the assignment was done so in the allotted time and according to instructions that were given. There was a lot to be learned from this lab. Mainly, how to put together and utilize subroutines in the context of MSP430 assembly. This is an extremely important concept for one to understand moving

forward as subroutines are a vital part of any program. There was also a lot to be learned in terms of passing parameters between subroutines. Registers, the stack and memory addresses are all tools that are used to help the programmer store and manipulate data to achieve their goal. All of these things are very important and should be understood in full as they are very useful and will be needed in the future.

Appendix:

Main:

```
; -------------------------------------------------------------------------
; File:         Lab5MainFinal.asm
; Function:     Finds a sum of squares of two integer arrays using two subroutines HMUL and SMUL
; Description:  This program calls SMUL and HMUL as subroutines in order to find the sum of squares of the given arrays
; Input:        The input arrays are signed 16-bit integers in arr1, arr2, arr3, and arr4
; Output:       P1OUT, P2OUT, P3OUT, P4OUT
; Author(s):    Michael Agnew, ma0133@uah.edu
; Date:         February 12, 2024
; Revised:      Febryary 13, 2024
; -------------------------------------------------------------------------
        .cdecls C, LIST, "msp430.h"  ; Include device header file
;-------------------------------------------------------------------------
```

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


        .ref    SMUL
;-------------------------------------------------------------------------------
        .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 code here
;-------------------------------------------------------------------------------
main:
         bis.b    #0xFF, &P1DIR ; port 1
         bis.b    #0xFF, &P2DIR ; port 2
         bis.b    #0xFF, &P3DIR ; port 3
         bis.b    #0xFF, &P4DIR ; port 4
         clr.w     P1OUT
```

```
 clr.w   P3OUT
 clr.w    P5OUT
 clr.w   P7OUT
push    #arr1           ; Push the address of arr1
push    #8              ; Push the number of elements
sub.w   #2, SP                  ; Allocate space for the sum
call    #SMUL
mov.w   @SP, P1OUT          ; Store the sum in P2OUT&P1OUT
add.w   #6, SP                  ; Collapse the stack

push    #arr3           ; Push the address of arr1
push    #8              ; Push the number of elements
sub.w   #2, SP                  ; Allocate space for the sum
 call    #HMUL
 mov.w    @SP, P3OUT
 add.w    #6, SP

push    #arr2           ; Push the address of arr1
push    #7              ; Push the number of elements
sub     #2, SP                  ; Allocate space for the sum
call    #SMUL
mov.w   @SP, P5OUT          ; Store the sume in P4OUT&P3OUT
mov.w    P5OUT, R12
```

```
        add.w   #6, SP                  ; Collapse the stack


        push    #arr4           ; Push the address of arr1
        push    #7              ; Push the number of elements
        sub.w   #2, SP                  ; Allocate space for the sum
        call    #HMUL
        mov.w    @SP, P7OUT
        add.w   #6, SP


        jmp     $


arr1:   .int    1, 7, -5, 4, 2, -3, 9, 6
arr2:   .int    2, 4, 1, 6, -1, -1, -1
arr3:   .int    1, 7, -5, 4, 2, -3, 9, 6
arr4:   .int    2, 4, 1, 6, -1, -1, -1


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

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

HMUL:

```
; -------------------------------------------------------------------------------
; File:             HMUL.asm
; Function:         Finds the sum of squares of an integer array
; Description:      This program is a subroutine that uses hardware multiplication to sum the squares
of an array
; Output:           The output is handled by the caller function
; Author(s):        Michael Agnew, ma0133@uah.edu
; Date:             February 12, 2024
; Revised:          February 13, 2024
; -------------------------------------------------------------------------------
        .cdecls C, LIST, "msp430.h"   ; Include device header file
; -------------------------------------------------------------------------------
        .def    HMUL
        .text
HMUL:                       ; Save the registers on the stack
        push    R13             ; 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
        clr.w   R13

lnext:  mov.w   @R4+, R7    ; get the next element
        mov.w   R7, &MPY    ; load in the value in R7
        mov.w   R7, &OP2    ; multiply it by itself
        nop  ; delay
        nop     ; delay
        nop  ; delay

        mov.w   RESLO, R8 ; store the result in R8
        add.w   R8, R13

        dec.w   R6              ; Decrement array length
        jnz     lnext           ; Repeat if not done
        mov.w   R13, 8(SP)              ; Store the sum on the stack

lend:   pop     R4              ; Restore R4
        pop     R6              ; Restore R6
```

```
        pop     R7              ; Restore R7
        ret                     ; Return
        .end
```

SMUL:

```
; -------------------------------------------------------------------------
; File:             SMUL.asm
; Function:         Finds the sum of squares of an integer array
; Description:      This program is a subroutine that uses shift-add algorithm to sum the squares of
an array
; Output:           The output is handled by the caller function
; Author(s):        Michael Agnew, ma0133@uah.edu
; Date:             February 12, 2024
; Revised:          February 13, 2024
; -------------------------------------------------------------------------

        .cdecls C,LIST,"msp430.h" ; Include device header file12

        .def    SMUL

        .text

SMUL:   push    R7                      ; Make register 7 temporal sum of multiplication
```

```
        push    R6                  ; Make register 6 array length
        push    R4                  ; Make register 4 pointer to array
        clr.w   R12                 ; Clear register 8 so you can do math operations on it
        mov.w   10(SP), R6          ; Retrieve array length
        mov.w   12(SP), R4          ; Retrieve the first element of the array

lnext:  mov.w   @R4+, R9            ; Extract the first element of the array
        mov.w   R9, R8              ; Copy the element of the array into R8


check:  and.w   #0x7FFF, R9     ; clear the carry bit
        cmp.w   #0x00,R9        ; Check if 0
        jeq     end             ; if 0, break because we're done
        mov.w   #0x01, R10      ; Initialize Register 10 to 1
        and.w   R9, R10         ; AND R9 and R10 Together to see if the LSB is 1
        cmp.w   #0x01, R10      ; Comparison
        jeq     sum             ; Add this number to the current sum if LSB = 1

rot:    cmp.w   #0x01, R9       ; If R9 is 1, then branch to 0 block
        jeq     zero            ; if equal to 1, jump to zero
        rlc.w   R8              ; Rotate R8
        dec.w   R8              ; Decrement R8 to get rid of extra num
        rrc.w   R9              ; Rotate R9
```

```
        jmp     check                   ; jump to check

sum:    add.w   R8, R12                 ; sum R8
        jmp     rot                     ; jump to rot

zero:   mov.w   #0x00, R9               ; set register to 0 so I don't have to worry about garbage
        rlc.w   R8                      ; Rotate R8
        jmp     check                   ; jump back to check


end:    dec.w   R6                      ; decrement array length
        jnz     lnext               ; if not zero, continue

lend:
        mov.w   R12, 8(SP)              ; assign R8 to the stack pointer
        pop     R4                      ; Restore R4
        pop     R6                      ; Restore R6
        pop     R7                      ; Restore R7
        ret

        .end
```