CPE 325: Intro to Embedded Computer System

Lab05

Sum of Squares

Submitted by: Gianna Foti

Date of Experiment:______23 September 2024_______________

Report Deadline:_______24 September 2024________________

Lab Section: ______CPE353-02______

Demonstration Deadline: _____24 September 2024___________

Introduction

In this lab, we had to develop an assembly program for the MSP430 microcontroller that computes the sum of squares for all elements in an integer array. The task involved writing two subroutines: SW_MUL, which uses the Shift-and-Add multiplication algorithm, and HW_MUL, which utilizes the MSP430's hardware multiplier. The array and its length were initialized in the main program, and both subroutines received the base address and length through the program stack. The results from SW_MUL and HW_MUL were stored in registers R12 and R13, respectively. The program correctly performed the sum of squares on an array of signed and unsigned integers with no fewer than five elements. Additionally, we measured the clock cycles for both subroutines. HW_MUL proved significantly faster and more efficient than SW_MUL, thanks to its reliance on hardware acceleration, whereas SW_MUL, while functional, required more clock cycles due to its algorithmic complexity. For the bonus task, I created a subroutine that converts a string of at least five digits into its numerical value using the hardware multiplier. The result was stored in memory, and I utilized the accumulator to maximize performance.

Theory Topics

1. Subroutines: what are they, and how did you use them in the lab
    a. A subroutine is a small section of code that performs a specific task, often used multiple times within a program. It helps streamline code by avoiding repetition, making it easier to manage and conserve memory. When a subroutine is called, the program temporarily jumps to its location, executes the defined instructions, and then returns to where it left off in the main program. Subroutines are especially useful for tasks that are consistent and reusable.
    b. In the lab, subroutines were used to modularize specific tasks:

i. SW_MUL Subroutine: This used the Shift-and-Add algorithm for software-based multiplication. The array's base address and length were passed via the stack, and the sum of squares was computed, returning the result through register R12.

ii. HW_MUL Subroutine: This subroutine used the Hardware Multiplier for faster multiplication, with parameters passed via the stack and the result returned through R13.

iii. These subroutines made the code more organized and allowed efficient comparison of software vs. hardware multiplication.

2. Passing parameters: Describe 3 different ways data can be input to a subroutine
    a. Registers: Data is passed directly through the processor's registers, which is the fastest method since accessing registers is quicker than accessing memory. However, it is limited by the number of available registers
    b. Stack: Parameters are pushed onto the program stack before calling the subroutine. The subroutine can then access these values by referencing the stack pointer. This method is flexible, especially when dealing with multiple parameters or larger data structures.
    c. Memory: Parameters are stored at specific memory addresses. The subroutine accesses data directly from these locations, which can be efficient for persistent data that needs to be shared or accessed repeatedly.
3. Hardware Multiplier: What it is/does Why it is useful
    a. A Hardware Multiplier is a specialized circuit within a processor designed specifically for performing multiplication operations. Unlike software-based multiplication, which requires multiple steps and clock cycles, the hardware multiplier executes the multiplication in fewer cycles by leveraging dedicated hardware.
    b. It is useful because of:

i. Speed: It significantly improves performance, especially when large numbers or frequent multiplications are involved, as it reduces the number of clock cycles needed.

ii. Efficiency: It frees up the processor to handle other tasks, making it ideal for real-time systems or applications that require high computational speed, such as signal processing or embedded systems.

Results & Observation

Part 1:

Program Description:

This program calculates the sum of squares of the elements in an integer array using two different multiplication methods: hardware-based multiplication (using the MSP430's hardware multiplier) and software-based multiplication (Shift-and-Add algorithm). I will describe the process of Main.asm, HW_MUL.asm, and SW_MUL.asm.

Process Main.asm:

1. First, I initialized an array (ARRAY_1) of integers and pushed its base address and length onto the stack.
2. Then I called two subroutines:
    a. HW_MUL: Uses the hardware multiplier to calculate the sum of squares of the array elements.
    b. SW_MUL: Uses the Shift-and-Add algorithm for software-based multiplication to perform the same task.
3. Then we stored the results in registers: R12 (from SW_MUL) and R13 (from HW_MUL).

Process HW_MUL.asm:

1. Register Management:
    a. Push registers R4 and R5 onto the stack to save their current values.
2. Load Parameters:
    a. Retrieve the base address of the array (located at index 8 of the stack) into R4.
    b. Retrieve the length of the array (located at index 6 of the stack) into R5.
3. Clear Accumulator:
    a. Clear R13 to ensure it starts from zero for accumulating the sum.
4. Process Each Array Element:
    a. Enter the HMultLoop:
        i. Load the current element of the array into the multiplier register (MPY).
        ii. Move the value from MPY to OP2 for multiplication.
        iii. Execute three no-operation (nop) instructions to allow for hardware operation time.
        iv. Add the result from RESLO (the lower part of the multiplication result) to R13.
        v. Decrement the loop counter (R5), which tracks how many elements remain.
        vi. If R5 is not zero, jump back to the start of the loop to process the next element.
5. Restore Registers and Return:
    a. Once all elements have been processed, pop the saved values of R5 and R4 from the stack.
    b. Return control to the calling function with the accumulated result in R13.

Process SW_MUL:

1. Register Management:
    a. Push registers R4, R5, R6, R7, R8, and R11 onto the stack to save their current values.
2. Load Parameters

    a. Retrieve the base address of the array (located at index 16 of the stack) into R4.
    b. Retrieve the length of the array (located at index 14 of the stack) into R5.
    c. Clear the result register R12 to start from zero.

3. Process Each Array Element:
    a. Enter the LoopyLoop:
        i. Check if the length (R5) is zero. If it is, jump to EndyEnd.
        ii. Load the current element from the array into R6 and increment the pointer.
        iii. Copy the current element to R7, which will serve as the multiplier.
        iv. Clear R11 to hold the temporary result of the multiplication.
        v. Set the loop counter (R8) to 16 (the bit length of the multiplier).
4. Perform Multiplication:
    a. Enter the AddyAdder loop:
    b. Check the LSB of R7:
        i. If the LSB is 1, add the multiplicand (R6) to R11.
        ii. If the LSB is 0, shift the multiplicand (R6) left and the multiplier (R7) right.
    c. Decrement the loop counter (R8).
    d. Repeat until the loop counter reaches zero.
5. Accumulate Results:
    a. After the multiplication loop:
        i. Check the LSB of the multiplier (R7):
            1. If the LSB is negative, subtract the multiplicand (R6) from R11.
        ii. Add the multiplication result from R11 to the running sum in R12.
        iii. Decrement the array length (R5) and jump back to LoopyLoop to process the next element.
6. Restore Registers and Return:

a. Once all elements have been processed, pop the saved values of registers from the stack in reverse order.

b. Return control to the calling function with the accumulated result in R12.

Clock Cycles:

1. Software Multiplication
   a. 917= 917 clock cycles Sec = Hz/CC → 1,048,576 = 1143.34 Elements/Second
   b. 

2. Hardware Multiplication
   a. 126 = 126 clock cycles Sec = Hz/CC → 1,048,576 Hz = 8329.17 Elements/Second
   b. 

Program 1 Output:

```
ARRAY_1:    .int    -1, 2, 3, 4, 5
STRING:     .string "78632", ' '
RESULT:     .space 10
```

R12	55 (Decimal)	Core
R13	55 (Decimal)	Core

Figure 1: Program 1 Output in Decimal with Data

Flowchart:

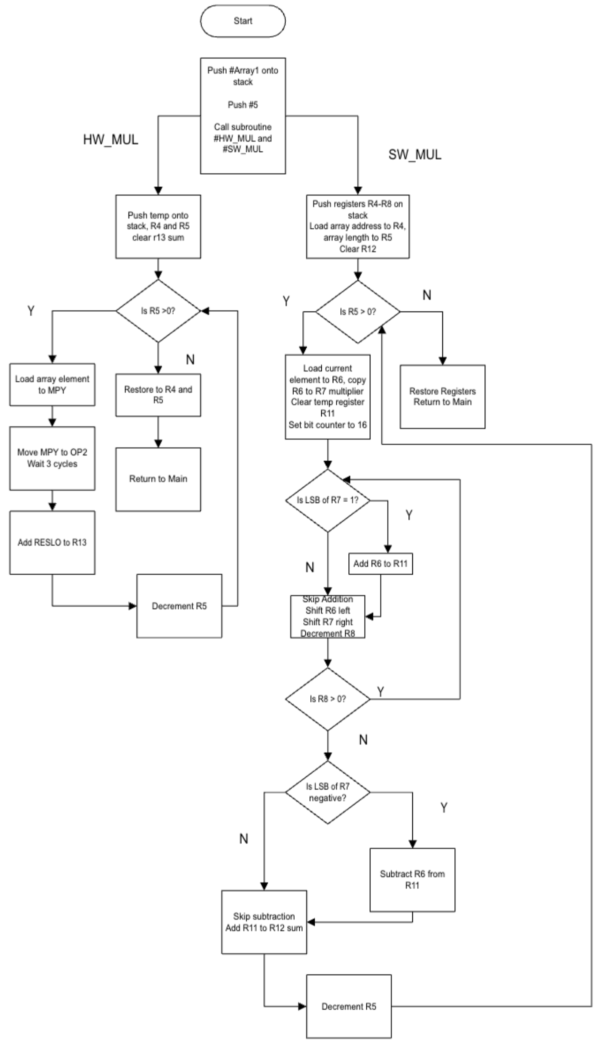

Figure 2: Program 1 Flowchart

Report Questions:

1. Comment on the efficiency of each subroutine. Which one do you prefer? Why?

    a. I prefer hardware multiplication. Hardware multiplication is faster because it leverages a dedicated hardware multiplier peripheral, allowing multiplication to occur directly at the hardware level. This bypasses the loops, shifts, and additions necessary in software-based multiplication, leading to faster and more efficient processing.

Part 2:

BONUS Description:

This assembly program processes a string of at least five digits to convert its ASCII representation into a numerical value using the Hardware Multiplier. The subroutine receives the base address of the string "78632" and the address of the variable for storing the result through the program stack. After processing, the program will compute the corresponding numerical value and store it in the specified memory location. The final result stored will be 78632. The subroutine utilizes the accumulator to ensure full credit is awarded.

Process:

1. Initialize Registers:
    a. Push registers (R4, R5, and R6) onto the stack to save their current values.
2. Load Parameters:
    a. Retrieve the base address of the string and the address of the result variable from the stack, storing them in R4 and R5, respectively.
3. Clear Registers:
    a. Clear the multiplication registers (MPY, RESLO, and OP2) to prepare for processing.
4. Process Each Character:
    a. Loop through each character of the string until a null terminator is encountered:
        i. Load the current character into the multiplication register (MPY).
        ii. Convert the ASCII character to its numerical value by subtracting 0x30.
        iii. Store the computed value into the result variable at the address in R5 and increment R5.
5. Finalize and Return:
    a. Once all characters are processed, restore the saved registers from the stack.
    b. Return control to the calling function.

Bonus Output:

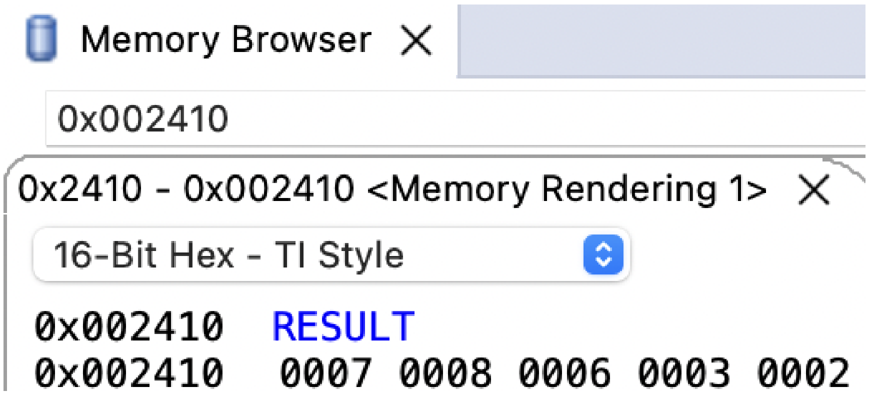

Figure 3: Bonus Output

Conclusion

In conclusion, I developed an assembly program for the MSP430 microcontroller to compute the sum of squares for all elements in an integer array. The task involved creating two subroutines: SW_MUL, which used the Shift-and-Add multiplication algorithm, and HW_MUL, which utilized the MSP430's hardware multiplier. The array and its length were passed to these subroutines via the program stack, and the results were stored in R12 (SW_MUL) and R13 (HW_MUL). Both subroutines successfully computed the sum of squares for an array of signed and unsigned integers with at least five elements. The performance comparison revealed that HW_MUL was significantly faster and more efficient due to its reliance on hardware acceleration, while SW_MUL, though functional, required more clock cycles because of its algorithmic complexity. For the bonus task, I created a subroutine that converts a string of at least five digits into its numerical value using the hardware multiplier, storing the result in memory. I utilized the accumulator to optimize performance further. I encountered no troubles completing the lab, and each task was executed smoothly. This lab effectively demonstrated the advantages of both software-based algorithms and hardware-accelerated solutions in embedded systems.

Appendix

Table 1: Program 1 Main source code

```
;* Name: Gianna Foti
;* File: main.asm
;* Description: Sum of squares using two subroutines (SW_MUL and HW_MUL).
;* Input: Base address and length of integer array passed via stack.
;* Output: Results stored in R12 (SW_MUL) and R13 (HW_MUL).
;* Lab Section: 02
;* Date: 24 September 2024
;-------------------------------------------------------------------------------
; MSP430 Assembler Code Template for use with TI Code Composer Studio
;
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file

;-------------------------------------------------------------------------------
            .def    RESET                   ; Export program entry-point to
                                            ; make it known to linker.
            .ref    HW_MUL
            .ref        SW_MUL
            .ref        BONUS
;-------------------------------------------------------------------------------
            .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
;-------------------------------------------------------------------------------
; Initialize array and pass parameters to subroutines
;-------------------------------------------------------------------------------
main:
            push    #ARRAY_1                ; push the address of array 1
            push    #5                      ; push the size of array
            call    #HW_MUL                 ; call hardware subroutine
            add     #4, SP                  ; adjust stack pointer, collapse stack
            push    #ARRAY_1                ; push the address of the array again
            push    #5                      ; push the size of the array again
            call    #SW_MUL                 ; calling the software subroutine
            add     #4, SP                  ; collapsing my stack
            push    #STRING
            push    #RESULT
            call    #BONUS
            add     #4, SP
            jmp     $                       ; infinite loop
            .data
ARRAY_1:    .int    -1, 2, 3, 4, 5
STRING:     .string "78632", ' '
RESULT:     .space 10
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
            .global __STACK_END, RESULT
            .sect   .stack

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

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

Table 2: Program 1 HW source code

```
;THIS IS MY HARDWARE MULTIPLICATION PROGRAMMMM
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file
;-------------------------------------------------------------------------------
            .def HW_MUL
;-------------------------------------------------------------------------------
HW_MUL:
            push R4                         ; pushing R4 onto the SP
            push R5                         ; pushing R5 onto the SP
            mov     8(SP), R4               ; accessing the 8 index
            mov     6(SP), R5               ; accessing the 6 index, length of array
            clr     R13                     ; clearing r13
HMultLoop:
            mov @R4+, MPY                   ; increment through the array, using the built
            mov MPY, OP2                    ; moving the value of MPY into OP2
            nop                             ; 1 clock cycle
            nop                             ; 2 clock cycle
            nop                             ; 3 clock cycle
            add RESLO, R13                  ; moving the value of RESLO into r13 as specified
            dec     R5                      ; decrements till i reach 0
            jnz HMultLoop                   ; if not equal to zero, hit the loop again
            pop R5                          ; else pop r5
            pop R4                          ; else pop r4
            ret
            .end
```

Table 3: Program 1 SW source code

```
; THIS IS MY SOFTWARE MULTIPLICATION
;-------------------------------------------------------------------------------
            .cdecls C,LIST,"msp430.h"       ; Include device header file
;-------------------------------------------------------------------------------
            .def SW_MUL
SW_MUL:
            push R4                                 ; saving registers on the stack
            push R5
            push R6
            push R7
            push R8
            push R11
            mov 16(SP), R4                          ; load address of array into R4
            mov 14(SP), R5                          ; load length of array into R5
            clr.w R12                               ; clear result register
LoopyLoop:
            jz EndyEnd                      ; jump to end if length is zero
            mov @R4+, R6                    ; load current element into R6 and increment pointer
```

```
                        mov R6, R7                  ; copy element into R7 (multiplier)
                        clr R11                     ; clearing the temp register thingy
                        mov #16, R8                 ; set loop counter (bit length of multiplier)
AddyAdder:
            bit.w   #1, R7                  ; check if LSB is 1
            jz      ShiftyShifter           ; if lsb is 0, then shift
            add.w   R6, R11                 ; if result is 1 then add multiplicant to the result for multiplication
ShiftyShifter:
            rla.w   R6                                  ; shift multiplicand (a) left
            rra.w   R7                                  ; shift multiplier (b) right
            dec.w   R8                                  ; decrementing the loopy counter
            jnz     AddyAdder                           ; loop until the loopy counter is 0
            bit.w   #1, R7                              ; check if lsb of multiplier is negative
            jz      ResultyResult                       ; if lsb is positive, go to the result
            sub.w   R6, R11                             ; if lsb is negative, subtract multiplicand from result
ResultyResult:
            add.w   R11, R12                    ; add multiplication result to running sum
            dec.w   R5                          ; decrement array length
            jmp     LoopyLoop                   ; repeat for next array element
EndyEnd:
            pop R11                 ; restore registers from stack
            pop R8
            pop R7
            pop R6
            pop R5
            pop R4
            ret
            .end
```

Table 4: BONUS source code

```
    ;THIS IS MY BONUS CODE

 .cdecls C,LIST,"msp430.h"      ; Include device header file
                .def BONUS
    .text               ; Assemble into program memory.
BONUS:      push R4                             ;save R4, the address of the str
                push R5                         ;save R5, the address of the place to store it
                push R6                         ;save R6, the element of the str
                mov.w 10(SP), R4                ;put address of str into R4
                mov.w 8(SP), R5                 ;put address of result to R5
                clr.w MPY
                clr.w RESLO
                clr.w OP2
loop:       mov.b @R4+, &MPY                    ;put element into R6
                cmp     #0, &MPY                ;is it the null value?
                jeq     done                    ;if it is, done
                sub     #0x30, &MPY
                mov.w   #0x1, &OP2
                nop
                nop
                nop
                mov RESLO, 0(R5)
                add     #0x2, R5
```

```
                jmp     loop
done:
                pop     R6
                pop     R5
                pop     R4
                ret
                .end
```