CPE 325: Intro to Embedded Computer System

**Lab05**

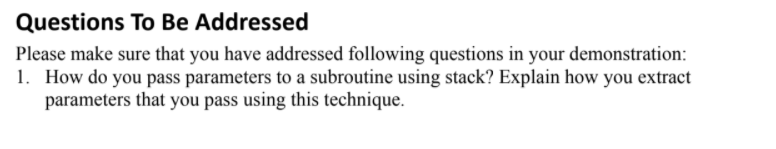
**Subroutines, passing parameters, Hardware multiplier**

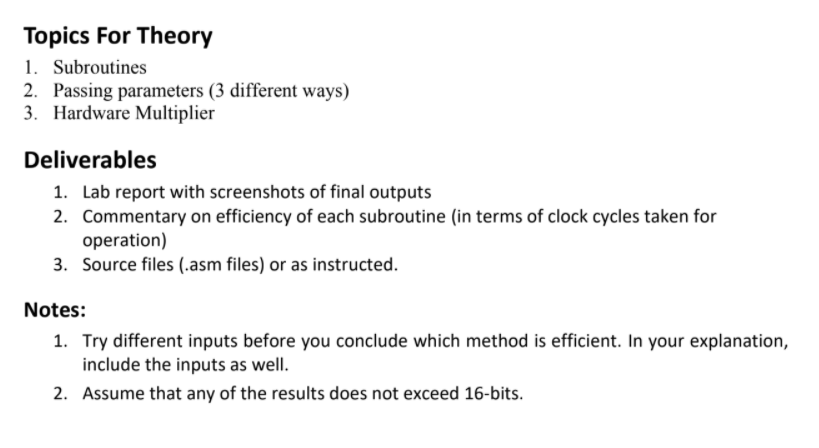
**Submitted by**: Nolan Anderson

**Date of Experiment**: 09/30/2020

**Report Deadline**: 09/30/2020

# 





# Introduction

This lab covers subroutines, passing parameters, and hardware multiplication. We use subroutines to make our code more readable and simpler than writing everything out manually and essentially “hard” coding it in. These subroutines, in different files, makes changes occur much faster and more straight forward, and also helps to eliminate confusion on what you are working on.

Passing parameters is very important in this lab because we do not want to overwrite anything we are doing in our subroutines. Using the stack helps us do this where we push and “pop” (really moving) the values to and from the stack. This is especially important in the hardware multiplication.

Hardware multiplication is using the actual instructions built in to do multiplication instead of implementing and algorithm such as shift add multiply. As you can see, using the specific and general purpose registers makes the code much simpler and faster than doing it with an algorithm.

# Theory

Write short notes on each topic discussed in lab.:

**Subroutines**:

Subroutines help to simplify code and perform specific tasks. They can take in parameters and you go to them with the CALL instruction. You can get out of a subroutine with the RETURN call. Subroutines are very helpful in making your code simpler and more readable.

**Passing Parameters (3 different ways):**

When passing parameters, you can either add it to the stack and pull it from there later on or you can store the values into registers or in memory locations. Personally, I think the best way to store the data is to put it into specific spots in memory because it allows you to not overwrite any registers or mess with the stack too much. It does require a little more coding but I believe it would be a little more robust.

**Hardware Multiplier:**

The hardware multiplier, unlike multiplying using software, uses the general-purpose registers to multiply data together. It is simpler than something like the software multiplier that we implemented here in this lab. The run time is also significantly lower. It can be done up to 16-bit multiplication and the register you move it to determines the type of multiplication that will be performed.

Results & Observation

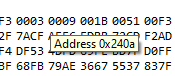
Copy the question from the assignment here:

**1. How do you pass parameters to a subroutine using stack? Explain how you extract**

**parameters that you pass using this technique.**

* I pushed the values onto the stack using push #swarr etc. and then in the calc\_power.asm file, I moved the position on the stack pointer to the register that I wanted to work with.

## Results Screenshots/Pictures:

* **Using b = 3**
* **Hardware Multiplier:**
* 
* **Software Multiplier:**
* 

## **3. Measure the number of clock cycles used by each subroutine for a small range of values.**

## **Comment of the efficiency of each subroutine.**

I have inserted images of my calculation for the number of clock cycles each instruction takes. As you can see, the hardware multiplier is over 2 times faster than the software multiplier. Total is the number of clock cycles at the end.

**Calc power efficiency: Hardware Multiplication efficiency:**

**Table

Description automatically generated**Table

Description automatically generated

**Software Multiplication efficiency:**

**Table

Description automatically generated**

## Observations:

Hardware implementations are much easier to code and use. I am very interested to see if we keep doing something like hardware multipliers because it is honestly really cool. We do enough software in my opinion and using hardware to our advantage ( and to its advantage ) seems to be a much better use of our time.

**Conclusion**

Write your conclusion. (Explain what you have learnt and issues you faced)

Folder Link:

https://drive.google.com/drive/folders/1\_Y3ABMDhCUxc9phtQ8JKHCM8LDOK7k7J?usp=sharing

Video link:

<https://drive.google.com/file/d/1DmtWEoQLX_pTyNtjfBN_lhFrb2UboLKI/view?usp=sharing>

**Appendix on the next page**

**Appendix**

Appendix 1 – main.asm

|  |
| --- |
| ;-------------------------------------------------------------------------------  ; File : main.asm  ; Function : Push data onto the stack and call a function in another file  ; Description: This file creates an array of 10 bytes for a hardware and software  ; multiplier  ; Input : A b value to be multiplied  ; Output : No output, see registers and memory browser.  ; Author : N. Anderson npa0002@uah.edu  ; Date : 09/30/2020  ;-------------------------------------------------------------------------------  .cdecls C,LIST,"msp430.h" ; Include device header file  ;-------------------------------------------------------------------------------  .def RESET ; Export program entry-point to  ; make it known to linker.  .ref calc\_power  ;-------------------------------------------------------------------------------  ; Allocation for result  **.bss** swarr, 10 ; 10 bytes for software array  **.bss** hwarr, 10 ; 10 bytes for hardware array  **.data**  **b:** .int 3 ; int variable to intput into the code.  **result:** .int ; result variable  ;-------------------------------------------------------------------------------  **.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:**  **push** #swarr ; Push the software array address onto the stack.  **push** #hwarr ; Push the hardware array onto the stack.  **push** result ; Push the result onto the stack.  **mov** b, R4 ; Mov b into R4.  **mov** #5, R8 ; Counter for the loops in calc\_power.  **call** #calc\_power ; A call to the calc\_power.asm file to start the loops.  **jmp** $ ; jump infinetly.  **lend:**  **nop**  ;-------------------------------------------------------------------------------  ; Stack Pointer definition  ;-------------------------------------------------------------------------------  **.global** \_\_STACK\_END  **.sect** .stack  ;-------------------------------------------------------------------------------  ; Interrupt Vectors  ;-------------------------------------------------------------------------------  **.sect** ".reset" ; MSP430 RESET Vector  **.short** RESET |

Appendix 2 – calc\_power.asm

|  |
| --- |
| ;-------------------------------------------------------------------------------  ; File : calc\_power.asm  ; Function : This file takes the data off of the stack, stores it into registers  ; and uses those register values to perform operations  ; Description: calc\_power takes the data off the stack and initializes different data.  ; Input : Input comes from main.asm which is really just the stack data.  ; Output : Output is in the memory browser.  ; Author : N. Anderson npa0002@uah.edu  ; Date : 09/30/2020  ;-------------------------------------------------------------------------------  .cdecls C,LIST,"msp430.h" ; Include device header file  .def calc\_power ; Define the calc\_power function.  .ref SW\_Mult ; Reference to the Software multiplier file.  .ref HW\_Mult ; Reference to the Hardware multiplier file.  **.text**  **calc\_power:**  **mov** 6(SP), R5 ; Move the Software array address into R5  **mov** 4(SP), R6 ; Move the Hardware array address into R6  **mov** 2(SP), R7 ; Move the result into R7  **mov** #1, R12 ; R12 will get 1 for the first iteration of the software loop.  **jmp** SWLoop ; Jump to the SWLoop, it will carry out from there.  **SWLoop:** **call** #SW\_Mult ; Call the software multiplier.  **incd** R5 ; Increment R5 to the next index.  **dec** R8 ; Decrement R8 (counter).  **jnz** SWLoop ; Jump not zero back to software loop.  **mov** #5, R8 ; reset power counter  **HWLoop:** **call** #HW\_Mult ; Call the hardware muliplier.  **incd** R6 ; Increment to the next index.  **dec** R8 ; Decrement the counter.  **jnz** HWLoop ; Jump not zero back to hardware loop.  **lend:**  **nop** |

Appendix 3 – SW\_Mult.asm

|  |
| --- |
| ;-------------------------------------------------------------------------------  ; File : SW\_Mult.asm  ; Function : This files function is to perform software multiplication on a value  ; Description: This file takes in R7 and R4 and uses shift - add - multiplication  ; to find the powers of 4 from 1-5  ; Input : R4 and R7 from main and calc\_power.asm  ; Output : See memory browser  ; Author : N. Anderson npa0002@uah.edu  ; Date : 09/30/2020  ;-------------------------------------------------------------------------------  .cdecls C,LIST,"msp430.h" ; Include device header file  .def SW\_Mult  **.text**  **SW\_Mult:**  **mov** #16, R9 ; Initializing a bit counter.  **mov** #0, R10 ; Initialize the result number to R10.  **mov** R4, R11 ; Move b into R11 so that we can perform operations w/o changing its value  **jmp** bitcheck ; Jump straight to bitcheck.  **bitcheck:**  **mov** R12, R13 ; Mov b into R13 so that we can perform operations without changing its value  **and** #0x01, R13 ; bitchecking to see if the LSB is 1  **cmp** #0x01, R13 ; Cmopare operation for previous AND instruction  **jne** noadd ; If it is not equal (i.e. = 0) then skip the add instruction.  **add** R11, R10 ; Add to the running total  **jmp** noadd ; Go straight to s  **noadd:**  **rrc** R12 ; Shift B to right by 1.  **rla** R11 ; Shift A left by 1.  **dec** R9 ; Decrement the bit counter.  **jnz** bitcheck ; If bit counter is not zero, go to next bit to add.  **mov** R12, R13 ; If it is 0, move the result into R13.  **and** #0x10, R13 ; Check to see if it is a negative number  **cmp** #0x10, R13 ; If it is negative, jump to subtract from A from result.  **jeq** neg ; Jump if equal to negative  **jmp** end ; Unconditionally jump to end  **neg:**  **sub** R11, R10 ; Sub instruction for negative numbers  **jmp** end ; Unconditionally jump to end.  **end:**  **mov** R10, 0(R5) ; Move the result into the next part of the  **mov** R10, R12 ; Put the result into B for the next power operation.  **ret** ; Return back to calc\_power.asm |

Appendix 4 – HW\_Mult.asm

|  |
| --- |
| ;-------------------------------------------------------------------------------  ; File : HW\_mult.asm  ; Function : To perform a hardware multiplication on predefined values  ; Description: This file will take in values and use hardware multiplication to  ; find the powers from 1-5  ; Input : R4 and R7 from main and calc\_power.asm  ; Output : See memory browser  ; Author : N. Anderson npa0002@uah.edu  ; Date : 09/30/2020  ;-------------------------------------------------------------------------------  .cdecls C,LIST,"msp430.h" ; Include device header file  .def HW\_Mult  **.text**  **HW\_Mult:**  **mov** R4, &**MPY** ; move R4 into the unsinged 16 bit multiplication register  **mov** R7, &OP2 ; move R4 into the general purpose operator. multiply by...  **nop**  **nop**  **nop**  **mov** RESLO, R7 ; move the result the R7.  **mov** R7, 0(R6) ; move result into R6  **ret** |