Laboratory 4

EGCP 280: Microcontrollers

Updated 10/5/16

Due: ***Wednesday, October 12, 2016 by 6:30PM*** *via TITANium*

Checkoff: ***Monday, October 10, 2016 by 8:20PM*** *during the second half of lab*

Lab Objectives

The objective for this lab is to become familiar with the stack. This will be achieved through the simulation of a simple postfix notation (a.k.a., Reverse-Polish Notation or RPN) calculator.

Description

A stack is a Last-In, First-Out (LIFO) data structure. A stack is characterized by two fundamental operations: push and pull (a.k.a., pop). The push operation adds a new item to the top of the stack and the pull operation removes an item from the top of the stack. The HCS12 contains several other instructions that can be used to manipulate and access data stored in the stack. In this lab, you will use the pull, push, and the other stack instructions to carry out the following tasks.

The stack is probably the one concept in assembly programming that is most critical to embedded programming, and is rarely used when programming in higher level languages. The stack is a section of memory that is allocated for temporary data storage. A very rough metaphor, not to be taken too far, would be to compare the stack to the RAM of your PC. The stack does not hold data that can be manipulated; it's simply a scratch pad for holding things briefly. Most high level languages such as Java and C++ utilize the stack to hold all of their temporary data values instead of registers. This makes them more general between architectures.

Postfix Notation

Postfix notation is a mathematical notation wherein every operator follows all of its operands, in contrast to prefix position. In postfix notation the operators follow their operands. For instance, to add 3 and 4, one would write "3 4 +" rather than "3 + 4". If there are multiple operations, the operator is given immediately after its second operand. So, the expression written "3 − 4 + 5" in conventional infix notation would be written "3 4 − 5 +" in postfix notation: first subtract 4 from 3, then add 5 to that. An advantage of postfix notation is that it obviates the need for parentheses that are required by infix. While "3 − 4 x 5" can also be written "3 − (4 x 5)", that means something quite different from "(3 − 4) x 5". In postfix, the former would be written "3 4 5 x −", which unambiguously means "3 (4 5 x) −" which reduces to "3 20 −"; the latter would be written "3 4 - 5 x", which unambiguously means "(3 4 -) 5 x".

Interpreters of postfix notation are often stack-based. That is, operands are pushed onto a stack, and when an operation is performed, its operands are popped from a stack and its result pushed back on. So, the value of postfix expression is on the top of the stack. Stacks, and therefore postfix notation, have the advantage of being easy to implement and very fast.

Despite the name, postfix notation is not exactly the reverse of Polish notation. For the operands of non-commutative operations are still written in the conventional order. For example, "/ 6 3" in prefix notation and "6 3 /" in postfix both evaluating to 2, whereas "3 6 /" in postfix notation would evaluate to ½.

Examples

Load the stack with 16-bit values from the index registers X and Y and then store the top two 8-bit values in the stack into accumulators A and B.

```
LDS     #PROG       ; Initialize the stack pointer to $2000
LDX     #$0000
LDY     #$FFFF
PSHX                ; X (16-bit) → stack
PSHY                ; Y (16-bit) → stack
```

After executing this code, if you were to look at the memory addresses $1FFF to $1FFC and the registers it would look like:

	$1FFB	--	X	$0000
Stack Pointer →	$1FFC	$FF	Y	$FFFF
	$1FFD	$FF	A	--
	$1FFE	$00	B	--
	$1FFF	$00		
	$2000	--		

Figure 1: Stack pointer and register contents after pushing index registers onto stack.

The stack pointer is now pointing to address $1FFC. Continuing the program:

```
PULA                ; 8-bit of stack → A
PULB                ; 8-bit of stack → B
```

The stack pointer now points to $1FFE:

	$1FFB	--	X	$0000
	$1FFC	--	Y	$FFFF
	$1FFD	--	A	$FF
Stack Pointer →	$1FFE	$00	B	$FF
	$1FFF	$00		
	$2000	--		

Figure 2: Stack pointer and register contents after pulling two 8-bit values from stack into accumulators A and B.

The stack will grow without bound. If you keep pushing values onto it, the stack pointer will continue to decrease, and you will write over more and more memory. If you were using a specific value of RAM to store a temporary data value not in the stack, and the stack pointer then points to that address, you will lose the old value. So be careful!

Work Tasks

Step 1

Allocate your stack such that the stack is located in what we defined as the data section of RAM. Remember, we defined our data section with the label "PROG" using the ORG assembler directive. Use the LDS instruction to initialize your stack. Now, load the stack using a loop with the values shown below. After the stack is loaded, use the TSX command to transfer the SP to index register X and then to load accumulator A with $23 and accumulator B with $67 (HINT: take a look at the lecture on stacks).

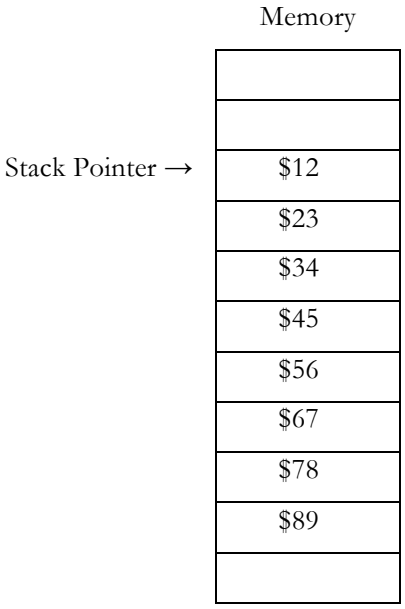

Figure 3: Stack and its contents.

Step 2

Code the simple postfix notation calculator that performs the operation shown in Figure 4. This calculator will only perform one operation ((35) (5) (-2) + (10)x-(4)+). The infix expression of the operation to be performed is "35 + ((5 + -2) x 10) + 4". Don't worry about saving the operators on the stack. Always push values on the stack. However, when you reach an operator, pull the values off the stack into registers, perform the arithmetic operation, and push the result back on the stack according to Figure 4. Your final result should be 9 ($09) and should be on the top of the stack.

NOTE: the multiply instruction (MUL) multiplies two 8-bit numbers and returns a 16-bit number. In this case, only return the lower 8-bit as the result.

Input	Operation	Stack	Comment
35	Push Value	35	
5	Push Value	5 35	
-2	Push Value	$FE 5 35	
+	Add	3 35	Pull two value (5, -2) and push result (3)
10	Push Value	10 3 35	
x	Multiply	30 35	Pull two value (3, 10) and push result (30)
-	Subtract	5	Pull two value (35, 30) and push result (5)
4	Push Value	4 5	
+	Add	9	Pull two value (5, 4) and push result (9)
	Result	(9)	

Figure 4: Postfix notation operations. Note, if a number doesn't have a prefix, like $ or %, it is in decimal.

Step 3

Now take the postfix notation calculator a step further. Using a loop, branches, and the stack, create a postfix notation calculator that performs the +, -, x, and / operations, as shown in the example in CALC_IN. Add the variables below to the DATA section of your code. Use these variables to implement your calculator. The operations should be the machine opcode as follows: **+: $8A, -: $8B, x: $8C, and /: $8D**. For example, if you want to use ABA to implement +, the value should be $18. Make sure to replace those ??s with what you want to use before you try to use the code. CALC_IN contains the variable with the operation to perform. Use conditional statements and your knowledge of the opcode to determine if there is an opcode, and if so which operation to perform. The values in CALC _IN below are initially 84/2*21+- and the result should be 1. Try another operation of your choosing by changing CALC_IN and discuss it in the methodology section of your report. Keep in mind you won't be able to use any values that are the same as the opcodes you are using, but any other value should work, including negative values.

```
CALC_IN     FCB     $08,$04,$8D,$02,$8C,$02,$01,$8A,$8B
                    ; 8   4   /   2   *   2   1   +   -   =1
CALC_OUT    RMB     1
```

```
CALC_START FDB     CALC_IN    ; Pointer to start of RPN array
CALC_END   FDB     CALC_OUT-1 ; Pointer to end of RPN array
```

HINTS: You will traverse the CALC_IN array in a similar fashion to lab 3. You will jump out of the array when an operator is found, pull two values off the stack, perform that particular operation, push the result onto the stack, and then return to the loop. Otherwise, you will just push the value on the stack.

NOTE: the divide instruction (IDIV) requires two 16-bit registers and returns a 16-bit value. Only return the lower 8-bit as your result.

What to Turn In (Please read this carefully)

You are to put your code as text and screenshots into the "lab_4_template.docx" in the designated areas. This will be the file that you upload to TITANium as your submission. Provide your code as text, not an image. Points will be deducted if it is an image. You are then to take screenshots of your program in both CodeWarrior and terminal (i.e. on the Dragon board). The screenshot should show the relevant CPU registers and memory locations (NOTE: a convenient way of getting screenshots is to use the "snipping tool"). Do NOT crop the screenshot, for any credit I want a full screen screenshot of your computer screen including the task bar. Also you must answer the questions on the "lab_4_template.docx" file.