CSCE 230L – Lab 3: Subroutines and Stacks

January, 2014

Objectives

- Learn instructions about ‘shift’ and ‘multiplication’
- Write simple assembly programs with subroutines
- Learn how to make use of the stack for subroutines and parameter passing

Useful References

Appendix B in the textbook (sections 1 through 6)

List of instructions for the NIOS II Processor (Instruction Set Reference):
http://www.altera.com/literature/hb/nios2/n2cpu_nii51017.pdf

What to Turn In

- Pre-lab questions (due at the beginning of lab; on paper)
  - All questions total to 20 points
- Code and questions from lab (submit on handin)
  - Part one assembly files (30 points)
  - Part two assembly files (50 points)

Make separate code files for each task and answer all the questions in a single text file. For the code, all you need to submit are the assembly files (.s files). Other files from a monitor program project are not needed.

ATTENTION:

Use different file names; otherwise they may replace each other in ‘web handin’ system.

Name (1 point):

Pre-lab (19 points)

1. Read through the entire lab.
2. Examine these short segments of code. (You can look up instruction usage in 'Instruction Set Reference'.)

   case A:

   ```
   addi    r2, r0, 3
   addi    r3, r0, 4
   mul     r5, r2, r3
   ```

   case B:

   ```
   addi    r2, r0, 3
   addi    r3, r0, 2
   sll     r5, r2, r3
   ```

   a. In each case, what is the ending value in r5?

   b. If we change the last instruction in case B to 'sll r5, r3, r2', what will be the final value of r5?

   c. Using shift instructions and add instructions, write code to implement muli r2, r2, 20 (suppose we are given the value of r2 at the beginning). Hint: $20=2^2+2^4$

The following questions refer to Figure B.5 in the textbook on page 543.

Reference: Figure 2.18 and 2.19.

3. Why do we push 'r2, r3…r5' on the stack in subroutine 'LISTADD'? Why do we pop 'r2, r3... r5' from the stack at the end of the subroutine?

4. Using a figure like the one in Figure 2.19, draw the stack just after instruction ‘stw r3, 20(sp)’ (the instruction right after the loop).

5. Why is it better to have the **called** routine store registers on the stack, rather than having the **calling** routine store its registers on the stack beforehand?

6. Explain the differences among the three instructions: br, jmp, call. Which registers will change its value?

7. How can a subroutine return to the main program? Which register will change its value?

8. What else needs to be stored on the stack when there are nested subroutine calls? Why?

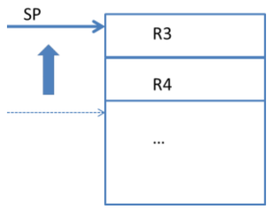
Lab

Part 1

Hopefully you know by now that programs are broken up into several functions. In assembly, if we want to pass parameters to a subroutine (functions), not worry about subroutines overwriting registers used by other routines, or be able to return when we have nested subroutine calls, we need a way to store values that can be retrieved later. This is done through use of a stack, which is implemented as a portion of memory set aside for this particular use.

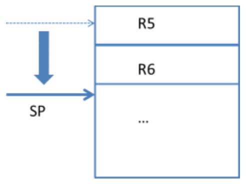
To use the stack in our programs, we are given a pointer to the "top" of the stack, with the top being the most recent value pushed on. At the beginning of our programs, we need to initialize the stack. When you have a value you want to put on it, you first need to decrement the pointer and then store the value at the new stack pointer address. The following code gives an example of pushing two values onto the stack.

```
addi    sp, sp, -8      /* decrement stack pointer */
stw     r4, 4(sp)       /* push r4 to stack */
stw     r3, (sp)        /* push r3 to stack */
```



At some point we will need to retrieve the value we put on the stack (or else there was no need to put them on). This process is simply the reverse of putting them on. We load the values from the stack and then restore the stack pointer to its previous address (increment the stack pointer). The following code undoes what the previous code did.

```
ldw     r5, (sp)        /* get value that was in r3 off stack */
ldw     r6, 4(sp)       /* get value that was in r2 off stack */
addi    sp, sp, 8       /* restore (increment) stack pointer */
```



As a general rule, **each subroutine shall save (preserve) all registers that it will use before it does anything else**. For example, say routA wants to call and pass one parameter to routB, and routB will use r2-r5:

- The first thing routB will do is put values in those registers on the stack in order to not lose their current value during execution of routB.
- Then routB does what it needs to do, and overwrites the top parameter on the stack with the return value.
- Finally, routB restores the values of r2-r5 so that routA can resume with them intact when routB returns to routA.

The following code should help show the idea.

```
routA:
        …
        addi    sp, sp, -4      /* decrement stack pointer */
        stw     r4, 0(sp)       /* push value of r4 as parameter */
        call routB
        ldw     r5, 0(sp)       /* pop return value into r5 */
        addi    sp, sp, 4       /* restore stack pointer */
        …

routB:
        addi    sp, sp, -16     /* make room on stack */
        stw     r2, 12(sp)      /* save value of r2 */
        stw     r3, 8(sp)       /* save value of r3 */
        stw     r4, 4(sp)       /* save value of r4 */
        stw     r5, 0(sp)       /* save value of r5 */
        ldw     r2, 16(sp)      /* load parameter into r2 */

        …                       /* perform some function, assume end result is in r3 */

        stw     r3, 16(sp)      /* push return value on stack */
        ldw     r2, 12(sp)      /* pop value of r2 */
        ldw     r3, 8(sp)       /* pop value of r3 */
        ldw     r4, 4(sp)       /* pop value of r4 */
        ldw     r5, 0(sp)       /* pop value of r5 */
        addi    sp, sp, 16      /* restore stack pointer */
        ret                     /* return to routA */
```

You may have been wondering what the **.global _start** means at the beginning of your file. This gives your code a name by which other code can reference your code. You can think of it as a **function name**. For one program to call another, the call instruction is used. You can also use it to call a procedure defined elsewhere in the program. Then when the other program or procedure is finished, it executes the instruction **ret**, which will return to the location where it was called.

Task 1 (30 points)

We'll start off with a simple program. Write the file main01.s that gives values to r2 and r3 (any value) and puts it on the stack, and calls a subroutine mult.s, which multiplies the number in r2 by 12 with shift instructions, and add the number in r3, the final result will send to r4. An overview of the two programs is given below.

For main01.s, partial code:

```
.text

.global _start

.extern LABEL

_start:
            movia sp, 0x007FFFFC   /* initialize stack */
            addi r2, r0, X         /* give a value (let's say X) to r2 */

            addi r3, r0, Y         /*give a value (let's say Y) to r3*/
            …                      /* push r2, r3 onto the stack */
            call LABEL             /* call subroutine 'LABEL' */
            …                      /* pop return value of mult from top of the stack into r4 */
            …                      /* restore stack pointer to where it was */
```

For mult.s, partial code:

```
.text

.global LABEL

      LABEL:

                  …                /* push the registers this subroutine will use on stack
      to
                                     preserve their value */
                  …                /* read parameter from stack, put into r5 and r6 (i.e.
      r5      has a value of X and r6 has a value of Y */
                  sll …            /* compute return value (12*X) */

                  add …            /*add up 12*X+Y*/
                  …                /* put return value on stack */
                  …                /* restore value to registers that were put on stack */
                  ret              /* return to main01.s */
```

Part 2

With the stack, we can pass parameters, and subroutine code can use the registers without worry of overwriting values that were being used in the calling routine. Also, when we have nested subroutines, **the return address needs to be pushed on the stack by the first subroutine before it calls another subroutine** because when it does, the link register will be updated to the address of the call function in the first subroutine. Hopefully this simple example will make this clearer. We have three routines A, B, and C. Routine A calls routine B, and routine B calls routine C. As we can see, routine B stores the current return address onto the stack before it calls routine C. That way, when routine C returns, routine B can restore the value of the return address so that it can return to routine A.

```
A:      movia sp, 0x007FFFFC   /* initialize stack to largest memory value */
        …
        call B
        …


B:      …
        addi    sp, sp, -4     /* decrement stack pointer */
        stw     ra, (sp)       /* push return address on stack */
        call C
        ldw     ra, (sp)       /* pop return address from stack */
        addi    sp, sp, 4      /* increment stack pointer */
        …
        ret                    /* return to routine A */


C:      …
        ret                    /* return to routine B */
```

Task 2 (50 points)

Now it's time to do something a little more complicated. You get an array with 10 arbitrary integers; you can assign values to them. Your task is to subtract each number by its index in that array, and then sum up.

- You need to have a main function get the address of that array.

- In your first subroutine, load each number from memory, and subtract it by its index, then store back to memory; push the return address on the stack, and call the second subroutine.
- In your second subroutine, you sum up the new array and push the result on the stack, then return to your main function.
- Your main function can fetch the result from the stack.

The formula is:

$$\text{Result}=\sum_{0}^{9}(a[i]-i)$$

The following offers a template.

In your main function 'main02.s'	In your first subroutine	In your second subroutine
.text	.text	.text
.global _start	.global SUB	.global SUM
.extern SUB	.extern SUM	
		SUM:
_start:	SUB:	…
	…	…
…	addi sp,sp, -4	ret
call SUB	stw ra, 0(sp)	
…	call SUM	
	…	
	ldw ra, 0(sp)	
	addi sp,sp, XX	
.data	ret	
ARRY:		
…		
.end		