ICSI404 – Assignment 10 – Stack It Up!

This assignment builds on the previous assignment.

Our computer so far can implement all the instructions that can go inside a function (think about C). What it cannot do is call functions, pass parameters, return values and return to where we called from. With this assignment, we will implement the last 4 assembly instructions that we need in order to do these things.

The Stack

CPUs typically have a stack implementation built in. Remember that a stack is like a stack of cards. The last thing in is the first thing out (LIFO). With a stack, we can push and pop values.

Consider, for a moment, how this C program might work:

```
int add (int a, int b) {
      return a+b;
}
void main() {
    int c = add(3,4);
}
```

Of course, it does not output anything, but that's not our emphasis here. Compilers would typically output something like this in assembly:

```
add:  pop R15 // stack:  3 4
      pop R1 // stack: 3
      pop R2 // stack: empty
      add R1 R2 R3
      push R3 // stack: 7
      push R15 // stack: 7 RN
      return // after this, stack: 7

main: move 3 R1 // stack: empty
      move 4 R2
      push R1   // stack:3
      push R2 // stack: 3 4
      call add // stack 3 4 RN
RN:   return // stack: 7
```

The parameters to the function are pushed onto the stack from registers. Inside the function they are popped off of the stack into registers. There are two other new instructions here – call and return. **Call is just like jump, but it pushes the address of the NEXT instruction after the call onto the stack. When return (in add) happens, it pops the return address from the stack and moves it into PC.**

The example above has a feature that we did not implement into our assembler – labels. Labels are really convenient to the assembly language programmer; without them, we have to count instructions and multiply by 2 to figure out the destination of JUMP or CALL. Now imagine if you edit your code and have to re-count all of your JUMP or CALL instructions! In an "real" assembler, the labels indicate the target of the CALL or JUMP and the assembler does the counting for you.

Since we are **not** implementing labels, you will have to do the counting yourselves on your examples. A simple CALL example might look like this:

```
CALL 6
INTERRUPT 0
HALT
INTERRUPT 1
RETURN
```

In this example, CALL pushes the address of the next instruction (INTERRUPT 0) onto the stack; that value is 2 in this case. CALL jumps to 6.

Interrupt 1 happens. RETURN pops the 2 from the stack and jumps there.

Interrupt 0 happens and then we halt.

For this assignment, you will need to create these four instructions (call, return, push, pop). The opcode for all four instructions will be: 0110.

Push will look like this:

0110 0000 0000 RRRR // R = register bit

Pop will look like this:

0110 0100 0000 RRRR // R = register bit

Call will look like this:

0110 10AA AAAA AAAA // A = address bit. These are like jump – absolute and not an offset

Return will look like this:

0110 1100 0000 0000 // No variable bits

You will need to create a stack pointer in your CPU. This is a longword that points to an address in memory that will indicate the NEXT place to write the stack. You should call this SP (stack pointer). It should be pre-initialized to 1020. Why? We are going to start at the end of memory (remember that we have 1024 bytes, so memory ranges from 0 – 1023). We read/write 4 bytes (32 bits) because that's the

size of a register. So the first push would go into bytes 1020, 1021, 1022, 1023. We would subtract 4 from the SP, giving 1016. The next push would go into bytes 1016, 1017, 1018, 1019. Pop would add 4 to the SP.

Call is a combination of push and jump. Return is a combination of pop and jump. Push doesn't change the register that it is copying from.

You must add these instructions to your assembler.

You must test these new instructions using your assembler (as we did before). Create a small program to test these instructions and place it in cpu_test3.

You must submit buildable .java files for credit.

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Unit Tests	None (0)	Partial Coverage (7)	All methods covered, needs more cases (13)	All methods/cases covered (20)
Push	None (0)	Attempted (8)		Completely working (15)
Pop	None (0)	Attempted (8)		Completely working (15)
Call	None (0)	Attempted (8)		Completely working (15)
Return	None (0)	Attempted (8)		Completely working (15)