TCSS 372 Project Description

Simulating the LC-3 Computer

Purpose

In this project you will be working with two or three classmates to build a simulation of the LC-3 computer similar to the one in the textbook. This project will solidify your understanding of the architecture of the machine and provide some opportunities for you to hone your programming skills. In this project you will have some options on the level of challenge you are willing to take on. Students who are aggressively ready to step up to those challenges will also be candidates for higher grade points as explained in the section on grading.

The LC-3 is a minimal von Neumann computer but it has all of the features that are necessary for implementing a Turing-complete machine. Building a simulator for this machine will demonstrate your mastery of the concepts of computing as well as your mastery of fundamental programming capabilities. Students who have met the higher challenges of this project in the past have been able to obtain significant employment in top firms that recognize the importance of being able to handle this kind of work.

Description

The basic LC-3 architecture is fully described in the textbook by Patt and Patel. Reference is made to Appendix C in particular. You will be writing a large C program that implements this architecture. Several optional enhancements will allow those taking on the challenge to explore more fully the advantages of more advanced machine architectures such as cache memories and pipeline design.

Requirements

The Basic LC-3 ISA

Complete implementation and testing of all instructions except the RTI. You will also implement a push and pop instruction (using the reserved instruction as we did in problem 1). Implement the following features for the debug monitor:

2) Save. This means save to a file. Prompt the user for a file name, check for overwriting the existing file, then write the current contents of memory down to and including the HALT (TRAP 25) memory location in the .hex file format (for future loading).

6) Edit. Allows the user to change a value in a specific memory location. Prompt the user for the memory address to be edited, display that address along with the current contents and prompt the user for the new contents to be entered in hex. After translating the hex string into an integer and putting it into the memory location indicated, redraw the memory dump with the address edited centered (e.g. item 8 on the screen).

8) (Un)Set Brkpt. As per suggestion in problem 4, prompt the user for an address at which to set or unset a breakpoint (up to 4 would be more than enough breakpoints). This facility can only be used in Run mode. You will need to modify the FETCH-DECODE-EXECUTE cycle so as to check at each value of the PC if it is in the array (table) of breakpoints. If it is, set the run mode to step and call the monitor to display the current state of the machine and memory. The cycle will remain in step mode until the user presses #4 (Run) again. Play with the book simulator to set and unset breakpoints to see how this works. If the address input by the user is already in the array it means that they wish to unset the breakpoint, in which case set the array slot to a signal value (e.g. 99999) indicating the slot is available.

Testing

Using example code in Chapter 10 of Patt & Patel, implement push and pop algorithms in LC-3 code with overflow and underflow checks, respectively. You may write a relatively simple application, such as a calculator similar to the one in Chapter 10. However you are going to adopt the use of the stack for saving the return address (R7) to support nested subroutine calls, and passing the arguments (pushed in order of use as in a C function call) to subroutines. The return value from the subroutine may be sent back in R0 as the TRAP values are.

The calling subroutine does the following:

1. push the arguments onto the stack
2. JSR or JSRR to the subroutine
3. upon return, uses the value returned in R0 (which could be an error flag or a computed value.

The called subroutine does the following

1. pops the arguments off the stack and saves them locally or simply keeps them in registers as needed.
2. pushes R7 onto the stack if it will be calling another subroutine
3. puts the result into R0
4. pops R7 if it had pushed it in #2
5. RET

The objective is to test all of the implemented instructions not previously implemented in problems. In particular design an algorithm that uses the LDI and STI instructions.

Assemble the program and test it on the textbook simulator to make sure it does what it is supposed to do. Once you are confident that the program works port the object file (HEX) to your simulator and run it.

Run the program in step mode to obtain screen shots of those instructions not previously tested. You will also take screen shots of the program processing breakpoints (i.e. when a breakpoint is encountered the program stops and the debug monitor is displayed – take a screen shot of that happening.)

EXTENDING THE INSTRUCTION SET

Next, take this program and replace the push and pop *subroutines* with simpler subroutines that use the reserved instruction to implement push/pop similarly to what we did for shifting in problem 1. This will take some careful thought on your part. You cannot just use the real LC-3 editor to do this since the reserved instruction (1101) throws an exception you can't use it on the LC-3 simulator. So you will have to figure out how to modify the .hex file (hint: use the .lst file for clues about how to write replacement subroutines.) This will involve a little bit of hand assembling. However the strategy is to take out the LC-3 code implementing the actual store and load plus decrementing or incrementing R6, and replace it with a single instruction as follows:

```
PUSH Sr        ;  Sr is taken from the DR field of the instruction format.
               ; R6--; M[R6] ← Sr
Op    Sr      Dir
1101: xxx: 000000000
                ^
               +------------------ bit 5 used to signal direction, 0 means push

POP Dr         ; Memory at top of stack (R6) is loaded to Dr
               ; Dr ← M[R6]; R6++
Op    Dr      Dir
1101: xxx: 000100000
                ^
               +------------------ bit 5 used to signal direction, 1 means pop
```

Note that in this implementation we are automatically incrementing and decrementing R6 as needed, so there is no need to ADD R6, R6, #1 or #-1 explicitly. Of course this means you need to plan on R6 only being used as the stack pointer and nothing else.

This code will only run on your simulator! You can try it on the LC-3 simulator but it should throw an illegal opcode exception! However, when you run it on your simulator it should work just fine. When you do this you can go to management and say you are ready to launch the upgraded LC-3 cpu instruction set!!!

Take screen shots of your program running with this new instruction. Set breakpoints at the push and pop subroutine calls and then step into the revised subroutines showing what happens when your new instruction is run. This means showing the stack location in memory and what happens to R6.

Advanced Features – Challenges for Getting More Points

1. **Simulating a Cache Memory Feature**

This challenge will promote any grade you get on the basic implementation to the next highest level (e.g. from B to B+). This add on challenge requires you to implement a simple L1 Cache for instructions and an L1-L2 cache for data (these are the I-MEM and D-MEM boxes in the pipelined LC-3, but you don't need a pipeline to have them.) The main aspects of these caches will be cache coherency (i.e. memory and caches have to be consistent) and minimizing miss rates (sizing the caches will be an issue).

I will provide more details on what is expected when we get to caches in week 8.

2. Simulating a Pipelined LC-3

If you really want to do something challenging then implementing the buffered pipeline architecture for the LC-3 is what you want! This will require some rewriting of the controller since we turn from instruction states to pipeline stages. The basic for-loop will remain, but each of the stages will be working on a different instruction. So each of the FETCH, DECODE/RR, EXECUTE, MEMORY, and STORE (write-back) will be doing different things to different instructions moving through the pipeline buffers. You will need to implement simple stall and NOP bubble treatments for some hazards as well as pipeline flushing for branch handling. I don't expect you to get into branch prediction (unless you are a masochist!)

A successful implementation of this capability is definitely worthy of an A grade range. I will work with the groups that elect to attempt this option.

3. **Simulating the Interrupt Capabilities**

Don't even ask. I have **NOT** done this for the LC-3 myself (keep meaning to get to it). But there are a few hotshots in the class who might want to give it a go. Points that could backfill the exams are on the table!

Grading

You will be giving a public demonstration in the last week of classes (Friday participation day mandatory without prior notice). If you can show that your program does all of the requirements set forth above I will only peruse your code submission for any anomalies I think you should know about. Basically if it runs in demonstration, that is 50% of your project grade. The demo consist of you running your program showing your monitor on the projection screen, each person providing a short overview of their part of the project, and then answering any questions from the other students. Another 50% comes from your documentation so do not go slack on the comments and your .docx file with labeled screen shots.

SUBMISSION

Submissions will be handled in the same manner as Problem 4.