Lab 7: Ibcm last modified by Aaron Bloomfield on October 25, 2013 8:51:22 AM EDT

Laboratory 7: IBCM Programming

Week of Monday, October 28, 2013

Objective

To become familiar with programming with IBCM, and understand how high-level language programs are represented at the machine level.

Background:

IBCM (Itty Bitty Computing Machine) is a simulated computer with a minimal instruction set. Despite its tiny small instruction set, the IBCM can compute anything that a modern "powerful" computer can compute.

Reading(s):

1. Read the slides on IBCM
2. Read IBCM book chapter
3. Run IBCM code online at http://www.cs.virginia.edu/~cs216/ibcm/. The sample code in the book chapter and also online: code-ex-1.txt and code-ex-2.txt

Procedure

Pre-lab

1. The online IBCM simulator is available at http://www.cs.virginia.edu/~cs216/ibcm. If that site is down, mirror websites are listed in the pre-lab section.
2. Write the two IBCM programs described in the pre-lab section: addition.ibcm and array.ibcm.
3. Note that some browsers have problems with the online simulator. If in doubt, use Firefox.
4. Your submitted files MUST have an .ibcm extension (not .ibcm.txt), and can NOT have any blank lines!
5. Files to download: IBCM reference files (sample programs, documentation, etc.) listed in the Readings section, above
6. Files to submit: addition.ibcm, array.ibcm. Make sure they are not named addition.ibcm.txt and array.ibcm.txt!

In-lab

1. Implement the bubble sort algorithm in IBCM from the C++ code. See the in-lab section for details.
2. Discuss any problems getting your programs to work with a TA.
3. If you are unable to finish the lab during the in-lab time, submit an extension request.
4. Your submitted files MUST have an .ibcm extension (not .ibcm.txt), and can NOT have any blank lines!
5. Files to download: bubblesort.cpp
6. Files to submit: bubblesort.ibcm. Make sure it's not named bubblesort.ibcm.txt!

Post-lab

1. Read chapter 3 of the shell script tutorial (http://www.freeos.com/guides/lsst/) for this lab.
2. Complete the shell script described in the post-lab section.
3. Your shell script MUST have 'a.out' as the executable name, not 'a.exe'.
4. Implement a "quine" in IBCM (see "What is a Quine?" section, below) into a quine.ibcm file.
5. You may discuss this question with your classmates, as long as it is high-level design issues, not IBCM-specific implementation issues (i.e. what you turn in should be your own work).
6. Submit a report, called postlab7.pdf (see the post-lab section for formatting details), that contains your thoughts on IBCM. What did you think? How easy was it to use? Would modifications to the simulator make life easier for you? How confident do you feel in writing IBCM code?
7. Your submitted files MUST have an .ibcm extension (not .ibcm.txt), and can NOT have any blank lines!
8. Files to download: counter.cpp

9. Files to submit: averagetime.sh, quine.ibcm (don't name it quine.ibcm.txt!), postlab7.pdf

---

Pre-lab

Using the IBCM Simulator

The easiest way to use the simulator is via the online version, available at http://www.cs.virginia.edu/~cs216/ibcm. There are a number of mirrors of this website available (all are identical):

- http://libra.cs.virginia.edu/~aaron/ibcm/
- http://people.virginia.edu/~asb2t/ibcm/

We request that you use the first URL (http://www.cs.virginia.edu/~cs216/ibcm), and use the others if that one is not available.

There are pros and cons to the online version of the emulator. The online version does not require installation, allows for inline memory modification, but will hang your browser if it gets stuck in an infinite loop. Also, the online simulator gets rather unhappy if there are extra blank lines at the end of your input file.

You will be developing a somewhat more complicated IBCM program in the in-lab. So, as preparation, study the IBCM documentation and lecture notes. Additionally, you will be expected to come to the in-lab with two working IBCM programs. You will not get much from the lab unless you are thoroughly familiar with the documentation and lecture notes — so make sure you are familiar with the material!

A separate simulator (with GUI interface) is available at http://www.eemta.org/~jwelsh/progs/ibcm/. Directions on that page describe how to download and install.

Writing IBCM Code

You **must** comment your IBCM code copiously. This means (practically) every line should have a comment describing what you are doing. The examples provided in the handouts posted on the course website and discussed in class illustrate a good way of doing this, where there are columns for each of the following:

1. the hexadecimal values that will go into that memory location;
2. the address of that memory location;
3. labels for jumps or variable names.
4. the operation-name for the instruction on that line;
5. a symbolic name for the address the instruction references
6. comments (that explain what's happening in a higher-level form).

Note that together the operation name and the address are sort of an assembly language version of the hexadecimal version of the operations in the first column. You probably want to write those first, and then turn these into the hex instruction that will go into column 1.

The simulator will only read the first 4 character on each line of a file. So you can't have entire lines with comments, or blank lines. The simulator can't handle these (and doesn't check for this), and you will get a strange error.

Even though you will have your program well-thought out and written out in symbolic IBCM before you start typing it in, alas you may still find that you have forgotten an extra variable or an extra instruction or two that you need to add in. To make these corrections easier, be sure to leave a bit of extra space for variables at the top of your program. You may also want to leave extra nop (B) instructions in your code that could be replaced later with actual instructions if needed. Make use of labels in your symbolic IBCM code to aid readability.

Note that under Windows, Alt-PrintScrn will copy a screen shot, which can then be pasted into Word or another editor for printing. Most Linux distributions use the PrintScreen button. Under Mac OS X, it's Apple-Shift-4. class="heading-h4">The IBCM Simulator

For the pre-lab, you will need to write two IBCM programs, as described below. Note that the programs will need to have an .ibcm extension when submitting, but they are text files, so you can still edit them in Emacs.

addition.ibcm

Write a *single* IBCM program that does the following:

- Gets three values from user input
- Stores the values into three variables
- Adds the variables together, and prints the sum (you may use additional memory if you wish)

- If the sum is zero, it prints the three values and stops
- If the sum is not zero, it starts over (tries to get three values, etc.)

array.ibcm

Write a second IBCM program that finds the maximum value in an array of values.

- The array base address is hard-coded into memory – meaning it's a pre-set value, and this is not obtained by user input. You can have the array be all or part of the IBCM program, or a section of memory after the program with values that you have selected.
- You may also hard code other values, such as the number of elements in the array, into your program
- Before your program halts, it prints out the maximum value of the array

Submitting your code

Your code MUST have comments in the file so that the TAs can grade it. No comments will earn a zero for the grade.

In-lab

Bubble sort

Download and look at the bubblesort.cpp algorithm from the Collab site. This algorithm is what needs to be implemented in IBCM, although you should NOT implement the output in the IBCM version.

To encode this program, follow these steps:

1. Write up high-level pseudo-code for your design on paper (make sure that it is absolutely correct, or you will probably regret it later)
2. Refine this pseudo-code, making it closer to the assembly code level
3. Alter your code into IBCM assembly code with labels instead of addresses
4. Run through a sufficient number of test cases by hand of your IBCM code from step (c) to convince yourself that it is correct
5. Encode into actual hex IBCM code and addresses, and test it using the simulator
6. Identify and fix the errors that you did not pick up in the previous steps

The file should be called bubblesort.ibcm. It MUST have comments in the file so that the TAs can grade it. No comments will earn a zero for the grade.

Post-lab

Post-lab report

One of the deliverables for the post-lab is a PDF document named postlab7.pdf. It must be in PDF format! See How to convert a file to PDF for details.

Submit a report, called postlab7.pdf, that contains your thoughts on IBCM. What did you think? How easy was it to use? Would modifications to the simulator make life easier for you? How confident do you feel in writing IBCM code? A (single-spaced) quarter to half a page is fine.

What is a quine

Based on the experience from the in-lab, you should now be able to write an IBCM program on your own. For the postlab, you should individually write an IBCM program that prints itself. This type of program is known as a "quine."

{quotation} quine: /kwi:n/ /n./ from the name of the logician Willard van Orman Quine, via Douglas Hofstadter A program that generates a copy of its own source text as its complete output. Devising the shortest possible quine in some given programming language is a common hackish amusement. {quotation}

Wikipedia has a good article about quines at http://en.wikipedia.org/wiki/Quine_%28computing%29, including examples in a few programming languages.

While at first this idea may sound like a serious mind-bender, in reality it is a rather short program that is not too tough to do in IBCM. This is not a fully general program, it is a carefully crafted program that will only print itself out. The program may contain very specific information such as a variable that is initialized to contain the length of the program. For example, if your quine is 25 lines long (data and instructions), then when it runs, it will print out 25 lines where each line consists of four hex digits. The 25 lines you print out may differ from the original file read into the IBCM simulator in a couple of places – these may be variables and instructions which you have modified between the time the program was loaded and the time that particular line is printed. It is possible to write this program in as few as 8 lines of IBCM code, but most likely you will have closer to 15-20 lines.

You may not submit a zero line quine! Even though this is technically valid, it will not earn credit for this lab.

We will test your program by running it, recording the output, and running that output as a program. You should do the same.

Bash shell script

For this lab, you will need to work a bit more on the shell script that you wrote for the last lab. The shell script will also compute the average running time for 5 executions of a program. The difference is that you will be using control structures, such as conditionals (if-then-else) and loops (for or while) in this shell script.

First, download the counter.cpp file from Collab. This program contains the timer code from lab 6, although it has been modified to print out the time in milliseconds. Note that the program doesn't actually do anything useful it just takes in a numeric command line parameter, and runs through an idle loop many times. We'll call the command line parameter taken in e, and thus the program will run through the idle loop 10e times. Thus, you should not enter a value for e greater than 9 (as 109 (1 billion) is the largest power of 10 that an int value can hold). On a modern computer, entering 9 as the parameter should take between 1 and 5 seconds to run – but keep in mind that the output is in milliseconds. Note that if you compile it with –O2, some compilers (including g++ on Linux systems) will recognize that there is an idle loop (i.e. a loop that does nothing), and will remove that code from the final binary – thus, your time will report as zero. If this is the case, lower the optimization level so that you get a non-zero value when you run it with a high number of iterations.

Your shell script should take in a single input value (as regular input, not as a command line parameter), which will be the number of iterations (i.e. the command-line parameter to pass to the binary program). If that input is 'quit', then the script should exit. Otherwise, you execute the program a total of 5 times, printing and keeping track of the execution time taken for each one. Your script should then print the average time taken for each execution. **You MUST call your executable program 'a.out' in your shell script.** If you are developing this under Windows, you can write the script with 'a.exe', and then change it to 'a.out' before you submit.

Your shell script **MUST** have an if statement (to see if it should exit), and **MUST** have a for or while loop. The number of times to iterate through the for or while loop (initially set to 5) should be a variable set previously in the script. Math in bash can be done with the 'expr' command, as discussed in the tutorial from the last lab. Integer division is fine when computing the average.

Recall that the back quote (on the same key as the tilde (~), which is usually to the left of the digit 1) tells a shell script to run the program, and use the output for something else (as opposed to displaying the output to the screen). For example, the following line would run the program (called a.out), only keep the last line, and save that output to a variable.

```
RUNNING TIME=`./a.out`
```

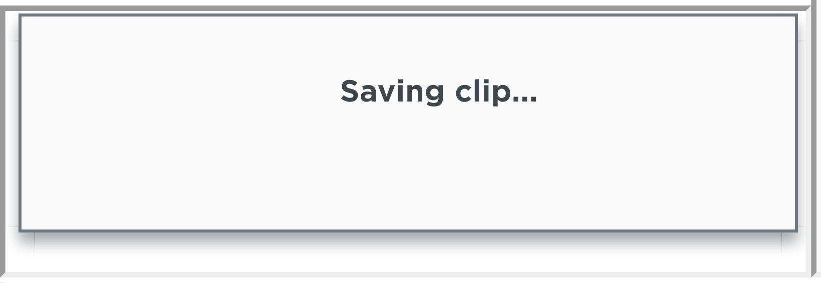
If you want to compare values in a while expression (such as the bash equivalent of "while (i < s)"), you should see http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-7.html. In particular, you need to use '-lt' for the less than operator, and square brackets instead of the parenthesis.

With the two things described above, the rest of the required concepts for the shell script are in the online tutorial. This script should be named averagetime.sh. Below are a few notes to keep in mind when writing your shell script. Bash is a very powerful language, but it can be rather finicky and unforgiving with syntax.

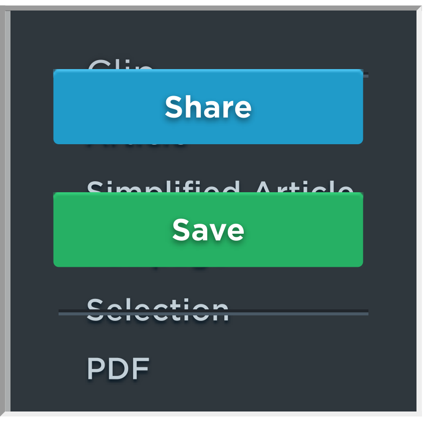
- Your program should be called 'averagetime.sh', and should have '#!/bin/bash' as the very first line of the script
- Bash is a bit finicky with having Boolean operators within an if clause, so try to avoid that (it can do it, but the syntax is very particular)
- When setting variables, do NOT have spaces around the equals sign
- When adding up values (using the back-quote and the 'expr' command), there SHOULD be spaces around the arithmetic operators
- A for loop requires a 'do' keyword before the for loop body; likewise, an if statement has a 'then' before the body. Either these words must be on the next line, or a semi-colon must be there before the 'do' or 'then'
- Keep in mind that to grab program output (such as the output of the binary program, or the result of a mathematical calculation using 'expr'), you use back quotes (i.e. `)
- To execute your script, you can just enter, './averagetime.sh'. If you get a complaint about that ('permission denied', for example), enter this command: 'chmod 755 averagetime.sh'. This tells your Unix system that averagetime.sh is a program that can be executed (remember chmod?).

Below is the output when we wrote this shell script. Obviously, your times may vary.

```
enter the exponent for counter.cpp:
9
Running iteration 1…
time taken: 1256 ms
Running iteration 2…
time taken: 1232 ms
Running iteration 3…
time taken: 1238 ms
Running iteration 4…
time taken: 1240 ms
Running iteration 5…
time taken: 1256 ms
5 iterations took 6222 ms
Average time was 1244 ms
```


Saving clip...

To:


Share
Simplified Article
Save
Selection
PDF