CS-GY 6133: Assignment #1

Due on Friday, September 30, 2016

Instructor: Siddharth Garg

Contents

Problem 1

How many instructions does an ISA need for it to be Turing Complete (roughly, the ISA can be used to compute anything that can be computed)? It turns out that there exist Turing Complete ISAs with that contain only a single instruction. One such ISA is the ~~empl~~subleq (subtract and branch if less than or equal to) ISA that contains only the following instruction:

$$subleq\, a, b, c$$

The semantics of the subleq instructions are as follows:

$$Mem[a] \leftarrow Mem[a] - Mem[b];$$

$$if\ Mem[a] \leq 0\, goto\, c$$

That is, first subtract the contents of memory location b from the contents of memory location a, and store the result in memory location a. If the result is less than or equal to zero, fetch the next instruction from memory location c. Else fetch the next instruction. Assume that your computer has a 4 GB byte-addressable address space.

(a) [10 Points]

What is the minimum number of bits you need to encode a subleq instruction. Design an instruction format for the subleq instruction.

(b) [10 Points]

Classify the subleq ISA as either a 3-, 2-, 1-,or 0-Address ISA. Is it a memory-memory or a load-store ISA?

Solution 3-address memory-memory ISA.

(c) [20 Points]

Consider the following *subleq* code snippet.

L_0: subleq C, C, L_1
L_1: subleq D, D, L_2
L_2: subleq A, B, L_6
L_3: subleq D, B, L_4
L_4: subleq C, D, L_5
L_5: subleq D, D, L_9
L_6: subleq D, A, L_7
L_7: subleq C, D, L_8
L_8: subleq D, D, L_9
L_9: //Rest of code

Here, A, B, C and D are memory locations. L_0 through L_9 represent memory locations of the respective instructions in the code snippet. Assume that before the code executes, the values store in location A, B, C and D are $Mem[A] = 7$, $Mem[B] = -9$, $Mem[C] = 4$ and $Mem[D] = 3$.
What are the values stored in these location after the code snippet executes? More generally, what does this snippet of code do?

Problem 1 continued on next page...

(d) [20 Points]

Write a subleq assembly routine to determine the sum of integers from 0 to N, where N is an input parameter. You can assume that the parameter N is already loaded in memory location M_0. At the end of your routine, the desired sum should be in memory location M_1. You can use memory locations M_2, M_3, ... as temporary variables. You are allowed to update/change the value in M_0.
For convenience, constants 0 and 1 are available in memory locations C_0 and C_1, respectively. Your code should not modify these values.
The subleq instructions in your code will be loaded in locations L_0, L_1, L_2, ..., starting from the first instruction. Your code is expected to terminate at location L_{end}.
For example, your code might look like this:

L_0: subleq M_0, M_1, L_1
L_1: subleq ...
...
L_{end}: **exit** //Your code should jump here when it terminates

Problem 2

The goal of this problem is for you to get familiar with the MIPS ISA. For a description of the MIPS ISA, you can refer to the following links:
`https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf`
`http://www.cburch.com/cs/330/reading/mips-ref.pdf`
or to the reference textbook (H&P).
The Intel FDIV bug is the stuff of folklore. In 1994, it was discovered that the Intel Pentium process had a design flaw in its floating point divider, resulting in faulty output when the FDIV instruction was executed with certain inputs.
In this problem we will understand how design bugs that affect certain instructions can be worked around by re-writing the faulty instruction(s) using other correctly working ones.
In each of the following sub-problems, we have indicated one instruction of the MIPS ISA that has been determined to be buggy. Your job is to using the remaining MIPS instructions to emulate the buggy instruction.

(a) [10 Points]

Assume that the *add rs*, *rt*, *rd* instruction is buggy. Write MIPS assembly code to re-execute *add* using the remaining MIPS instructions. In your implementation, you are allowed to use one additional register rx (besides rs, rt and rd). Your implementation should result in the same values in registers rs, rt and rd as the corresponding call to *add rs*, *rt*, *rd*. However, the value in register rx can be anything.

(b) [10 Points]

Assume that the *and rs*, *rt*, *rd* instruction is buggy. Write MIPS assembly code to re-execute *and* using the remaining MIPS instructions. In your implementation you are only allowed to use registers rs, rt and rd. Your implementation should result in the same values in registers rs, rt and rd as the corresponding call to *and rs*, *rt*, *rd*.

(c) [10 Points]

Assume that the *lw rs*, *rt*, *immediate* instruction is buggy. Write MIPS assembly code to re-execute *lw*

Problem 2 [(c) [10 Points]] continued on next page...

using the remaining MIPS instructions. In your implementation, you are allowed to use one additional register rx (besides rs and rt). Your implementation should result in the same values in registers rs and rt as the corresponding call to $lw\,rs$, rt, $immediate$. However, the value in register rx can be anything. Note: although the load-linked (ll) instruction looks identical to the load-word (lw) instruction, it is in fact not. Bottom-line, don't use ll to replace lw! (Hint: you can assume that main memory is byte-addressable.)

(d) [10 Points]

Assume that the $j\,jmpAddress$ instruction is buggy. Write MIPS assembly code to re-execute j using the remaining MIPS instructions. In your implementation, you are allowed to use one additional register rx. The final value in register rx can be anything. (If requires, recall that register $0 is a constant zero register.)

Problem 3

The ARM Thumb ISA is another example of a RISC ISA. The instruction manual for the ARM Thumb ISA can be found here: `https://ece.uwaterloo.ca/~ece222/ARM/ARM7-TDMI-manual-pt3.pdf`

(a) [10 Points]

Give one example each of a 3-address and a 2-address arithmetic or logical instruction in the ARM Thumb ISA.

(a) [30 Points]

Which of the 6 memory addressing modes we discussed in class (absolute, register indirect, based, indexed, memory indirect, auto-increment) are supported by the ARM Thumb ISA and which are not. For each addressing mode that is supported, provide an example of the corresponding load/store instruction from the ARM Thumb ISA. (Note: you will find that for some addressing modes, the correspondence between the instruction semantics discussed in class and the equivalent ARM Thumb instruction is not exact but the flavor remains the same.)