CMPT 295 Assignment 6 (2%)

Submit your solutions by Friday, March 8, 2019 10am.
Remember, when appropriate, to justify your answers.

1. [10 marks] *Operand Reduction*

   Consider the design of an instruction set for a machine with:

   - a one-byte opcode;
   - a word-size of 32 bits;
   - a byte-addressable memory of size $2^{20} \times 8$;
   - a register file of size 8×32.

   Note that the word-size does not equal the smallest addressable grain-size of memory: it is a multiple. That is typical for most machines. It is possible, therefore, that several instructions (up to 4) could occupy a single word of memory. An efficiently designed CPU would typically fetch that word only once and retain the result internally, rather than waste time on 4 separate fetches. This will be factored into our calculations by counting the number of *bytes* accessed on fetch, decode and execute, rather than the number of words.

   (a) [3 marks] Consider the design of a 3-operand machine. The desired addressing modes include immediate mode, direct mode and relative mode.

`movi`	`$val, rC`	$\texttt{rC} \leftarrow \texttt{\$val}$
`movmr`	`addr, rC`	$\texttt{rC} \leftarrow \texttt{M[addr]}$
`movrr`	`rA, rC`	$\texttt{rC} \leftarrow \texttt{rA}$
`movrm`	`rA, addr`	$\texttt{M[addr]} \leftarrow \texttt{rA}$
`add`	`rA, rB, rC`	$\texttt{rC} \leftarrow \texttt{rA} + \texttt{rB}$
`jle`	`rA, rB, disp`	if $\texttt{rA} \leq \texttt{rB}$, $\texttt{PC} \mathrel{+}= \texttt{disp}$

   Design a set of instruction formats for these 6 instructions. Your design should allow each explicit operand to encode its full range of values.

   Present your answer in table format, one instruction per row. The two rightmost columns should contain a measure of how many *bytes* would be loaded on *fetch* + *decode*, and how many would be read/written on *execute*.

   (b) [2 marks] Create alternate versions of `movi` and `jle` that offer a restricted range of values, but a shorter instruction length:

`movi`	$\texttt{\$val}_{21}$`, rC`	$\texttt{rC} \leftarrow \texttt{\$val}_{21}$	21-bit signed `$val`
`jle`	`rA, rB, `$\texttt{disp}_{10}$	if $\texttt{rA} \leq \texttt{rB}$, $\texttt{PC} \mathrel{+}= \texttt{disp}_{10}$	10-bit signed `disp`

   Add two more rows to your table to include these versions. State the range of each operand.

(c) [2 marks] Redevelop the instruction formats for a 2-operand machine. The only instructions that need to change would be `add` and `jle`.

`add`	`rA, rC`	$\mathtt{rC} \leftarrow \mathtt{rC} + \mathtt{rA}$	
`jle`	`rA, disp`	if $\mathtt{rA} \leq 0$, $\mathtt{PC} \mathrel{+}= \mathtt{disp}$	
`jle`	$\mathtt{rA,\ disp}_5$	if $\mathtt{rA} \leq 0$, $\mathtt{PC} \mathrel{+}= \mathtt{disp}_5$	5-bit signed `disp`

Add these to your table.

Note: To achieve full marks, your instruction formats should be as consistent as possible with those developed in parts (a) and (b). You may need to re-tune your answer to (a) and (b) to arrive at the best overall result.

(d) [3 marks] Write a program that, for a trio of 32-bit values `x`, `y`, `z`, computes $\mathtt{z} \leftarrow (\mathtt{x} + \mathtt{y}) * (\mathtt{x} - \mathtt{y})$. You may also use `sub` and `mul`, which will have the same instruction format as `add`.

Write two versions: one for the 3-operand machine and one for the 2-operand machine. Total the number of bytes of memory access that are done during each of *fetch* + *decode*, and *execute*, and conclude which system is better.

(e) [2 BONUS marks] Using your ISA from (a), (b), (c), write a version of the program that consumes no more than 30 bytes of memory access.

2. [10 marks] *Branch Reduction*

The *linear search* algorithm is one of the classic linear-time algorithms you study in Computing Science. The most standard implementation is within your care package. Have a look.

If you build and run the executable, you will see that it benchmarks a pair of linear searches on a randomized array. For this problem, you will code an alternate version of linear search, and demonstrate that it is measurably better.

(a) [3 marks] *C:* Though the algorithm cannot run in sub-linear time, the algorithm can be optimized to reduce the cost of each loop. A good first approximation is to go after the expensive operations: the comparisons. A comparison in your C code will generate a corresponding branch in the assembly code; a branch has the potential to be mis-predicted by the CPU's instruction pipeline.

The standard algorithm does two comparisons per loop: the comparison between `A[i]` and `target` and the comparison between `i` and `n`. The number of comparisons can be roughly cut in half using the following algorithm:

```
search(A[n], target)
   if n <= 0 then return -1

   tmp <- A[n-1]
   A[n-1] <- target

   i <- 0
   while A[i] != target do
      i <- i+1

   A[n-1] <- tmp

   if i < n-1 then return i
   else if A[n-1] == target then return n-1
   else return -1
```

Code this algorithm in `lsearch_2.c`. Replace the code for the function `lsearch_2()` with your own code.

(b) [3 marks] *Hardcopy:* Benchmark `lsearch_1()` and `lsearch_2()` for $\texttt{N} \in \{5000000, 10000000, 15000000, 20000000\}$, and $\texttt{NTESTS} = 400$. Use the computers in ASB 9840.

- Collect three samples for each `N`, and present your raw data and the average times in a table format.
- Plot the average times on a graph, and connect each collection with a straight line. As usual, use a ruler.
- Compute the slope of each line.
- Determine which algorithm is better, and divide its slope by the slope of the other. Express your answer as a percentage.

(c) [4 marks] *x86-64:* Open `lsearch_2.s` in your favourite editor, and observe what assembly `gcc` has created on your behalf. It contains several directives and labels that would make it difficult for a human to understand the code. Your goal for this part of the problem is to clean things up.

- [0 marks] *Delete all directives.*

  These are the lines that begin with a period "`.`". You will need to keep one, however: `.globl lsearch_2`, otherwise the code will no longer link properly.

- [1 mark] *Change all label names.*

  The sequentially numbered labels do not help describe the meaning of the algorithm. Adjust them so that it would assist *your* understanding of the code as an assembly programmer. Some examples of labels you have used so far are `loop`, `endif`, and `found`. There are no hard and fast rules for good label names, except to balance expressiveness against the length of the label.

- [3 marks] *Describe the algorithm that is used in the assembly code.*

  Because compiler optimization was turned on, `gcc` made some adjustments to your original algorithm in C. Figure out what the new algorithm has become and then
  - add a variable map to describe the variables held by each register; and
  - add comments that contain the C code equivalent to the new algorithm. (This might not be the same C code you produced in part (a).)

So you know what is expected, a sample is shown in `assembly.sample`.