CMPT 295 Assignment 7 (2%)

Submit your solutions by Friday, March 15, 2019 10am.
Remember, when appropriate, to justify your answers.

Express your times in picoseconds and your throughputs in GIPS.

1. [3 marks] *Equal-lengthed Stages*

   Consider the design of a processor, with a max instruction length of 600 ps. The propagation delay to load a register is 25 ps.

   (a) [1 mark] What is the minimum clock cycle time, the instruction latency and CPU throughput using serial execution?

   (b) [1 mark] What is the minimum clock cycle time, the instruction latency and CPU throughput using a pipelined execution with 8 equal stages?

   (c) [1 mark] Consider a design which used n equal stages. What is the minimum clock cycle time, the instruction latency and CPU throughput expressed as a function of n? (You may wish to check that your generalization agrees with your results from parts (a) and (b), i.e., by substituting $n = 1, 8$.)

2. [2 marks] *Serial vs Pipelined Instructions*

   Consider two systems: both have a clock cycle of 450 ps and a register load time of 50 ps. The difference? The first one is serially executed, while the second one uses a pipeline of 5 stages per instruction. What is the instruction throughput of each machine? What is the instruction latency?

3. [4 marks] *Oddly-lengthed Stages*

   Instruction execution design might not always result in stages that are equal in length.

   As an example, consider a system that can be cleanly divided into 6 stages, in the order (A, B, C, D, E, F), each with a propagation delay (in ps) of $(40, 80, 100, 150, 160, 70)$, for a grand total of 600 ps.

   The register loading time is 25 ps.

   (a) [1 mark] If you only had one extra set of registers to place between an adjacent pair of stages in order to form a 2-stage pipeline, where would you place them? Compute the minimum clock cycle time and the maximum possible CPU throughput.

   (b) [1 mark] If you had *two* extra sets of registers, where would you place them to form a 3-stage pipeline? Again, compute the min clock cycle time and max throughput.

   (c) [1 mark] If you had *five* extra sets of registers, and created a 6-stage pipeline, what would the min clock cycle time and max throughput be?

   (d) [1 mark] Suppose you had *two* extra sets of registers, like in part (b). If you could direct your designers to divide any of A, B, C, D, E, F into two stages, where would you tell them to concentrate their efforts?

4. [3 marks] *Persistent Dynamic Memory*

   Build and run the code in the `q4` subdirectory from the care package. When you compare the output with the corresponding C code, it appears that there is a bug in `printf()`. Perhaps in the other function. There is a bug, however it is not within either function.

   Explain where the bug really is, and why `printf()` displays what it does.

5. [8 marks] *Linked Lists and Pipeline Stalls*

To search a linked list of values costs $O(n)$, i.e., linear time, the same running time for a simple linear search of an array. But not all linear time algorithms are created equal: to choose, you can measure the *cost per element* (CPE) using benchmarking.

(a) [6 marks] *Hardcopy:* Open the care package and peruse the files. The driver sets up two data structures — an array `int A[N]` and a linked list `List *L` — with the values $0 \ldots \texttt{N} - 1$ in a random order. The benchmarking code follows, and you can choose between `lsearch()` for the array or `LLsearch()` for the linked list. The code for these subroutines is in `LL.c`. Have a look.

Compile the code and open `LL.s`. Comparing the main loops of the two functions:

```
LLsearch:                                   lsearch:
        . . .                                       . . .
.L10:                                       .L57:
        addl    $1, %eax                            addq    $1, %rcx
        cmpl    %esi, (%rdx)                        cmpl    %edx, -4(%rdi, %rcx, 4)
        je      .L9                                 je      .L54
.L4:                                        .L56
        movq    8(%rdx), %rdx                       cmpq    %rsi, %rcx
        testq   %rdx, %rdx                          movl    %ecx, %eax
        jne     .L10                                jne     .L57

        . . .                                       . . .
```

it looks like there are the same number of instructions per element on both sides, but not all instructions are created equal. The cost of `LLsearch()` will naturally be higher because of the von Neumann bottleneck, i.e., there are two memory references per element for `LLsearch()`, but only one memory reference per element for `lsearch()`. To level the playing field, add a memory reference instruction to `lsearch` as follows:

```
lsearch:
        . . .
.L57:
        addq    $1, %rcx
        movl    -8(%rdi, %rcx, 4), %r8d      <--- add this line and recompile <---
        cmpl    %edx, -4(%rdi, %rcx, 4)
        . . .
```

Re-build the code and microbenchmark each function for $\texttt{N} \in \{150, 200, 250, 300\}$. For each, run the code 11 times and throw out the shortest 3 and longest 3 times. Tabulate your results, one quintet of measurements per row.

Compute the average of each set of 5 times. Then, plot both data sets on a graph and draw a straight line through each set. (Use a ruler.)

Compute the slope of each line and report the the cycles per element (CPE) for each algorithm.

(b) [2 marks] *Hardcopy:* The main reason for the difference in the CPE is a data dependency in the linked list search that isn't present in the array search. Explain the sequence of instructions that causes the delay.