COMP 740 Homework 3

Prof. Montek Singh
Fall 2016
Due Mon, Nov 14, 2016 (in class)

1. (15 points) *Absolute and Relative Improvements.* Suppose a machine's performance is a function of two parameters, *x* and *y,* where each parameter is positive and has the dimension of time:

$$Performance = \frac{1}{x} + \frac{4}{y}$$

Suppose the designer has a choice of improving the system's performance by either decreasing *x* or *y* (but not both). This problem explores which choice would the preferred one.

   a. Suppose the cost of improving *x* is identical to the cost of improving *y, for the same absolute improvement (δx=δy).* Under which conditions would: (i) an improvement in *x* be preferable, (ii) an improvement in *y* be preferable, and (iii) both improvements be equally good.
   b. Now suppose instead that the cost of improving *x* is identical to the cost of improving *y, for the same relative improvement (δx /x = δy/y).* Under which conditions would: (i) an improvement in *x* be preferable, (ii) an improvement in *y* be preferable, and (iii) both improvements be equally good.

2. (25 points) *MIPS Integer Pipeline: Hazards, Stalls and Forwarding.* Consider the following program fragment to compute values for variable `D`. All variables are memory-resident, integer-valued, and located at distinct memory addresses.

```
D = (A*X+B)*X+C;
```

A compiler for a MIPS machine generates the following code for this program fragment: (The memory addresses are symbolic names.)

```
1:  LW R1, X
2:  LW R2, A
3:  MULT R10, R1, R2
4:  LW R3, B
5:  ADD R10, R10, R3
6:  MULT R10, R10, R1
7:  LW R4, C
8:  ADD R10, R10, R4
9:  SW D, R10
```

Note: Make sure that the code fragment actually does compute `D` correctly; if not, fix it.

(a) Enumerate all of the data hazards that exist among the instructions. Classify them as RAW, WAR, and WAW hazards. Show your answer as a graph with one circle (node) for each of the nine instructions, and a directed edge from node *i* to node *j* if there is a dependence between instructions *i* and *j* (instruction *i* occurs before instruction *j*). Label this edge with the type of hazard (RAW, WAR or WAW). Such a graph is called a precedence graph.

(b) Assume a five-stage linear pipelined implementation of MIPS with the two additional assumptions (both admittedly unrealistic): (i) all operations spend a single cycle in the EX stage, and (ii) there are no forwarding paths whatsoever.[1] Represent the execution of the program above with the help of a space-time chart (i.e., the spreadsheets we have been using in class), and determine the number of clock cycles it will take to complete execution. Clearly indicate all stalls.

(c) Still assuming no forwarding, *rearrange* the order of instructions in this program to reduce execution time without violating any dependencies. Draw a space-time chart to represent the execution of the rearranged program.

(d) Now assume that all reasonable forwarding paths are available. Draw a space-time diagram for the execution of the *original* program (without any reordering).

[1] Note, however, that a combination of a register write and a register read in the same clock cycle are allowed because each can complete in half a clock cycle.

(e) Still assuming that all reasonable forwarding paths are available, draw a space-time diagram for the execution of the *rearranged* program.

(f) Summarize the execution times of the four program executions in a two-dimensional table. Label the y-axis with the lack (or availability) of forwarding, and the x-axis with the lack (or availability) of compile-time scheduling.

3. (30 points) *Scoreboarding.* We will look at a dynamically scheduled version of the DAXPY loop, the double precision a * x + y. You may have heard of this as SAXPY (single precision). It is a common loop in computations such as Gaussian elimination. Specifically, we will compute y[i] = a * x[i] + y[i]. Remember that each x[i] and y[i] are 8 bytes long. The MIPS code for DAXPY is:

```
loop: L.D F2, 0(R1)           ; load x[i]
      MULT.D F4, F2, F0       ; a * x[i] // F0 has a
      L.D F6, 0(R2)           ; load y[i]
      ADD.D F6, F4, F6        ; add y[i]
      S.D F6, 0(R2)           ; store result into y[i]
      ADDI R1, R1, 8          ; increment x index
      ADDI R2, R2, 8          ; increment y index
      SGTI R3, R1, END        ; set-if-greater-than-immediate //R3 ← (R1 > END)
      BEQZ R3, loop           ; branch if R3 is zero/false
```

For this problem you will use a pipeline identical to that discussed in the lecture on Scoreboarding (or Fig. C.54 in textbook). Thus, as assumed in the examples discussed in class: the execution units are not pipelined, there are no forwarding paths, etc. The number of clock cycles required to execute an instruction is as follows:

- integer operations, including loads/stores: 1 cycle
- floating-point adds: 4 cycles
- floating-point multiplies: 8 cycles

Note: Scheduling will be of only one basic block; do not schedule the branch instruction or subsequent iterations.

Show at which clock cycle each instruction performs the various steps of execution. In particular, draw the "Instruction Status" part of the scoreboard, and show the clock cycle number (beginning with '1') when each of the steps --- Issue, Read Operands, Start Execution, Complete Execution and Write Result --- takes place.

4. (30 points) *Tomasulo's Algorithm:* Do exercise 3.15 part (a) only from the textbook.