CS252
Prerequisite Quiz
Solutions

Krste Asanovic
Fall 2007

Problem 1 (29 points)

The followings are two code segments written in MIPS64 assembly language:

Segment A:

```
Loop: LD      r5, 0(r1)      # r5 ← Mem[r1+0]
      LD      r6, 0(r2)      # r6 ← Mem[r2+0]
      DADD    r5, r5, r6     # r5 ←  r5 + r6
      SD      r5, 0(r3)      # Mem[r3+0] ← r5
      LD      r5, 0(r1)      # r5 ← Mem[r1+0]
      LD      r6, 0(r2)      # r6 ← Mem[r2+0]
      DSUB    r5, r5, r6     # r5 ←  r5 - r6
      SD      r5, 0(r4)      # Mem[r4+0] ← r5
      DADDUI  r1, r1, 8      # r1 ← r1 + 8
      DADDUI  r2, r2, 8      # r2 ← r2 + 8
      DADDUI  r3, r3, 8      # r3 ← r3 + 8
      DADDUI  r4, r4, 8      # r4 ← r4 + 8
      BNE     r1, r9, Loop   # branch to Loop if r1 ≠ r9
```

Segment B:

```
Loop: LD      r5, 0(r1)      # r5 ← Mem[r1+0]
      LD      r6, 0(r2)      # r6 ← Mem[r2+0]
      DADD    r7, r5, r6     # r7 ←  r5 + r6
      DSUB    r8, r5, r6     # r8 ←  r5 - r6
      SD      r7, 0(r3)      # Mem[r3+0] ← r7
      SD      r8, 0(r4)      # Mem[r4+0] ← r8
      DADDUI  r1, r1, 8      # r1 ← r1 + 8
      DADDUI  r2, r2, 8      # r2 ← r2 + 8
      DADDUI  r3, r3, 8      # r3 ← r3 + 8
      DADDUI  r4, r4, 8      # r4 ← r4 + 8
      BNE     r1, r9, Loop   # branch to Loop if r1 ≠ r9
```

In both segments, assume r1, r2, r3, r4 initially hold valid memory addresses. Register r9 is pre-computed to be 80 larger than the initial value of r1. All instructions operate on 64-bit doubleword values and the memory address space is byte-addressable.

a) If both segments are expected to perform the same task, can you guess what the task is? You can write the answer in C-like pseudo code. (10 points)

```
for (i = 0 ; i < 10 ; i= i+ 1) {
    c[i] = a[i] + b[i];
    d[i] = a[i] – b[i];
}
```

b) In general, which segment do you expect to perform better when executed? (9 points)

Segment B is expected to perform better because it executes two less memory load instructions than segment A per iteration of the loop.

c) Do the two segments always produce the same results at all situations? If not, can you specify a situation which makes them behave differently? (10 points)

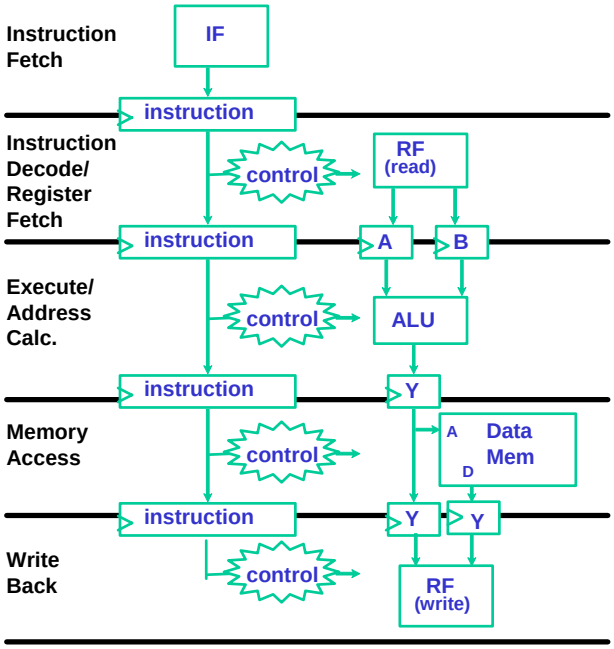
If the initial value of register r3 is same as that of r1, memory address alias occurs. In this case, the two segments behave differently. When segment A is executed, the value of r5 for calculating the addition is different from the value of r5 for calculating the subtraction in the same iteration because the first memory store to address pointed by r3 in the iteration overwrites the previous value of the memory address pointed by r1, thus affecting the value returned by the second load to the same address. On the other hand, when segment B is executed, the value of r5 for calculating the addition is the same as the value of r5 for calculating the subtraction in the same iteration. Similar scenario occurs when r3 has the same initial value as r2.

Problem 2 (36 points)

The following figure shows a 5-stage pipelined processor. The pipelined processor should always compute the same results as an unpipelined processor. Answer the following questions for each of the instruction sequences below:

- Why does the sequence require special handling (what could go wrong)?
- What are the minimal hardware mechanisms required to ensure correct behavior?
- What additional hardware mechanisms, if any, could help preserve performance?

Assume that the architecture does not have any branch delay slots, and assume that branch conditions are computed by the ALU.



a) (12 points)

```
BEQ     r1, r0, 200     # branch to PC+200 if r1 == r0
ADD     r2, r3, r5      # r2 ← r3 + r5
SUB     r4, r5, r6      # r4 ← r5 + r6
…
```

To determine the outcome of the branch instruction, the registers r1 and r0 have to be read and compared for equality in the ALU stage. By this time, the add instruction will be in the Register File stage, and the subtract instruction will be in the Instruction Fetch stage. If the branch is taken, then these two instructions should not be executed. This is called a *control hazard*.

We can achieve correct behavior if the branch is taken by killing the two following instructions and inserting bubbles into the pipeline in their place.

We can improve performance by using branch prediction to guess the outcome of the branch during the Register File stage. If we correctly predict a taken branch, we only have to kill one instruction instead of two. To further improve performance, we could use a branch target buffer to predict the instruction address to fetch next during the Instruction Fetch stage (even before the branch is decoded in the Register File stage).

b) (12 points)

```
ADD     r1, r0, r2      # r1 ← r0 + r2
SUB     r4, r1, r2      # r4 ← r1 - r2
…
```

The add instruction writes its result to register r1 when it reaches the Write Back stage, at which time the subtract instruction will be in the Memory stage. However, the subtract instruction needs to use register r1 to compute its result in the ALU stage. This is called a *data hazard*.

We can achieve correct behavior by stalling the subtract instruction in the Register File stage until register r1 gets written. We would stall the subtract instruction for three cycles and insert bubbles into the pipeline.

We can improve performance by adding bypass muxes to the pipeline and forwarding the result of the add instruction at the end of the ALU stage to the subtract instruction at the end of the Register File stage. With this solution, the pipeline does not need to stall.

c) (12 points)

```
LD      r1, 0(r2)       # r1 ← Mem[r2+0]
ADD     r3, r1, r2      # r3 ← r1 + r2
…
```

This is a *data hazard*, similar to part (b) above. We can again achieve correct behavior by stalling the dependent instruction until register r1 gets written. We can improve perfomance by forwarding the result of the load instruction at the end of the Memory stage to the add instruction at the end of the Register File stage. However, we still must stall the add instruction for one cycle.

Problem 3 (15 points)

For each of the following statements about making a change to a cache design, circle **True** or **False** and provide a one sentence explanation of your choice. Assume all cache parameters (capacity, associativity, line size) remain fixed except for the single change described in each question. Please provide a one sentence explanation of your answer.

a) Doubling the line size halves the number of tags in the cache. (3 points)

True / False

True. Since capacity (the amount of data that can be stored in the cache) and associativity are fixed, doubling the line size would halve the number of lines in the cache. Since there is one tag per line, halving the number of lines would halve the number of tags.

b) Doubling the associativity doubles the number of tags in the cache. (3 points)

True / False

False. Since capacity and line size are fixed, doubling the associativity will not change the number of lines in the cache. Therefore the number of tags in the cache remains the same.

c) Doubling cache capacity of a direct-mapped cache usually reduces conflict misses. (3 points)

True / False

True. Assuming that line size remains the same, doubling the cache capacity would double the number of lines, reducing the probability of a conflict miss.

d) Doubling cache capacity of a direct-mapped cache usually reduces compulsory misses. (3 points)

True / False

False. Since line size remains the same, the amount of data loaded into the cache on each miss remains the same. Therefore the number of compulsory misses remain the same.

e) Doubling the line size usually reduces compulsory misses. (3 points)

True / False

True. Doubling the line size increases the amount of data loaded into the cache on each miss. If the program exhibits spatial locality, it would experience fewer compulsory misses.