

CS 211 - HW 2 Solutions

Ques.1: This exercise analyzes the trade-off between adding a new instruction that decreases the number of instructions executed at the cost of increasing the clock cycle by 5%.

(a) Because the new design has a clock cycle time equal to 1.05 times the original clock cycle time, the new design must execute fewer instructions to achieve the same execution time. For the original design we have

$$CPU\ Time_{old} = CPI_{old} \times CC_{old} \times IC_{old}$$

For the new design the equation is:

$$CPU_{new} = CPI_{new} \times CC_{new} \times IC_{new} = CPI_{old} \times (CC_{old} \times 1.05) \times (IC_{old} - R)$$

where the CPI of the new design is the same as the original (per the exercise statement), the new clock cycle is 5% longer, and the new design executes R fewer instructions than the original design. To find out how many loads must be removed to match the performance of the original design set the above two equations equal and solve for R :

$$CPI_{old} \times CC_{old} \times IC_{old} = CPI_{old} \times (CC_{old} \times 1.05) \times (IC_{old} - R) \implies R = 0.048IC_{old}$$

Thus the instruction count must decrease by 4.8% overall to achieve the same performance, and this 4.8% is comprised entirely of loads. Figure B.27 shows that 25.1% of the gcc instruction mix is loads so $(0.048/0.251 = 0.19) = 19\%$ of the loads must be replaced by the new register-memory instruction format for the performance of the old and new designs to be the same. If more than 19% of the loads can be replaced then performance of the new design is better.

b. Consider the code sequence

```
LOAD R1,0(R1)
ADD R1,R1,R1
```

The result written to R1 is $MEM[0 + R1] + MEM[0 + R1]$. However, the register-memory form of this code sequence is

```
ADD R1,0(R1)
```

The result written to R1 in this case is $R1 + MEM[0 + R1]$ which is not the same unless the initial value in R1 just happens to be equal to the value at the memory location with address $[0 + R1]$. In general the values will not be equal, and a compiler would not be able to use the new instruction form in this case.

Ques.2:

1. load R0, (R4)
2. add R1, R0, #10
3. load R2, (R5)
4. mul R0, R2, #10
5. mul R4, R2, R1
6. store R0, (R5)

(a) There are all the three types of dependencies (RAW, WAR, WAW) in the above code. The RAW dependencies are: between instructions 1 and 2 on R0, between instructions 3 and 4 on R2, between instructions 2,5 on R1, between instructions 3,5 on R2, between instructions 4,6 on R0. (Note that by transitivity, there are other RAW dependencies, since 2 depends on 1 and 5 depends on 2 we have 5 depends on 1, etc.). The WAR dependencies are: between 1,5 on R4; between 2,4 on R0. The WAW dependencies are: between 1,4 on R0.

(b) Due to the RAW dependencies, there are stalls in the above code without any forwarding.

(c) With internal forwarding, the stalls due to RAW between 4,6 no longer exist. But there is still a stall cycle between 1,2 and 3,4 which cannot be removed by internal forwarding.

(d) We can rearrange the code to eliminate stalls (with internal forwarding) in the following manner:

```
1. load R0, (R4)
3. load R2, (R5)
2. add  R1, R0, #10
4. mul R0, R2, #10
5. mul R4, R2, R1
6. store R0, (R5)
```