CS61c Fall 2014 Discussion 11 – Pipelining

Pipelining Hazards

Structural – Hazards that occur due to competition for the same resource (register file read vs. write back, instruction fetch vs. data read). Caching and clever register timing can solve these hazards.
Control – Hazards that occur due to non-sequential instructions (jumps and branches). These cannot be solved completely by forwarding, so we're forced to introduce a branch-delay slot (MIPS) or use branch prediction.
Data – Hazards that occur due to data dependencies (instruction requires result from earlier instruction). These are mostly solved by forwarding, but `lw` still requires a bubble.

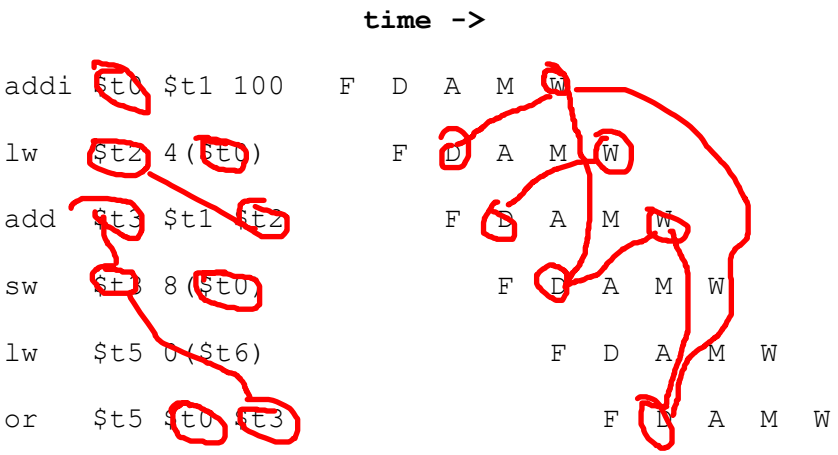
1. Suppose you've designed a MIPS processor implementation in which the stages take the following lengths of time: IF=20ns, ID=10ns, EX=20ns, MEM=35ns, WB=10ns. What is the minimum clock period for which your processor functions properly? Where should the bulk of your R&D budget go for the next generation of processors?

35 ns. For MEM so as to reduce the clock period

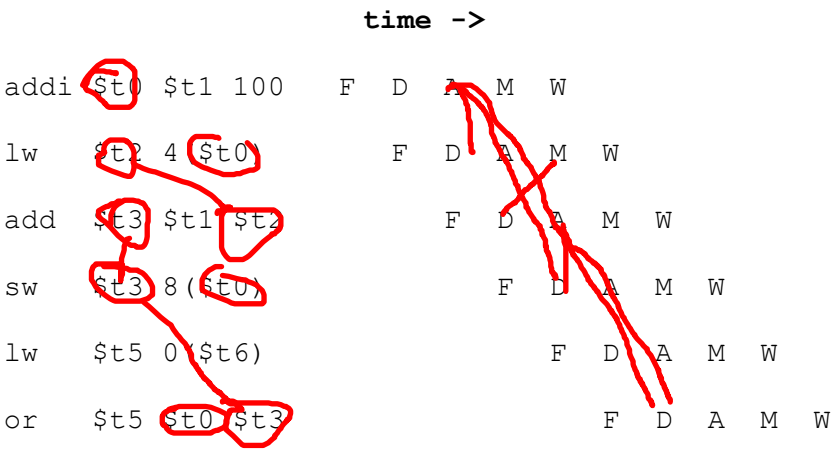
2. Your friend tells you that his processor design is 10x better than yours, since it has 50 pipeline stages to your 5. Is he right? Why or why not? (This is intentionally vague)

Yes and No. Yes as long as there are no branches or loads/stores. And Yes if all the stages are equal.
No if the stages are not equal and the branches/loads/stores have to stall many clock cycles in order to work.

3. Spot all data dependencies (including ones that do not lead to stalls). Draw arrows from the stages where data is made available, directed to where it is needed. Circle the involved registers in the instructions. **Assume no forwarding.**



4. Redraw the arrows for the above question assuming that our hardware provides forwarding.



5) How many stalls will we have to add to the pipeline to resolve the hazards in Exercise 3? How many stalls to resolve the hazards in Exercise 4?

6; 1

6) Rewrite the following delayed branch MIPS excerpt to maximize performance assuming forwarding.

```
Loop:   addi $v0, $v0, 1
        addi $t1, $a0, 4
        lw $t0, 0($t1)
        add $a0, $t0, $a1
        addi $a0, $a0, 4
        bne $t0, $0, Loop
        nop
        jr $ra
```

Loop:

addi $t1, $a0, 3
lw $t0, 0($t1)
addi $v0, $v0, 1
add $a0, $t0, $a1
bne $t0, $0, Loop
addi $a0, $a0, 4
jr $ra

7) Now, assume for the delayed branch code from Exercise 6 that our hardware can execute Static Dual Issue for any two instructions at once. Using reordering (with nops for padding), but no loop unrolling, schedule the instructions to make the loop take as few clock cycles as possible.

addi $t1, $a0, 4 and addi $v0, $v0, 1
lw $t0, 0($t1)
stall
add $a0 $t0 $a1 and bne $t0, $0, Loop
addi $a0, $a0, 4
jr $ra