CS61c Spring 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.

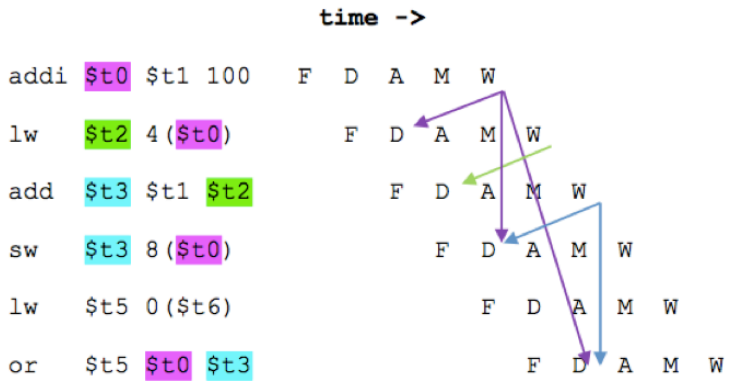
1. Suppose you've designed a MIPS processor implementation in which the stages take the following lengths of time: IF=20ns, ID=10ns, EX=20ns, MEM=35ns, WB=10ns. What is the minimum clock period for which your processor functions properly? Where should the bulk of your R&D budget go for the next generation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

MEM is the bottleneck – limiting the minimum clock period to 35ns.

2. Your friend tells you that his processor design is 10x better than yours, since it has 25 pipeline stages to your 5. Is he right? Why or why not? (This is intentionally vague)

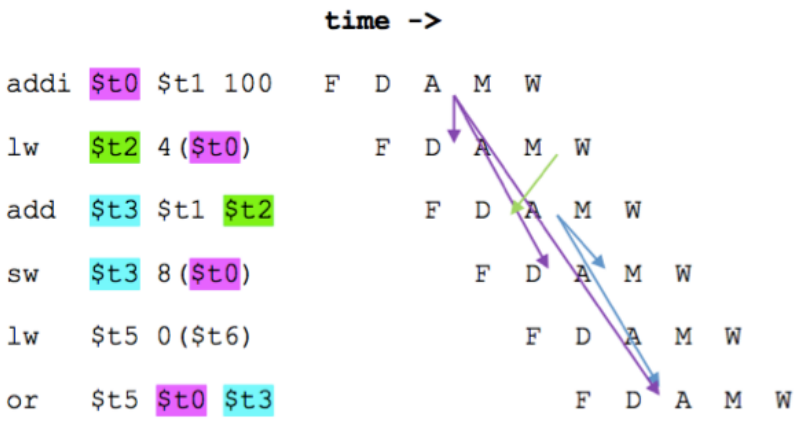
No, for many, many reasons:
-Much higher power consumption
-More hardware required to implement, more expensive to manufacture.
-Increased complexity in implementation, more hazards
-Overhead from implementing the pipeline stages would result in <10x speedup, even at best - Unlikely to evenly split the logic into 50 stages, also resulting in <10x speedup.
-Other technologies (for example, caches) might also limit the performance of any one stage. - Increased penalty for missed branch predictions / longer to fill the pipeline

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.**



Without forwarding, the register values become available after the write-back (WB) stage, and are needed in the instruction-decode (ID) stage.

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



With forwarding, the register values become available as soon as they are computed/retrieved, and are needed as late as possible in the computation.

Note that arithmetic operations with forwarding do not cause stalls, but load word still does.

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 stalls without forwarding, and 1 stall with forwarding.

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

#A fills the load delay, #B fills the branch delay slot.

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