CS 61C Fall 2013 – 11 – Pipelining and Hazards

Pipelining Hazards:

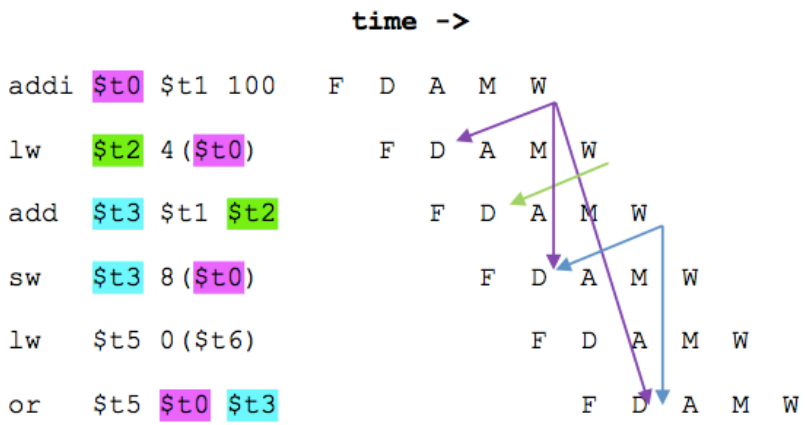
Structural – Hazards that occur due to competition for the same resource (register file read vs. write back, instruction fetch vs. data read). These are solved by caching and clever register timing.
Control – Hazards that occur due to non-sequential instructions (jumps and branches). These cannot be 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 where the stages take the following lengths of time: IF=15ns, ID=5ns, EX=25ns, MEM=40ns, WB=15ns. What is the minimum clock period where your processor functions properly? What should be the focus for the next generation?
Memory is the bottleneck, limiting to a period to 40ns; it should have the main focus in development.

2) Your friend tells you that his processor design is 5x better than yours, since it has 25 pipeline stages to your 5. Is he right?

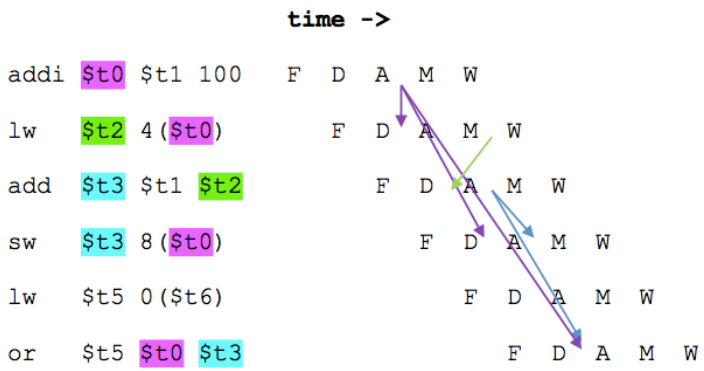
No, for many, many reasons:
-Much higher power consumption
-More hardware required to implement, more expensive to manufacture.
-Increased complexity in implementation, more hazards
-Overhead from implementing the pipeline stages would result in <10x speedup, even at best
-Unlikely to evenly split the 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 the data dependencies! 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.** One dependency has been drawn for you.



Without forwarding, the register values become available in the write-back phase, and are needed in the decode phase.

4) Redo 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.

Notice 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 **3)**? How many stalls to resolve the hazards in **4)**?

6 stalls without forwarding, 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, 1
        lbu $t0, 0($t1)
        sb $t0, 0($a0)
        addi $a0, $a0, 1
        bne $t0, $0, Loop
        nop
        jr $ra
```

```
Loop:   addi $t1, $a0, 1
        lbu $t0, 0($t1)
        addi $v0, $v0, 1 #A
        sb $t0, 0($a0)
        bne $t0, $0, Loop
        addi $a0, $a0, 1 #B
        jr $ra
```

A: Fills the load delay slot with instruction that executes anywhere in loop without changing result.
B: Fills the branch delay slot with an instruction that can be moved down.
Result: Two fewer clock cycles to execute each loop.

7) Now, assume for the delayed branch code from **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.

load/store		ALU/branch
noop	|	addi $t1 $a0 1
lbu $t0 0($t1)	|	addi $v0 $v0 1
noop	|	noop
sb $t0 0($a0)	|	bne $t0 $0 Loop
noop	|	addi $a0 $a0 1
noop	|	jr $ra

Saves one cycle.