Monday, March 9, 2020

Method 2: pipelining

Note resources are idle 43% of the time

Pipelined execution of instructions

Consider our MIPS subset: add, sub, slt, and, or, lw, sw, beq.

Goal: Break each instruction into 5 steps that each take roughly the same amount of time

Pipeline stages

Fetch (F): Read instruction from I-mem; update PC.

Decode (D): Produce outputs of control unit; read GPRs from R-file

Execute (E): Get result from ALU

Memory (M): D-mem access for loads and stores

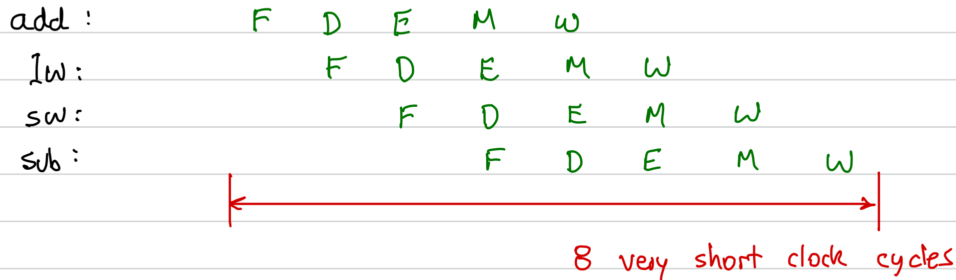
Writeback (W): Write result to a GPR for R-type and lw instructions.

Not all stages are needed for every instruction

- R-type – no D-mem access
- sw – no writeback

Suppose we have :

```
add  $t2, $t0, $t1
lw   $t4, ($t3)
sw   $t5, ($t6)
sub  $t9, $t8, $t7
```



Both the single-cycle and pipelined processors start a new instruction every clock cycle

INSTRUCTION PHASES ...

lw $t0, 20($t1)	Fetch	Decode	Execute	Memory	Write back		
or $t2, $t3, $t4		Fetch	Decode	Execute	Memory	Write back	
sw $t5, 40($t6)			Fetch	Decode	Execute	Memory	Write back

ACTIVITY DETAILS ...

lw $t0, 20($t1)	Read lw from I-mem	Read $t1 out of R-file	Compute $t1 + 20	Read data from D-mem	Update $t0 in R-file		
or $t2, $t3, $t4		Read OR from I-mem	Copy $t3 and $t4 from R-file	Compute $t3 OR $t4	no D-mem access	update $t2 in R-file	
sw $t5, 40($t6)			Read sw from I-mem	Copy $t5 and $t6 from R-file	Compute $t6 + 40	write $t5 to D-mem	No R-file update

Response time of pipeline stages much faster than single-cycle machine – only doing <u>one</u> operation per clock cycle

- clock can be much faster
- higher instruction throughput

What can go wrong – pipeline hazards

Three main kinds:
- structural hazards
- data hazards
- control hazards

Structural hazards

This is when a single unit might be asked to do or more incompatible things.

Example: a computer with one memory unit

- can't simultaneously fetch an instruction and read or write data.

Solution: proper design; e.g., two memory units.

Data hazards:

What's wrong with this:

```
add  $t0, $t1, $t2
sub  $t3, $t4, $t0
```

($t0 in add:) writeback is the last operation in the pipeline

($t0 in sub:) not available yet! (will use last-saved $t0)

Control hazards

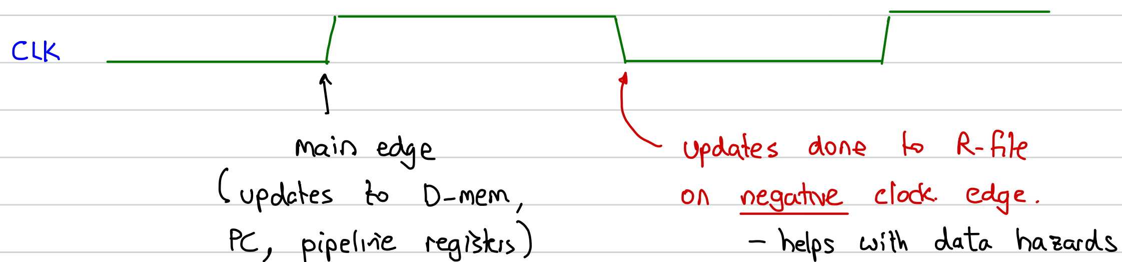
Branching could go wrong:

```
     beq  $t0, $t1, L1
     and  $t2, $t3, $t4
      ⋮
L1:  sub  $t5, $t6, $t7
```

do we branch? → and

beq hasn't even started the comparison yet!

Edge-triggering and the R-file



R-file uses negative-edge triggered devices

Three solutions to data hazards

Method 1: Stalling

```
add   $t0, $t1, $t2
sub   $t3, $t4, $t0
```

← wait for $t0 to be written back to R-file

- suspend execution for one or more clock cycles.