Pipelining review

Pipeline Control – big picture

Pipeline in action

Let's execute a program at address 0x1234:

- lw $1, 4($3)
- add $2, $3, $4
- or $3, $1, $2
- brn $1, $3, L

Suppose: $1 = 1, $2 = 2, $3 = 1000, $4 = 6

Pipeline in action 1

Clock: 1

$1 = 1, $2 = 2, $3 = 1000, $4 = 4

Pipeline in action 2

Clock: 2

$1 = 1, $2 = 2, $3 = 1000, $4 = 6
**Pipeline in action 3**

- Clock: 1 2 3
- **$1 = 1, $2 = 2, $3 = 1000, $4 = 6**

**Data Hazards!**

- Clock: 1 2 3 4
- **$1 = 1, $2 = 2, $3 = 1000, $4 = 6**

**Resolving data dependencies**

- Have the compiler generate no-ops
  - Don’t have to deal with data hazards in hardware.
- Stall the pipeline, i.e., create **bubbles**
  - The resulting delays are the same as for no-ops
- Send the result generated in stage 3 or stage 4 to the appropriate input of the ALU.
  - This is **forwarding** or **bypassing**.
  - More performance at the cost of more hardware
    - For one simple pipeline, cost is slightly more control and extra buses.
    - For several pipelines, say $n$, communication grows as $O(n^2)$

**Pipeline in action 4 (anything wrong?)**

- Clock: 1 2 3 4
- Let’s assume memory returns 42 for lw:
- **$1 = 1, $2 = 2, $3 = 1000, $4 = 6**

**Pipeline in action 5**

- Clock: 1 2 3 4 5
- **$1 = 42, $2 = 2, $3 = 1000, $4 = 6**

**Data Hazards**

- Data dependence
  - result of an operation needed before it is stored back in reg. file:
    - lw $55, 0($3)
    - add $55, $3, $4
  - The data hazard above is read after write (RAW)
- Data dependence (RAW) occurs when:
  - An instruction wants to read a register in stage 2, and
  - One instruction in stage 3 or stage 4 is going to write that register.
  - Note: if the instruction writing the register is in stage 5, this is fine since we can write a register and read it in the same cycle.
- Hazard detection unit compares register fields across stages.
Forwarding from WB to EX

IFF \\, $4, 3, 5$

Instruction memory

CLK

PC

bne $1, 3, L$

or $3, 1, 2$

or $3, 1, 2$

$2$ invalid

$4$ valid

lw $1, 4(3)$

$1 = 42, 2 = 2, 3 = 1000, 4 = 6$

Forwarding from MEM to EX

IFF \\, $4, 3, 5$

Instruction memory

CLK

PC

bne $1, 3, L$

or $3, 1, 2$

or $3, 1, 2$

$2$ invalid

$4$ valid

lw $1, 4(3)$

$1 = 42, 2 = 2, 3 = 1000, 4 = 6$

Generally, need to forward to both inputs!

IFF \\, $4, 3, 5$

Instruction memory

CLK

PC

bne $1, 3, L$

or $3, 1, 2$

or $3, 1, 2$

$2$ invalid

$4$ valid

lw $1, 4(3)$

$1 = 42, 2 = 2, 3 = 1000, 4 = 6$

Forwarding a register twice?

What if had:

- add $1, 2, 2$
- or $1, 3, 4$
- and $5, 1, 1$

Where do you forward $1$ from?

Another look at forwarding...

Data hazards and loads

lw $1, 4(3)$

add $2, 3, 4$

or $3, 1, 2$

bne $1, 3, L$

Can’t do forwarding in this case: can’t go back in time.

Have to insert a bubble between lw and or

lw $1, 4(3)$

or $3, 1, 2$

bne $1, 3, L$

If we switch these?

add $2, 3, 4$
**Inserting a bubble**

- Have to do it when you load into a register which is used in the next instruction

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st $</td>
<td>$ 2nd $</td>
<td>$ 3rd $</td>
<td>$ 4th $</td>
<td>$ 5th $</td>
<td>$ 6th $</td>
<td>$ 7th $</td>
<td>$ 8th $</td>
</tr>
</tbody>
</table>

- or $3, 1, 2$
- add $2, 3, 4$
- Insert bubble here

**Other hazards**

- We've seen data hazards
  - when an instruction in the pipeline is dependent on another instruction still in the pipeline.
  - Resolved by:
    - Bypassing/forwarding
    - Stalling
- Other types of hazards:
  - Structural hazards
    - where two instructions at different stages want to use the same resource.
    - Solving by using more resources (e.g., instruction and data memory; several ALUs). Won't happen in our pipeline.
  - Control hazards
    - happen on a taken branch.
    - Evaluation of branch condition and calculation of branch target is not completed before next instruction is fetched.

**Branch path**

- Stall until result of the condition & target are known.
- Too slow
- Reduce penalty by redesigning the pipeline:
  - move branch calculation to ID stage
  - move branch comparison to ID stage
  - how to do quick compare?
  - use both edges of the clock
- Delay slots
  - specify in ISA that instruction after branch is always executed.
- Branch prediction
  - in IF stage, guess branch outcome
  - correct if it turns out to be wrong.

**Resolving control hazards**

- Consider a memory copy program:
  - loop: `lw $1, 0($2)`
  - `sw $1, 0($2)`
  - `add $3, $3, $4`
  - `add $4, $4, $4`
  - `bne $4, $5, Loop`

- Too many stalls!
- How do we fix this?