Can pipelining get us into trouble?

° Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - data hazards: attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in branch instructions

° Can always resolve hazards by waiting
  - pipeline control must detect the hazard
    - take action (or delay action) to resolve hazards

---

Single Memory is a Structural Hazard

Detection is easy in this case! (right half highlight means read, left half write)
Structural Hazards limit performance

° Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  • average CPI  1.3
  • otherwise resource is more than 100% utilized

Control Hazard Solutions

° Stall: wait until decision is clear
  • Its possible to move up decision to 2nd stage by adding hardware to check registers as being read

° Impact: 2 clock cycles per branch instruction
  => slow
Control Hazard Solutions

° Predict: guess one direction then back up if wrong
  - Predict not taken

° Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)
° More dynamic scheme: history of 1 branch (- 90%)

Control Hazard Solutions

° Redefine branch behavior (takes place after next instruction) “delayed branch”

° Impact: 0 clock cycles per branch instruction if can find instruction to put in “slot” (- 50% of time)
° As launch more instruction per clock cycle, less useful
Data Hazard on r1:

- Dependencies backwards in time are hazards

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```
Data Hazard Solution:

- “Forward” result from one stage to another

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

- Can’t solve with forwarding:
  - Must delay/stall instruction dependent on loads
Designing a Pipelined Processor

° Go back and examine your datapath and control diagram
° associated resources with states
° ensure that flows do not conflict, or figure out how to resolve
° assert control in appropriate stage

Pipelined Datapath (as in book); hard to read
Pipelined Processor (almost) for slides

What happens if we start a new instruction every cycle?

Control and Datapath

IR ← Mem[PC]; PC ← PC+4;

A ← R[rs]; B ← R[rt]

S ← A + B;
S ← A or ZX;
S ← A + SX;
If Cond
PC ← PC + SX;

M ← Mem[S]
Mem[S] ← B

R[rd] ← S;
R[rt] ← S;
R[rd] ← M;

Next PC

Reg File

Exec

Mem File

Data Mem

IR

Inst. Mem

Dc0 Ctrl

Ex Ctrl

Mem Ctrl

WB Ctrl

Equal
## Pipelining the Load Instruction

The five independent functional units in the pipeline datapath are:

- Instruction Memory for the Ifetch stage
- Register File’s Read ports (bus A and busB) for the Reg/Dec stage
- ALU for the Exec stage
- Data Memory for the Mem stage
- Register File’s Write port (bus W) for the Wr stage

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1st lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
</tr>
<tr>
<td>2nd lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
</tr>
<tr>
<td>3rd lw</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
<td></td>
</tr>
</tbody>
</table>

## The Four Stages of R-type

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - ALU operates on the two register operands
  - Update PC
- **Wr:** Write the ALU output back to the register file
Pipelining the R-type and Load Instruction

We have pipeline conflict or structural hazard:
- Two instructions try to write to the register file at the same time!
- Only one write port

Important Observation
- Each functional unit can only be used once per instruction
- Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File’s Write Port during its 5th stage
    
    \[
    \begin{array}{ccccc}
    \text{1} & 2 & 3 & 4 & 5 \\
    \text{Load} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Mem} & \text{Wr}
    \end{array}
    \]
  - R-type uses Register File’s Write Port during its 4th stage
    
    \[
    \begin{array}{ccccc}
    \text{1} & 2 & 3 & 4 \\
    \text{R-type} & \text{Ifetch} & \text{Reg/Dec} & \text{Exec} & \text{Wr}
    \end{array}
    \]
- 2 ways to solve this pipeline hazard.
Solution 1: Insert “Bubble” into the Pipeline

- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Solution 2: Delay R-type’s Write by One Cycle

- Delay R-type’s register write by one cycle:
  - Now R-type instructions also use Reg File’s write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.
The Four Stages of Store

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Write the data into the Data Memory
The Three Stages of Beq

- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory

- **Reg/Dec:**
  - Registers Fetch and Instruction Decode

- **Exec:**
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Control Diagram

- \( IR \leftarrow \text{Mem}[PC]; \ PC \leftarrow PC + 4; \)
- \( A \leftarrow R[rs]; B \leftarrow R[rt]; \)
- \( S \leftarrow A + B; \)
- \( S \leftarrow A \text{ or } ZX; \)
- \( S \leftarrow A + SX; \)
- \( S \leftarrow A + SX; \)
- \( M \leftarrow \text{Mem}[S]; \)
- \( \text{Mem}[S] \leftarrow B; \)
- \( S \leftarrow A \text{ or } ZX; \)
- \( M \leftarrow \text{Mem}[S]; \)
- \( \text{Mem}[S] \leftarrow B; \)
- \( M \leftarrow S; \)
- \( R[rd] \leftarrow S; \)
- \( R[rt] \leftarrow S; \)
- \( R[rd] \leftarrow M; \)
- If Cond PC \( < \text{PC} + SX; \)
- Next PC
- PC
- Inst. Mem
- Reg File
- Exec
- Mem Access
- Data Mem
- Equal
- M
- Reg File
Datapath + Data Stationary Control