Pipelined Processors
Data and Control Hazards
Reminder: Processor Performance

- "Iron Law" of performance:
  \[
  \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
  \]

  \[
  \text{Perf} = \frac{1}{\text{Time}}
  \]

- Pipelining Goals:
  - Lower CPI: Keep CPI as close to 1 as possible
  - Lower cycle time since each pipeline stage does less work than a single cycle processor.
Reminder: Pipelining with Data Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages
  - Simple, wastes cycles, higher CPI

- **Strategy 2: Bypass.** Route data to the earlier pipeline stage as soon as it is calculated
  - More expensive, lower CPI
  - Still needs stalls when result is produced after EXE stage
  - Can trade off having fewer bypasses with stalling more often

Nov 18, 2021  MIT 6.004 Fall 2021
Resolving Data Hazards by Stalling

- **Strategy 1:** Stall. Wait for the result to be available by freezing earlier pipeline stages.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Stalls increase CPI!**
Resolving Data Hazards by Bypassing

- **Strategy 2: Bypass.** Route data to the earlier pipeline stage as soon as it is calculated.

- **addi** writes to x11 at the end of cycle 5... but the result is produced during cycle 3, at the EXE stage!

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
</tr>
</tbody>
</table>

- **addi** result computed
- x11 updated
Variable Memory Response Time

- Timing of clocked read assuming cache hit (returns data by next clock cycle)

- Timing of clocked read on cache miss. The cache will produce a stall signal, telling the pipeline to wait until the memory responds.
Handling Instruction Cache Miss by Stalling

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages.

### Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stall

Instruction cache hasn’t responded to fetch of **xor**

Instruction cache returns **xor** instruction. Begins fetch of **sub**

*Nov 18, 2021*  
MIT 6.004 Fall 2021
**Stall Logic for Instruction Cache Miss**

- **STALL==1**
  - Disables PC and IF pipeline register
  - Instruction cache keeps working to fetch data from memory
  - Injects NOP instruction into EXE stage

- Control logic sets STALL=1 if instruction cache misses (in addition to setting it when a data hazard exists.)
Resolving Data Cache Miss by Stalling

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>lw</td>
<td>sub</td>
<td>xori</td>
<td>ori</td>
<td>nextI</td>
<td>nextI</td>
<td>nextI</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>lw</td>
<td>sub</td>
<td>xori</td>
<td>ori</td>
<td>ori</td>
<td>ori</td>
<td>ori</td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>lw</td>
<td>sub</td>
<td>xori</td>
<td>xori</td>
<td>xori</td>
<td>xori</td>
<td>xori</td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>lw</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
</tbody>
</table>

Data cache miss on `lw` request of cycle 5

addi x9, x10, 2  
`lw` x13, 0(x11)  
`sub` x17, x15, x16  
xori x19, x18, 0xF  
`ori` x2, x1, 0x3

MIT 6.004 Fall 2021
Nov 18, 2021
Control Hazards
Which instruction to fetch next?

- So far, we have only considered sequential execution where nextPC = PC + 4.

- Now, we will add support for branch and jump instructions.
Control Hazards

- What do we need to compute nextPC?
  - We always need **opcode** to know how to compute nextPC
  - JAL: nextPC = pc + immJ
  - JALR: nextPC = \((\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\)
  - Branches: nextPC = brFun(\text{reg}[\text{rs1}], \text{reg}[\text{rs2}])? pc + immB : pc + 4
  - All other instructions: nextPC = PC + 4

- In what stage is nextPC available?
  - Depends on the pipeline and instruction type
Resolving Control Hazards

- In what stage is \textit{nextPC} available?

\begin{itemize}
  \item pc available in IF
  \item opcode, imm available in DEC
  \item operations on pc, imm, reg[rs1], reg[rs2] available in EXE
\end{itemize}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Instruction} & \textbf{Stage} \\
\hline
JAL & EXE \\
JALR & EXE \\
Branches & EXE \\
Others & DEC \\
\hline
\end{tabular}
\end{center}
Resolving Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages

- **Strategy 2: Bypass (aka Forward).** Route data to the earlier pipeline stage as soon as it is calculated

- **Strategy 3: Speculate**
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
Resolving Control Hazards By Stalling

- Assume `bne` is taken in this example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>NOP</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
</tr>
</tbody>
</table>

Opcode not known yet nextPC unknown → Stall

Opcode = `addi`
nextPC = PC + 4

Opcode = `bne`
nextPC unknown (branch outcome in EXE) → Stall once more

CPI = 7 cycles / 3 instructions!
Might as well not pipeline...
Resolving Hazards

- **Strategy 1: Stall.** Wait for the result to be available by freezing earlier pipeline stages.

- **Strategy 2: Bypass (aka Forward).** Route data to the earlier pipeline stage as soon as it is calculated.

- **Strategy 3: Speculate**
  - Guess a value and continue executing anyway.
  - When actual value is available, two cases:
    - Guessed correctly → do nothing
    - Guessed incorrectly → kill & restart with correct value
Resolving Control Hazards with Speculation

- **What’s a good guess for nextPC?** PC+4

- **Assume** bne is not taken in example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
</tr>
</tbody>
</table>

Start fetching at PC+4 (and) but bne not resolved yet...

Guessed right, keep going

Nov 18, 2021

MIT 6.004 Fall 2021

L20-17
Resolving Control Hazards with Speculation

- *What’s a good guess for nextPC?* PC+4

- Assume `bne` is taken in example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>sub</td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

Start fetching at PC+4 (and) but `bne` not resolved yet …

Guesses wrong, annul and & xor and restart fetching at loop

Start fetching at PC+4 (and) but `bne` not resolved yet …

GUessed wrong, annul and & xor and restart fetching at loop

Nov 18, 2021
Speculation Logic

- When EXE finds a jump or taken branch, it supplies nextPC and sets ANNUL==1
  - Writes NOPs in IF/DEC and DEC/EXE pipeline registers, annulling instructions currently in IF and DEC stages (called branch annulment)
  - Loads the branch or jump target into PC register
Interaction Between Stalling and Speculation

- Suppose that, on the same cycle,
  - EXE wants to annul DEC and IF due to a control hazard
  - DEC wants to stall due to a data hazard

- Example: Assume \texttt{bne} is taken

```
loop:  addi x12, x11, -1  
lw x14, 0(x15)  
bne x13, x0, loop  
and x16, x14, x18  
xor x19, x20, x21
```

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>DEC</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>lw</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>bne</td>
<td>lw</td>
<td>addi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>and</td>
<td>bne</td>
<td>lw</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>xor</td>
<td>and</td>
<td>bne</td>
<td>addi</td>
<td></td>
</tr>
</tbody>
</table>

\texttt{bne} wants to annul; \texttt{and} wants to stall

- \textbf{Which should take precedence, ANNUL or STALL?}

**ANNUL, because it comes from an earlier instruction**
Putting It All Together

- Let’s see an example with stalls, bypassing, and (mis)speculation
- Assume bne is taken once, then not taken

```
Assume bne is taken once, then not taken.
```

### Instruction Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>and</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>and</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bne taken, annuls and and xor, and stalls on x14.

lw value bypassed.
Summary

- Stalling can address all pipeline hazards
  - Simple, but hurts CPI
- Bypassing improves CPI on data hazards
- Speculation improves CPI on control hazards
  - Speculation works only when it’s easy to make good guesses
Thank you!

Next lecture: Synchronization