Pipelined Processors
Data and Control Hazards

Notes:
- Quiz 2 on Thu April 20
- Quiz 2 will cover L7-L15
Reminder: Processor Performance

- "Iron Law" of performance:

\[
\text{Perf} = \frac{1}{\text{Time}} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

- 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
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Stalls increase CPI!

April 6, 2023

MIT 6.191 Spring 2023
Resolving Data Hazards by Bypassing

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

- **addi** writes to \( x_{11} \) 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>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( \text{addi } x_{11}, x_{10}, 2 \)
- \( \text{xor } x_{13}, x_{11}, x_{12} \)
- \( \text{sub } x_{17}, x_{15}, x_{16} \)
- \( \text{xori } x_{19}, x_{18}, 0xF \)
Load-To-Use Stalls

- Bypassing cannot eliminate load delays because their data is not available until the WB stage
- Bypassing from WB still saves a cycle:

```assembly
lw x11, 0(x10)
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

<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>lw</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xori</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

lw data available → 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.

---

*MIT 6.191 Spring 2023*
Handling Instruction 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>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>NOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Instruction cache returns **xor** instruction

Begins fetch of **sub**
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.

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

Data cache miss on lw request of cycle 5

lw completes
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 = {(reg[rs1] + immI)[31:1], 1'b0}
    - Branches: nextPC = brFun(reg[rs1], reg[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 nextPC available?

- pc available in IF
- opcode, imm available in DEC
- operations on pc, imm, reg[rs1], reg[rs2] available in EXE

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>EXE</td>
</tr>
<tr>
<td>JALR</td>
<td>EXE</td>
</tr>
<tr>
<td>Branches</td>
<td>EXE</td>
</tr>
<tr>
<td>Others</td>
<td>DEC</td>
</tr>
</tbody>
</table>

April 6, 2023

MIT 6.191 Spring 2023
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>NOP</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>NOP</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>NOP</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**loop:**

- `addi x12, x11, -1`
- `sub x14, x15, x16`
- `bne x13, x0, loop`

---

**Opcode not known yet**

- `nextPC = PC + 4`
- **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...

---

April 6, 2023

MIT 6.191 Spring 2023

L15-16
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>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Guesed right, keep going
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>addi</td>
<td>sub</td>
<td>bne</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

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

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

April 6, 2023

MIT 6.191 Spring 2023

L15-19
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 `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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>lw</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
</tr>
</tbody>
</table>

bne wants to annul; and wants to stall

- 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

```
<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></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

- `bne` taken, annuls `and` and `xor`
- `and` 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: Operating Systems