Pipelined Processors
Data and Control Hazards
“Iron Law” of performance:

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

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
Resolving Data Hazards by Stalling

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

```
IF  addi  xor  sub  sub  sub  sub  xori
DEC addi  xor  xor  xor  xor  sub  xori
EXE addi  NOP  NOP  NOP  xori  sub
MEM addi  NOP  NOP  NOP  xori
WB  addi  NOP  NOP  NOP  NOP
```

Stalls increase CPI!

November 17, 2020

MIT 6.004 Fall 2020
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>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>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>x11 updated</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```
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>
<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>sub</td>
<td>xori</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</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

November 17, 2020

MIT 6.004 Fall 2020
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></td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td></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
```

Stall

lw completes
Control Hazards
Which instruction to fetch next?

- So far, we have only considered sequential execution where $\text{nextPC} = \text{PC} + 4$.

- 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}[rs1] + \text{immI})[31:1], 1'b0\}
  - Branches: nextPC = \text{brFun}(\text{reg}[rs1], \text{reg}[rs2])? \text{pc} + \text{immB} : \text{pc} + 4
  - All other instructions: nextPC = PC + 4

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

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

- In what stage is nextPC available?

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Next 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>
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>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
</tr>
</tbody>
</table>

Opcode = `addi`  
nextPC = PC + 4

Opcode = `bne`  
nextPC unknown (branch outcome in EXE) \(\rightarrow\) 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

```
addi x12, x11, -1
sub x14, x15, x16
bne x13, x0, loop
and x16, x17, 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>
<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>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
</tr>
</tbody>
</table>

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

Guessed 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>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 ...

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

loop:  
addi x12, x11, -1  
sub x14, x15, x16  
bne x13, x0, loop  
and x16, x17, x18  
xor x19, x20, x21
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 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></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>addi</td>
<td></td>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>addi</td>
<td></td>
<td></td>
<td></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

```assembly
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>
<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>and</td>
<td>xor</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>

`bne` taken, annuls `and` and `and xor` and stalls on `x14` `lw` value bypassed

November 17, 2020
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