Processor Pipelining:
Data and Control Hazards

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
“Iron Law” of processor performance:

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

$$\text{Perf} = \frac{1}{\text{Time}}$$
Reminder: Pipelining and Hazards

- "Iron Law" of processor 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 seeks to improve performance by reducing cycle time
Reminder: Pipelining and Hazards

- “Iron Law” of processor performance:

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

- Pipelining seeks to improve performance by reducing cycle time

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value → Data hazard
  - The program counter → Control hazard (branches and jumps)
“Iron Law” of processor performance:

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

- Pipelining seeks to improve performance by reducing cycle time
- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value → Data hazard
  - The program counter → Control hazard (branches and jumps)

Hazards may increase cycles per instruction (CPI)
Classic 5-Stage RISC Pipeline
Classic 5-Stage RISC Pipeline

- PC
- Instruction Memory
- Decode
- Register File
- Execute
- Data Memory
- Register File
Pipeline registers separate different stages
Classic 5-Stage RISC Pipeline

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
Classic 5-Stage RISC Pipeline

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
- For now, nextPC = PC+4 (no branches or jumps)
Classic 5-Stage RISC Pipeline

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
- For now, nextPC = PC+4 (no branches or jumps)
**Classic 5-Stage RISC Pipeline**

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
- For now, nextPC = PC+4 (no branches or jumps)
- Instruction and data memories have clocked reads
Classic 5-Stage RISC Pipeline

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
- For now, nextPC = PC+4 (no branches or jumps)
- Instruction and data memories have clocked reads
Clarification: Memories with Combinational vs. Clocked Reads

- Last lecture showed a pipeline where instruction and data memories have combinational reads: reads complete in the same cycle.
Clarification: Memories with Combinational vs. Clocked Reads

- Last lecture showed a pipeline where instruction and data memories have combinational reads: reads complete in the same cycle.
Clarification: Memories with Combinational vs. Clocked Reads

- Last lecture showed a pipeline where instruction and data memories have combinational reads: reads complete in the same cycle.

- We will use memories with clocked reads: read data is available at the beginning of the next cycle.
Clarification: Memories with Combinational vs. Clocked Reads

- Last lecture showed a pipeline where instruction and data memories have combinational reads: reads complete in the same cycle.

- We will use memories with clocked reads: read data is available at the beginning of the next cycle.
Clarification: Memories with Combinational vs. Clocked Reads

- Last lecture showed a pipeline where instruction and data memories have combinational reads: reads complete in the same cycle

- We will use memories with clocked reads: read data is available at the beginning of the next cycle

- Clocked reads are more common
- Clocked reads simplify some pipelining issues
Reminder: Pipelined Execution

- IF: PC -> Instruction Memory
- DEC: Instruction Memory -> Decode
  Decode -> Register File
- EXE: Register File -> Execute
- MEM: Execute -> Data Memory
- WB: Data Memory -> Register File

PC +4

November 1, 2018
Reminder: Pipelined Execution

lw x11, 4(x12)
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
Reminder: Pipelined Execution

IF

DEC

EXE

MEM

WB

PC

Instruction Memory

+4

Decode

Register File

Execute

Data Memory

Register File

<table>
<thead>
<tr>
<th>Cycles</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reminder: Pipelined Execution

When do register reads and writes happen?

```
lw x11, 4(x12)
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1
```

<table>
<thead>
<tr>
<th>Stages</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
</tr>
</tbody>
</table>

November 1, 2018
Reminder: Pipelined Execution

When do register reads and writes happen?

Reads in DEC stage

Writes at end of WB stage

IF

DEC

EXE

MEM

WB

PC

Instruction Memory

Decode

Register File

Execute

Data Memory

Register File

lw x11, 4(x12)
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1

Cycles

<table>
<thead>
<tr>
<th>Stages</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td>lw</td>
<td>lw</td>
</tr>
</tbody>
</table>
Reminder: Pipelined Execution

IF

DEC

EXE

MEM

WB

PC

Instruction Memory

Decode

Register File

Execute

Register File

Data Memory

When do register reads and writes happen?

Reads in DEC stage

Writes at end of WB stage

lw x11, 4(x12)

lw x13, 8(x14)

sub x15, x16, x17

xor x19, x20, x21

add x22, x23, x24

addi x25, x26, 1

Cycles

Stages

<table>
<thead>
<tr>
<th>Stages</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
</tr>
</tbody>
</table>

Read x12

November 1, 2018
Reminder: Pipelined Execution

When do register reads and writes happen?
Reads in DEC stage
Writes at end of WB stage

IF
lw x11, 4(x12)
lw x13, 8(x14)
sub x15, x16, x17
xor x19, x20, x21
add x22, x23, x24
addi x25, x26, 1

DEC
lw
lw
sub
xor
add
addi

EXE
lw
lw
sub
xor
add

MEM
lw
lw
sub

WB
lw
lw
lw

Cycles

<table>
<thead>
<tr>
<th>Stages</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td>addi</td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>lw</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read x12
Write x11

PC
Instruction Memory

+4

Decode
Register File

Execute

Register File

Data Memory

November 1, 2018

MIT 6.004 Fall 2018
Reminder: Data Hazards

- Consider this instruction sequence:

  ```
  addi x11, x10, 2
  xor  x13, x11, x12
  sub  x17, x15, x16
  xori x19, x18, 0xF
  ```
Reminder: Data Hazards

- Consider this instruction sequence:

```
addi x11, x10, 2
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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Reminder: Data Hazards

- Consider this instruction sequence:

```plaintext
addi x11, x10, 2
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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

November 1, 2018
## Reminder: Data Hazards

- Consider this instruction sequence:

```plaintext
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td></td>
</tr>
</tbody>
</table>
Reminder: Data Hazards

- Consider this instruction sequence:

```
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xor 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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td></td>
</tr>
</tbody>
</table>

- xor reads x11 on cycle 3, but addi does not update it until end of cycle 5 → x11 is stale!
Reminder: Data Hazards

- Consider this instruction sequence:

\[
\begin{align*}
&\text{addi } x11, x10, 2 \\
&\text{xor } x13, x11, x12 \\
&\text{sub } x17, x15, x16 \\
&\text{xori } x19, x18, 0xF \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Cycle</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>addi</td>
<td>addi</td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>4</td>
<td>xori</td>
<td>xori</td>
<td>xori</td>
<td>xor</td>
<td>xor</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- xor reads \(x11\) on cycle 3, but addi does not update it until end of cycle 5 \(\rightarrow x11\) is stale!
- Pipeline must maintain correct behavior...
Resolving Hazards

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

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

\[
\begin{align*}
\text{addi } x11, x10, 2 \\
\text{xor } x13, x11, x12 \\
\text{sub } x17, x15, x16 \\
\text{xori } x19, x18, 0xF
\end{align*}
\]
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>xori</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>

Stall: addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
Resolving Data Hazards by Stalling

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

![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><strong>sub</strong></td>
<td><strong>sub</strong></td>
<td><strong>sub</strong></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><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td><strong>NOP</strong></td>
<td></td>
</tr>
</tbody>
</table>

- stall

- addi x11, x10, 2
- xor x13, x11, x12
- sub x17, x15, x16
- xori x19, x18, 0xF

- x11 updated
Resolving Data Hazards by Stalling

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

```
1 2 3 4 5 6 7 8
IF addi xor sub sub sub xori
DEC addi xor xor xor xor sub xori
EXE addi NOP NOP NOP xor sub
MEM addi NOP NOP NOP xor
WB addi NOP NOP NOP
```

- Stalls increase CPI!

- x11 updated
Stall Logic

- New **STALL** control signal
- **STALL==1**
  - Disables PC and IF pipeline register (freezing IF and DEC)
  - Injects NOP instruction into EXE stage
- NOP = No-operation, e.g., addi x0, x0, 0

![Pipeline Diagram]

PC -> Instruction Memory

IF stage

+4

Instruction Memory

Decode

Register File

NOP

Decoding

Execute

Data Memory

Register File

WB

November 1, 2018
Stall Logic

- New **STALL** control signal
- STALL\(==1\)
  - Disables PC and IF pipeline register (freezing IF and DEC)
  - Injects NOP instruction into EXE stage
- NOP = No-operation, e.g., `addi x0, x0, 0`
- Control logic sets STALL\(=1\) if source registers of instruction in DEC match destination register on EXE, MEM, or WB *(except when source is x0!)*
Data Hazards due to Loads & Stores

- Is there any possible data hazard in this instruction sequence?

  \[
  \text{st } x11, 8(x10) \\
  \text{ld } x13, 4(x12) \\
  \ldots
  \]

  \[
  \begin{align*}
  \text{mem}[\text{reg}[10] + 8] & \leftarrow \text{reg}[11] \\
  \text{reg}[13] & \leftarrow \text{mem}[\text{reg}[12] + 4]
  \end{align*}
  \]
Data Hazards due to Loads & Stores

- Is there any possible data hazard in this instruction sequence?

\[
\text{st } x_{11}, \ 8(x_{10}) \\
\text{ld } x_{13}, \ 4(x_{12}) \\
\text{mem}[\text{reg}[10] + 8] \leftarrow \text{reg}[11] \\
\text{reg}[13] \leftarrow \text{mem}[\text{reg}[12] + 4] \\
\ldots
\]

- What if \( \text{reg}[10] + 8 = \text{reg}[12] + 4 \)?
Data Hazards due to Loads & Stores

- Is there any possible data hazard in this instruction sequence?
  - `st x11, 8(x10)`, `ld x13, 4(x12)`


- Our pipeline has no hazards through memory because writes complete in one cycle, but this is a major issue in more complex processors.
  - In general, load & store hazards may be solved in the pipeline or in the memory system.
Resolving Data Hazards by Bypassing

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

```
addi x11, x10, 2
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>
</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></td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
</tr>
</tbody>
</table>
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></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>xor</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
</tr>
</tbody>
</table>

`addi` result computed
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></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
</tr>
</tbody>
</table>

- addi result computed
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></td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
</tr>
</tbody>
</table>
```

`addi x11, x10, 2`  
`xor x13, x11, x12`  
`sub x17, x15, x16`  
`xori x19, x18, 0xF`
Bypass Logic

- Add bypass muxes to DEC outputs
- Route EXE, MEM, WB outputs to mux inputs
Add bypass muxes to DEC outputs
Route EXE, MEM, WB outputs to mux inputs
Bypass value if destination register of instruction in EXE, MEM, or WB matches source register of instruction in DEC
Bypass Logic

- Add bypass muxes to DEC outputs
- Route EXE, MEM, WB outputs to mux inputs
- Bypass value if destination register of instruction in EXE, MEM, or WB matches source register of instruction in DEC
  - No bypass for x0
Bypass Logic

- Add bypass muxes to DEC outputs
- Route EXE, MEM, WB outputs to mux inputs
- Bypass value if destination register of instruction in EXE, MEM, or WB matches source register of instruction in DEC
  - No bypass for x0
- If multiple matches, use value from most recent instruction (EXE > MEM > WB)
Full vs. Partial Bypassing
Full vs. Partial Bypassing

- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time...

IF
- PC
- Instruction Memory
- +4
- Decode
- Register File
- STALL

DEC
- NOP
- STALL

EXE
- Execute

MEM
- Data Memory

WB
- Register File

November 1, 2018
Full vs. Partial Bypassing

- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time...

- But full bypassing is not needed! We can always stall
  - e.g., just bypass from EXE and WB

---

Bypasses are expensive
- Lots of wires & large muxes
- May affect clock cycle time...

But full bypassing is not needed! We can always stall
- e.g., just bypass from EXE and WB
Full vs. Partial Bypassing

- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time...

- But full bypassing is not needed! We can always stall
  - e.g., just bypass from EXE and WB
Full vs. Partial Bypassing

- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time...

- But full bypassing is not needed! We can always stall
  - e.g., just bypass from EXE and WB
Full vs. Partial Bypassing

- Bypasses are expensive
  - Lots of wires & large muxes
  - May affect clock cycle time...

- But full bypassing is not needed! We can always stall
  - e.g., just bypass from EXE and WB

- With a fully bypassed pipeline, do we still need the stall signal?
Load-To-Use Stalls

- Bypassing cannot eliminate load delays because their data is not available until the WB stage

\begin{align*}
  & \text{lw } x11, 0(x10) \\
  & \text{xor } x13, x11, x12 \\
  & \text{sub } x17, x15, x16 \\
  & \text{xori } x19, x18, 0xF
\end{align*}
Load-To-Use Stalls

- Bypassing cannot eliminate load delays because their data is not available until the WB stage.

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

\[
\begin{align*}
&\text{lw } x_{11}, 0(x_{10}) \\
&\text{xor } x_{13}, x_{11}, x_{12} \\
&\text{sub } x_{17}, x_{15}, x_{16} \\
&\text{xori } x_{19}, x_{18}, 0xF
\end{align*}
\]
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:

```
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>NOP</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

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

lw data available
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:

```
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>xori</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>xori</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

lw data available
### 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:

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

**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>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>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- `lw` data available
- `x11` updated

November 1, 2018
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

  2 stalls (w/ bypasses)
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

2 stalls (w/ bypasses)
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

2 stalls (w/ bypasses)
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

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

2 stalls (w/ bypasses)
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away
- Example:

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

2 stalls (w/ bypasses)

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

No stalls
Compilers Can Help

- Compilers can rearrange code to put dependent instructions farther away

Example:

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

2 stalls (w/ bypasses)

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

No stalls

- Only works well when compiler can find independent instructions to move around!
Summary: Pipelining with Data Hazards

- Strategy 1: Stall. Wait for the result to be available by freezing earlier pipeline stages
  - Simple, wastes cycles, higher CPI
Summary: 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 ALU stage
  - Can use fewer bypasses & stall more often
Summary: 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 ALU stage
  - Can use fewer bypasses & stall more often

- More pipeline stages $\rightarrow$ More frequent data hazards
  - Lower $t_{CK}$, but higher CPI
Resolving Control Hazards

- What do we need to compute nextPC?
  - We always need opcode to know how to compute nextPC
Resolving Control Hazards

- What do we need to compute nextPC?
  - We always need opcode to know how to compute nextPC
  - JAL: \( \text{nextPC} = \text{pc} + \text{immJ} \)
Resolving 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}
Resolving 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
Resolving 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
Resolving 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?
Resolving 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?**

<table>
<thead>
<tr>
<th>JAL</th>
<th>JALR</th>
<th>Branches</th>
<th>Others</th>
</tr>
</thead>
</table>

November 1, 2018
Resolving Control Hazards

In what stage is nextPC available?

- JAL
- JALR
- Branches
- Others
Resolving Control Hazards

In what stage is nextPC available?

- pc available in IF
- opcode, imm available in DEC

<table>
<thead>
<tr>
<th>JAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JALR</td>
</tr>
<tr>
<td>Branches</td>
</tr>
<tr>
<td>Others</td>
</tr>
</tbody>
</table>

November 1, 2018
Resolving Control Hazards

- **IF**
  - Instruction Memory
  - pc available in IF

- **DEC**
  - Decode
  - Register File
  - opcode, imm available in DEC
  - operations on pc, imm, reg[rs1], reg[rs2] available in EXE

- **EXE**
  - Execute

- **MEM**
  - Data Memory

- **WB**
  - Register File

- **In what stage is nextPC available?**

- **Table**
  - JAL
  - JALR
  - Branches
  - Others
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>JAL</th>
<th>EXE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JALR</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td></td>
</tr>
</tbody>
</table>
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</th>
<th>Memory</th>
<th>Decode</th>
<th>Register File</th>
<th>Execute</th>
<th>Data Memory</th>
<th>Register File</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resolving Control Hazards

- In what stage is nextPC available?

```
nextPC
IF
| Instruction Memory
| pc available in IF

DEC
| Decode
| Register File
| opcode, imm available in DEC
| operations on pc, imm, reg[rs1], reg[rs2] available in EXE

EXE
| Execute

MEM
| Data Memory

WB
| Register File
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td>EXE</td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td>EXE</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td>EXE</td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resolving Control Hazards

- **In what stage is nextPC available?**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory</th>
<th>Decode</th>
<th>Register File</th>
<th>Execute</th>
<th>Data Memory</th>
<th>Register File</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Others</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>
</tr>
</tbody>
</table>

- **pc available in IF**
- **opcode, imm available in DEC**
- **operations on pc, imm, reg[rs1], reg[rs2] available in EXE**
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 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
Resolving Control Hazards By Stalling

- Assume `bne` is taken in this example

```
loop:  addi x13, x11, -1
       sub x14, x15, x16
       bne x13, x0, loop
```
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>addi</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>bne</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 x13, x11, -1`  
`sub x14, x15, x16`  
`bne x13, x0, loop`
Resolving Control Hazards By Stalling

- Assume `bne` is taken in this example

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

<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>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></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
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 not known yet

nextPC unknown → Stall

Opcode = addi

nextPC = PC + 4

loop: addi x13, x11, -1
sub x14, x15, x16
bne x13, x0, loop
### Resolving Control Hazards By Stalling

- **Assume** `bne` is taken in this example

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

<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><strong>IF</strong></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><strong>DEC</strong></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><strong>EXE</strong></td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>addi</td>
<td></td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td></td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>NOP</td>
<td>bne</td>
<td></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
```
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></td>
<td></td>
<td></td>
<td></td>
<td></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...

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

November 1, 2018
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.
What’s a good guess for nextPC?
Resolving Control Hazards with Speculation

- *What’s a good guess for nextPC?*  PC+4
Resolving Control Hazards with Speculation

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

- Assume bne is not taken in example

```assembly
loop:  addi x13, x11, -1
       sub  x14, x15, x16
       bne x13, x0, loop
       and x16, x17, x18
       xor x19, x20, x21
       ...
```
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>

loop:  
```plaintext
addi x13, x11, -1
sub x14, x15, x16
bne x13, x0, loop
and x16, x17, x18
xor x19, x20, x21
...```

November 1, 2018
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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```
loop:  addi x13, x11, -1
       sub x14, x15, x16
       bne x13, x0, loop
       and x16, x17, x18
       xor x19, x20, x21
       ...
```
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>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

```
loop:   addi x13, x11, -1
sub x14, x15, x16
bne x13, x0, loop
and x16, x17, x18
xor x19, x20, x21
...`

Resolving Control Hazards with Speculation

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

- Assume `bne` is **taken** in example

```
loop:  addi x13, 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>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>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

loop: addi x13, x11, -1
sub x14, x15, x16
bne x13, x0, loop
and x16, x17, x18
xor x19, x20, x21
...
## 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 ...

```
loop: addi x13, x11, -1
     sub x14, x15, x16
     bne x13, x0, loop
     and x16, x17, x18
     xor x19, x20, x21
     ...
```

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

---

November 1, 2018
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>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</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>bne</td>
<td>and</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>sub</td>
<td>bne</td>
<td>and</td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>sub</td>
<td>bne</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

November 1, 2018
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 ...

Loop:  

```
addi x13, x11, -1
sub x14, x15, x16
bne x13, x0, loop
and x16, x17, x18
xor x19, x20, x21
...  
```

Guessed wrong, annul and & xor and restart fetching at loop.
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
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 x13, 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
**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

```plaintext
loop:  addi x13, x11, -1  
dl w 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><strong>IF</strong></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td>xor</td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td></td>
<td></td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td>lw</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
</tr>
</tbody>
</table>

*bne* wants to annul; *and* wants to stall
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

```assembly
loop:    addi x13, 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?
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 x13, 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>addi</td>
<td>lw</td>
<td>bne</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td>addi</td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

\textbf{ANNUL, because it comes from an earlier instruction}
Putting It All Together

- Let’s see an example with stalls, bypassing, and (mis)speculation
Let’s see an example with stalls, bypassing, and (mis)speculation

Assume `bne` is taken once, then not taken

```
loop:    addi x13, x11, -1
         lw x14, 0(x15)
         bne x13, x0, loop
         and x16, x14, x18
         xor x19, x20, x21
```
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>xor</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>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

```
loop: addi x13, 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>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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`bne` taken, annuls `and` and `xor`
Putting It All Together

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

```
loop: addi x13, 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>NOP</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>and</td>
<td>xor</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>NOP</td>
<td>NOP</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>NOP</td>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

bne taken, annuls and and xor
Putting It All Together

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

```
loop:  addi x13, 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></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>xor</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></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>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`bne` taken, annuls `and` and `xor` and stalls on `x14`
## 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>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>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</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>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `bne` taken, annuls `and` and `xor` and stalls on `x14`
- `lw` value bypassed
Putting It All Together

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

```
loop:   addi x13, 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></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>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</td>
<td>NOP</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>NOP</td>
<td>addi</td>
<td>lw</td>
<td>bne</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
Summary

- Stalling can address all pipeline hazards
  - Simple, but hurts CPI
- Bypassing improves CPI on data hazards
- Speculation improves CPI on control hazards
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
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
- This is all you need to analyze simple pipelined processors
- Next week: How to build them!
Thank you!

Next lecture: Implementing pipelined processors