Introduction to Pipelining

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Performance Measures

- Two metrics of interest when designing a system:
Performance Measures

• Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced
Performance Measures

- Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed
Performance Measures

• Two metrics of interest when designing a system:

1. Latency: The delay from when an input enters the system until its associated output is produced

2. Throughput: The rate at which inputs or outputs are processed

• The metric to prioritize depends on the application
Performance Measures

• Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

• The metric to prioritize depends on the application
  – *Airbag deployment system*?
Performance Measures

• Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

• The metric to prioritize depends on the application
  – *Airbag deployment system?*  Latency
Performance Measures

• Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

• The metric to prioritize depends on the application
  – *Airbag deployment system?*  **Latency**
  – *General-purpose processor?*
Performance Measures

- Two metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced.

2. Throughput: The *rate* at which inputs or outputs are processed.

- The metric to prioritize depends on the application:
  - *Airbag deployment system?* Latency
  - *General-purpose processor?* Throughput (maximize instructions/second)
For combinational logic:
latency = $t_{PD}$
throughput = $1/t_{PD}$

We can’t get the answer any faster, but are we making effective use of our hardware at all times?
For combinational logic:
latency = $t_{PD}$
throughput = $1/t_{PD}$

We can’t get the answer any faster, but are we making effective use of our hardware at all times?
For combinational logic:
- latency = $t_{PD}$
- throughput = $1/t_{PD}$

We can’t get the answer any faster, but are we making effective use of our hardware at all times?

F & G are “idle”, just holding their outputs stable while H performs its computation.
Use registers to hold H’s input stable!

Now F & G can be working on input $X_{i+1}$ while H is performing its computation on $X_i$. We’ve created a 2-stage pipeline: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers:

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipeline</td>
<td>______</td>
<td>______</td>
</tr>
</tbody>
</table>
Pipelined Circuits

Use registers to hold H’s input stable!

Now F & G can be working on input $X_{i+1}$ while H is performing its computation on $X_i$. We’ve created a 2-stage pipeline: if we have a valid input $X$ during clock cycle $j$, $P(X)$ is valid during clock $j+2$.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers:

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipeline</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>
Use registers to hold H’s input stable!

Now F & G can be working on input \( X_{i+1} \) while H is performing its computation on \( X_i \). We’ve created a 2-stage pipeline: if we have a valid input \( X \) during clock cycle \( j \), \( P(X) \) is valid during clock \( j+2 \).

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers:

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipeline</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>worse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Use registers to hold H’s input stable!

Now F & G can be working on input $X_{i+1}$ while H is performing its computation on $X_i$. We’ve created a 2-stage pipeline: if we have a valid input X during clock cycle $j$, $P(X)$ is valid during clock $j+2$.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers:

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipeline</td>
<td>50</td>
<td>1/25</td>
</tr>
</tbody>
</table>

worse
Pipelined Circuits

Use registers to hold H’s input stable!

Now F & G can be working on input $X_{i+1}$ while H is performing its computation on $X_i$. We’ve created a 2-stage pipeline: if we have a valid input $X$ during clock cycle $j$, $P(X)$ is valid during clock $j+2$.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers:

<table>
<thead>
<tr>
<th></th>
<th>latency</th>
<th>throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipeline</td>
<td>50</td>
<td>1/25</td>
</tr>
</tbody>
</table>

worse better!
Pipeline Diagrams

Clock cycle

<table>
<thead>
<tr>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F &amp; G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline stages
Pipeline Diagrams

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F &amp; G</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline stages

X

Stable here

April 19, 2018 L17-5
Pipeline Diagrams

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th>Clock cycle</th>
<th>F &amp; G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>F(X_i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i+3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- F(15)
- G(20)
- H(25)

X_i stable here

April 19, 2018
Pipeline Diagrams

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>F &amp; G</td>
<td>F(X_i)</td>
<td>F(X_{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>G(X_i)</td>
<td>G(X_{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>H(X_i)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline stages

X_i stable here

April 19, 2018
Pipeline Diagrams

Clock cycle

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F &amp; G</strong></td>
<td>F(X_i)</td>
<td>F(X_{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>G(X_i)</td>
<td>G(X_{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>H</strong></td>
<td>H(X_i)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline stages

- **F**
- **G**
- **H**

- **X_i** stable here
- **P(X_i)** available here

April 19, 2018
The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.
The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.
The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.
Definition:
A well-formed *K-Stage Pipeline* (“K-pipeline”) is an acyclic circuit having exactly K registers on *every* path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.
Definition:
A well-formed \textit{K-Stage Pipeline} (“K-pipeline”) is an acyclic circuit having exactly K registers on every path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:
Every pipeline stage, hence every K-Stage pipeline, has a register on its \textit{output} (not on its input).
Definition:
A well-formed **K-Stage Pipeline** (“K-pipeline”) is an acyclic circuit having exactly K registers on every path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:
Every pipeline stage, hence every K-Stage pipeline, has a register on its **output** (not on its input).

Clock period:
The clock must have a period $t_{\text{CLK}}$ sufficient to cover $t_{\text{PD}}$ over all combinational paths plus the register $t_{\text{SETUP}}$. 
Definition:
A well-formed \textit{K-Stage Pipeline} ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:
Every pipeline stage, hence every K-Stage pipeline, has a register on its output (not on its input).

Clock period:
The clock must have a period $t_{CLK}$ sufficient to cover $t_{PD}$ over all combinational paths plus the register $t_{SETUP}$.

\[
\text{K-pipeline latency } L = K \times t_{CLK} \\
\text{K-pipeline throughput } T = \frac{1}{t_{CLK}}
\]
Consider a BAD job of pipelining:

For what value of $K$ is the following circuit a $K$-Pipeline?
Consider a BAD job of pipelining:

For what value of K is the following circuit a K-Pipeline?
Ill-Formed Pipelines

Consider a BAD job of pipelining:

For what value of K is the following circuit a K-Pipeline?
Ill-Formed Pipelines

Consider a BAD job of pipelining:

For what value of K is the following circuit a K-Pipeline?
Ill-Formed Pipelines

Consider a BAD job of pipelining:

For what value of K is the following circuit a K-Pipeline? none
Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happens because some paths from inputs to outputs have 2 registers, and some have only 1!

This can’t happen on a well-formed K pipeline!
A Pipelining Methodology

A → B → C → D → E → F

- A: 4ns
- B: 3ns
- C: 8ns
- D: 4ns
- E: 2ns
- F: 5ns
Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

\[ T = \frac{1}{8\text{ns}} \]
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

\[ T = \frac{1}{8\text{ns}} \]
\[ L = 24\text{ns} \]
A Pipelining Methodology

Step 1:
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:
Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate pipeline stages.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

Strategy:
Focus your attention on placing pipelining registers around the slowest circuit elements (bottlenecks).

\[ T = \frac{1}{8\text{ns}} \]
\[ L = 24\text{ns} \]
Pipeline Example

OBSERVATIONS:

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2-pipe:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-pipe:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

April 19, 2018
Pipeline Example

OBSERVATIONS:

<table>
<thead>
<tr>
<th>PIPE</th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-pipe:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Example

OBSERVATIONS:

- 1-pipeline improves neither L nor T.

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-pipe:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Pipeline Example**

OBSERVATIONS:

- 1-pipeline improves neither L nor T.

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Example

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.

<table>
<thead>
<tr>
<th>PIPE</th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.

<table>
<thead>
<tr>
<th>0-pipe:</th>
<th>LATTENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Pipeline Example

X → A → B → C → Y

April 19, 2018
Pipeline Example

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don’t improve T.

<table>
<thead>
<tr>
<th></th>
<th>LATEENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>
**Pipeline Example**

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don’t improve T.
- Back-to-back registers are sometimes needed to keep pipeline well-formed.

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>
Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period.
Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period
- Must account for new pipeline stages in our plan

4-stage pipeline, throughput=1
Summary: Design Alternatives

- Combinational (C)
- Pipeline (P)
- Folded (F)

Reuse a block, multicycle
Summary: Design Alternatives

- **Combinational (C)**
  - Direct connection of functions $f_1$, $f_2$, and $f_3$
  - No pipeline stages

- **Pipeline (P)**
  - Functions $f_1$, $f_2$, and $f_3$ connected in a pipeline
  - Multicycle operation

- **Folded (F)**
  - Reuse a block
  - Multicycle operation

April 19, 2018
Summary: Design Alternatives

- **Combinational (C)**
- **Pipeline (P)**
- **Folded (F)**
  
  Reuse a block, multicycle
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

- **Combinational (C)**
- **Pipeline (P)**
- **Folded (F)**
  - Reuse a block, multicycle
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

- **Combinational (C)**
- **Pipeline (P)**
- **Folded (F)**
  - Reuse a block, multicycle
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

- Combinational (C)
- Pipeline (P)
- Folded (F)

Reuse a block, multicycle
Summary: Design Alternatives

- Combinational (C)
- Pipeline (P)
- Folded (F)
  - Reuse a block, multicycle

April 19, 2018
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

- Combinational (C)
- Pipeline (P)
- Folded (F)

Reuse a block, multicycle
Summary: Design Alternatives

- **Combinational (C):**
  - $f_1$ → $f_2$ → $f_3$ →

- **Pipeline (P):**
  - $f_1$ → $f_2$ → $f_3$ →

- **Folded (F):**
  - $f_i$ →

Reuse a block, multicycle
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle
Summary: Design Alternatives

- Combinational (C)
- Pipeline (P)
- Folded (F)

Reuse a block, multicycle

Clock?  Area?  Throughput?
Summary: Design Alternatives

- **Combinational (C)**
- **Pipeline (P)**
- **Folded (F)**

Clock: \( P \approx F < C \)

<table>
<thead>
<tr>
<th>Clock</th>
<th>Area?</th>
<th>Throughput?</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>F</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>C</td>
<td>🗨️</td>
<td>🗨️</td>
</tr>
</tbody>
</table>

April 19, 2018
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle

Clock: P \approx F < C
Area: F < C < P

Throughput?
Summary: Design Alternatives

Combinational (C)

Pipeline (P)

Folded (F)
Reuse a block, multicycle

Clock: $P \approx F < C$

Area: $F < C < P$

Throughput: $F < C < P$
Pipelined Processors
Processor Performance

- “Iron Law” of performance:

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

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

• “Iron Law” of performance:

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

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

• Options to reduce execution time:
  - Executed instructions $\downarrow$ (work/instruction $\uparrow$)
  - Cycles per instruction (CPI) $\downarrow$
  - Cycle time $\downarrow$ (frequency $\uparrow$)
Single-Cycle Processor Performance
Single-Cycle Processor Performance

- CPI = 1
Single-Cycle Processor Performance

- CPI = 1
- $t_{CK} =$ Longest path for any instruction
Single-Cycle Processor Performance

- CPI = 1
- \( t_{CK} = \) Longest path for any instruction

\[
t_{CK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}
\]
Single-Cycle Processor Performance

- CPI = 1
- \( t_{CK} = \) Longest path for any instruction
  \[
  t_{CK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}
  \]
Single-Cycle Processor Performance

- **CPI = 1**
- $t_{CK} = \text{Longest path for any instruction}$

$$t_{CK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}$$
Single-Cycle Processor Performance

- CPI = 1
- \( t_{CK} = \) Longest path for any instruction

\[
t_{CK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB}
\]

Slow!
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{CK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx$ 1.0
- We’ll study the classic 5-stage pipeline:
Pipelined Implementation

• Divide datapath in multiple pipeline stages to reduce $t_{ck}$
  – Each instruction executes over multiple cycles
  – Consecutive instructions are overlapped to keep $CPI \approx 1.0$
• We’ll study the classic 5-stage pipeline:

  IF

  Instruction Fetch stage: Maintains PC, fetches instruction and passes it to
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{CK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$

- We’ll study the classic 5-stage pipeline:

  ![Diagram](IF DEC)

  **Instruction Fetch stage**: Maintains PC, fetches instruction and passes it to

  **Decode & Read Registers stage**: Decodes instruction and reads source operands from register file, passes them to
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{\text{CK}}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$
- We’ll study the classic 5-stage pipeline:

  **IF**  
  Instruction Fetch stage: Maintains PC, fetches instruction and passes it to

  **DEC**  
  Decode & Read Registers stage: Decodes instruction and reads source operands from register file, passes them to

  **EXE**  
  Execute stage: Performs indicated operation in ALU, passes result to
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{CK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$
- We’ll study the classic 5-stage pipeline:

  **Instruction Fetch stage**: Maintains PC, fetches instruction and passes it to
  **Decode & Read Registers stage**: Decodes instruction and reads source operands from register file, passes them to
  **Execute stage**: Performs indicated operation in ALU, passes result to
  **Memory stage**: If it’s a load, use input as the address, pass read data (or ALU result if not a load) to
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{CK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$
- We’ll study the classic 5-stage pipeline:

  - **Instruction Fetch stage**: Maintains PC, fetches instruction and passes it to
  - **Decode & Read Registers stage**: Decodes instruction and reads source operands from register file, passes them to
  - **Execute stage**: Performs indicated operation in ALU, passes result to
  - **Memory stage**: If it’s a load, use input as the address, pass read data (or ALU result if not a load) to
  - **Write-Back stage**: writes result back into register file.
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{CK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$
- We’ll study the classic 5-stage pipeline:

  - **Instruction Fetch stage**: Maintains PC, fetches instruction and passes it to
  - **Decode & Read Registers stage**: Decodes instruction and reads source operands from register file, passes them to
  - **Execute stage**: Performs indicated operation in ALU, passes result to
  - **Memory stage**: If it’s a load, use input as the address, pass read data (or ALU result if not a load) to
  - **Write-Back stage**: writes result back into register file.

$$t_{CK} = \max\{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}\}$$
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?

![Diagram of a computer's instruction execution pipeline]

- PC
- Decode
- Execute
- Inst Memory
- Data Memory
- Register File
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?

- Processor has state: PC, Register file, Memories
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?

- Processor has state: PC, Register file, Memories
- There are loops we cannot break!
  - To compute the next PC
  - To write result into the register file
Why isn’t this a 30-minute lecture?

We know how to pipeline combinational circuits, what’s the big deal?

- Processor has state: PC, Register file, Memories
- There are loops we cannot break!
  - To compute the next PC
  - To write result into the register file
- Can’t produce a well-formed pipeline with what we know so far...
Pipeline Hazards

- 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, jumps, exceptions)
Pipeline Hazards

• 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, jumps, exceptions)

• Plan of attack:
  1. Design a 5-stage pipeline that works with sequences of independent instructions
  2. Handle data hazards
  3. Handle control hazards
Pipeline Hazards

- Pipelining tries to overlap the execution of multiple instructions, but an instruction may depend on something produced by an earlier instruction
  - A data value $\rightarrow$ Data hazard
  - The program counter $\rightarrow$ Control hazard
    (branches, jumps, exceptions)

- Plan of attack:
  1. Design a 5-stage pipeline that works with sequences of independent instructions
  2. Handle data hazards
  3. Handle control hazards

*Today: Concepts*  
*Next week: Implementation*
Simplified Single-Cycle Datapath
nextPC = PC+4 (we’ll worry about control hazards later)
Simplified Single-Cycle Datapath

- nextPC = PC+4 (we’ll worry about control hazards later)

- Same register file appears twice in the diagram
  - Top: reads
  - Bottom: writes
5-Stage Pipelined Datapath

- Pipeline registers separate different stages
- Each stage services one instruction per cycle
Example: Pipelined Execution
Example: 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
Example: 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
```

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

<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></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>

April 19, 2018
Example: 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>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></td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>lw</td>
<td></td>
<td></td>
<td>lw</td>
<td>lw</td>
</tr>
</tbody>
</table>
Example: Pipelined Execution

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

<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>

April 19, 2018
Example: Pipelined Execution

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

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

<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>PC</td>
<td>+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register File</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

April 19, 2018 L17-20
Example: Pipelined Execution

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

<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>add</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>

Cycles

Read x12
Write x11

April 19, 2018 L17-20
Data Hazards

• Consider this instruction sequence:

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

- Consider this instruction sequence:

<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>

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

• Consider this instruction sequence:

<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>

\[
\begin{align*}
\text{addi } & x11, x10, 2 \\
\text{xor } & x13, x11, x12 \\
\text{sub } & x17, x15, x16 \\
\text{xori } & x19, x18, 0xF
\end{align*}
\]
## 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><strong>IF</strong></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DEC</strong></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></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!
Data Hazards

- Consider this instruction sequence:

  \[
  \begin{align*}
  &\text{addi } x_{11}, x_{10}, 2 \\
  &\text{xor } x_{13}, x_{11}, x_{12} \\
  &\text{sub } x_{17}, x_{15}, x_{16} \\
  &\text{xori } x_{19}, x_{18}, 0xF
  \end{align*}
  \]

  \[
  \begin{array}{cccccc}
  1 & 2 & 3 & 4 & 5 & 6 \\
  \hline
  \text{IF} & \text{addi} & \text{xor} & \text{sub} & \text{xori} \\
  \text{DEC} & \text{addi} & \text{xor} & \text{sub} & \text{xori} \\
  \text{EXE} & \text{addi} & \text{xor} & \text{sub} & \text{xori} \\
  \text{MEM} & \text{addi} & \text{xor} & \text{sub} \\
  \text{WB} & \text{addi} & \text{xor} \\
  \end{array}
  \]

  - xor reads \textit{x11} on cycle 3, but addi does not update it until end of cycle 5 → \textit{x11} is stale!
  - Pipeline must maintain correct behavior...

April 19, 2018
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 (1)

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

  ```assembly
  addi x11, x10, 2
  xor x13, x11, x12
  sub x17, x15, x16
  xori x19, x18, 0xF
  ```
Resolving Data Hazards (1)

- 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></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>xori</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</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 (1)

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

```plaintext
<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>subst</td>
<td>subst</td>
<td>subst</td>
<td></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>
```

- x11 updated

### Code Snippet

```plaintext
addi x11, x10, 2
xor x13, x11, x12
sub x17, x15, x16
xori x19, x18, 0xF
```
Resolving Data Hazards (1)

- **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>subst</td>
<td>subst</td>
<td>subst</td>
<td></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>

**Stalls increase CPI!**

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

x11 updated
Resolving Data Hazards (2)

- 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></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>
Resolving Data Hazards (2)

- **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 available at the end of 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
xor x19, x18, 0xF

x11 updated

addi result computed
Resolving Data Hazards (2)

- **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 available at the end of 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>addi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

April 19, 2018
Control Hazards

• What do we need to compute nextPC?
  – JAL: nextPC = pc + immJ
Control Hazards

• What do we need to compute nextPC?
  - JAL: \( \text{nextPC} = \text{pc} + \text{immJ} \)
  
  - JALR: \( \text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\} \)
Control Hazards

• What do we need 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
Control Hazards

• What do we need to compute nextPC?
  – JAL: $\text{nextPC} = \text{pc} + \text{immJ}$

  – JALR: $\text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$

  – Branches: $\text{nextPC} = \text{brFun}(\text{reg}[\text{rs1}], \text{reg}[\text{rs2}]) ? \text{pc} + \text{immB} : \text{pc} + 4$

  – All other instructions: $\text{nextPC} = \text{PC} + 4$
Control Hazards

What do we need to compute nextPC?
- JAL: \( \text{nextPC} = \text{pc} + \text{immJ} \)

- JALR: \( \text{nextPC} = \{ (\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0 \} \)

- Branches: \( \text{nextPC} = \text{brFun} (\text{reg}[\text{rs1}], \text{reg}[\text{rs2}])? \text{pc} + \text{immB} : \text{pc} + 4 \)

- All other instructions: \( \text{nextPC} = \text{PC} + 4 \)
- (Exceptions also change PC, we’ll deal with them later)
Control Hazards

• What do we need to compute nextPC?
  – JAL: \( \text{nextPC} = \text{pc} + \text{immJ} \)
  – JALR: \( \text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\} \)
  – Branches: \( \text{nextPC} = \text{brFun}((\text{reg}[\text{rs1}], \text{reg}[\text{rs2}]))? \text{pc} + \text{immB} : \text{pc} + 4 \)
  – All other instructions: \( \text{nextPC} = \text{PC} + 4 \)
  – (Exceptions also change PC, we’ll deal with them later)

• *What’s the earliest stage we can compute nextPC?*

<table>
<thead>
<tr>
<th>JAL</th>
<th>JALR</th>
<th>Branches</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What do we need 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
- (Exceptions also change PC, we’ll deal with them later)

What’s the earliest stage we can compute nextPC?

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

• What do we need 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
  – (Exceptions also change PC, we’ll deal with them later)

• What’s the earliest stage we can compute nextPC?

<table>
<thead>
<tr>
<th></th>
<th>JAL</th>
<th>JALR</th>
<th>Branches</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Hazards

• What do we need to compute nextPC?
  – JAL: $\text{nextPC} = \text{pc} + \text{immJ}$
  
  – JALR: $\text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$
  
  – Branches: $\text{nextPC} = \text{brFun}(\text{reg}[\text{rs1}], \text{reg}[\text{rs2}])? \text{pc} + \text{immB} : \text{pc} + 4$
  
  – All other instructions: $\text{nextPC} = \text{PC} + 4$
  
  – (Exceptions also change PC, we’ll deal with them later)

• What’s the earliest stage we can compute nextPC?

<table>
<thead>
<tr>
<th>JAL</th>
<th>JALR</th>
<th>Branches</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>EXE</td>
<td>EXE</td>
<td></td>
</tr>
</tbody>
</table>
Control Hazards

• What do we need to compute nextPC?
  – JAL: $\text{nextPC} = \text{pc} + \text{immJ}$
  – JALR: $\text{nextPC} = \{(\text{reg}[\text{rs1}] + \text{immI})[31:1], 1'b0\}$
  – Branches: $\text{nextPC} = \text{brFun(reg[rs1], reg[rs2])}\? \text{pc} + \text{immB} : \text{pc} + 4$
  – All other instructions: $\text{nextPC} = \text{PC} + 4$
  – (Exceptions also change PC, we’ll deal with them later)

• What’s the earliest stage we can compute nextPC?

<table>
<thead>
<tr>
<th></th>
<th>JAL</th>
<th>JALR</th>
<th>Branches</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>EXE</td>
<td>EXE</td>
<td></td>
<td>IF</td>
</tr>
</tbody>
</table>
Resolving Control Hazards with Speculation

- 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 \texttt{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: 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>

Start fetching at PC+4 (and) but **bne** not resolved yet...
Resolving Control Hazards with Speculation

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

2. Assume `bne` is not taken in example

```plaintext
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></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...

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
      ...

April 19, 2018
Resolving Control Hazards with Speculation

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

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

```
IF          addi  sub  bne  and  xor  addi  sub  bne  and
DEC         addi  sub  bne  and  NOP  addi  sub  bne
EXE          addi  sub  bne  NOP  NOP  addi  sub
MEM          addi  sub  bne  NOP  NOP  addi
WB           addi  sub  bne  NOP  NOP
```

```
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><strong>IF</strong></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><strong>DEC</strong></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>and</td>
<td>NOPO</td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td></td>
</tr>
<tr>
<td><strong>EXE</strong></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOPO</td>
<td>NOPO</td>
<td>addi</td>
<td>sub</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOPO</td>
<td>NOPO</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOPO</td>
<td>NOPO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

April 19, 2018
Resolving Control Hazards with Speculation

- What’s a good guess for nextPC? PC+4
  - Assume bne is taken in example
  - Start fetching at PC+4 but bne not resolved yet …
  - Guesses wrong, annul and & xor and restart fetching at 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>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>

April 19, 2018 L17-27
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>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></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 19, 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>addi</td>
<td>sub</td>
<td>bne</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</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
...

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

April 19, 2018
Thank you!

Next lecture: Implementing pipelined processors