Introduction to Pipelining

Silvina Hanono Wachman
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)
Performance of Combinational Logic

For combinational logic:
- latency = \( t_{PD} \)
- throughput = \( 1/t_{PD} \)
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?
Performance of Combinational Logic

For combinational logic:

\[
\text{latency} = t_{PD}
\]

\[
\text{throughput} = \frac{1}{t_{PD}}
\]

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

\[
\begin{align*}
X & \quad F(X) \\
F(X) & \quad G(X) \\
G(X) & \quad P(X)
\end{align*}
\]
Performance of Combinational Logic

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
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 $j+2$. 

Diagram:

- F
- G
- H
- P(X)
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 ($t_{PD} = 0, t_{SETUP} = 0$):

<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>
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 ($t_{PD} = 0$, $t_{SETUP} = 0$):

<table>
<thead>
<tr>
<th>Unpipelined</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>1/45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2-stage pipeline</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</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 ($t_{PD} = 0$, $t_{SETUP} = 0$):

<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><strong>worse</strong></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 ($t_{PD} = 0$, $t_{SETUP} = 0$):

<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>
<tr>
<td>worse</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 ($t_{PD} = 0$, $t_{SETUP} = 0$):

<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

\[
\begin{array}{cccc}
i & i+1 & i+2 & i+3 \\
\hline
F & G \\
H \\
\end{array}
\]
### Pipeline Diagrams

Clock cycle: $i \quad i+1 \quad i+2 \quad i+3$

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th></th>
<th></th>
<th></th>
<th></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>
<tr>
<td>$X_i$ stable here</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing pipeline stages $F$, $G$, and $H$ with input $X$ and output $P(X)$.
Pipeline Diagrams

Clock cycle

\[ i \quad i+1 \quad i+2 \quad i+3 \]

**Pipeline stages**

<table>
<thead>
<tr>
<th>F &amp; G</th>
<th>F((X_i))</th>
<th>G((X_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(X_i\) **stable here**
# Pipeline Diagrams

<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(Xᵢ)</td>
<td>F(Xᵢ₊₁)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G(Xᵢ)</td>
<td>G(Xᵢ₊₁)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>H</strong></td>
<td></td>
<td></td>
<td>H(Xᵢ)</td>
<td></td>
</tr>
</tbody>
</table>

Clock cycle

Pipeline stages

Xi

stable here
Pipeline Diagrams

Clock cycle

\[
\begin{array}{cccc}
  i & i+1 & i+2 & i+3 \\
  F(X_i) & F(X_{i+1}) & G(X_{i+1}) & \\
  H(X_i) & \\
\end{array}
\]

Pipeline stages:
- **F & G**
  - F(X_i)
  - G(X_i)
  - F(X_{i+1})
  - G(X_{i+1})

- **H**
  - H(X_i)

Xi stable here
P(Xi) available here

October 30, 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.
Pipeline Diagrams

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.
Pipeline Conventions

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.
Pipeline Conventions

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).
Pipeline Conventions

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).

Clock period:
The clock must have a period $t_{CLK}$ sufficient to cover the longest register to register propagation delay plus setup time.
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 $t_{CLK}$ sufficient to cover the longest register to register propagation delay plus setup time.

K-pipeline latency $L = K \times t_{CLK}$

K-pipeline throughput $T = \frac{1}{t_{CLK}}$
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?
For what value of $K$ is the following circuit a $K$-Pipeline?

Consider a BAD job of pipelining:
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 in a well-formed K pipeline!

For what value of K is the following circuit a K-Pipeline? none
A Pipelining Methodology

A 4ns

B 3ns

C 8ns

D 4ns

E 2ns

F 5ns
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.
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.
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.

\[ t_{\text{CLK}} = 8\text{ns} \]
\[ T = 1/(8\text{ns}) \]
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.
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).

INPUTS

A
4ns

B
3ns

C
8ns

D
4ns

E
2ns

F
5ns

OUTPUTS

t_{CLK} = 8ns
T = 1/(8ns)
L = 24ns
### Pipeline Example

**OBSERVATIONS:**


![Diagram of pipeline](image)

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

<table>
<thead>
<tr>
<th>1-pipe:</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>2-pipe:</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>3-pipe:</th>
<th></th>
</tr>
</thead>
</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.
- T improved by breaking long combinational paths, allowing faster clock.

<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>
**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>pipeline</th>
<th>LATTENCY</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
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)

Clock? Area? Throughput?
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)

Clock: \(B \approx C < A\)

Area? \hspace{2cm} Throughput?
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)

Clock: $B \approx C < A$

Area: $C < A < B$

Throughput?
Pipelining Design Alternatives

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multicycle (C)

Clock: $B \approx C < A$

Area: $C < A < B$

Throughput: $C < A < B$
Pipelined Processors
"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:
  
  \[
  \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
  \]

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

- Options to reduce execution time:
  - Executed instructions ↓ (work/instruction ↑)
  - Cycles per instruction (CPI) ↓
  - Cycle time ↓ (frequency ↑)
Single-Cycle Processor Performance
Single-Cycle Processor Performance

- CPI = 1
Single-Cycle Processor Performance

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

\( t_{\text{CLK}} = \text{Longest path for any instruction} \)

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

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

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

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

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

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

$$t_{CLK} \approx t_{IMEM} + t_{DEC} + t_{RF} + t_{EXE} + t_{DMEM} + t_{WB} \quad \text{Slow!}$$
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{\text{CLK}}$
  - 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 $t_{CLK}$
  - 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
Pipelined Implementation

- Divide datapath in multiple pipeline stages to reduce $t_{\text{CLK}}$
  - 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 DEC Diagram]

  **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 $t_{CLK}$
  - 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 of pipeline stages]

  - **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 $t_{CLK}$
  - 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 $t_{CLK}$
  - 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 $t_{CLK}$
  - Each instruction executes over multiple cycles
  - Consecutive instructions are overlapped to keep CPI $\approx 1.0$
- We’ll study the classic 5-stage pipeline:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction Fetch stage: Maintains PC, fetches instruction and passes it to</td>
</tr>
<tr>
<td>DEC</td>
<td>Decode &amp; Read Registers stage: Decodes instruction and reads source operands from register file, passes them to</td>
</tr>
<tr>
<td>EXE</td>
<td>Execute stage: Performs indicated operation in ALU, passes result to</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory stage: If it’s a load, use input as the address, pass read data (or ALU result if not a load) to</td>
</tr>
<tr>
<td>WB</td>
<td>Write-Back stage: writes result back into register file.</td>
</tr>
</tbody>
</table>

$t_{CLK} = \max\{t_{IF}, t_{DEC}, t_{EXE}, t_{MEM}, t_{WB}\}$
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?
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 → 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

Today: 1 and begin 2
Next time: 2 and 3
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

- **IF**
  - PC
  - Instruction Memory
- **DEC**
  - Decode
  - Register File
- **EXE**
  - Execute
- **MEM**
  - Data Memory
- **WB**
  - Register File
Example: Pipelined Execution

```assembly
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>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></td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
<td>sub</td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
<td>lw</td>
</tr>
</tbody>
</table>

October 30, 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
```

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>

October 30, 2018

MIT 6.004 Fall 2018
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
```

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

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

October 30, 2018
Example: Pipelined Execution

When do register reads and writes happen?

Reads in DEC stage
Writes at end of WB stage

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

Read x12

October 30, 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

Cycles

<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>sub</td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
</tr>
</tbody>
</table>

Read x12
Write x11
Data Hazards

- Consider this instruction sequence:

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

```
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>sub</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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></td>
<td>addi</td>
<td>xor</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 → `x11` is stale!
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>

- xor reads x11 on cycle 3, but addi does not update it until end of cycle 5 → x11 is stale!
- Pipeline must maintain correct behavior...
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><em>sub</em></td>
<td><em>sub</em></td>
<td><em>sub</em></td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td><em>xor</em></td>
<td><em>xor</em></td>
<td><em>xor</em></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></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.

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>xor</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></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`
- **Stall:** `xor x13, x11, x12`
- **Stall:** `sub x17, x15, x16`
- **Stall:** `xori x19, x18, 0xF`

- **x11 updated**
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><em>sub</em></td>
<td><em>sub</em></td>
<td><em>sub</em></td>
<td>xor</td>
<td>xori</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>sub</td>
<td>xori</td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>sub</td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</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>

*Stalls increase CPI!*

Stalls increase CPI!
Take-Home Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module’s latency.

- What is the latency and throughput of your pipelined circuit?
Thank you!

Next lecture: Data and Control Hazards