Introduction to Pipelining

Reminders:
Quiz 1 tonight! 7:30-9:30PM
A-K in 32-123, L-Z in 26-100

Makeup on Monday, 3-5PM in 34-101
Accommodations on Monday, 3-6PM in 34-101
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 $= 1/t_{PD}$
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?
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:
 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.
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>Pipeline Type</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>
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>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 ($t_{PD} = 0$, $t_{SETUP} = 0$):

<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><strong>worse</strong></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></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

![Pipeline Diagram](image)

**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</td>
<td>G</td>
<td>H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F15</td>
<td>G20</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pipeline stages**

- F & G
- H
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

Xi

stable here

October 17, 2019

MIT 6.004 Fall 2019
# 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>Pipeline stages</td>
<td>F &amp; G</td>
<td><strong>F(Xᵢ)</strong></td>
<td><strong>G(Xᵢ)</strong></td>
<td>H</td>
</tr>
</tbody>
</table>

- F(15)
- G(20)
- H(25)
## Pipeline Diagrams

### Pipeline Stages

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

- F(X_i)
- F(X_{i+1})
- G(X_i)
- G(X_{i+1})
- H(X_i)

### Clock Cycle

<table>
<thead>
<tr>
<th>i</th>
<th>i+1</th>
<th>i+2</th>
<th>i+3</th>
</tr>
</thead>
</table>

- Xi

- Stable here
### Pipeline Diagrams

#### Pipeline Stages

<table>
<thead>
<tr>
<th>F &amp; G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(X&lt;sub&gt;i&lt;/sub&gt;)</td>
<td>H(X&lt;sub&gt;i&lt;/sub&gt;)</td>
</tr>
<tr>
<td>G(X&lt;sub&gt;i&lt;/sub&gt;)</td>
<td></td>
</tr>
<tr>
<td>G(X&lt;sub&gt;i+1&lt;/sub&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

#### 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>X&lt;sub&gt;i&lt;/sub&gt; stable here</td>
<td>F(X&lt;sub&gt;i&lt;/sub&gt;)</td>
<td>F(X&lt;sub&gt;i+1&lt;/sub&gt;)</td>
<td></td>
</tr>
<tr>
<td>P(X&lt;sub&gt;i&lt;/sub&gt;) available here</td>
<td>G(X&lt;sub&gt;i&lt;/sub&gt;)</td>
<td>G(X&lt;sub&gt;i+1&lt;/sub&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

#### Pipeline Diagram

- **F**: Input
- **G**: Intermediate stage
- **H**: Output

---

October 17, 2019
Pipeline Diagrams

The results associated with a particular set of input data moves \textit{diagonally} through the diagram, progressing through one pipeline stage each clock cycle.
The results associated with a particular set of input data moves \textit{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.
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 *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.
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 $T = 1 / t_{CLK}$
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?
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
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!
A Pipelining Methodology

![Pipelining Diagram]

- 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.

$t_{CLK} = 8\text{ns}$
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.

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

\[
\begin{align*}
\text{CLK} & = 8\text{ns} \\
T & = \frac{1}{8\text{ns}} \\
L & = 24\text{ns}
\end{align*}
\]
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).

\[ t_{CLK} = 8\text{ns} \]
\[ T = \frac{1}{(8\text{ns})} \]
\[ L = 24\text{ns} \]
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></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>
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>LATTENGY</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

![Diagram of pipeline example](image)

**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>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>
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.
Sample Pipelining Problem

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

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

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

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

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

\[ t_{CLK} = 4 \]
\[ T = 1/(4) \]
\[ L = 4 \times 4 = 16 \]
Design Tradeoffs
Introduction:
Multiplier Case Study
### Multiplication by repeated addition

<table>
<thead>
<tr>
<th>b Multiplicand</th>
<th>1101</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a Multiplier</td>
<td>* 1011</td>
<td>(11)</td>
</tr>
</tbody>
</table>

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier.

<table>
<thead>
<tr>
<th>tp</th>
<th>m0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>tp</td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>m0</td>
<td>01101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>m1</td>
<td>10011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>0000</td>
</tr>
<tr>
<td>m2</td>
<td>010011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>1101</td>
</tr>
<tr>
<td>m3</td>
<td>1000111</td>
<td>(143)</td>
</tr>
</tbody>
</table>
### Multiplication by repeated addition

| b Multiplicand | 1101 | (13) |
| a Multiplier * | 1011 | (11) |

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[
\text{mi} = (a[i]==0) ? 0 : b;
\]
Multiplication by repeated addition

b Multiplicand    1101   (13)
a Multiplier      * 1011   (11)

\[
\begin{array}{c}
\text{tp} \\
m0 \\
\text{tp} \\
m1 \\
\text{tp} \\
m2 \\
\text{tp} \\
m3 \\
\text{tp}
\end{array}
\begin{array}{c}
0000 \\
+ 1101 \\
01101 \\
+ 1101 \\
100111 \\
+ 0000 \\
0100111 \\
+ 1101 \\
10001111 (143)
\end{array}
\]

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[
m_i = (a[i]==0)? 0 : b;
\]
Multiplication by repeated addition

b Multiplicand 1101 (13)
a Multiplier * 1011 (11)

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[ m_i = (a[i]==0)? 0 : b; \]

\[
\begin{array}{c}
\text{tp} \\
m_0 \\
\text{tp} \\
m_1 \\
\text{tp} \\
m_2 \\
\text{tp} \\
m_3 \\
\text{tp}
\end{array}
\begin{array}{c}
0000 \\
+ 1101 \\
\hline
01101 \\
+ 1101 \\
\hline
100111 \\
+ 0000 \\
\hline
0100111 \\
+ 1101 \\
\hline
10001111 (143)
\end{array}
Multiplication by repeated addition

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

$mi = (a[i]==0)? 0 : b;$
Multiplication by repeated addition

\[ b \text{ Multiplicand } \times 1011 \text{ (11)} \]

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[ m_i = (a[i]==0)? 0 : b; \]
Multiplication by repeated addition

b Multiplier: 1101 (13)
a Multiplier: * 1011 (11)

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

mi = (a[i]==0)? 0 : b;

We also shift the result by one position at every step

m0
+ 1101

m1
+ 1101

m2
+ 0000

m3
+ 1101

10001111 (143)
Multiplication by repeated addition

b Multiplcand  1101  (13)
a Multiplier  *  1011  (11)

\[
\begin{array}{c}
<table>
<thead>
<tr>
<th>t_p</th>
<th>m_0</th>
<th>m_1</th>
<th>m_2</th>
<th>m_3</th>
<th>t_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1101</td>
<td>01101</td>
<td>1101</td>
<td>00000</td>
<td>100111</td>
</tr>
</tbody>
</table>
\end{array}
\]

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[ m_i = (a[i]==0)? 0 : b; \]

We also shift the result by one position at every step

Notice, the first addition is unnecessary because it simply yields \( m_0 \)
Multiplication by repeated addition

b Multiplicand   1101   (13)
a Multiplier      * 1011   (11)

\[
\begin{array}{rcl}
tp & & 0000 \\
m0 & + & 1101 \\
tp & & 01101 \\
m1 & + & 1101 \\
tp & & 100111 \\
m2 & + & 0000 \\
tp & & 0100111 \\
m3 & + & 1101 \\
tp & & 10001111   (143)
\end{array}
\]

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[mi = (a[i]==0)? 0 : b;\]

We also shift the result by one position at every step

Notice, the first addition is unnecessary because it simply yields m0

Also note that these are unsigned binary numbers.
Multiplication by repeated addition circuit

b Multiplicand  1101  (13)
a Multiplier  *  1011  (11)

\[
\begin{array}{c}
| \text{tp} | \text{m0} | \text{m1} | \text{m2} | \text{m3} | \text{tp} |\\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+</td>
<td>1101</td>
<td>+</td>
<td>1101</td>
<td>01101</td>
</tr>
<tr>
<td>01101</td>
<td>+</td>
<td>1101</td>
<td>+</td>
<td>0000</td>
<td>100111</td>
</tr>
<tr>
<td>100111</td>
<td>+</td>
<td>1101</td>
<td>+</td>
<td>0100111</td>
<td>0100111</td>
</tr>
<tr>
<td>1000111</td>
<td>+</td>
<td>1101</td>
<td>+</td>
<td>0100111</td>
<td>1000111</td>
</tr>
</tbody>
</table>
\end{array}
\]

\( m_i = (a[i]==0)? 0 : b; \)
Multiplication by repeated addition circuit

b Multiplicand  1101  (13)
a Multiplier    *  1011  (11)

\[
\begin{align*}
\text{tp} & \quad 0000 \\
m0 & + \quad 1101 \\
\text{tp} & \quad 01101 \\
m1 & + \quad 1101 \\
\text{tp} & \quad 100111 \\
m2 & + \quad 0000 \\
\text{tp} & \quad 0100111 \\
m3 & + \quad 1101 \\
\text{tp} & \quad 10001111  \quad (143)
\end{align*}
\]

\[m_i = (a[i]==0)? 0 : b;\]
Multiplication by repeated addition circuit

\[ b \text{ Multiplicand } \begin{array}{c} 1101 \\ \end{array} \]  \quad \begin{array}{c} (13) \\ \end{array} \\
\[ a \text{ Multiplier } \begin{array}{c} \times \end{array} \begin{array}{c} 1011 \\ \end{array} \]  \quad \begin{array}{c} (11) \\ \end{array} \\
\[ \begin{array}{c} 0000 \\ + \end{array} \begin{array}{c} 1101 \\ \end{array} \]  \quad \begin{array}{c} tp \\ m0 \\ \end{array} \\
\[ \begin{array}{c} 01101 \\ + \end{array} \begin{array}{c} 1101 \\ \end{array} \]  \quad \begin{array}{c} tp \\ m1 \\ \end{array} \\
\[ \begin{array}{c} 100111 \\ + \end{array} \begin{array}{c} 0000 \\ \end{array} \]  \quad \begin{array}{c} tp \\ m2 \\ \end{array} \\
\[ \begin{array}{c} 0100111 \\ + \end{array} \begin{array}{c} 1101 \\ \end{array} \]  \quad \begin{array}{c} tp \\ m3 \\ \end{array} \\
\[ \begin{array}{c} 10001111 \end{array} \quad (143) \]

\[ m_i = (a[i]==0)? 0 : b; \]
Multiplication by repeated addition circuit

\[ \text{b \ Multiplicand} \quad 1101 \quad (13) \]
\[ \text{a \ Multiplier} \quad * \quad 1011 \quad (11) \]

\[
\begin{array}{c}
\text{tp} \\
\text{m0} \\
\text{tp} \\
\text{m1} \\
\text{tp} \\
\text{m2} \\
\text{tp} \\
\text{m3} \\
\text{tp}
\end{array}
\begin{array}{c}
0000 \\
+ \quad 1101 \\
01101 \\
+ \quad 1101 \\
100111 \\
+ \quad 0000 \\
0100111 \\
+ \quad 1101 \\
10001111 \quad (143)
\end{array}
\]

\[ m_i = (a[i]==0)? 0 : b; \]
Multiplication by repeated addition circuit

b Multiplicand  1101  (13)
a Multiplier  *  1011  (11)

\[
\begin{align*}
\text{tp} & \quad 0000 \\
m0 & \quad + \ 1101 \\
\text{tp} & \quad 01101 \\
m1 & \quad + \ 1101 \\
\text{tp} & \quad 100111 \\
m2 & \quad + \ 0000 \\
\text{tp} & \quad 0100111 \\
m3 & \quad + \ 1101 \\
\text{tp} & \quad 10001111 \quad (143)
\end{align*}
\]

\[mi = (a[i]==0)? 0 : b;\]
Multiplication by repeated addition circuit

**b** Multiplicand  \(1101\)  (13)  
**a** Multiplier  \(*\ 1011\)  (11)  

\[
\begin{align*}
&\text{tp} & & 0000 \\
&m0 & + & 1101 \\
\hline
&\text{tp} & & 0101 \\
&m1 & + & 1101 \\
\hline
&\text{tp} & & 100111 \\
&m2 & + & 0000 \\
\hline
&\text{tp} & & 0100111 \\
&m3 & + & 1101 \\
\hline
&\text{tp} & & 10001111  \quad (143)
\end{align*}
\]

\[
\text{mi} = (a[i]==0)? 0 : b;
\]
Multiplication by repeated addition circuit

b Multiplicand 1101 (13)
a Multiplier * 1011 (11)

\[
\begin{array}{c}
| \text{tp} & \text{m0} & 0000 \\hline
| \text{tp} & \text{m1} & 01101 \hline
| \text{tp} & \text{m2} & 100111 \hline
| \text{tp} & \text{m3} & 0100111 \hline
| \text{tp} & 10001111 (143) \hline
\end{array}
\]

\[m_i = (a[i]==0)? 0 : b;\]
Multiplication by repeated addition circuit

b Multiplicand \(1101\) (13)
a Multiplier \(* 1011\) (11)

\[
\begin{array}{c}
\text{tp} \\
m0 \\
\text{tp} \\
m1 \\
\text{tp} \\
m2 \\
\text{tp} \\
m3 \\
\text{tp}
\end{array}
\begin{array}{c}
0000 \\
+ 1101 \\
\hline
01101 \\
+ 1101 \\
\hline
100111 \\
+ 0000 \\
\hline
0100111 \\
+ 1101 \\
\hline
10001111 (143)
\end{array}
\]

\[m_i = (a[i] == 0)? 0 : b;\]
Implementation of $mi$

\[
\begin{array}{cccc}
B_3 & B_2 & B_1 & B_0 \\
\times & A_3 & A_2 & A_1 & A_0 \\
\hline
B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 \\
B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 \\
B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 \\
\hline
\end{array}
\]
Implementation of mi

\[
\text{mi} = (a[i]==0)\,?\,0 : b;
\]
Implementation of $m_i$

The “Binary” Multiplication Table

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$m_i = (a[i]==0)? 0 : b;$

\[
\begin{array}{cccc}
\times & A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
\hline
B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 \\
B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 \\
+ & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 \\
\end{array}
\]

$m0$ $m1$ $m2$ $m3$
# Implementation of \( m_i \)

The "Binary" Multiplication Table

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ m_i = (a[i]==0)? 0 : b; \]

\[
\begin{array}{cccc}
B_3 & B_2 & B_1 & B_0 \\
\times & A_3 & A_2 & A_1 & A_0 \\
\hline
B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 \\
B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 \\
+ B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 \\
\end{array}
\]

\[ m_0 \]
\[ m_1 \]
\[ m_2 \]
\[ m_3 \]
Implementation of $m_i$

The “Binary” Multiplication Table

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

= AND

$B_3A_0$ $B_2A_0$ $B_1A_0$ $B_0A_0$

$m0$

$B_3A_1$ $B_2A_1$ $B_1A_1$ $B_0A_1$

$m1$

$B_3A_2$ $B_2A_2$ $B_1A_2$ $B_0A_2$

$m2$

$+$ $B_3A_3$ $B_2A_3$ $B_1A_3$ $B_0A_3$

$m3$

$mi = (a[i]==0)? 0 : b;$
Implementation of $m_i$

The “Binary” Multiplication Table

$$
\begin{array}{c|cc}
* & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
= \text{AND}
$$

$BA_i$ called a “partial product”

$$
\begin{array}{cccc}
B_3 & B_2 & B_1 & B_0 \\
\times & A_3 & A_2 & A_1 & A_0 \\
\hline
B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 \\
B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 \\
+ & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 \\
\hline
m_0 & m_1 & m_2 & m_3
\end{array}
$$

$m_i = (a[i]==0)? 0 : b;$

Multiplying N-digit number by M-digit number gives (N+M)-digit result
Implementation of $m_i$

The “Binary” Multiplication Table

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$B_{A_i}$ called a “partial product”

$$m_i = (a[i]==0)? 0 : b;$$

$$\begin{array}{cccc}
B_3 & B_2 & B_1 & B_0 \\
A_3 & A_2 & A_1 & A_0 \\
\times & & & \\
\hline
B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 \\
B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 \\
+ & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 \\
\hline
m_0 & m_1 & m_2 & m_3
\end{array}$$

Multiplyng $N$-digit number by $M$-digit number gives $(N+M)$-digit result

Easy part: forming partial products (bunch of AND gates)
Hard part: adding $M$ $N$-bit partial products
Combinational Multiplier Redrawn
Combinational Multiplier Redrawn

\[ s = a \oplus b \oplus c_{in} \]

\[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]
Combinational Multiplier Redrawn

\[ s = a \oplus b \oplus c_{in} \]
\[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]

\[ s = a \oplus b \]
\[ c_{out} = a \cdot b \]
Combinational Multiplier Redrawn

\[ s = a \oplus b \oplus c_{in} \]

\[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]

\[ s = a \oplus b \]

\[ c_{out} = a \cdot b \]

Latency = \( \Theta(N) \)
Combinational Multiplier Redrawn

\[ s = a \oplus b \oplus c_{in} \]

\[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]

\[ s = a \oplus b \]

\[ c_{out} = a \cdot b \]

Latency = \( \Theta(N) \)

Throughput = \( \Theta(1/N) \)
Combinational Multiplier Redrawn

\[ s = a \oplus b \oplus c_{in} \]

\[ c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in} \]

\[ s = a \oplus b \]

\[ c_{out} = a \cdot b \]

Latency = \( \Theta(N) \)

Throughput = \( \Theta(1/N) \)

Area = \( \Theta(N^2) \)
Increase Throughput with Pipelining - First Attempt
Increase Throughput with Pipelining - First Attempt

L12-17
Increase Throughput with Pipelining - First Attempt

[Diagram of a pipeline with binary inputs and outputs, showing the flow of data through the pipeline.]
Increase Throughput with Pipelining - First Attempt
Increase Throughput with Pipelining - First Attempt

$\mathbf{t_{CLK}} = \Theta(N)$
Increase Throughput with Pipelining - First Attempt

\[ t_{CLK} = \Theta(N) \]

\# stages = \Theta(N)
Increase Throughput with Pipelining - First Attempt

$t_{CLK} = \Theta(N)$

$\# \text{ stages} = \Theta(N)$

Latency = $\Theta(N^2)$
Increase Throughput with Pipelining - First Attempt

$t_{CLK} = \Theta(N)$

# stages = $\Theta(N)$

Latency = $\Theta(N^2)$

Throughput = $\Theta(1/N)$
Increase Throughput with Pipelining - First Attempt

Need to break carry chain

$t_{CLK} = \Theta(N)$

$\# \text{ stages} = \Theta(N)$

Latency = $\Theta(N^2)$

Throughput = $\Theta(1/N)$
Increase Throughput with Pipelining
Increase Throughput with Pipelining

![Diagram of pipelined carry-lookahead adder]

- **a** (input carry)
- **b** (sum bits)
- **z** (output)

- **FA** (full adder)
- **HA** (half adder)
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining
Increase Throughput with Pipelining

\[ t_{\text{CLK}} = \Theta(1) \]
Increase Throughput with Pipelining

$t_{\text{CLK}} = \Theta(1)$

# stages = $\Theta(N)$
Increase Throughput with Pipelining

\[ t_{CLK} = \Theta(1) \]

\# stages = \Theta(N)

Latency = \Theta(N)
Increase Throughput with Pipelining

$t_{CLK} = \Theta(1)$

# stages = $\Theta(N)$

Latency = $\Theta(N)$

Throughput = $\Theta(1)$
Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times.
Folded Multiplier

**Reduce Area With Sequential Logic**
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. *Tradeoff increased latency for reduced area.*
Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times.  **Tradeoff increased latency for reduced area.**

![Diagram of the folded multiplier](image)

**Init:** $P \leftarrow 0$, load A&B

**Repeat** M times {
  \[
P \leftarrow P + (A_{\text{LSB}} == 1 \ ? \ B : 0)\]
  shift $S_N, P, A$ right one bit
}

**Done:** $(N+M)$-bit result in $P, A$
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand \( B \) has \( N \) bits and the multiplier \( A \) has \( M \) bits. If we only want to invest in a single \( N \)-bit adder, we can process adds sequentially using the same adder \( M \) times. Tradeoff increased latency for reduced area.

Init: \( P \leftarrow 0 \), load \( A \& B \)

Repeat \( M \) times \{ 
\[
P \leftarrow P + (A_{\text{LSB}} == 1 \ ? \ B : 0) \\
\text{shift } S_N,P,A \text{ right one bit}
\}

Done: \((N+M)\)-bit result in \( P,A \)

Using Ripple Carry Adder
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. **Tradeoff increased latency for reduced area.**

Using Ripple Carry Adder

\[ t_{CLK} = \Theta(N) \]

Init: \( P \leftarrow 0 \), load A&B

Repeat M times {
\begin{align*}
P &\leftarrow P + (A_{\text{LSB}} == 1 \ ? \ B \ : \ 0) \\
\text{shift } S_N, P, A \text{ right one bit}
\end{align*}
}

Done: (N+M)-bit result in P,A
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

Init: P ← 0, load A&B
Repeat M times {
  P ← P + (A_{LSB} == 1 ? B : 0)
  shift S_N,P,A right one bit
}
Done: (N+M)-bit result in P,A

Using Ripple Carry Adder
\[ t_{CLK} = \Theta(N) \]
\[ \# \text{ stages} = \Theta(N) \]
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

Init: P←0, load A&B
Repeat M times {
P ← P + (A_{LSB}==1 ? B : 0)
shift S_{N}, P, A right one bit
}
Done: (N+M)-bit result in P,A

Using Ripple Carry Adder
\[ t_{CLK} = \Theta(N) \]
\[ \text{# stages} = \Theta(N) \]
Latency = \Theta(N^2)
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

Init: P ← 0, load A&B
Repeat M times {
    P ← P + (A_{LSB}==1 ? B : 0)
    shift S_N, P, A right one bit
}
Done: (N+M)-bit result in P,A

Using Ripple Carry Adder
\[ t_{CLK} = \Theta(N) \]
\[ \# \text{ stages} = \Theta(N) \]
\[ \text{Latency} = \Theta(N^2) \]
\[ \text{Throughput} = \Theta(1/N^2) \]
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

Init: \( P \leftarrow 0 \), load A&B

Repeat M times {
    \[ P \leftarrow P + (A_{LSB}==1 \ ? \ B \ : \ 0) \]
    shift \( S_N, P, A \) right one bit
}

Done: \((N+M)\)-bit result in \( P, A \)

Using Ripple Carry Adder
\[ t_{CLK} = \Theta(N) \]
\[ \# \text{ stages} = \Theta(N) \]
\[ \text{Latency} = \Theta(N^2) \]
\[ \text{Throughput} = \Theta(1/N^2) \]
\[ \text{Area} = \Theta(N) \]
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times.

\[
\text{Init: } P \leftarrow 0, \text{ load } A\&B
\]

\[
\text{Repeat M times} \{
\quad \text{P} \leftarrow \text{P} + (A_{\text{LSB}} == 1 ? B : 0)
\quad \text{shift } S_N, P, A \text{ right one bit}
\}\n\]

\[
\text{Done: } (N+M)\text{-bit result in } P, A
\]

Using Ripple Carry Adder
- \( t_{\text{CLK}} = \Theta(N) \)
- \# stages = \( \Theta(N) \)
- Latency = \( \Theta(N^2) \)
- Throughput = \( \Theta(1/N^2) \)
- Area = \( \Theta(N) \)

Using Diagonal Partial Products
- \( t_{\text{CLK}} = \Theta(1) \)
Reduce Area With Sequential Logic

Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

```
Init: P ← 0, load A&B
Repeat M times {
    P ← P + (A_{LSB}==1 ? B : 0)
    shift S_{N}, P, A right one bit
}
Done: (N+M)-bit result in P,A
```

Using Ripple Carry Adder

\[ t_{CLK} = \Theta(N) \]
\[ # \text{ stages} = \Theta(N) \]
\[ \text{Latency} = \Theta(N^2) \]
\[ \text{Throughput} = \Theta(1/N^2) \]
\[ \text{Area} = \Theta(N) \]

Using Diagonal Partial Products

\[ t_{CLK} = \Theta(1) \]
\[ # \text{ stages} = \Theta(N) \]
Reduce Area With Sequential Logic

Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. **Tradeoff increased latency for reduced area.**

```
Init: P ← 0, load A&B
Repeat M times {
  P ← P + (A_{LSB} == 1 ? B : 0)
  shift S_N, P, A right one bit
}
Done: (N+M)-bit result in P, A
```

Using Ripple Carry Adder

$t_{CLK} = \Theta(N)$

# stages = $\Theta(N)$

Latency = $\Theta(N^2)$

Throughput = $\Theta(1/N^2)$

Area = $\Theta(N)$

Using Diagonal Partial Products

$t_{CLK} = \Theta(1)$

# stages = $\Theta(N)$

Latency = $\Theta(N)$
Folded Multiplier

Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. **Tradeoff increased latency for reduced area.**

Using Ripple Carry Adder
- \( t_{CLK} = \Theta(N) \)
- \# stages = \( \Theta(N) \)
- Latency = \( \Theta(N^2) \)
- Throughput = \( \Theta(1/N^2) \)
- Area = \( \Theta(N) \)

Using Diagonal Partial Products
- \( t_{CLK} = \Theta(1) \)
- \# stages = \( \Theta(N) \)
- Latency = \( \Theta(N) \)
- Throughput = \( \Theta(1/N) \)

Init: \( P \leftarrow 0, \) load A&B
Repeat M times {
  - \( P \leftarrow P + (A_{LSB}==1 ? B : 0) \)
  - shift \( S_N, P, A \) right one bit
}
Done: \( (N+M) \)-bit result in \( P,A \)
Reduce Area With Sequential Logic
Assume the multiplicand (B) has N bits and the multiplier (A) has M bits. If we only want to invest in a single N-bit adder, we can process adds sequentially using the same adder M times. Tradeoff increased latency for reduced area.

Init: P ← 0, load A&B
Repeat M times {
  P ← P + (A_{LSB}==1 ? B : 0)
  shift S_N, P, A right one bit
}
Done: (N+M)-bit result in P, A

Using Ripple Carry Adder
\[ t_{CLK} = \Theta(N) \]

Using Diagonal Partial Products
\[ t_{CLK} = \Theta(1) \]

Using Ripple Carry Adder
# stages = \Theta(N)
Latency = \Theta(N^2)
Throughput = \Theta(1/N^2)
Area = \Theta(N)

Using Diagonal Partial Products
# stages = \Theta(N)
Latency = \Theta(N)
Throughput = \Theta(1/N)
Area = \Theta(N)
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)

\[ \ldots f_1 \rightarrow f_2 \rightarrow f_3 \ldots \]

One module per pipeline stage (B)

\[ \ldots f_1 \rightarrow f_2 \rightarrow f_3 \ldots \]

Folded reuse a block, multicycle (C)

\[ \ldots f_1 \rightarrow \quad \rightarrow f_i \quad \rightarrow \quad \rightarrow \ldots \]
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 ≈ C < A

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: $C < A < B$

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: \( C < A < B \)
Good luck on the quiz!