Design Tradeoffs in Sequential Logic
Lecture Goals

- Finish discussion of pipelining and design tradeoffs in sequential logic

- Study how to generalize an FSM to solve multiple problems
  - First step towards building a general-purpose processor!
Software vs. Hardware Design
Timing is the key difference

1. Software interfaces (even instructions) are timing-independent
   - Specify *what* should happen, not *when*
1. Software interfaces (even instructions) are **timing-independent**
   - Specify *what* should happen, not *when*

```
while (b != 0) {
    a = a * b;
    b = b - 1;
}

loop: mv a1, s0
     call mul
     addi s0, s0, -1
     beqz s0, loop
```
Software vs. Hardware Design
Timing is the key difference

1. Software interfaces (even instructions) are timing-independent
   - Specify what should happen, not when

   ```
   while (b != 0) {
     a = a * b;
     b = b - 1;
   }
   ```

2. Hardware design is all about timing
   - Specify what happens on every clock cycle...
   - ...which itself determines the length of the clock cycle

   ```
   module Factorial;
   Reg#(Word) a(0);
   Reg#(Word) b(0);
   rule step;
     ... 
   endrule
   ```
Recap: Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps
Recap: Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

- Even when combinational circuits suffice, sequential circuits allow more design tradeoffs
Recap: Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

- Even when combinational circuits suffice, sequential circuits allow more design tradeoffs
  - Pipelined circuits improve throughput by increasing frequency and overlapping multiple computations
  - Folded circuits reduce area by reusing a small amount of combinational logic over multiple cycles
Reminder: Pipelined Circuits

- Pipelining breaks a combinational circuit over multiple stages using registers.

```
  F 15
  |    |
  |    |   G 20
  |    |   H 25
  |    |
X   P(X)
```
Reminder: Pipelined Circuits

- Pipelining breaks a combinational circuit over multiple stages using registers

![Diagram of pipelined circuit]
Reminder: Pipelined Circuits

- Pipelining breaks a combinational circuit over multiple stages using registers
  - Each computation takes multiple cycles
  - On each cycle, each stage processes a different value
  - $t_{\text{CLK}} \downarrow \rightarrow$ Throughput $\uparrow$
Reminder: Pipelined Circuits

- Pipelining breaks a combinational circuit over multiple stages using registers
  - Each computation takes multiple cycles
  - On each cycle, each stage processes a different value
  - $t_{CLK} \downarrow \rightarrow \text{Throughput} \uparrow$

- Pipeline diagrams

<table>
<thead>
<tr>
<th>Pipeline stages</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td>F &amp; G</td>
<td>F($X_i$)</td>
</tr>
<tr>
<td>G($X_i$)</td>
<td>G($X_{i+1}$)</td>
</tr>
<tr>
<td>H</td>
<td>H($X_i$)</td>
</tr>
</tbody>
</table>
Reminder: Pipelined Multiplier
Reminder: Pipelined Multiplier
Reminder: Pipelined Multiplier
Reminder: Pipelined Multiplier

Combinational Area = \Theta(N^2)
Reminder: Pipelined Multiplier

Combinational
Area = \Theta(N^2)
\text{t}_{PD} = \Theta(N)
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Reminder: Pipelined Multiplier

Combinational
- Area = $\Theta(N^2)$
- $t_{PD} = \Theta(N)$
- Latency = $\Theta(N)$
- Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

Combination

- \[ \text{Area} = \Theta(N^2) \]
- \[ t_{PD} = \Theta(N) \]
- \[ \text{Latency} = \Theta(N) \]
- \[ \text{Throughput} = \Theta(1/N) \]
Reminder: Pipelined Multiplier

Combinational

Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

Combinational

Area = \( \Theta(N^2) \)

\( t_{PD} = \Theta(N) \)

Latency = \( \Theta(N) \)

Throughput = \( \Theta(1/N) \)
Reminder: Pipelined Multiplier

Combinational

Area = \Theta(N^2)

\( t_{PD} = \Theta(N) \)

Latency = \Theta(N)

Throughput = \Theta(1/N)
Reminder: Pipelined Multiplier

Combinational

Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

Combinational

Area = \Theta(N^2)

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

Latency = \Theta(N)

Throughput = \Theta(1/N)
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Reminder: Pipelined Multiplier

\[ \text{Combinational} \]
- \text{Area} = \Theta(N^2)
- \text{Latency} = \Theta(N)
- \text{Throughput} = \Theta(1/N)

\[ \text{Pipelined} \]
- \text{Area} = \Theta(N^2)

October 22, 2019
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$

Pipelined
Area = $\Theta(N^2)$
$t_{CLK} = \Theta(1)$
Reminder: Pipelined Multiplier

Combinational
Area = $\Theta(N^2)$
$t_{PD} = \Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$

Pipelined
Area = $\Theta(N^2)$
$t_{CLK} = \Theta(1)$
#stages = $\Theta(N)$
Reminder: Pipelined Multiplier

**Combinational**
- Area = $\Theta(N^2)$
- $t_{PD} = \Theta(N)$
- Latency = $\Theta(N)$
- Throughput = $\Theta(1/N)$

**Pipelined**
- Area = $\Theta(N^2)$
- $t_{CLK} = \Theta(1)$
- #stages = $\Theta(N)$
- Latency = $\Theta(N)$
Reminder: Pipelined Multiplier

Combinational
- Area = $\Theta(N^2)$
- $t_{PD} = \Theta(N)$
- Latency = $\Theta(N)$
- Throughput = $\Theta(1/N)$

Pipelined
- Area = $\Theta(N^2)$
- $t_{CLK} = \Theta(1)$
- #stages = $\Theta(N)$
- Latency = $\Theta(N)$
- Throughput = $\Theta(1)$
Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking ~N cycles to perform the additions
Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking \( \sim N \) cycles to perform the additions

```
Init: P ← 0, load A&B
Repeat N times {
    P ← P + (A_{LSB}==1 ? B : 0)
    shift S_{N}, P, A right one bit
}
Done: 2N-bit result in P, A
```
Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking \( \sim N \) cycles to perform the additions

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

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

Done: 2N-bit result in P, A
```

Tradeoff: reduced area, but lower throughput
Summary: Design Alternatives

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

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

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

Folded: Reuse a single block, multicycle (C)

Clock?  Area?  Throughput?
Summary: Design Alternatives

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

Folded: Reuse a single block, multicycle (C)

Clock: $B \approx C < A$

Area?  Throughput?
Summary: Design Alternatives

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

Folded: Reuse a single block, multicycle (C)

Clock: $B \approx C < A$

Area: $C < A < B$

Throughput?
Summary: Design Alternatives

Several combinational blocks in one pipeline stage (A)

One block per pipeline stage (B)

Folded: Reuse a single block, multicycle (C)

Clock: $B \approx C < A$

Area: $C < A < B$

Throughput: $C < A < B$
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$
    lower latency & higher throughput
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD}$ → lower $t_{CLK}$ → lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \Rightarrow$ lower $t_{CLK} \Rightarrow$
    lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{\text{CLK}}$ depends only on our circuit
  - So lower $t_{\text{PD}} \rightarrow$ lower $t_{\text{CLK}} \rightarrow$ lower latency & higher throughput

- In practice, other constraints may set $t_{\text{CLK}}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{\text{CLK}}$, throughput and latency tradeoffs change
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$ lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{CLK}$, throughput and latency tradeoffs change
  - Example: 4-stage vs. 2-stage pipeline
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$ lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{CLK}$, throughput and latency tradeoffs change
  - Example: 4-stage vs. 2-stage pipeline
    - If $t_{CLK,4\text{stage}} = t_{CLK,2\text{stage}}/2$ ?
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$ lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{CLK}$, throughput and latency tradeoffs change
  - Example: 4-stage vs. 2-stage pipeline
    - If $t_{CLK,4stage} = t_{CLK,2stage}/2$? Throughput: 2x, Latency: 1x
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$
    - lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{CLK}$, throughput and latency tradeoffs change
  - Example: 4-stage vs. 2-stage pipeline
    - If $t_{CLK,4stage} = t_{CLK,2stage}/2$ ?  Throughput: 2x, Latency: 1x
    - If $t_{CLK,4stage} = t_{CLK,2stage}$ ?
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed $t_{CLK}$ depends only on our circuit
  - So lower $t_{PD} \rightarrow$ lower $t_{CLK} \rightarrow$
    lower latency & higher throughput

- In practice, other constraints may set $t_{CLK}$
  - Propagation delay of other circuits
  - Limits on power consumption

- When our own circuit is not limiting $t_{CLK}$, throughput and latency tradeoffs change
  - Example: 4-stage vs. 2-stage pipeline
    - If $t_{CLK,4stage} = t_{CLK,2stage}/2$ ? Throughput: 2x, Latency: 1x
    - If $t_{CLK,4stage} = t_{CLK,2stage}$ ? Throughput: 1x, Latency: 2x
Pipeline Extensions
Pipeline Extensions

Producer → $f_1$ → $f_2$ → $f_3$ → Consumer
Pipeline Extensions

- Producer may not have input every cycle
  → Valid bits
Pipeline Extensions

- Producer may not have input every cycle → Valid bits
- Consumer may not be able to accept output every cycle → Stall logic to freeze/pause the pipeline
Pipeline Extensions

- Producer may not have input every cycle → **Valid bits**
- Consumer may not be able to accept output every cycle → **Stall logic** to freeze/pause the pipeline
- With large pipelines, may need to decouple stages further → Use **queues** instead of registers
Pipelines with Valid Bits

- If the producer won’t give an input every cycle, tag each stage with a valid bit
  - In Minispec, use Maybe types
Pipelines with Valid Bits

- If the producer won’t give an input every cycle, tag each stage with a valid bit
  - In Minispec, use Maybe types
Pipelines with Valid Bits

- If the producer won’t give an input every cycle, tag each stage with a valid bit
  - In Minispec, use Maybe types
Pipelines with Valid Bits

- If the producer won’t give an input every cycle, tag each stage with a valid bit
  - In Minispec, use Maybe types

- Invalid inputs propagate through the pipeline, produce invalid outputs:

<table>
<thead>
<tr>
<th>Cycle</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>Stage 1</td>
<td>V1</td>
<td>V2</td>
<td>Inv</td>
<td>V3</td>
<td>Inv</td>
<td>Inv</td>
<td>V4</td>
<td>V5</td>
</tr>
<tr>
<td>Stage 2</td>
<td>V1</td>
<td>V2</td>
<td>Inv</td>
<td>V3</td>
<td>Inv</td>
<td>Inv</td>
<td>V4</td>
<td></td>
</tr>
<tr>
<td>Stage 3</td>
<td>V1</td>
<td>V2</td>
<td>Inv</td>
<td>V3</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
</tr>
</tbody>
</table>
Pipelines with Stall Logic

- If the consumer can’t accept an output every cycle, we need to freeze the pipeline (and the producer!)
Pipelines with Stall Logic

- If the consumer can’t accept an output every cycle, we need to freeze the pipeline (and the producer!)
- Solution: Stall signal + registers with enable circuit
  - If stall is True, all pipeline registers retain their values

![Diagram showing producer, f1, f2, f3, consumer with stall signal]
Pipelines with Stall Logic

- If the consumer can’t accept an output every cycle, we need to freeze the pipeline (and the producer!)
- Solution: Stall signal + registers with enable circuit
  - If stall is True, all pipeline registers retain their values

<table>
<thead>
<tr>
<th>Cycle</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>Stage 1</td>
<td>V1</td>
<td>V2</td>
<td>V3</td>
<td>V4</td>
<td>V4</td>
<td>V5</td>
<td>V5</td>
<td>V5</td>
</tr>
<tr>
<td>Stage 2</td>
<td>V1</td>
<td>V2</td>
<td>V3</td>
<td>V3</td>
<td>V4</td>
<td>V4</td>
<td>V4</td>
<td>V4</td>
</tr>
<tr>
<td>Stage 3</td>
<td>V1</td>
<td>V2</td>
<td>V2</td>
<td>V2</td>
<td>V3</td>
<td>V3</td>
<td>V3</td>
<td>V3</td>
</tr>
<tr>
<td>Stall</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:
Combining Valid Bits + Stall Logic

- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:

<table>
<thead>
<tr>
<th>Cycle</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>Stage 1</td>
<td>V1</td>
<td>V2</td>
<td>Inv</td>
<td>V3</td>
<td>V4</td>
<td>Inv</td>
<td>V5</td>
<td>V5</td>
</tr>
<tr>
<td>Stage 2</td>
<td>Inv</td>
<td>V1</td>
<td>V2</td>
<td>Inv</td>
<td>V3</td>
<td>V4</td>
<td>V4</td>
<td>V4</td>
</tr>
<tr>
<td>Stage 3</td>
<td>Inv</td>
<td>Inv</td>
<td>V1</td>
<td>V2</td>
<td>V2</td>
<td>V3</td>
<td>V3</td>
<td>V3</td>
</tr>
<tr>
<td>Output</td>
<td>Inv</td>
<td>Inv</td>
<td>Inv</td>
<td>V1</td>
<td>V1</td>
<td>V2</td>
<td>V2</td>
<td>V2</td>
</tr>
<tr>
<td>Stall</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Stalling Delay in Large Pipelines

- We can’t stall large pipelines immediately
  - Stall signal drives a huge number of register enables → excessive fan-out causes delay
  - Stall delay eventually sets $t_{PD}$, and limits $t_{CLK}$!
Stalling Delay in Large Pipelines

- We can’t stall large pipelines immediately
  - Stall signal drives a huge number of register enables → excessive fan-out causes delay
  - Stall delay eventually sets $t_{PD}$, and limits $t_{CLK}$!

Solution: Use queues with >1 element instead of registers to separate pipeline stages
  - Stages don’t stall unless queue is full
  - Allows making stall decisions local
Example: 2-Element FIFO Queue
First-In, First-Out

- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
Example: 2-Element FIFO Queue
First-In, First-Out

- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
- Possible implementation:
Example: 2-Element FIFO Queue
First-In, First-Out

- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
- Possible implementation:

```
enqueue    first
isFull     dequeue
```

```

depqueue    sel0
enqueue      sel1

Invalid     e1
sel1        first

e0
sel0
```
Example: 2-Element FIFO Queue
First-In, First-Out

- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
- Possible implementation:

```
module FIFO2#(type T);
    Reg#(Maybe(T)) e0(Invalid);
    Reg#(Maybe(T)) e1(Invalid);
    method Maybe#(T) first = e0;
    method Bool isFull = isValid(e0) && isValid(e1);
    input Bool dequeue default = False;
    input Maybe#(T) enqueue default = Invalid;
    rule tick;
        if (!isValid(e0) && isValid(enqueue))
            e0 <= enqueue;
        else if (isValid(e0) && dequeue)
            e0 <= e1;
        if (isValid(e0) &&
            !isValid(e1) && isValid(enqueue))
            e1 <= enqueue;
        else if (isValid(e1) && dequeue)
            e1 <= Invalid;
    endrule
endmodule
```
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:

![Diagram of queue system with producer, f1, f2, f3, and consumer stages with stalls indicated.]

October 22, 2019
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:

  - A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:

  - A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full

![Diagram showing the decoupling process with queues](image_url)
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  
  - A stage stalls only if its output queue is full.
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:

- A stage stalls only if its output queue is full
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
  - Tradeoff: Can’t enqueue to full queue, even if an element is being dequeued on the same cycle
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
  - Tradeoff: Can’t enqueue to full queue, even if an element is being dequeued on the same cycle
    - We could build a queue that allowed this, but this would add a combinational path from dequeue to isFull (so we’d still have the problem of high stall $t_{PD}$!)
Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:
  - A stage stalls only if its output queue is full
  - Tradeoff: Can’t enqueue to full queue, even if an element is being dequeued on the same cycle
    - We could build a queue that allowed this, but this would add a combinational path from dequeue to isFull (so we’d still have the problem of high stall $t_{PD}$!)

- Queues also provide tolerance to variable latencies
  - Buffer multiple results without stalling producer when consumer takes variable number of cycles
From Special-Purpose FSMs to General-Purpose Processors
6.004 So Far

- Finite State Machines
- Sequential Elements
- Combinational Logic
- CMOS Gates
- Transistors
6.004 So Far

- What can you do with these?
  - Take a (solvable) problem
  - Design a procedure (recipe) to solve the problem
  - Design a finite state machine that implements the procedure and solves the problem
What can you do with these?
- Take a (solvable) problem
- Design a procedure (recipe) to solve the problem
- Design a finite state machine that implements the procedure and solves the problem

What you’ll be able to do after this week:
- Design a machine that can solve any solvable problem, given enough time and memory (a general-purpose computer)
Example: Factorial FSM
Example: Factorial FSM

Let’s design a circuit to compute factorial(N)
Example: Factorial FSM

Let’s design a circuit to compute factorial(N)

Python:
```python
a = 1
b = N
while b != 0:
    a = a * b
    b = b - 1
```

C:
```c
int a = 1;
int b = N;
while (b != 0) {
    a = a * b;
    b = b - 1;
}
```
Example: Factorial FSM

Let’s design a circuit to compute factorial(N)

Python:
```python
a = 1
b = N
while b != 0:
    a = a * b
    b = b - 1
```

C:
```c
int a = 1;
int b = N;
while (b != 0) {
    a = a * b;
    b = b - 1;
}
```

High-level FSM:
- States (start, loop, done)
- Boolean transitions (b == 0, b != 0)
- Register assignments in states (e.g., a ← a * b)
- Describes cycle-by-cycle behavior
- Registers (a, b)
- States (start, loop, done)
- Boolean transitions (b == 0, b != 0)

October 22, 2019
L13-19
Datapath for Factorial

```
start

loop

b != 0

b == 0

b <= b - 1

done

b <= N

a <= 1

a <= a * b

b <= b

a <= a

b <= b - 1

b <= b
```
Datapath for Factorial

- Implement registers

```
\begin{align*}
\text{start} & \rightarrow \text{loop} & b \neq 0 & \rightarrow \text{done} \\
& \rightarrow \text{loop} & b = 0 & \rightarrow \text{done}
\end{align*}
```

```
\text{a} \leftarrow 1 & \text{a} \leftarrow \text{a} \times \text{b} & \text{a} \leftarrow \text{a} \\
\text{b} \leftarrow \text{N} & \text{b} \leftarrow \text{b} - 1 & \text{b} \leftarrow \text{b}
```

October 22, 2019
Datapath for Factorial

- Implement registers

```
start -> loop (b != 0) -> done

a <= 1  a <= a * b  a <= a
b <= N  b <= b - 1  b <= b
```
Datapath for Factorial

- Implement registers

```
start -> loop -> done

b != 0
b == 0

a <= 1  a <= a * b  a <= a
b <= N  b <= b - 1  b <= b
```

```
> a
32

> b
32
```
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

```
a <= 1
a <= a * b
a <= a
b <= N
b <= b - 1
b <= b
```

\[
\begin{align*}
\text{start} & \quad \text{loop} & \quad \text{done} \\
b \neq 0 & \quad b = 0
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

\[
\begin{align*}
\text{a} & \leftarrow 1 \\
\text{a} & \leftarrow \text{a} \times \text{b} \\
\text{a} & \leftarrow \text{a} \\
\text{b} & \leftarrow \text{N} \\
\text{b} & \leftarrow \text{b} - 1 \\
\text{b} & \leftarrow \text{b}
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

```
a <= 1  a <= a * b  a <= a
b <= N  b <= b - 1  b <= b
```
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

\[
\begin{align*}
& a \leftarrow 1 \\
& a \leftarrow a \times b \\
& a \leftarrow a \\
& b \leftarrow N \\
& b \leftarrow b - 1 \\
& b \leftarrow b
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

\[ a \leq 1 \]
\[ a \leq a \times b \]
\[ a \leq a \]
\[ b \leq N \]
\[ b \leq b - 1 \]
\[ b \leq b \]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

\[
\begin{align*}
    a &\leftarrow 1 \\
    a &\leftarrow a \times b \\
    a &\leftarrow a \\
    b &\leftarrow N \\
    b &\leftarrow b - 1 \\
    b &\leftarrow b
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

\[
\begin{align*}
a & \leq 1 & a & \leq a \times b & a & \leq a \\
b & \leq N & b & \leq b - 1 & b & \leq b
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment

```
a <= 1  a <= a * b  a <= a
b <= N  b <= b - 1  b <= b
```
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes

\[
\begin{align*}
\text{start} & \rightarrow \text{loop} & b \neq 0 \\
\text{loop} & \rightarrow \text{done} & b = 0 \\
\text{start} & \rightarrow \text{loop} & b = N \\
\text{done} & \rightarrow \text{start} & b = b - 1
\end{align*}
\]

a <= 1, a <= a * b, a <= a
b <= N, b <= b - 1, b <= b

\[
\begin{align*}
a & \leq 1 & a & \leq a \times b & a & \leq a \\
b & \leq N & b & \leq b - 1 & b & \leq b
\end{align*}
\]
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes

```
start → loop → done
```

```
\[
\begin{align*}
\text{start} & \quad \text{loop} & \quad \text{done} \\
\text{b} \neq 0 & \quad \text{b} = 0
\end{align*}
\]
```

```
a \leq 1 & \quad a \leq a \times b & \quad a \leq a \\
b \leq N & \quad b \leq b - 1 & \quad b \leq b
```

```
\begin{align*}
\text{wa} & \quad \text{SEL} \\
0 & \quad 1 & \quad 2
\end{align*}
```

```
\begin{align*}
N & \quad 32 \\
1 & \quad 32
\end{align*}
```

```
\begin{align*}
a & \quad b
\end{align*}
```

```
\begin{align*}
1 & \quad 32 \\
\times & \quad 32 \\
+ & \quad 32
\end{align*}
```

```
\begin{align*}
0 & \quad 1 & \quad 2
\end{align*}
```

```
\begin{align*}
w & \quad a
\end{align*}
```

```
\begin{align*}
b & \quad a \times b
\end{align*}
```

```
\begin{align*}
b - 1 & \quad b
\end{align*}
```

```
\begin{align*}
b & \quad a
\end{align*}
```

```
\begin{align*}
b & \quad a \times b
\end{align*}
```

```
\begin{align*}
b & \quad b - 1
\end{align*}
```

October 22, 2019

MIT 6.004 Fall 2019
Datapath for Factorial

- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes

October 22, 2019
Control FSM for Factorial

```
start -> loop -> done

b != 0
b == 0

a <= 1
a <= a * b
a <= a

b <= N
b <= b - 1
b <= b

wa_SEL 0 1 2
wb_SEL 0 1 2

a

b

*

+

N

1
```
Control FSM for Factorial

- Implement combinational logic for transition conditions

```
<table>
<thead>
<tr>
<th>Transition</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>b != 0</td>
</tr>
<tr>
<td>Loop</td>
<td>b == 0</td>
</tr>
<tr>
<td>Done</td>
<td></td>
</tr>
</tbody>
</table>
```

- Logic for transition conditions:
  - a <= 1
  - a <= a * b
  - a <= a
  - b <= N
  - b <= b - 1
  - b <= b

```
wa_SEL | wa_SEL = 0, 1, 2
wb_SEL | wb_SEL = 0, 1, 2

a

b

* =>

0

-1

+ =>

== =>

z
```
Control FSM for Factorial

- Implement combinational logic for transition conditions
- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition conditions
  - Outputs: Mux select signals
Control FSM for Factorial

- Implement combinational logic for transition conditions
- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition conditions
  - Outputs: Mux select signals

\[
\begin{align*}
a & \leq 1 \\
b & \leq N \\
a & \leq a \times b \\
b & \leq b - 1 \\
b & \neq 0 \\
b & = 0
\end{align*}
\]
Programming the Datapath

- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
  - Multiplication
  - Squaring
Programming the Datapath

- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
  - Multiplication
  - Squaring

- But very limited problems. Reasons:
  - Limited storage (only two registers!)
  - Limited set of operations, and inputs to those operations
  - Limited inputs to the control FSM
A Simple Programmable Datapath

- Each cycle, this datapath:
  - Reads two operands \((a, b)\) from 4 registers \((x1-x4)\)
  - Performs one operation of \(+, -, *, &\) on operands
  - Optionally writes result to a register
A Simple Programmable Datapath

Each cycle, this datapath:
- Reads two operands \((a, b)\) from 4 registers \((x1-x4)\)
- Performs one operation of +, -, *, & on operands
- Optionally writes result to a register

Control FSM:
A Control FSM for Factorial

- Assume initial register contents:
  - x1 value = 1
  - x2 value = N
  - x3 value = -1
  - x4 value = 0

- Control FSM:

```
loop mul
  asel = x1
  bsel = x2
  opsel = 2 (*)
  wen = 1
  wsel = x1

loop sub
  asel = x2
  bsel = x3
  opsel = 0 (+)
  wen = 1
  wsel = x2

loop beq
  asel = x2
  bsel = x3
  opsel = X
  wen = 0
  wsel = X

done
  eq == 0
  asel = X
  bsel = X
  opsel = X
  wen = 0
  wsel = X
```
A Control FSM for Factorial

- Assume initial register contents:
  - x1 value = 1
  - x2 value = N
  - x3 value = -1
  - x4 value = 0

- Control FSM:

```
loop mul
  asel = x1
  bsel = x2
  opsel = 2 (*)
  wen = 1
  wsel = x1
  x1 <= x1 * x2

loop sub
  asel = x2
  bsel = x3
  opsel = 0 (+)
  wen = 1
  wsel = x2
  x2 <= x2 + x3

loop beq
  asel = x2
  bsel = x3
  opsel = X
  wen = 0
  wsel = x2
  eq == 1

done
  asel = X
  bsel = X
  opsel = X
  wen = 0
  wsel = X
  eq == 0
  N! in x1
```
A Control FSM for Factorial

- Assume initial register contents:
  - x1 value = 1
  - x2 value = N
  - x3 value = -1
  - x4 value = 0

- Control FSM:
  
  **Loop**
  - asel = x1
  - bsel = x2
  - opsel = 2 (*)
  - wen = 1
  - wsel = x1
  - x1 <= x1 * x2
  - mul x1, x1, x2

  **Loop**
  - asel = x2
  - bsel = x3
  - opsel = 0 (+)
  - wen = 1
  - wsel = x2
  - x2 <= x2 + x3

  **Loop beq**
  - asel = x2
  - bsel = x3
  - opsel = X
  - wen = 0
  - wsel = X
  - eq == 0

  **Loop beq**
  - asel = x2
  - bsel = x3
  - opsel = X
  - wen = 0
  - wsel = X
  - eq == 1

  **Done**
  - asel = X
  - bsel = X
  - opsel = X
  - wen = 0
  - wsel = X
  - N! in x1
A Control FSM for Factorial

- Assume initial register contents:
  - x1 value = 1
  - x2 value = N
  - x3 value = -1
  - x4 value = 0

- Control FSM:

```
 Loop
 mul
 asel = x1
 bsel = x2
 opsel = 2 (*)
 wen = 1
 wsel = x1
 x1 <= x1 * x2
 mul x1, x1, x2

 Loop
 sub
 asel = x2
 bsel = x3
 opsel = 0 (+)
 wen = 1
 wsel = x2
 x2 <= x2 + x3
 add x2, x2, x3

 Loop
 beq
 asel = x2
 bsel = x3
 opsel = X
 wen = 0
 wsel = X
 eq == 0

 Done
 asel = X
 bsel = X
 opsel = X
 wen = 0
 wsel = X
 eq == 1
 N! in x1
```
A Control FSM for Factorial

- Assume initial register contents:
  - x1 value = 1
  - x2 value = N
  - x3 value = -1
  - x4 value = 0

- Control FSM:

```
x1 <= x1 * x2
x2 <= x2 + x3
mul x1, x1, x2
add x2, x2, x3
beq x2, x3, loopmul
```

```
x1 <= x1 * x2
x2 <= x2 + x3
```

```
N! in x1
```
A Control FSM for Factorial

- Assume initial register contents:
  - \( x_1 \) value = 1
  - \( x_2 \) value = \( N \)
  - \( x_3 \) value = \(-1\)
  - \( x_4 \) value = 0

- Control FSM:

```
x1 <= x1 * x2
x2 <= x2 + x3
N! in x1
j done
```
New Problem → New Control FSM

- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, ...
  - But nothing that requires more than four registers
New Problem → New Control FSM

- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, ...
  - But nothing that requires more than four registers

- By designing a control FSM, we are programming the datapath
New Problem → New Control FSM

- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, ...
  - But nothing that requires more than four registers

- By designing a control FSM, we are programming the datapath

- Early digital computers were programmed this way!
  - ENIAC (1943):
    - First general-purpose digital computer
    - Programmed by setting huge array of dials and switches
    - Reprogramming it took about 3 weeks
The von Neumann Model

- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
The von Neumann Model

-Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)

-Components:
The von Neumann Model

- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:

  - Main memory holds programs and their data
The von Neumann Model

- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:

  - **Main memory** holds programs and their data
  - **Central processing unit** accesses and processes memory values
The von Neumann Model

- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:

- **Main memory** holds programs and their data
- **Central processing unit** accesses and processes memory values
- **Input/output devices** to communicate with the outside world
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

```
<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>rs</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd &lt;= op(rs,rt)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Central Processing Unit

Main Memory

```
Instruction
Instruction
Instruction
Data
Data
Data
```

October 22, 2019

MIT 6.004 Fall 2019
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

![Diagram showing main memory with instructions and data, central processing unit, and an operation rd <= op(rs,rt) with memory address 0xba5eba11]
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

How does CPU distinguish between instructions and data?
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Internal storage

address

data

control

status

address

instructions

October 22, 2019
Anatomy of a von Neumann Computer

Diagram showing the anatomy of a von Neumann computer with blocks labeled Datapath, Control Unit, and Main Memory. Arrows indicate flow of data, address, control, and status signals between these components.
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Internal storage

address

data

control

status

address

instructions

dest

asel

bsel

fr

ALU

Cc’s
Anatomy of a von Neumann Computer
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

registers

operations

Internal storage

address

data

control

status

address

instructions

dest

asel

bsel

fn

ALU

Cc’s

October 22, 2019
Anatomy of a von Neumann Computer

- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath
Thank you!

Next lecture: Building a RISC-V processor