Design Tradeoffs in Sequential Circuits
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;
     ... 
     ```
Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design
  4. Power consumption
  5. Energy of executing a task
  6. ...
Lecture Outline

- Examine design tradeoffs in digital logic: throughput, latency, and area
  - Power & energy are important, but out of scope for 6.004

- Extend pipelined designs to integrate them with other circuits
  - Valid bits, stall logic, queues

- Study how to generalize an FSM to solve multiple problems
  - First step towards building a general-purpose processor!
Throughput, Latency, and Area 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
  - Each computation takes multiple cycles
  - On each cycle, each stage processes a different value
  - $t_{CLK} \downarrow \rightarrow$ Throughput $\uparrow$

- Pipeline diagrams

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>Pipeline stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$F(X_i)$</td>
</tr>
<tr>
<td>$i+1$</td>
<td>$F(X_{i+1})$</td>
</tr>
<tr>
<td>$i+2$</td>
<td>$F(X_{i+2})$</td>
</tr>
<tr>
<td>$i+3$</td>
<td>$F(X_{i+3})$</td>
</tr>
<tr>
<td>$i+1$</td>
<td>$G(X_i)$</td>
</tr>
<tr>
<td>$i+2$</td>
<td>$G(X_{i+1})$</td>
</tr>
<tr>
<td>$i+3$</td>
<td>$G(X_{i+2})$</td>
</tr>
<tr>
<td>$i+2$</td>
<td>$H(X_i)$</td>
</tr>
<tr>
<td>$i+3$</td>
<td>$H(X_{i+1})$</td>
</tr>
<tr>
<td>$i+4$</td>
<td>$H(X_{i+2})$</td>
</tr>
</tbody>
</table>

Diagram:

- Input $X$ flows through stages $F$, $G$, and $H$.
- Each stage processes a different value.
- Throughput increases as $t_{CLK}$ decreases.

Diagram showing pipeline stages and inputs:

- $F(15)$
- $G(20)$
- $H(25)$
- $P(X)$
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
- **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

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

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

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

- 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 $t_{CLK,4\text{stage}} = t_{CLK,2\text{stage}}/2$ ? Throughput: 2x, Latency: 1x
    - If $t_{CLK,4\text{stage}} = t_{CLK,2\text{stage}}$ ? Throughput: 1x, Latency: 2x
Increasing Throughput with Replication

- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel

- Processes two values each cycle
- Metrics vs a single pipeline: 
  - Clock? \( \text{Same} \)
  - Throughput? \( 2x \)
  - Area? \( 2x \)
Example: Pipeline or Replicate?

- Consider the following two multipliers
  - PipedMul
  - FoldedMul

  4-stage pipelined multiplier
  Throughput = 1/t_{\text{CLK}}

  Folded multiplier that takes 4 cycles per output
  Throughput = 1/(4t_{\text{CLK}})

  Similar t_{\text{CLK}} vs. PipedMul, lower area

- Can you design a circuit that uses FoldedMul to achieve the same throughput as PipedMul?

  Replicate FoldedMul 4 times
  Each FoldedMul produces an output and takes a new input every 4 cycles
  Throughput = 4*1/(4t_{\text{CLK}}) = 1/t_{\text{CLK}}
Pipeline Extensions
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

- 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></td>
<td></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!)
- 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:

### Table

<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 $t_{CLK}$!

- 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
- Possible implementation:

```plaintext
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;
        let next0 = dequeue? e1 : e0;
        let next1 = dequeue? Invalid : e1;

        if (isValid(enqueue)) begin
            if (!isValid(next0)) next0 = enqueue;
            else if (!isValid(next1)) next1 = enqueue;
        end
        e0 <= next0;
        e1 <= next1;
    endrule
endmodule
```

```
Invalid  sel_1
  |      |  e1  
  |      |  e0  
  |      | sel_0
  |      |  first
  dequeue
  enqueue

isFull  sel_1
  |      | sel_0

enquepe  first
```
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

- 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*).

<table>
<thead>
<tr>
<th>Finite State Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Elements</td>
</tr>
<tr>
<td>Combinational Logic</td>
</tr>
<tr>
<td>CMOS Gates</td>
</tr>
<tr>
<td>Transistors</td>
</tr>
</tbody>
</table>
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: \( b \neq 0 \) and \( b = 0 \)
- Register assignments in states:
  - start: \( a \leq 1 \), \( b \leq N \)
  - loop: \( a \leq a \times b \), \( b \leq b - 1 \)
  - done: \( a \leq a \), \( b \leq b \)

- Describes cycle-by-cycle behavior
- Registers: \((a, b)\)
- States: \((\text{start}, \text{loop}, \text{done})\)
- Boolean transitions: \((b=0, b\neq 0)\)
- Register assignments in states:
  - (e.g., \( a \leftarrow a \times b \))
Datapath for Factorial

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

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

\[
\begin{array}{ccc}
& a \leq 1 & a \leq a \times b & a \leq a \\
& b \leq N & b \leq b - 1 & b \leq b \\
\end{array}
\]
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

- 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
  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
  eq == 0
  asel = x2
  bsel = x3
  opsel = 0
  wen = 0
  wsel = x2
  x2 <= x2 + x3
  add x2, x2, x3
  beq x2, x3, loopmul

done
  eq == 1
  asel = X
  bsel = X
  opsel = X
  wen = 0
  wsel = X
  N! in x1
  j done
```
New Problem $\rightarrow$ 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

- Modern computers instead store programs in memory, coded as a sequence of instructions

more on next lecture...
Thank you!

Next lecture: Building a RISC-V processor