Design Tradeoffs in Sequential Circuits

Good luck on the quiz tonight!
Lecture Outline

- Examine design tradeoffs in digital logic: throughput, latency, and area
  - Power & energy are important, but out of scope for 6.191
  - Case study: Multiplier

- Study how to generalize an FSM to solve multiple problems
  - First step towards building a general-purpose processor!
Reminder: Multiplication by repeated addition

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

```
| tp | 0000 |
| m0 | + 1101 |
| tp | 01101 |
| m1 | + 1101 |
| tp | 100111 |
| m2 | + 0000 |
| tp | 0100111 |
| m3 | + 1101 |
| tp | 10001111 | (143)
```

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

Implementation: Cascade of N-1 N-bit adders
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>

\( BA_i \) called a “partial product”

\[
B_3 A_0 \quad B_2 A_0 \quad B_1 A_0 \quad B_0 A_0 \\
B_3 A_1 \quad B_2 A_1 \quad B_1 A_1 \quad B_0 A_1 \\
B_3 A_2 \quad B_2 A_2 \quad B_1 A_2 \quad B_0 A_2 \\
+ B_3 A_3 \quad B_2 A_3 \quad B_1 A_3 \quad B_0 A_3
\]

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

Multiplying \( N \)-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
Using ripple-carry adders

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

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

\[ s = a \oplus b \]

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

Latency = \( \Theta(N) \)
Throughput = \( \Theta(1/N) \)
Area = \( \Theta(N^2) \)
Pipelining to Increase Throughput

First Attempt

Need to break carry chain

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

# stages = $\Theta(N)$

Latency = $\Theta(N^2)$

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

\[ t_{CLK} = \Theta(1) \]
\[ \# \text{ stages} = \Theta(N) \]
\[ \text{Latency} = \Theta(N) \]
\[ \text{Throughput} = \Theta(1) \]
Folded (Multi-Cycle) Multiplier

- 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

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

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

\[
\text{Done: } 2N\text{-bit result in } P,A
\]

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

Several combinational modules in one pipeline stage (A)

One module per pipeline stage (B)

Folded reuse a block, multi-cycle (C)

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

Latency: \( A < B < C \)

Area: \( C < A < B \)

Throughput: \( C < A < B \)
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 decreasing clock period and overlapping multiple computations
  - Multi-cycle / folded circuits *reduce area* by reusing a small amount of combinational logic over multiple cycles
Clock Frequency Constraints

- To analyze latency and throughput, so far we’ve assumed \( t_{\text{CLK}} \) depends only on our circuit
  - So lower \( t_{\text{PD}} \) → lower \( t_{\text{CLK}} \) → 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
  - Example: 4-stage vs. 2-stage pipeline
    - If \( t_{\text{CLK},4\text{stage}} = t_{\text{CLK},2\text{stage}}/2 \)  \textbf{Throughput: 2x, Latency: 1x}
    - If \( t_{\text{CLK},4\text{stage}} = t_{\text{CLK},2\text{stage}} \)  \textbf{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? $\text{Same}$
  - Latency? $\text{Same}$
  - Throughput? $2x$
  - Area? $2x$
Example: Pipeline or Replicate?

- Consider the following two multipliers

  - PipedMul
    - 4-stage pipelined multiplier
    - Throughput = \( \frac{1}{t_{CLK}} \)
  - FoldedMul
    - Folded multiplier that takes 4 cycles per output
    - Throughput = \( \frac{1}{4t_{CLK}} \)
    - Similar \( t_{CLK} \) 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 = \( 4 \times \frac{1}{4t_{CLK}} = \frac{1}{t_{CLK}} \)
From Special-Purpose FSMs to General-Purpose Processors
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 next week:
- Design a machine that can solve any solvable problem, given enough time and memory (a general-purpose computer)
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, b <= b - 1`)
- Describes cycle-by-cycle behavior
- Registers `(a, b)`
- States (start, loop, done)
- Boolean transitions (`b==0, b!=0`)
- Register assignments in states (e.g., `a <= a * 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

\[a \leq 1 \quad a \leq a \times b \quad a \leq a\]
\[b \leq N \quad b \leq b - 1 \quad b \leq b\]
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:

```
<table>
<thead>
<tr>
<th>State</th>
<th>asel</th>
<th>bsel</th>
<th>opsel</th>
<th>wen</th>
<th>wsel</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>x1</td>
<td>x2</td>
<td>2</td>
<td>1</td>
<td>x1</td>
</tr>
<tr>
<td>mul</td>
<td>x1</td>
<td>x1</td>
<td>x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>x2</td>
<td>x3</td>
<td>0</td>
<td>1</td>
<td>x2</td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>x2</td>
<td>x4</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j done</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Transition:
  - eq == 0
  - eq == 1

- Equations:
  - x1 <= x1 * x2
  - x2 <= x2 + x3
  - N! in x1

- Notes:
  - 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

- 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 next week...
Thank you!

Next lecture: Compilers and RISC-V assembly