Combinational Logic
and Introduction to Minispec

Reminders:
Quiz 1 review tonight 7:30-9:30pm
Quiz 1 on Thursday 7:30-9:30pm
Lecture Goals

- Learn how to design large combinational circuits through three useful examples:
  - Adder
  - Multiplexers
  - Shifter

- Learn how to implement combinational circuits in the Minispec hardware description language (HDL)
  - Design each combinational circuit as a function, which can be simulated or synthesized into gates
Building a Combinational Adder

- **Goal:** Build a circuit that takes two n-bit inputs \(a\) and \(b\) and produces \((n+1)\)-bit output \(s = a + b\)

- **Approach:** Implement the binary addition algorithm we have seen (called the standard algorithm)
The $i$th step of each addition:
- Takes three 1-bit inputs: $a_i$, $b_i$, $c_i$ (carry-in)
- Produces two 1-bit outputs: $s_i$, $c_{i+1}$ (carry-out)
- The 2-bit output $c_{i+1}s_i$ is the binary sum of the three inputs

Can you build a circuit that performs a single step with what you’ve learned so far?
Combinational Logic for an Adder

- First, build a full adder (FA), which
  - Adds three one-bit numbers: \(a\), \(b\), and carry-in \(c\)
  - Produces a sum bit and a carry-out bit

- Then, cascade FAs to perform binary addition

- Result: A ripple-carry adder (simple but slow)
# Deriving the Full Adder

## Truth table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c&lt;sub&gt;in&lt;/sub&gt;</th>
<th>c&lt;sub&gt;out&lt;/sub&gt;</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

## Boolean expressions

\[
\begin{align*}
\text{s} &= \quad \quad \quad \quad a \oplus \text{b} \\
\text{c}_{\text{out}} &= \quad \quad \quad \quad \text{a} \quad \text{b} \\
\end{align*}
\]
## Deriving the Full Adder

### Truth Table

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Boolean Expressions

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

\[ c_{\text{out}} = \]

---

October 5, 2021  
MIT 6.004 Fall 2021  
L08-7
Deriving the Full Adder

Truth table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>cout</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Boolean expressions

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

\[ c_{\text{out}} = a \cdot b + a \cdot c_{\text{in}} + b \cdot c_{\text{in}} \]
Describing a 32-bit Adder

- Truth table with \(2^{64}\) rows and 64 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - \(s_k = a_k \oplus b_k \oplus c_k\)
  - \(c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k\) \(\forall 0 \leq k \leq 31\)
- Circuit diagrams: tedious to draw, error-prone

- A hardware description language (HDL), *i.e.*, a programming language specialized to describe hardware
  - Precisely specify the structure and behavior of digital circuits
  - Designs can be automatically simulated or synthesized to hardware
  - Enables building hardware with same principles used to build software (write and compose simple, reusable building blocks)
  - Uses a familiar syntax (functions, variables, control-flow statements, etc.)

But be aware of the differences!
Introduction to Minispec

A simple HDL based on Bluespec
Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions

```plaintext
function Bool inv(Bool x);
    Bool result = !x;
    return result;
endfunction
```

- All values have a fixed type, which is known statically (e.g., result is of type `Bool`)
- Note: Types Start With An Uppercase Letter, variable and function names are lowercase
Bool Type and Operations

- Values of type \texttt{Bool} can be \texttt{True} or \texttt{False}
- \texttt{Bool} supports Boolean and comparison operations:

  ```
  Bool a = True;
  Bool b = False;

  Bool x = !a;    // False since a == True
  Bool y = a && b; // False since b == False
  Bool z = a || b; // True since a == True

  Bool n = a != b; // True; equivalent to XOR
  Bool e = a == b; // False; equivalent to XNOR
  ```

- \texttt{Bool} is the simplest type, but working with many single-bit values is tedious
  - Need a type that represents multi-bit values!
Bit#(n) Type and Operations

- Bit#(n) represents an n-bit value
- Bit#(n) supports the following basic operations:
  - Bitwise logical: ~ (negation), & (AND), | (OR), ^ (XOR)
  - Bit selection
  - Concatenation

```verbatim
Bit#(4) a = 4'b0011;   // 4-bit binary 3
Bit#(4) b = 4'b0101;   // 4-bit binary 5
Bit#(4) x = ~a;       // 4'b1100
Bit#(4) y = a & b;     // 4'b0001
Bit#(4) z = a ^ b;     // 4'b0110

Bit#(1) l = a[0];     // 1'b1 (least significant)
Bit#(3) m = a[3:1];   // 3'b001

Bit#(8) c = {a, b};   // 8'b00110101
```

L08-13
Full Adder in Minispec

function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    Bit#(1) s = a ^ b ^ cin;
    Bit#(1) cout = (a & b) | (a & cin) | (b & cin);
return {cout, s};
endfunction

s = a ⊕ b ⊕ cin

cout = a·b + a·cin + b·cin
2-bit Ripple-Carry Adder

- Functions are **inlined**: Each function call creates a new instance (copy) of the called circuit
  - Allows composing simple circuits to build larger ones

```cpp
function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) cin);
    Bit#(2) lower = fullAdder(a[0], b[0], cin);
    Bit#(2) upper = fullAdder(a[1], b[1], lower[1]);
    return {upper, lower[0]};
endfunction
```
4-bit Ripple-Carry Adder

Composing functions lets us build larger circuits, but writing very large circuits this way is tedious

- Next lecture: Writing an n-bit adder in a single function

function Bit#(5) rca4(Bit#(4) a, Bit#(4) b, Bit#(1) cin);
    Bit#(3) lower = rca2(a[1:0], b[1:0], cin);
    Bit#(3) upper = rca2(a[3:2], b[3:2], lower[2]);
return {upper, lower[1:0]};
endfunction
Multiplexers
A 2-way multiplexer or mux selects between two inputs $a$ and $b$ based on a single-bit input $s$ (select input).

Gate-level implementation:
- What if $a$ and $b$ are $n$-bit wide?
- Replicate the single-bit 2-way mux $n$ times
- Use the same $s$ as the select input for all the replicated structures

$$y = a \cdot \bar{s} + b \cdot s$$
4-way Multiplexer

- A 4-way multiplexer selects between four inputs based on the value of a 2-bit select input $s$
  - Typically implemented using 2-way multiplexers

- An $n$-way multiplexer can be implemented with a tree of $n-1$ 2-way multiplexers
8-way Multiplexer

- An n-way multiplexer can be implemented with a tree of \( n-1 \) 2-way multiplexers
Multiplexers in Minispec

- 2-way mux → Conditional operator

\[
\begin{align*}
\text{Minispec} & : s? b : a \\
\text{Python} & : \text{if } s: b; \text{ else: } a;
\end{align*}
\]

s has type \text{Bool}; True is treated as 1 and False as 0

- N-way mux → Case expression

\[
\begin{align*}
\text{Minispec} & : \text{case } (s) \\
& : \begin{cases}
0 : a; \\
1 : b; \\
2 : c; \\
3 : d;
\end{cases}
\end{align*}
\]

s has type \text{Bit#}(2)
Aside: No Conditional Execution!

- Given this conditional statement...
  
  $$s? \ foo(x) : \ bar(y)$$

- In software, the program would first evaluate $s$, then run either $\text{foo}(x)$ or $\text{bar}(y)$

- But in hardware, this statement instantiates and evaluates both $\text{foo}(x)$ and $\text{bar}(y)$, in parallel!
Selecting a Wire: $x[i]$

- **Constant selector:** e.g., $x[2]$
  - No hardware; $x[2]$ is just the name of a wire

- **Dynamic selector:** $x[i]$
  - 4-way mux
Shift operators
Fixed-Size Shifts

- Fixed-size shift operation is cheap in hardware
  - Just wire the circuit appropriately

- Arithmetic shifts are similar

Arithmetic right shift by n divides integer in two’s complement representation by $2^n$
Logical Right Shift by $s$

- Suppose we want a shifter that right-shifts an N-bit input $x$ by $s$, where $N=32$ and $0 \leq s \leq 31$
- Naïve approach: Create 32 different fixed-size shifters and select using a mux

How many 2-way one-bit muxes are needed to implement this structure?

$$ (32-1) \times 32 = 992 $$

$\approx 4k$ gates

We can do better!

Tree-like 32-input mux

Each input is 32-bit
Barrel Shifter
An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by $s$ using a series of fixed-size power-of-2 shifts
  - For example, shift by 5 ($=4+1$) can be done with shifts of sizes 4 and 1
  - The bit encoding of $s$ tells us which shifts are needed: if the $i^{th}$ bit of $s$ is 1, then we need to shift by $2^i$
    - Ex: $5 = 0b00101$
  - Implementation: A cascade of $\log_2 N$ muxes that choose between shifting by $2^i$ and not shifting

How many 2-way 1-bit muxes?

$N*\log_2 N = 32*5 = 160$
Barrel Shifter Implementation

- Example in Minispec for N=4
  - Only need 2 bits for s, why?

- Use conditional operator for 2-way muxes

- Use concatenation and bit selection for fixed shifts

```plaintext
function Bit#(4) barrelShifter(Bit#(4) x, Bit#(2) s);
  Bit#(4) r1 = (s[1] == 0) ? x : {2'b00, x[3:2]};
  Bit#(4) r0 = (s[0] == 0) ? r1 : {1'b0, r1[3:1]};
  return r0;
endfunction
```
Thank you!

Next lecture:
Complex combinational circuits and advanced Minispec