Combinational Logic and Introduction to Minispec
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)

\[
\begin{array}{c}
\begin{array}{cccc}
& a_{n-1} & a_0 & b_{n-1} & b_0 \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{cccc}
\text{n-bit adder} & \equiv & \text{n-bit adder} \\
\cdots & \cdots & \cdots & \cdots \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{cccc}
s_n & s_0 \\
\cdots & \cdots \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
1110 \\
\downarrow \\
1110 \\
\downarrow \\
10101
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
1110 \\
\downarrow \\
0111 \\
\downarrow \\
10101
\end{array}
\end{array}
\]

\[
1110 \quad \text{carry}
\]
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 \textit{carry-in} \( c_{\text{in}} \)
  - Produces a \textit{sum} bit and a \textit{carry-out} bit

- Then, cascade FAs to perform binary addition

- Result: A \textit{ripple-carry adder} (simple but slow)
Deriving the Full Adder

Truth table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c_{in}</th>
<th>c_{out}</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

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

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

alternatives

- Truth table with $2^{64}$ rows and 33 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  - $s_k = a_k \oplus b_k \oplus c_k$
  - $c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$
- 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.)
Introduction to Minispec

A simple HDL based on Bluespec
Combinational Logic as Functions

- In Minispec, combinational circuits are described using functions.

```
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 **Bool** can be *True* or *False*
- 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
  ```

- 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)

```plaintext
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 selection

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

- Concatenation

```plaintext
Bit#(8) c = {a, b}; // 8'b00110101
```
Full Adder in Minispec

\[
\begin{align*}
    s &= a \oplus b \oplus c_{\text{in}} \\
    c_{\text{out}} &= a \cdot b + a \cdot c_{\text{in}} + b \cdot c_{\text{in}}
\end{align*}
\]

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
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

```verbatim
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
```

---

September 26, 2019  MIT 6.004 Fall 2019  L07-13
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
Multiplexers
2-way Multiplexer

- 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:
  - If $a$ and $b$ are $n$-bit wide then this structure is replicated $n$ times; $s$ is the same input for all the replicated structures.

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

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

- Typically implemented using 2-way multiplexers

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

- 2-way mux $\rightarrow$ Conditional operator

$$s \ ? \ b \ : \ a$$

- N-way mux $\rightarrow$ Case expression

```minispec
case (s)
    0 : a;
    1 : b;
    2 : c;
    3 : d;
endcase
```

$s$ has type `Bool`; True is treated as 1 and False as 0.
Selecting a Wire: \( x[i] \)

- **Constant selector:** e.g., \( x[2] \)

  
  \[
  \begin{array}{cccc}
  x_0 & x_1 & x_2 & x_3 \\
  \end{array}
  \]

  ![Constant selector diagram]

  No hardware; \( x[2] \) is just the name of a wire.

- **Dynamic selector:** \( x[i] \)

  
  \[
  \begin{array}{cccc}
  x_0 & x_1 & x_2 & x_3 \\
  \end{array}
  \]

  ![Dynamic selector diagram]

  4-way mux

Assume \( x \) is 4 bits wide.
Shift operators
Fixed-size shifts

- Fixed size shift operation is cheap in hardware
  - Just wire the circuit appropriately
- Arithmetic shifts are similar

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{array}
\quad
\begin{array}{cccc}
a & b & c & d \\
0 & 0 & a & b \\
\end{array}
\]

\[
\begin{array}{cccc}
-8/4 & 1 & 0 & 0 & 0 \\
\end{array}
\quad
\begin{array}{cccc}
a & b & c & d \\
\end{array}
\]

\[
\begin{array}{cccc}
-2 & 1 & 1 & 1 & 0 \\
\end{array}
\quad
\begin{array}{cccc}
a & a & a & a & b \\
\end{array}
\]

useful for multiplication and division by \(2^n\)
Logical right shift by $n$

- Suppose we want to build a shifter that right-shifts a value $x$ by $n$ where $n$ is between 0 and 31
- One way to do this is by selecting from 32 different fixed-size shifters using a mux

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

$$n \cdot (n-1)$$

Can we do better?
Barrel shifter
An efficient circuit to perform logical right shift by \( n \)

- Shift by \( n \) can be broken down into \( \log n \) steps of fixed-length shifts of size 1, 2, 4, ...
  - For example, we can perform shift 5 (=4+1) by doing shifts of size 4 and 1
  - Thus, \( 8\text{'}b01100111 \) shift 5 can be performed in two steps:
    - \( 8\text{'}b01100111 \Rightarrow 8\text{'}b00000110 \Rightarrow 8\text{'}b00000011 \)
      - shift 4
      - shift 1

- For a 32-bit number, a 5-bit \( n \) can specify all the needed shifts
  - \( 3_{10} = 00011_2 \), \( 5_{10} = 00101_2 \), \( 21_{10} = 10101_2 \)
  - The bit encoding of \( n \) tells us which shifters are needed; if the value of the \( i^{th} \) (least significant) bit is 1 then we need to shift by \( 2^i \) bits
Conditional operation: Shift versus no-shift

- We need a mux to select the appropriate wires: if $s$ is 1 the mux selects the wires on the left, otherwise it selects the wires on the right

```vhdl
(s==0)?{a,b,c,d}:{2'b0,a,b};
```
Barrel shifter implementation

- A barrel shifter for an n-bit number uses a cascade of $\log n$ muxes, each performing a conditional fixed-size shift of sizes 1, 2, 4, ...

- Example: A barrel shifter for 4-bit numbers can be expressed as two conditional expressions:

```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[3:0]};
    return r0;
endfunction
```
Thank you!

Next lecture:
Complex combinational circuits
and advanced Minispec