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
  - 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>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

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

- Truth table with $2^{64}$ rows and 64 columns
- 32 sets of Boolean equations, where each set describes a FA
- Use some ad-hoc notation to describe recurrences
  \[ s_k = a_k \oplus b_k \oplus c_k \]
  \[ c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \]
  \[ 0 \leq k \leq 31 \]
- 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
In Minispec, combinational circuits are described using functions.

- 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.
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
  ```
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

![Diagram of 2-bit Ripple-Carry Adder]

```verilog
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
```

- Functions are **inlined**: Each function call creates a new instance (copy) of the called circuit
  - Allows composing simple circuits to build larger ones
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 → Conditional operator
  
  Minispec:  
  ```minispec
  a  b  s
  0  1 0
  s? b : a
  ```
  
  Python:  
  ```python
  b if s else a
  ```
  
  s has type `Bool`; True is treated as 1 and False as 0

- N-way mux → Case expression
  
  Minispec:  
  ```minispec
  a  b  c  d  s
  0  1  2  3 2
  case (s)
  0 : a;
  1 : b;
  2 : c;
  3 : d;
  endcase
  ```
  
  s has type `Bit#(2)`
Aside: No Conditional Execution!

- Given this conditional statement...
  \[ s ? \text{foo}(x) : \text{bar}(y) \]

- 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., $x[2]$
  
  \[ x_0 \quad x_1 \quad x_2 \quad x_3 \]
  \[ \quad [2] \quad \text{no hardware; } x[2] \text{ is just the name of a wire} \]

- **Dynamic selector:** $x[i]$
  
  \[ x_0 \quad x_1 \quad x_2 \quad x_3 \]
  \[ \quad [i] \quad \text{4-way mux} \]

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

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

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

- $8 \div 4 = -2$
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?

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

$= \sim 4k$ gates

We can do better!
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?*

$$N \cdot \log_2 N = 32 \cdot 5 = 160$$
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

```markdown
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