Complex Combinational Logic: Implementation and Design Tradeoffs
Lecture Goals

- Learn some advanced Minispec features that enable implementing large circuits succinctly
  - Parametric functions
  - Type inference and user-defined types
  - Loops and control-flow statements

- Study design tradeoffs in combinational logic by analyzing different adder implementations
Reminder: 4-bit Ripple-Carry Adder

- Problem 1: Have to write a function for every bit width
- Problem 2: If we build large functions from smaller ones, have to write many functions!
Parametric Types

- **Bit#(n)**, an n-bit value, is a **parametric type**
  - n is the **parameter** (an Integer value)
  - Using Bit#(n) requires specifying a fixed n (e.g., Bit#(4) is a 4-bit value)

- Minispec provides other parametric types, and lets you define your own
  - Parametric types are **generic**
  - They take one or more parameters
  - Parameters must be known at compile-time
  - Specifying the parameters yields a **concrete** type

- Parameters can be Integers or types
  - Example: Vector#(n, T) is an n-element vector of T’s (e.g., Vector#(4, Bit#(8)) = 4-elem vector of 8-bit values)
Parametric Functions

- Functions have fixed argument and return types
  - Problem 1: Have to write a function for every bit width
  - Problem 2: If we build large functions from smaller ones, have to write many functions! (e.g., rca2 \( \rightarrow \) rca4 \( \rightarrow \) rca8 \( \ldots \))

- Parametric functions solve these problems: We can write one *generic* function that covers every case
  - Example: \( rca\#(n) \), an \( n \)-bit ripple-carry adder

- A parametric function must be invoked with fixed parameters, which instantiates a *concrete* function
  - Example: Calling \( rca\#(32) \) instantiates a 32-bit adder
Example: Parametric Parity

function Bit#(1) parity#(Integer n)(Bit#(n) x);
   return (n == 1)? x : x[n-1] ^ parity#(n-1)(x[n-2:0]);
endfunction

- The parameter $n$ is used as a variable in the function
- Large circuits implemented by composing smaller ones: parity#(n) invokes parity#(n-1)!
- If another function calls parity#(3), compiler produces:

```plaintext
function Bit#(1) parity#(3)(Bit#(3) x);
   return x[2] ^ parity#(2)(x[1:0]);
endfunction
function Bit#(1) parity#(2)(Bit#(2) x);
   return x[1] ^ parity#(1)(x[0:0]);
endfunction
function Bit#(1) parity#(1)(Bit#(1) x);
   return x;
endfunction
```

![Diagram of parity circuits](image)
Integer is a Special Type
Always evaluated by the compiler

- Integer values are (positive or negative) numbers with an **unbounded number of bits**
  - Unbounded bits → Cannot be synthesized to hardware

- Integers are guaranteed to be evaluated at compile time, i.e., turned into fixed numbers
  - If the compiler cannot evaluate an Integer expression, it throws an error

- Integer supports the same operations as Bit#(n), (arithmetic, logical, comparisons, etc.)
  - But evaluated by compiler → operations on Integers never produce any hardware
N-bit Ripple-Carry Adder

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

function Bit#(n+1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) cin);
    Bit#(n) lower = rca#(n-1)(a[n-2:0], b[n-2:0], cin);
    Bit#(2) upper = fullAdder(a[n-1], b[n-1], lower[n-1]);
    return {upper, lower[n-2:0]};
endfunction

// Base case
function Bit#(2) rca#(1)(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    return fullAdder(a, b, cin);
endfunction
Type Inference

- You can omit the type of a variable by declaring it with the let keyword.
- The compiler infers the variable’s type from the type of the expression assigned to the variable.

```plaintext
Bit#(4) x = 4'b0011;
let y = x;       // y has type Bit#(4)
let z = {x, x}; // z has type Bit#(8)
let w = 2'b11;  // w has type Bit#(2)
let n = 42;     // n has type Integer
```
User-Defined Types

- **Type synonyms** allow giving a different name to a type

- **Structs** represent a group of member values with different types

- ** Enums** represent a set of symbolic constants

- Structs and enums are much clearer than using raw bits!
  - e.g., `Bit#(24) pixel; pixel[15:8]` versus `Pixel pixel; pixel.green`

```plaintext
typedef Bit#(8) Byte;

typedef struct {
    Byte red;
    Byte green;
    Byte blue;
} Pixel;

Pixel p;
p.red = 255;

typedef enum {
    Ready, Busy, Error
} State;

State state = Ready;
```
For Loops

- For loop statements allow compactly expressing a sequence of similar statements

  ```
  Bit#(6) w = 0;
  for (Integer i = 0; i < 6; i = i + 1)
      w[i] = z[i / 2];
  ```

- For loops are not like loops in software programming languages!
  - Fixed number of iterations (Integer induction variable!)
  - Unrolled at compile time

  ```
  w[0] = z[0];
  w[1] = z[0];
  w[2] = z[1];
  w[3] = z[1];
  w[4] = z[2];
  w[5] = z[2];
  ```
N-bit Ripple-Carry Adder with Loop

function Bit#(n+1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) cin);
Bit#(n) s = 0;
Bit#(n+1) c = {0, cin};
for (Integer i = 0; i < n; i = i + 1) begin
    let x = fullAdder(a[i], b[i], c[i]);
    s[i] = x[0];
    c[i+1] = x[1];
end
return {c[n], s};
endfunction
Conditional Statements

- If statements have a syntax similar to software:

  ```
  function Bit#(4) max(Bit#(4) a, Bit#(4) b); 
  Bit#(4) result = b;
  if (a > b) result = a;
  return result;
endfunction
  ```

- But they are **implemented very differently** from software programming languages!
  - Translated to muxes, like conditional expressions
  - Each variable assigned within an if statement uses a mux to select the right value (the one assigned in the if branch, else branch, or the previous value)

- Minispec also has case statements (see tutorial)
Minispec Takeaways

- Minispec lets you build circuits with constructs similar to those of software programming languages

- But keep in mind that the implementation of these features is often very different from software!
  - Parametric functions and types are instantiated
  - Functions are inlined
  - Conditionals (?:, if-else, case) are translated to multiplexers, and all their branches are evaluated
  - Loops are unrolled
  - What remains is an acyclic graph of gates

Never forget that you’re designing hardware
Design Tradeoffs in Combinational Circuits
Algorithmic Tradeoffs in Hardware Design

- Each function often allows many implementations with widely different delay, area, and power

- Choosing the right **algorithms** is key to optimizing your design
  - Tools cannot compensate for an inefficient algorithm (in most cases)
  - Just like programming software

- Case study: Building a better adder
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001

\[ t_{PD} = n \times t_{PD,FA} \approx \Theta(n) \]

- \( \Theta(n) \) is read “order n” and tells us that the latency of our adder grows linearly with the number of bits of the operands
Asymptotic Analysis

- Formally, \( g(n) = \Theta(f(n)) \) iff there exist \( C_2 \geq C_1 > 0 \) such that for all but finitely many integers \( n \geq 0 \),

\[
C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)
\]

- Example: \( n^2 + 2n + 3 = \Theta(n^2) \) (read “is of order \( n^2 \))

since \( 2n^2 > n^2 + 2n + 3 > n^2 \) except for a few small integers
Carry-Select Adder Trades Area for Speed

- **Propagation delay:** $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder.
  - If we apply the same strategy recursively (build each 16-bit adder from 8-bit carry-select adders, etc.), $t_{PD,n} = \Theta(\log n)$

**Drawbacks?** Consumes much more area than ripple-carry adder. Wide mux adds significant delay (lab 4)
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay

Key idea: Transform chain of carry computations into a tree

- Transforming a chain of associative operations (e.g., AND, OR, XOR) into a tree is easy
- But how to do this with carries?
Carry-Lookahead Adder Details

NOTE: Remaining slides are optional material that will not be on a quiz but will be helpful for Lab 2 and the Design Project
Building a Carry-Lookahead Adder

- Step 1: Generate the output carry in $\Theta(\log n)$ delay
- Step 2: Extend step 1 to generate all carries in $\Theta(\log n)$ delay

- We will use two main ideas that are broadly useful beyond CLAs!
  - Step 1 leverages that function composition is associative
  - Step 2 uses the parallel scan (a.k.a. parallel prefix) algorithm
Function composition is associative
Basic math reminder 😊

- Function composition is a binary operation \( \circ \) that takes two functions as inputs, \( f \) and \( g \), and produces a function \( h = f \circ g \) such that \( h(x) = g(f(x)) \)

- Function composition is always associative:
  \[
  f_1 \circ f_2 \circ f_3 = (f_1 \circ f_2) \circ f_3 = f_1 \circ (f_2 \circ f_3)
  \]
Deriving carry-out in $\Theta(\log n)$ delay

- Consider a ripple-carry adder:

- Suppose all inputs $(a, b, c_0)$ become valid and stable at $t=0$. If we focus on computing the output carry only, this circuit is equivalent to

- A chain of functions $f_i$, each with 1-bit input and output
- Each $f_i$ is determined by the values of $a_i$ and $b_i$
Turning function chains into trees

- Because function composition is associative, we can turn a chain of functions into a tree by first composing the functions...

\[
\begin{align*}
&c_4 \xleftarrow{C_3} f_3 \xleftarrow{C_2} f_2 \xleftarrow{C_1} f_1 \xrightarrow{} f_0 \\
&f_{32} = f_2 \circ f_3 \quad f_{10} = f_0 \circ f_1 \\
&f_{30} = f_{10} \circ f_{32}
\end{align*}
\]

...and then evaluating the final function: \( c_4 = f_{30}(c_0) \)

*How does delay grow with chain length \( n \)? \( \Theta(\log n) \)*

- Very general trick: Can turn any chain of functions into a tree, *if you can compose them efficiently*
1-bit input, 1-bit output functions

<table>
<thead>
<tr>
<th></th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>generate</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>propagate</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>invert</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **How many bits do we need to enumerate these functions?**  2 bits (only 4 choices!)
### Composing 1-bit functions

<table>
<thead>
<tr>
<th>f</th>
<th>g</th>
<th>f $\circ$ g</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill</td>
<td>kill</td>
<td>kill</td>
</tr>
<tr>
<td>kill</td>
<td>generate</td>
<td>generate</td>
</tr>
<tr>
<td>kill</td>
<td>propagate</td>
<td>propagate</td>
</tr>
<tr>
<td>kill</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>generate</td>
<td>kill</td>
<td>kill</td>
</tr>
<tr>
<td>generate</td>
<td>generate</td>
<td>generate</td>
</tr>
<tr>
<td>generate</td>
<td>propagate</td>
<td>propagate</td>
</tr>
<tr>
<td>generate</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>generate</td>
<td>kill</td>
<td>kill</td>
</tr>
<tr>
<td>generate</td>
<td>generate</td>
<td>generate</td>
</tr>
<tr>
<td>propagate</td>
<td>generate</td>
<td>propagate</td>
</tr>
<tr>
<td>propagate</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>propagate</td>
<td>kill</td>
<td>kill</td>
</tr>
<tr>
<td>propagate</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>propagate</td>
<td>generate</td>
<td>generate</td>
</tr>
<tr>
<td>propagate</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>invert</td>
<td>kill</td>
<td>kill</td>
</tr>
<tr>
<td>invert</td>
<td>generate</td>
<td>generate</td>
</tr>
<tr>
<td>invert</td>
<td>propagate</td>
<td>propagate</td>
</tr>
<tr>
<td>invert</td>
<td>invert</td>
<td>invert</td>
</tr>
<tr>
<td>invert</td>
<td>propagate</td>
<td>propagate</td>
</tr>
</tbody>
</table>

This is just a combinational function with two 2-bit inputs and one 2-bit output!
Deriving the initial functions

- Remember, to derive the carry-out, a ripple-carry adder can be seen as a chain of functions $f_i$, each determined by the values of $a_i$ and $b_i$.

- We can derive these functions from the full adder:

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

This is a function with two 1-bit inputs and one 2-bit output.

- kill
- propagate
- propagate
- generate

February 16, 2023
MIT 6.191 Spring 2023
Evaluating functions

- Given a function $f$ (as a 2-bit input value), we need another function that applies $f$ to an input carry to produce the output carry.
- This is also a simple combinational circuit.

<table>
<thead>
<tr>
<th>$f$</th>
<th>$c_{\text{in}}$</th>
<th>$c_{\text{out}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kill</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>generate</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>generate</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>propagate</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>propagate</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>invert</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>invert</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Invert not used in CLAs.
Step 1: Generating the output carry
Putting it all together

How does delay grow with number of bits?
\( \Theta(\log n) \)
Step 2: Generating all carries

- So far we have seen how to generate a single output carry, but we need all intermediate ones too.
- This is a specific application of the parallel scan (a.k.a. parallel prefix) algorithm.

Two main options:
- Brent-Kung CLA: Low area but some extra delay
- Kogge-Stone CLA: High area but lower delay
Option 1: Brent-Kung CLA
Option 2: Kogge-Stone CLA
CLA Nitty-Gritty: Choosing a good encoding for functions

- CLAs need to encode three possible functions: kill, propagate, generate (invert is not used)
- 2-bits required per function, but 3 values, so how they are encoded can affect logic cost

- A common encoding is $f = \{g, p\}$, where:
  - $g = ab$ (generate bit)
  - $p = a+b$ (propagate bit)

- With this encoding,  
  - $g = 0, p = 0 \rightarrow$ kill  
  - $g = 0, p = 1 \rightarrow$ propagate  
  - $g = 1, p = X \rightarrow$ generate
CLA Building Blocks with \( f = \{g, p\} \) encoding

- Produce initial \( f \) signals
  \[
  a \quad b
  \]
  \[
  f = \{g, p\}
  \]
  \[
  g = ab
  \]
  \[
  p = a + b
  \]

- Compose \( f \) signals
  \[
  f_{ij} \quad f_{(j-1)k}
  \]
  \[
  g_{ik} = g_{ij} + p_{ij}g_{(j-1)k}
  \]
  \[
  p_{ik} = p_{ij}p_{(j-1)k} \quad (i \geq j > k)
  \]

- Produce individual carries
  \[
  f_{ij} \quad c_j
  \]
  \[
  c_{i+1} = g_{ij} + p_{ij}c_j
  \]
Carry-Lookahead Adder Takeaways

- There are many CLA designs
  - We’ve seen Brent-Kung and Kogge-Stone CLAs
  - Some other types
  - Different variants for each type, e.g., using higher-radix trees to reduce depth

- This technique is useful beyond adders: computes any one-dimensional recurrence in $\Theta(\log n)$ delay
  - e.g., comparators, priority encoders, etc.
Summary

- Parametric functions let us write a generic description of a function that is then instantiated on demand.

- Use for loops and if-else statements with care: their similarity to software can be confusing and they can lead to poor circuits.

- Choosing the right algorithms is crucial to design good digital circuits—tools can only do so much!

- Carry-select and carry-lookahead adders achieve $\Theta(\log n)$ delay, but at the cost of extra area.
Thank you!

Next lecture: CMOS