Complex Combinational Logic: Implementation and Design Tradeoffs
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 \( 2^n \)

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
\downarrow & & & \downarrow \\
0 & 0 & 1 & 0
\end{array}
\quad
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
\downarrow & & & \downarrow \\
1 & 1 & 1 & 0
\end{array}
\quad
\begin{array}{c}
x \quad \gg \quad 2 \quad \downarrow \\
\downarrow & & \downarrow \\
4 & & 4
\end{array}
\]

\[-8 \div 4 = -2\]
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?

\[
(32-1) \times 32 = 992 = \sim 4k \text{ gates}
\]

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 $\log_2N$ muxes that choose between shifting by $2^i$ and not shifting

How many 2-way 1-bit muxes?

$N\times\log_2N = 32\times5 = 160$
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

```verilog
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
```
Implementing Large Circuits in Minispec
Parametric Types

- Bit#(n), an n-bit value, is a **parametric type**
  - n is the **parameter** (an Integer value)
  - Using Bit#(n) requires specifying 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 → rca4 → rca8 …)

- 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

```plaintext
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
```
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](image)

```plaintext
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.

```verilog
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

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

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

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

  Pixel p;
  p.red = 255;
  ```

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

  ```plaintext
typedef enum {
    Ready, Busy, Error
  } State;

  State state = Ready;
  ```

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

- Example: The loop above is translated into this sequence:
  
  ```
  w[0] = z[0];
  w[1] = z[0];
  w[2] = z[1];
  w[3] = z[1];
  w[4] = z[2];
  w[5] = z[2];
  ```
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 quite 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

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

- \( \Theta(n) \) 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?
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.
Good Luck on the Quiz! 😊

Next lecture: Sequential Circuits
Carry-Lookahead Adder Details

NOTE: Remaining slides are optional material which will not be on a quiz but can be helpful for Lab 4 and the Design Project
Carry Generation and Propagation

\[ s = a \oplus b \oplus c_{in} \]
\[ c_{out} = ab + ac_{in} + bc_{in} \]

- We can rewrite \( c_{out} = ab + (a+b)c_{in} \) as \( c_{out} = g + pc_{in} \) with \( g = ab \) (generate) and \( p = a+b \) (propagate)
  - \( g=1 \) \( \rightarrow \) \( c_{out} = 1 \) (FA generates a carry)
  - \( p=1 \) (and \( g=0 \)) \( \rightarrow \) \( c_{out} = c_{in} \) (FA propagates carry)

Note \( p \) and \( g \) don’t depend upon \( c_{in} \)
Consider a 2-bit ripple-carry adder. Let’s derive $c_2$ as a function of $c_0$ and the individual $g$’s and $p$’s.

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0)$$

$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$

What about a 4-bit adder?

$$g_{10} = g_1 + p_1 g_0$$
$$p_{10} = p_1 p_0$$

$$g_{32} = g_3 + p_3 g_2$$
$$p_{32} = p_3 p_2$$

$$g_{30} = g_{32} + p_{32} g_{10}$$
$$p_{30} = p_{32} p_{10}$$

$$c_4 = g_{30} + p_{30} c_0$$
## CLA Building Blocks

- **Step 1:** Generate individual $g$ & $p$ signals
  
  - $g = ab$
  - $p = a+b$

- **Step 2:** Combine adjacent $g$ & $p$ signals
  
  $$
g_{ik} = g_{ij} + p_{ij}g_{(j-1)k}$$
  $$\ p_{ik} = p_{ij}p_{(j-1)k} \quad (i \geq j > k)$$

- **Step 3:** Generate individual carries
  
  $$c_{i+1} = g_{ij} + p_{ij}c_j$$

There are many CLA variants. Let’s derive the Brent-Kung CLA.
Generating and Combining gp’s

How does delay grow with number of bits?

$\Theta(\log n)$
Generating the Carries
There are many CLA designs
- We’ve seen a Brent-Kung CLA
- Several other types (e.g., Kogge-Stone)
- Different variants for each type, e.g., using higher-radix trees to reduce depth

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