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
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: Shifts

- Fixed-size shifts are cheap
  - Just wires
Reminder: Shifts

- Fixed-size shifts are cheap
  - Just wires
Reminder: Shifts

- Fixed-size shifts are cheap
  - Just wires

- What about variable-size shifts?
  - Suppose we want to build a shifter that right-shifts a 32-bit value $x$ by $n$, where $n$ is between 0 and 31
Reminder: Shifts

- Fixed-size shifts are cheap
  - Just wires

- What about variable-size shifts?
  - Suppose we want to build a shifter that right-shifts a 32-bit value $x$ by $n$, where $n$ is between 0 and 31
  - Naïve approach: Select from 32 different fixed-size shifters using a mux
Reminder: Shifts

- Fixed-size shifts are cheap
  - Just wires

- What about variable-size shifts?
  - Suppose we want to build a shifter that right-shifts a 32-bit value $x$ by $n$, where $n$ is between 0 and 31
  - Naïve approach: Select from 32 different fixed-size shifters using a mux
    - Expensive!
    - $n \times (n-1)$ 2-way 1-bit muxes
Barrel Shifter
An efficient circuit to perform variable-size shifts

- A barrel shifter performs shift by $n$ using a series of fixed-size shifts by power-of-2 sizes
  - For example, shift by 5 ($=4+1$) can be done with shifts of sizes 4 and 1
  - The bit encoding of $n$ tells us which shifts are needed: if the $i^{th}$ bit of $n$ is 1, then we need to shift by $2^i$
Barrel Shifter
An efficient circuit to perform variable-size shifts

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

- A barrel shifter performs shift by \( n \) using a series of fixed-size shifts by power-of-2 sizes
  - For example, shift by 5 (=4+1) can be done with shifts of sizes 4 and 1
  - The bit encoding of \( n \) tells us which shifts are needed: if the \( i^{th} \) bit of \( n \) is 1, then we need to shift by \( 2^i \)
  - 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?
```
Barrel Shifter
An efficient circuit to perform variable-size shifts

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

\[
\text{How many 2-way 1-bit muxes?}
\]

\[
n \times \log_2 n
\]
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)
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

- **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
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
Parametric Functions

- Functions have fixed argument and return types
  - Problem 1: Have to write a function for every bit width
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 ...)

October 1, 2019
MIT 6.004 Fall 2019
L08-7
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
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

```
function Bit#(1) parity#(Integer n)(Bit#(n) x);
    return (n == 1)? x : x[n-1] ^ parity#(n-1)(x[n-2:0]);
endfunction
```
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.
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: \( \text{parity}##(n) \) invokes \( \text{parity}##(n-1) \)! 
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:
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:

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

  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
```
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:

  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
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:

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
Example: Parametric Parity

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

  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
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 $\rightarrow$ 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 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

\[
\begin{align*}
&\text{rca}(n) \\
&a_n \quad b_n \\
&\downarrow \quad \downarrow \\
&S \\
&\text{rca}(n-1) \\
&a_{n-1} \quad b_{n-1} \\
\end{align*}
\]
function Bit#(n+1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) cin);

endfunction
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);
endfunction
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]);
endfunction
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
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

```c
typedef Bit#(8) Byte;
```
User-Defined Types

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

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

```
typedef Bit#(8) Byte;

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

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

```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;
```
User-Defined Types

- **Type synonyms** allow giving a different name to a type
  
  ```
  typedef Bit#(8) Byte;
  typedef struct {
    Byte red;
    Byte green;
    Byte blue;
  } Pixel;
  
  Pixel p;
  p.red = 255;
  ```

- **Structs** represent a group of member values with different types
  
  ```
  typedef enum {
    Ready, Busy, Error
  } State;
  
  State state = Ready;
  ```

- ** 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.green
For Loops

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

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

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

```cpp
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
For Loops

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

```plaintext
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:
  ```plaintext
  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:

```plaintext
function Bit#(4) max(Bit#(4) a, 
                  Bit#(4) b); 

    Bit#(4) result = b; 
    if (a > b) result = a; 
    return result; 
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

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

- If statements have a syntax similar to software:

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

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

  ```
  function Bit#(4) max(Bit#(4) a, Bit#(4) b);
  Bit#(4) result;
  if (a > b) result = a;
  else result = b;
  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.
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.
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
Each function often allows many implementations with widely different delay, area, and power.
Algorithmic Tradeoffs in Hardware Design

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

Diagram:
- Problem
- Hardware designer
- High-level circuit description
- Synthesis tool
- Optimized circuit implementation
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.
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

\[ \text{full Adder} \]

\[ a_{n-1} \rightarrow b_{n-1} \]

\[ c_n \rightarrow c_{n-1} \]

\[ s_{n-1} \]

\[ a_1 \rightarrow b_1 \]

\[ c_2 \rightarrow c_1 \]

\[ s_1 \rightarrow s_0 \]

\[ a_0 \rightarrow b_0 \]

\[ c_{\text{in}} \]
Ripple-Carry Adder: Simple but Slow

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

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

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

\[ t_{PD} = n \cdot t_{PD,FA} \]
Ripple-Carry Adder: Simple but Slow

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

\[ t_{PD} = n \times t_{PD,FA} \approx \Theta(n) \]
Ripple-Carry Adder: Simple but Slow

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

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

- \( \Theta(n) \) 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 \( n \geq 0 \),

\[
C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)
\]
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 $n \geq 0$,

$$C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)$$

$g(n) = O(f(n))$  \[\Theta(... \text{ implies both inequalities; } O(... \text{ implies only the first.})\]
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 $n^2$”)
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

\[ a[31:16] \downarrow b[31:16] \downarrow \text{16-bit Adder} \downarrow 0 \]
\[ a[15:0] \downarrow b[15:0] \downarrow \text{16-bit Adder} \downarrow 0 \]
\[ \text{16-bit Adder} \downarrow 1 \]
\[ s[31:16] \]

\[ s[15:0] \]
Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”.

The carry-out of the low half selects the correct version of the high-half addition.
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”

- Propagation delay: $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$
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

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”

The carry-out of the low half selects the correct version of the high-half addition.
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.), \( t_{PD,n} = \Theta(\log n) \)

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”
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?**
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
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”.

- **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
Carry-Lookahead Adders (CLAs)

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

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

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

- Key idea: Transform chain of carry computations into a tree
Carry-Lookahead Adders (CLAs)

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

  - 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 Generation and Propagation

We can rewrite $c_{out} = ab + (a+b)c_{in}$ in $s = a \oplus b \oplus c_{in}$.

- We can rewrite $c_{out} = ab + (a+b)c_{in}$.
Carry Generation and Propagation

We can rewrite $c_{out} = ab + (a+b)c_{in}$ in as $c_{out} = g + pc_{in}$ with $g = ab$ (generate) and $p = a+b$ (propagate).
Carry Generation and Propagation

We can rewrite $c_{out} = ab + (a+b)c_{in}$ 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)

$s = a \oplus b \oplus c_{in}$

$c_{out} = ab + ac_{in} + bc_{in}$
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 \( c_{in} \)
Generate and Propagate
Compose Hierarchically!

\[ c_{out} = g + p \cdot c_{in} \]
where \( g = a \cdot b \) and \( p = a+b \)
Generate and Propagate
Compose Hierarchically!

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

\[ c_{\text{out}} = g + p \cdot c_{\text{in}} \]

where \( g = a \cdot b \) and \( p = a + b \)

October 1, 2019

MIT 6.004 Fall 2019
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.

\[ c_{\text{out}} = g + p \cdot c_{\text{in}} \]

where $g = a \cdot b$ and $p = a + b$

\[ c_2 = g_1 + p_1 c_1 \]
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:

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

where $g = a \cdot b$ and $p = a + b$.
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.

\[
c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0)
\]

\[
= g_1 + p_1 g_0 + p_1 p_0 c_0
\]

where \( g = a \cdot b \) and \( p = a + b \).
Generate and Propagate
Compose Hierarchically!

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

$c_{out} = g + p \cdot c_{in}$
where $g = a \cdot b$ and $p = a+b$

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

\[
c_2 = g_1 + p_1 c_1 = g_1 + p_1 (g_0 + p_0 c_0)
\]

\[
= g_1 + p_1 g_0 + p_1 p_0 c_0
\]
Generate and Propagate Compose Hierarchically!

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?

\[
c_{\text{out}} = g + p \cdot c_{\text{in}} \quad \text{where } g = a \cdot b \text{ and } p = a+b
\]
Generate and Propagate
Compose Hierarchically!

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?

\[
g_{10} = g_1 + p_1 g_0 \quad p_{10} = p_1 p_0
\]
Generate and Propagate
Compose Hierarchically!

\[ c_{\text{out}} = g + p \cdot c_{\text{in}} \]
where \( g = a \cdot b \) and \( p = a+b \)

- 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?

\[ g_{10} = g_1 + p_1 g_0 \quad p_{10} = p_1 p_0 \]
\[ g_{32} = g_3 + p_3 g_2 \quad p_{32} = p_3 p_2 \]
Generate and Propagate
Compose Hierarchically!

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?

\[
g_{10} = g_1 + p_1 g_0 \\
g_{32} = g_3 + p_3 g_2 \\
g_{30} = g_{32} + p_{32} g_{10}
\]

\[
p_{10} = p_1 p_0 \\
p_{32} = p_3 p_2 \\
p_{30} = p_{32} p_{10}
\]
Generate and Propagate
Compose Hierarchically!

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?

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

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

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

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

- Step 1: Generate individual g & p signals

\[ g = ab \]
\[ p = a + b \]
CLA Building Blocks

- **Step 1: Generate individual g & p signals**
  
  \[
  g = ab \quad \text{p} = a+b
  \]
  
  \[
  gp = \{g, p\}
  \]

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

- **Step 1:** Generate individual g & p signals
  
  \[ \begin{align*}
  &a \quad b \\
  \downarrow & \quad \downarrow \\
  g \quad p &= ab \quad a + b \\
  \end{align*} \]
  
  \( g_p = \{g, p\} \)

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

- **Step 3:** Generate individual carries
  
  \[ \begin{align*}
  &g_{ij} \quad c_j \\
  \downarrow & \quad \downarrow \\
  c_{i+1} &= g_{ij} + p_{ij}c_j \\
  \end{align*} \]

\[ \begin{align*}
  &g_{ij} \quad g_{(j-1)k} \\
  \downarrow & \quad \downarrow \\
  g_{ik} &= g_{ij} + p_{ij}g_{(j-1)k} \\
  p_{ik} &= p_{ij}p_{(j-1)k} & (i \geq j > k) \\
  \end{align*} \]
### 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
Generating and Combining gp’s
Generating and Combining gp’s

```
<table>
<thead>
<tr>
<th>a_7</th>
<th>b_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_6</td>
<td>b_6</td>
</tr>
<tr>
<td>a_5</td>
<td>b_5</td>
</tr>
<tr>
<td>a_4</td>
<td>b_4</td>
</tr>
<tr>
<td>a_3</td>
<td>b_3</td>
</tr>
<tr>
<td>a_2</td>
<td>b_2</td>
</tr>
<tr>
<td>a_1</td>
<td>b_1</td>
</tr>
<tr>
<td>a_0</td>
<td>b_0</td>
</tr>
</tbody>
</table>
```

GP

```
gp_76
  └── gp_74

GP
  └── gp_54

GP
  └── gp_32

GP
  └── gp_30

GP
  └── gp_10

GP
  └── gp_0
```
Generating and Combining gp’s
Generating and Combining gp’s

How does delay grow with number of bits?
Generating and Combining gp’s

How does delay grow with number of bits?

$\Theta(\log n)$
Generating and Combining gp’s

How does delay grow with number of bits? $\Theta(\log n)$
Generating the Carries

\[ \begin{align*}
    a_7 b_7 & \quad a_6 b_6 & \quad a_5 b_5 & \quad a_4 b_4 & \quad a_3 b_3 & \quad a_2 b_2 & \quad a_1 b_1 & \quad a_0 b_0 \\
    \text{gp}_7 & \quad \text{gp}_6 & \quad \text{gp}_5 & \quad \text{gp}_4 & \quad \text{gp}_3 & \quad \text{gp}_2 & \quad \text{gp}_1 & \quad \text{gp}_0 \\
    \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} \\
    \text{gp}_76 & \quad \text{gp}_54 & \quad \text{gp}_32 & \quad \text{gp}_30 & \quad \text{gp}_10 & \quad \text{gp}_1 & \quad \text{gp}_0 \\
    \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} \\
    \text{gp}_70 & \quad \text{gp}_74 & \quad \text{gp}_54 & \quad \text{gp}_30 & \quad \text{gp}_10 & \quad \text{gp}_1 & \quad \text{gp}_0 \\
\end{align*} \]
Generating the Carries

\[ a_7 \cdot b_7 \rightarrow \text{GP} \rightarrow a_6 \cdot b_6 \rightarrow \text{GP} \rightarrow a_5 \cdot b_5 \rightarrow \text{GP} \rightarrow a_4 \cdot b_4 \rightarrow \text{GP} \rightarrow a_3 \cdot b_3 \rightarrow \text{GP} \rightarrow a_2 \cdot b_2 \rightarrow \text{GP} \rightarrow a_1 \cdot b_1 \rightarrow \text{GP} \rightarrow a_0 \cdot b_0 \rightarrow \text{GP} \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

\[ \text{gp}_7 \rightarrow \text{GP} \rightarrow \text{gp}_6 \rightarrow \text{GP} \rightarrow \text{gp}_5 \rightarrow \text{GP} \rightarrow \text{gp}_4 \rightarrow \text{GP} \rightarrow \text{gp}_3 \rightarrow \text{GP} \rightarrow \text{gp}_2 \rightarrow \text{GP} \rightarrow \text{gp}_1 \rightarrow \text{GP} \rightarrow \text{gp}_0 \]

October 1, 2019
Generating the Carries
Generating the Carries

\[
\begin{array}{cccccccc}
\text{a}_7 \quad \text{b}_7 \\
gp_7 \\
\text{GP} \\
gp_76 \\
\text{GP} \\
gp_74 \\
\text{GP} \\
gp_70 \\
c_{in} \\
C \\
\text{c}_8 \\
\text{C} \\
\text{c}_6 \\
\text{C} \\
\text{c}_5 \\
\text{C} \\
\text{c}_4 \\
\text{C} \\
\text{c}_3 \\
\text{C} \\
\text{c}_2 \\
\text{C} \\
\text{c}_1 \\
\end{array}
\]
Generating the Carries
There are many CLA designs
- We’ve seen a Brent-Kung CLA
- Several other types (e.g., Kogge-Stone)
Carry-Lookahead Adder Takeaways

- 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
Carry-Lookahead Adder Takeaways

- 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.
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.
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-lookahead adders perform $\Theta(\log n)$ addition with modest area cost. This technique can be used to optimize a broad class of circuits.
Thank you!

Next lecture: CMOS