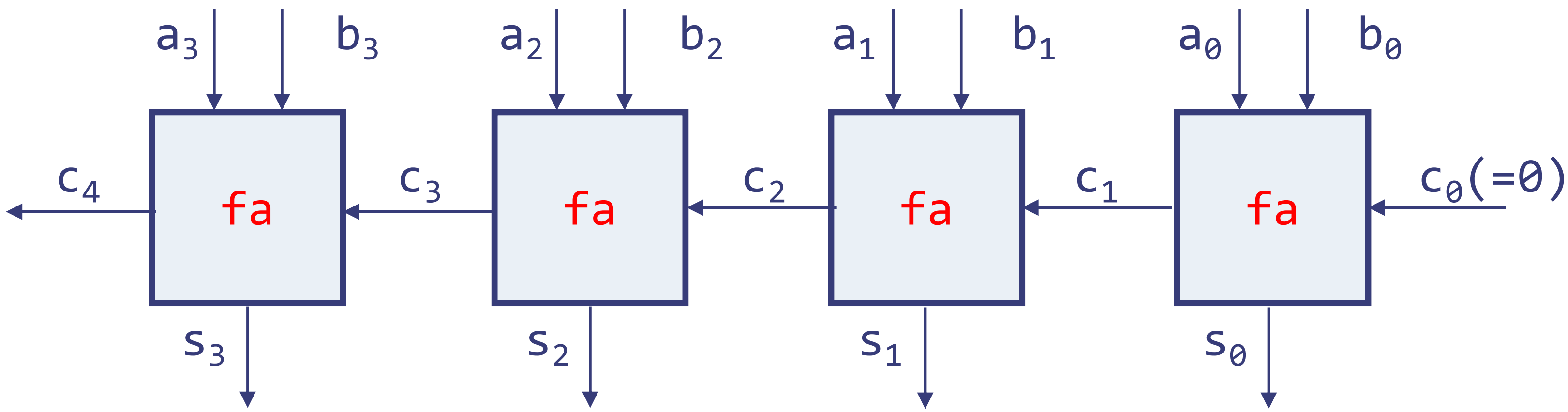
Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Ripple-carry adder

- Cascade FAs to perform binary addition



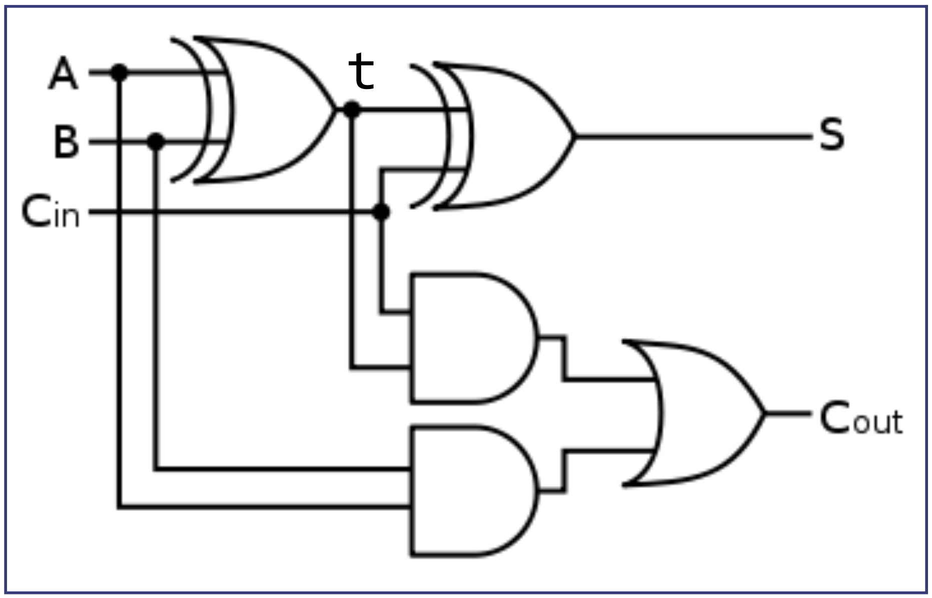
Full Adder: A one-bit adder

Boolean Equations

$t = a \oplus b$

$s = t \oplus c_{in}$

$c_{out} = a \cdot b + c_{in} \cdot t$




```
function fa(a, b, c_in);
   t = (a ^ b);
   s = t ^ c_in;
   c_out = (a & b) | (c_in & t);
   result[0] = s;
   result[1] = c_out;
   return result;
endfunction
```

Structural code – only specifies interconnection between boxes

Not quite correct - needs type annotations

Full Adder: A one-bit adder
corrected

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(1) t;
   Bit#(1) s;
   Bit#(1) c_out;
   Bit#(2) result;
   t = a ^ b;
   s = t ^ c_in;
   c_out = (a & b) | (c_in & t);
   result[0] = s;
   result[1] = c_out;
   return result;
endfunction
```

Type declaration "`Bit#(1) a`" says that `a` is one bit wide

Full Adder *more convenient syntax*

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(1) t = a ^ b;
   Bit#(1) s = t ^ c_in;
   Bit#(1) c_out = (a & b) | (c_in & t);


    return {c_out, s};
endfunction
```

A variable's type declaration and definition can be combined

`{c_out,s}` represents bit concatenation; can be used to avoid naming intermediate results

How big is `{c_out,s}`?

Type checking

- The Bluespec compiler checks if all the declared types are used consistently

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(1) t = a ^ b;
   Bit#(1) s = t ^ c_in;
   Bit#(1) c_out = (a & b) | (c_in & t);
   return {c_out, s};
endfunction
```

3

The compiler will flag this as an error because {c_out, abc[0]} is Bit#(2)

- In fact, the compiler can reduce the programmer's burden by deducing some types and not asking for explicit type declarations

⇒ The "let" syntax

"let" syntax

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   Bit#(1) t = a ^ b;
   Bit#(1) s = t ^ c_in;
   Bit#(1) c_out = (a & b) | (c_in & t);
   return {c_out, s};
endfunction
```

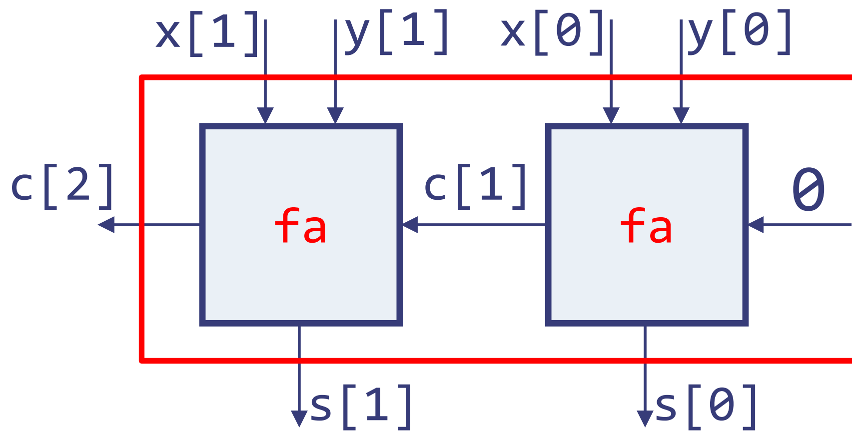
```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
   let t = a ^ b;
   let s = t ^ c_in;
   let c_out = (a & b) | (c_in & t);
   return {c_out, s};
endfunction
```

Types of t, s and c_out can be deduced from the types of the corresponding right-hand-side expressions

"let" syntax is very convenient, we will use it extensively

2-bit Ripple-Carry Adder

cascading full adders



Use fa as a black-box

```
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
   let s = 2b'00;        Bit#(3) c = 3b'000;
   c[0] = 0;
   let cs0 = fa(x[0], y[0], c[0]);
   s[0] = cs0[0];   c[1] = cs0[1];
   let cs1 = fa(x[1], y[1], c[1]);
   s[1] = cs1[0];   c[2] = cs1[1];
return {c[2],s};
endfunction
```

s has two wires, initially each s wire is zero

wire s[0] is updated

wire s[1] is updated

The same as writing {c[2],s[1],s[0]};

32-bit Ripple-Carry Adder (RCA)

- We could have written the chain of RCA explicitly, but we can also use loops!

```
function Bit#(33) add32(Bit#(32) x, Bit#(32) y, Bit#(1) c0);
   Bit#(32) s = 0;
   Bit#(33) c = 0;
   c[0] = c0;
   for (Integer i=0; i<32; i=i+1) begin
      Bit#(2) cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1];
      s[i] = cs[0];
   end
   return {c[32],s};
endfunction
```

Now we discuss how the gates are generated (synthesized) from a loop

Loop is unfolded by the compiler

```
for(Integer i=0; i<32; i=i+1) begin
   Bit#(2) cs = fa(x[i], y[i], c[i]);
   c[i+1] = cs[1];
   s[i] = cs[0];
end
```

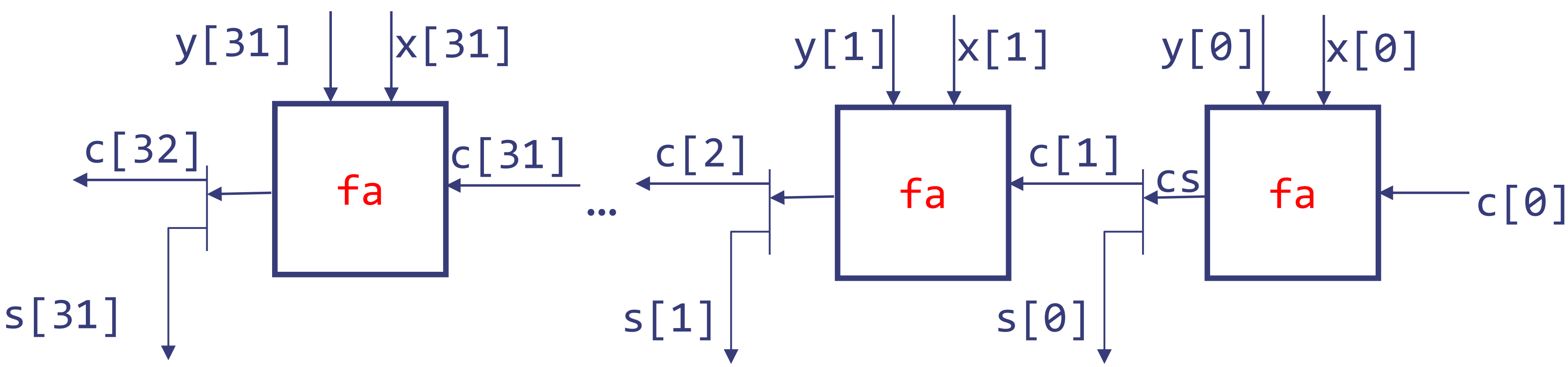
```
cs = fa(x[0], y[0], c[0]);
c[1] = cs[1];                      i = 0
s[0] = cs[0];
cs = fa(x[1], y[1], c[1]);
c[2] = cs[1];                      i = 1
s[1] = cs[0];
...
cs = fa(x[31], y[31], c[31]);
c[32] = cs[1];                     i = 31
s[31] = cs[0];
```

cs in the loop body is a local variable. Hence each of these cs refers to a different value. We could have named them cs0, … cs31.

Loops to gates

```
cs0 = fa(x[0], y[0], c[0]); c[1]=cs0[1]; s[0]=cs0[0];
cs1 = fa(x[1], y[1], c[1]); c[2]=cs1[1]; s[1]=cs1[0];
…
cs31 = fa(x[31], y[31], c[31]);
                    c[32] = cs31[1]; s[31] = cs31[0];
```

Unfolded loop defines an acyclic wiring diagram



Each instance of function fa is replaced by its body

Types in Bluespec

Types

- Every expression in a Bluespec program has a type
- A type is a *grouping* of values, examples
  - Bit#(16)   // 16-bit wide bit-vector (16 is a numeric type)
  - Bool   // 1-bit value representing True or False
  - Vector#(16,Bit#(8)) // Vector of size 16 containing Bit#(8)'s
- A type declaration can be parameterized by other types using the syntax '#', for example
  - `Bit#(n)` represents n bits, e.g., `Bit#(8)`, `Bit#(32)`, `...`
  - `Tuple2#(Bit#(8), Integer)` represents a pair of 8-bitvector and an integer.
  - `function Bit#(8) fname (Bit#(8) arg)` represents a function from Bit#(8) to Bit#(8) values
- A *type name* always begins with a capital letter, while a *variable identifier* begins with a small letter

Type synonyms

```
typedef Bit#(8) Byte;

typedef Bit#(32) Word;

typedef Tuple2#(a,a) Pair#(type a);

typedef 32 DataSize;

typedef Bit#(DataSize) Data;
```

type variable

numeric type

Enumerated types

A very useful typing concept

- Suppose we have a variable c whose values can represent three different colors
  - Declare the type of c to be Bit#(2) and adopt the convention that 00 represents Red, 01 Blue and 10 Green
- A better way is to create a new type called `Color`:

```
typedef enum {Red, Blue, Green}
Color deriving(Bits, Eq);
```

Why is this way better?

- Bluespec compiler automatically assigns a bit representation to the three colors and provides a function to test whether two colors are equal
- If you do not use “deriving” then you will have to specify your own encoding and equality function

Types prevent us from mixing colors with raw bits

Parameterized Circuits

n-bit Ripple-Carry Adder

```
function Bit#(n+1) addN(Bit#(n) x, Bit#(n) y, Bit#(1) c0);
   Bit#(n) s = 0;
   Bit#(n+1) c = 0;
   c[0] = c0;
   for (Integer i=0; i<n; i=i+1) begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1];
      s[i] = cs[0];
   end
   return {c[n],s};
endfunction
```

Now can instantiate different sized adders by specifying n

Unfortunately, there are several subtle type errors in this program – we will fix them one by one

Fixing the type errors

Parameterized Ripple-Carry Adder

```
function Bit#(n+1) addN(Bit#(n) x, Bit#(n) y, Bit#(1) c0);
   Bit#(n) s = 0;
   Bit#(n+1) c = 0;
   c[0] = c0;
   for (Integer i=0; i<n; i=i+1) begin
      let cs = fa(x[i],y[i],c[i]);
      c[i+1] = cs[1];
      s[i] = cs[0];
   end
   return {c[n],s};
endfunction
```

- n is numeric type and Bluespec does not allow arithmetic on types, e.g., n+1, i<n, c[n] are illegal!

Fixing the type errors
`valueOf(n)` versus `n`

- Each expression has a *type* and a *value*, and these two come from entirely disjoint worlds
- `n` in `Bit#(n)` is a *numeric type* variable and resides in the types world
- Sometimes we need to use values from the types world in actual computation. The function `valueOf` extracts the integer from a numeric type
  - Thus,

    `i<n` is not type correct

    `i<valueOf(n)` is type correct

Fixing the type errors
`TAdd#(n,1)` versus `n+1`

- Sometimes we need to perform operations in the types world that are very similar to the operations in the value world
  - Examples: Addition, Multiplication, Logarithm base 2, ...
- Bluespec defines a few special operators in the types world for such operations
  - `TAdd#(m,n), TSub#(m,n), TMul#(m,n), TDiv#(m,n), TLog#(n), TExp#(n), TMax#(m,n), TMin#(m,n)`
  - Thus,

    `Bit#(n+1)` is not type correct

    `Bit#(TAdd#(n,1))` is type correct

Parameterized Ripple-Carry Adder
corrected

```
function Bit#(TAdd#(n,1)) addN(Bit#(n) x, Bit#(n) y,
                                        Bit#(1) c0);
   Bit#(n) s = 0;
   Bit#(TAdd#(n,1)) c;
   c[0] = c0;
   let valn = valueOf(n);
   for (Integer i=0; i<valn; i=i+1) begin
      let cs = fa(x[i], y[i], c[i]);
      c[i+1] = cs[1];
      s[i] = cs[0];
   end
   return {c[valn],s};
endfunction
```

types world equivalent of n+1

Lifting a type into the value world

Instantiating the parametric Adder

```
function Bit#(Tadd#(n,1)) addN(Bit#(n) x, Bit#(n) y, Bit#(1) c0);
```

How do we define a add32, add3 … using addN ?

```
// concrete instances of addN!
function Bit#(33) add32(Bit#(32) x, Bit#(32) y, Bit#(1) c0)
                                = addN(x,y,c0);

function Bit#(4) add3(Bit#(3) x, Bit#(3) y, Bit#(1) c0)
                                = addN(x,y,c0);
```

The numeric type n on the RHS implicitly gets instantiated to 32 and 3, respectively, because of the LHS declaration

Bluespec is for describing circuits

- Bluespec is like a language for drawing pictures of interconnected boxes
- Boxes happen to be Boolean gates with inputs and outputs
- However, unlike ordinary pictures, our boxes, i.e., gates, have computational meaning, and therefore, we can ask what values a circuit would produce on its output lines, given a specific set of values on its input lines
- Even though the primary purpose of the Bluespec compiler is to synthesize a network of gates, the ability to simulate the functionality of the resulting circuit is extremely important

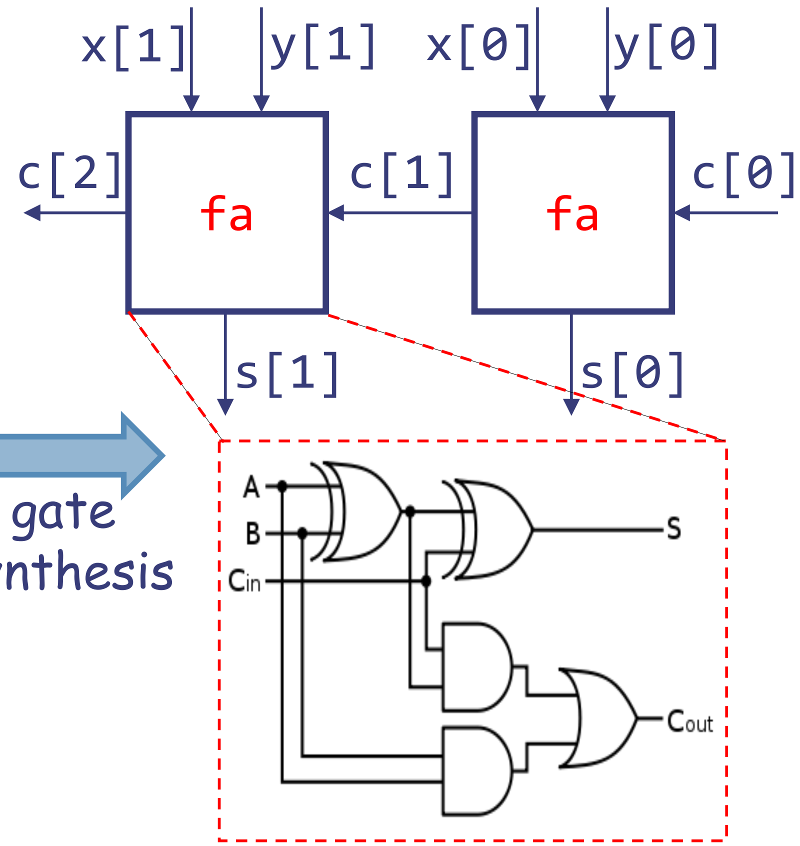
Bluespec: Gate synthesis versus simulation 2-bit adder

```
function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
   let s = 2b'00;     Bit#(3) c = 3b'000;
   c[0] = 0;
   let cs0 = fa(x[0], y[0], c[0]);
   s[0] = cs0[0];    c[1] = cs0[1];
   let cs1 = fa(x[1], y[1], c[1]);
   s[1] = cs1[0];    c[2] = cs1[1];
return {c[2],s};
endfunction
```

simulate

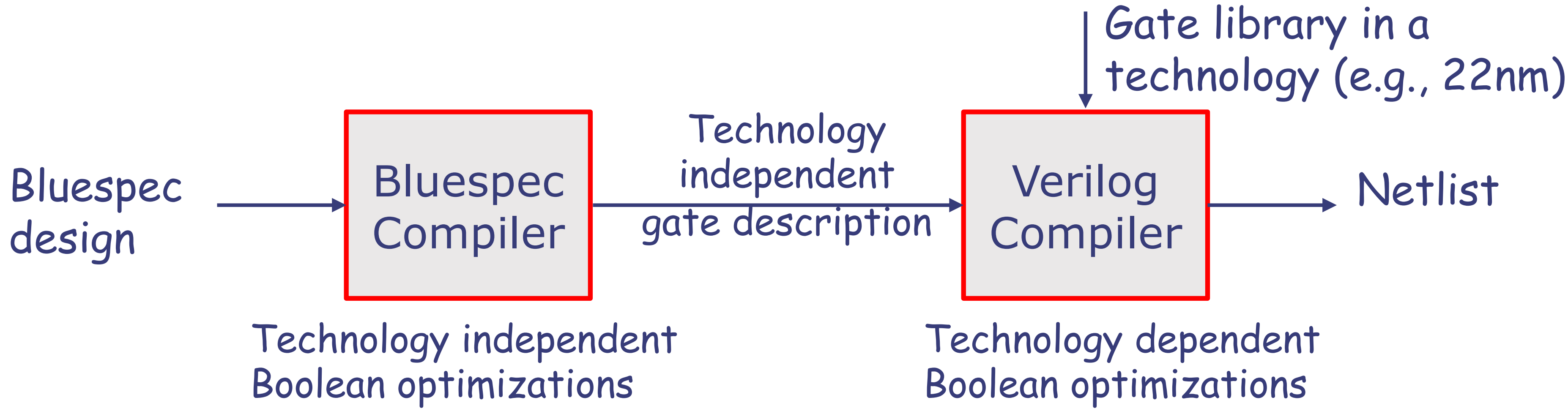
- add2(2'b11, 2'b01) ⇒ 3'b100
- add2(2'b01, 2'b01) ⇒ 3'b010

gate synthesis



Caution: In spite of the fact that Bluespec programs, like programs in other software languages, produce outputs given inputs, the purpose of Bluespec programs is to describe circuits

Compiling Bluespec into circuits



- *Static elaboration:* Bluespec compiler eliminates all constructs which have no direct hardware meaning
  - All data structures are converted into bit vectors
  - Loops are unfolded
  - Functions are in-lined
  - What remains is an acyclic graph of Boolean gates
    - The compiler complains if it detects a cycle in your circuit

Takeaway

- Once we define a combinational circuit, we can use it repeatedly to build larger circuits
- Bluespec compiler, because of the type signatures of functions, prevents us from connecting functions and gates in obviously illegal ways
- We can use loop constructs and functions to express combinational circuits, but all loops are unfolded and functions are in-lined during the compilation phase
- We can also write parameterized circuits in Bluespec, for example an n-bit adder. Once n is specified, the correct circuit is automatically generated

The best way to learn about types is to try writing a few expressions and feeding them to the compiler