Complex Combinational circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
2-bit Ripple-Carry Adder

cascading full adders

function Bit#(3) add2(Bit#(2) x, Bit#(2) y);
  Bit#(2) s = 0;   Bit#(3) c = 0;
  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

How to generalize this to a w-bit adder?
w-bit Ripple-Carry Adder

- For a parameterized w-bit adder, we cannot write a straight-line program as we did for the 2-bit adder
- Use loops!

```verbatim
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);
  Bit#(w) s = 0;
  Bit#(w+1) c = 0;
  c[0] = c0;
  for (Integer i=0; i<w; 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[w],s};
endfunction
```

This program has some subtle type errors in but before fixing them, we will discuss how the gates are generated (synthesized) from a loop.
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);
    Bit#(2) s = 0;    Bit#(3) c = 0;
    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

- add2(2’b11, 2’b01) \Rightarrow 3’b100
- add2(2’b01, 2’b01) \Rightarrow 3’b010

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: 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 actually generates Verilog, which is synthesized first into a generic network of gates and then later mapped into specific gates provided by the library of a specific hardware technology
  - Both the Bluespec compiler and the Verilog compiler do many different Boolean optimizations with the intention of reducing the circuit area, the propagation delay or both
Back to our w-bit ripple carry adder

```plaintext
for (Integer i=0; i<w; i=i+1) begin
    let cs = fa(x[i], y[i], c[i]);
    c[i+1] = cs[1];
    s[i] = cs[0];
end
```

Unfold the loop

Can be done only when the value of w is known

Suppose w = 3

```plaintext
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];
cs2 = fa(x[2], y[2], c[2]);
c[3] = cs2[1];
s[2] = cs2[0];
```

- i = 0
- i = 1
- i = 2
Loops to gates

Unfolded loop defines an acyclic wiring diagram

Each instance of function \( fa \) is replaced by its body
Instantiating the parametric Adder

- How do we define concrete instances like add3, add32 ... using addN?

```plaintext
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);

function Bit#(4) add3(Bit#(3) x, Bit#(3) y, Bit#(1) c0);
    return addN(x, y, c0);
endfunction

function Bit#(33) add32(Bit#(32) x, Bit#(32) y, Bit#(1) c0);
    return addN(x, y, c0);
endfunction
```

- The numeric type w on the right-hand side (RHS) implicitly gets instantiated to 3 because of the left-hand side (LHS) declarations
- Similarly,
Introduction to Types in Bluespec
Types

- A type is a *grouping* of values, examples
  - `Bit#(16)` // 16-bit wide bit-vector (16 is a numeric type)
  - `bit [15:0]` // synonym for `Bit#(16)`
  - `Bool` // 1-bit value representing True and False
  - `UIInt#(32)` // unsigned integers, 32 bits wide
  - `Vector#(16,Int#(8))` // Vector of size 16 containing `Int#(8)`'s

- Every expression in a Bluespec program has a type; sometimes it is specified explicitly and sometimes it is deduced by the compiler
  - Thus, we say an expression has a type or belongs to a type

- Compiler assigns a bit representation to every typed value that is implementable in hardware
  - `pack`: converts a typed value into bits
  - `unpack`: converts bits into a desired type value
Parameterized types: 

- A type declaration can be parameterized by other types using the syntax `#`, for example
  - `Bit#(n)` represents n bits and can be instantiated by specifying a value of n
    - `Bit#(1), Bit#(32), Bit#(8), ...`
  - `Tuple2#(Integer, Integer)` represents a pair of Integers
  - `function Integer fname (Integer arg)` represents a function from Integers to Integers and is named `fname`

- A *type name* begins with a capital letter, except for a numeric type which is just a number
  - `Bool, Bit#(32), Int#(32), Integer, ...`

- A *type variable identifier* always begins with a small letter
  - `t, wordSize, n, ...`
Type synonyms

```plaintext
typedef Bit#(8) Byte;
typedef Bit#(32) Word;
typedef Tuple2#(a,a) Pair#(type a);
typedef 32 DataSize;
typedef Bit#(DataSize) Data;
```

*type variable*
Enumerated types

- 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);
  ```
- 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

Why is this way better?
Type declaration versus deduction

- The programmer writes down types of some expressions in a program and the compiler infers the types of the rest of expressions.
- If the type inference cannot be performed or the type declarations are inconsistent then the compiler complains.

```
function Bit#(2) fa(Bit#(1) a, Bit#(1) b, Bit#(1) c_in);
  let ab = ha(a, b);
  let abc = ha(ab[0], c_in);
  Bit#(2) c_out = ab[1] | abc[1]; /* type error? */
  return {c_out, abc[0]};
endfunction
```

Type checking prevents lots of silly mistakes.
function Bit#(w+1) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);
    Bit#(w) s = 0;
    Bit#(w+1) c = 0;
    c[0] = c0;
    for (Integer i=0; i<w; 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[w],s};
endfunction

There are several subtle type errors in this program - now we will fix them one by one
Fixing the type errors

**valueOf(w) versus w**

- Each expression has a type and a value, and these two come from entirely disjoint worlds
- `w` in `Bit#(w)` is a numeric type variable and resides in the types world
- Sometimes we need to use values from the types world into actual computation. The function `valueOf` extracts the integer from a numeric
  - Thus,
    - `i<w` is not type correct
    - `i<valueOf(w)` is type correct
Fixing the type errors

TAdd(#(w,1)) versus w+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
- We define a few special operators in the types world for such operations
  - Examples: TAdd#(m,n), TMul#(m,n), ...
  - Thus,
    - Bit#(w+1) is not type correct
    - Bit#(TAdd#(w,1)) is type correct
A w-bit Ripple-Carry Adder

corrected

function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);

Bit#(w) s = 0;
Bit#(TAdd#(w,1)) c;
c[0] = c0;
let valw = valueOf(w);
for (Integer i=0; i<valw; 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[valw],s};
endfunction
Multiplication by repeated addition

\[
\begin{array}{cccc}
\text{b Multiplicand} & 1101 & (13) \\
\text{a Multiplier} & 1011 & (11) \\
\hline
\text{tp} & 0000 \\
\text{m0} & + 1101 \\
\hline
\text{tp} & 01101 \\
\text{m1} & + 1101 \\
\hline
\text{tp} & 100111 \\
\text{m2} & + 0000 \\
\hline
\text{tp} & 0100111 \\
\text{m3} & + 1101 \\
\hline
\text{tp} & 10001111 & (143)
\end{array}
\]

At each step we add either 1101 or 0 to the result depending upon a bit in the multiplier

\[
mi = (a[i]==0)? 0 : b;
\]

We also shift the result by one position at every step.

Notice, the first addition is unnecessary because it simply yields \(m_0\)
Multiplication by repeated addition circuit

b Multiplicand  1101  (13)
a Multiplier    *  1011  (11)

\[
\begin{array}{c}
tp \\
m0 \\
tp \\
m1 \\
tp \\
m2 \\
tp \\
m3 \\
\end{array}
\begin{array}{c}
+ 0000 \\
+ 1101 \\
01101 \\
+ 1101 \\
100111 \\
+ 0000 \\
0100111 \\
+ 1101 \\
10001111  (143)
\end{array}
\]

\[
m_i = (a[i]==0)? 0 : b;
\]
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
    Bit#(32) tp = 0;
    Bit#(32) prod = 0;
    for (Integer i = 0; i < 32; i = i+1)
        begin
            Bit#(32) m = (a[i] == 0)? 0 : b;
            Bit#(33) sum = add32(m, tp, 0);
            prod[i] = sum[0];
            tp = sum[32:1];
        end
    return {tp, prod};
extendfunction
Analysis of 32-bit multiply

```plaintext
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
    Bit#(32) tp = 0;
    Bit#(32) prod = 0;
    for (Integer i = 0; i < 32; i = i+1)
        begin
            Bit#(32) m = (a[i] == 0)? 0 : b;
            Bit#(33) sum = add32(m, tp, 0);
            prod[i] = sum[0];
            tp = sum[32:1];
        end
    return {tp, prod};
endfunction
```

- Can we design a faster adder?
  - yes!
- Can we reuse the adder circuit and reduce the size of the multiplier
  - stay tuned ...

Long chains of gates
- 32-bit multiply has 32 ripple carry adders in sequence!
- 32-bit ripple carry adder has a 32-long chain of gate

Take home problem: What is the propagation delay of mul32 in terms of FA delays?
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 write parameterized circuits in Bluespec, for example an n-bit adder. Once n is specified, the correct circuit is automatically generated.
- 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.

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