Binary Arithmetic Circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
A full adder adds two one-bit numbers and a carry and produces a sum bit and a carry bit.

A full adder is itself implemented using half adders which add two one-bit numbers and produce a sum bit and a carry bit.
function Bit#(3) add2(Bit#(2) x, Bit#(2) y, Bit#(1) c0);
Bit#(2) s = 0;
Bit#(3) c = 0;
c[0] = c0;
let cs0 = fa(x[0], y[0], c[0]);
c[1] = cs0[1];
s[0] = cs0[0];
let cs1 = fa(x[1], y[1], c[1]);
c[2] = cs1[1];
s[1] = cs1[0];
return {c[2], s};
endfunction

Use fa as a black-box
Assigning to Multi-Bit Words

Means \( c \) is three bits wide and each element is set to zero

\[ c[0] = c0; \]

\( c0 \) is assigned to element 0 of \( c \), but the value of the rest of the elements is not affected

Each bit in a multi-bit word must have an initial value
- An attempt to use uninitialized bits will raise a compiler warning and result in unexpected behavior
An 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!

```plaintext
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 some subtle type errors in this program but before we fix them, you may wonder what is the meaning of a loop in terms of gates?
All loops are unfolded by the compiler!

```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

Can be done only when the value of w is known

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];
...
csw = fa(x[w-1], y[w-1], c[w-1]);
c[w] = csw[1];
s[w-1] = csw[0];
```
Loops to gates

\[
cs_0 = fa(x[0], y[0], c[0]); c[1]=cs_0[1]; s[0]=cs_0[0];
\]
\[
cs_1 = fa(x[1], y[1], c[1]); c[2]=cs_1[1]; s[1]=cs_1[0];
\]
\[
...\]
\[
cs_w = fa(x[w-1], y[w-1], c[w-1]);
\]
\[
c[w] = cs_w[1]; s[w-1] = cs_w[0];
\]

Unfolded loop defines an acyclic wiring diagram
**Instantiating the parametric Adder**

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

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

function Bit#(4) add3(Bit#(3) x, Bit#(3) 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,

function Bit#(33) add32(Bit#(32) x, Bit#(32) y, Bit#(1) c0);
    return addN(x, y, c0);
endfunction
```
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#$w$ 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
    $i<\text{valueOf}(w)$ 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

Structural interpretation of a loop – unfold it to generate an acyclic graph

types world equivalent of w+1

Lifting a type into the value world
Can you design a comp function $f$ so that it can be chained together using a loop to form a comparator for $n$-bit unsigned numbers.

It should return:
- EQ if $a = b$,
- LT if $a < b$,
- GT if $a > b$

```verbatim
function CompResult comp
  (Bit#(n) a, Bit#(n) b);
  // chain f calls together
endfunction
```

```verbatim
typedef enum {EQ, LT, GT} CompResult deriving (Bits, Eq);
```
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:
```cpp
typedef enum {Red, Blue, Green} Color deriving(Bits, Eq);
```

BSV compiler automatically assigns a bit representation to the three colors and provides a function to test if the two colors are equal.

Why is this way better?

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.
Multiplication by repeated addition

\[ b \text{ Multiplicand } \quad 1101 \quad (13) \]
\[ a \text{ Multiplier } \quad * \quad 1011 \quad (11) \]

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

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

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

- Multiplicand: 1101 (13)
- Multiplier: 1011 (11)

```
\[ mi = (a[i] == 0)? 0 : b; \]
```
Combinational 32-bit multiply

```haskell
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
```

This circuit uses 32 add32 circuits
Lot of gates!

Can we do better?
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 gates
- Is total delay less than nxn FA delays? Yes, 3n-2 FA delays!
BSV Compiling phases

- Type checking: Ensures that type of each expression can be determined uniquely; Otherwise the program is rejected

- Static elaboration: Compiler eliminates all constructs which have no direct hardware meaning
  - Loops are unfolded
  - Functions are in-lined
  - After this stage the program does not contain any Integers because Integers are unbounded in BSV

- Gates are generated (actually Verilog)
Takeaway

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