Complex Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology
Selecting a wire: \(x[i]\)

- **Constant selector:** e.g., \(x[2]\)

  - \(x_0\)  
  - \(x_1\)  
  - \(x_2\)  
  - \(x_3\)

  - \(x_0\)  
  - \(x_1\)  
  - \(x_2\)  
  - \(x_3\)

  No hardware; \(x[2]\) is just the name of a wire

- **Dynamic selector:** \(x[i]\)

  - \(x_0\)  
  - \(x_1\)  
  - \(x_2\)  
  - \(x_3\)

  - \(x_0\)  
  - \(x_1\)  
  - \(x_2\)  
  - \(x_3\)

  4-way mux

Assume \(x\) is 4 bits wide.
A 2-way multiplexer

A mux is a simple conditional expression

Bluespec:  \((p)? \ b : \ a \ ;\)

True is treated as a 1 and False as a 0

Gate-level implementation

If \(a\) and \(b\) are \(n\)-bit wide then this structure is replicated \(n\) times; \(p\) is the same input for all the replicated structures
A 4-way multiplexer

```vhdl
case \{s1, s0\}
2'b00 : a;
2'b01 : b;
2'b10 : c;
2'b11 : d;
endcase
```

Syntax: writing 0, 1, 2, 3 would have also worked

(s1==0) & (s0==1), which is the same writing \~s1 & s0

n-way mux can be implemented using n-1 two-way muxes
Shift operators

- Fixed size shift operation is cheap in hardware – just wire the circuit appropriately

- Arithmetic shifts are similar

  Logical right shift by 2

  $$\begin{array}{cccc}
  1 & 0 & 0 & 1 \\
  0 & 0 & 1 & 0 \\
  \end{array}$$

  $$\begin{array}{cccc}
  a & b & c & d \\
  0 & 0 & a & b \\
  \end{array}$$

  -8/4

  Logical right shift by 2

  $$\begin{array}{cccc}
  1 & 0 & 0 & 0 \\
  \end{array}$$

  $$\begin{array}{cccc}
  a & b & c & d \\
  \end{array}$$

  useful for multiplication and division by $$2^n$$

  $$\begin{array}{cccc}
  1 & 1 & 1 & 0 \\
  \end{array}$$

  $$\begin{array}{cccc}
  a & a & a & b \\
  \end{array}$$

  -2
Logical right shift by \( n \)

- Shift \( n \) can be broken down into \( \log n \) steps of fixed-length shifts of size 1, 2, 4, ...
  - The bit encoding of \( n \) tells us which shifters are needed; if the value of the \( i^{th} \) (least significant) bit is 1 then we need to shift by \( 2^i \) bits
  - For example, we can perform shift 5 (=4+1) by doing shifts of size 4 and 1. Thus, 8’b01100111 shift 5 can be performed in two steps:
    - 8’b01100111 \( \Rightarrow \) 8’b00000110 \( \Rightarrow \) 8’b00000011
      - Shift 4
      - Shift 1
Conditional operation: shift versus no-shift

- We need a mux to select the appropriate wires: if \( s \) is one the mux will select the wires on the left (shift) otherwise it would select wires on the right (no-shift)

\[
\begin{align*}
&s = 1 \Rightarrow \{2'b0, a, b\} : \{a, b, c, d\};
\end{align*}
\]
Logical right shift circuit

- Define log \( n \) shifters of sizes 1, 2, 4, ...
- Define log \( n \) muxes to perform a particular size shift
- Suppose \( n = \{n1, n0\} \) is a two bit number. A shift by \( n \) can be expressed as two conditional expressions where the second uses the output of the first

```
Bit#(4) input = {a, b, c, d}
Bit#(4) tmp = (s1==1)? {2'b0, a, b}:input;
Bit#(4) output = (s0==1)? {1'b0, tmp[3], tmp[2], tmp[1]}:tmp;
```
Multiplication by repeated addition

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

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 m0
Multiplication by repeated addition circuit

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

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

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

```verbatim
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
```

Lot of gates!

This circuit uses 32 add32 circuits

Lot of gates!
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 a sequence!
- 32-bit ripple carry adder has a 32-long chain of gates

Take home problem: What is the propagation delay of mul32 in terms of FA delays?
All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...
4-way Butterfly Node

```typescript
function Vector#(4,Complex) bfly4
  (Vector#(4,Complex) t,  Vector#(4,Complex) x);
```

- t’s (twiddle coefficients) are mathematically derivable constants for each bfly4 and depend upon the position of bfly4 the in the network.
function Vector#(4, Complex#(s)) bfly4
    (Vector#(4, Complex#(s)) t, Vector#(4, Complex#(s)) x);

    Vector#(4, Complex#(s)) m, y, z;
    m[0] = x[0] * t[0]; m[1] = x[1] * t[1];

    y[0] = m[0] + m[2]; y[1] = m[0] - m[2];

    z[0] = y[0] + y[2]; z[1] = y[1] + y[3];

    return(z);
endfunction

Polymorphic code: works on any type of numbers for which *, + and - have been defined

Vector does not mean storage; a vector is just a group of wires with names
Language notes: Sequential assignments

- Sometimes it is convenient to reassign a variable (x is zero everywhere except in bits 4 and 8):

  Bit#(32) x = 0;
  x[4] = 1; x[8] = 1;

- This may result in the introduction of muxes in a circuit:

  Bit#(32) x = 0;
  let y = x+1;
  if (p) x = 100;
  let z = x+1;
Complex Arithmetic

- **Addition**
  - $z_R = x_R + y_R$
  - $z_I = x_I + y_I$

- **Multiplication**
  - $z_R = x_R * y_R - x_I * y_I$
  - $z_I = x_R * y_I + x_I * y_R$
Representing complex numbers as a `struct`

typedef struct{
    Int#(t) r;
    Int#(t) i;
} Complex#(numeric type t) deriving (Eq,Bits);

- Notice the Complex type is parameterized by the size of Int chosen to represent its real and imaginary parts.

- If x is a struct then its fields can be selected by writing x.r and x.i.
typedef struct{
    Int#(t) r;
    Int#(t) i;
} Complex#(numeric type t) deriving (Eq,Bits);

function Complex#(t) cAdd (Complex#(t) x, Complex#(t) y);
    Int#(t) real = x.r + y.r;
    Int#(t) imag = x.i + y.i;
    return(Complex{r:real, i:imag});
endfunction
Overloading (Type classes)

- The same symbol can be used to represent different but related operators using Type classes
- A type class groups a bunch of types with similarly named operations. For example, the type class Arith requires that each type belonging to this type class has operators +, -, *, / etc. defined
- We can declare Complex type to be an instance of Arith type class
Overloading +, *

```markdown
instance Arith#(Complex#(t));
function Complex#(t) +
    (Complex#(t) x, Complex#(t) y);
    Int#(t) real = x.r + y.r;
    Int#(t) imag = x.i + y.i;
    return(Complex{r:real, i:imag});
endfunction

function Complex#(t) *
    (Complex#(t) x, Complex#(t) y);
    Int#(t) real = x.r*y.r - x.i*y.i;
    Int#(t) imag = x.r*y.i + x.i*y.r;
    return(Complex{r:real, i:imag});
endfunction
```

... The context allows the compiler to pick the appropriate definition of an operator
Combinational IFFT

function Vector#(64, Complex#(n)) stage_f
  (Bit#(2) stage, Vector#(64, Complex#(n)) stage_in);

function Vector#(64, Complex#(n)) ifft
  (Vector#(64, Complex#(n)) in_data);

repeats stage_f three times
BSV Code: Combinational IFFT

```haskell
function Vector#(64, Complex#(n)) ifft
    (Vector#(64, Complex#(n)) in_data);

//Declare vectors
    Vector#(4,Vector#(64, Complex#(n))) stage_data;
    stage_data[0] = in_data;
    for (Bit#(2) stage = 0; stage < 3; stage = stage + 1)
        stage_data[stage+1] = stage_f(stage,stage_data[stage]);
    return(stage_data[3]);
endfunction
```

The for-loop is unfolded and stage_f is in-lined during static elaboration

No notion of loops or procedures during execution
function Vector#(64, Complex#(n)) stage_f
    (Bit#(2) stage, Vector#(64, Complex#(n)) stage_in);
Vector#(64, Complex#(n)) stage_temp, stage_out;
   for (Integer i = 0; i < 16; i = i + 1)
      begin
         Integer idx = i * 4;
         Vector#(4, Complex#(n)) x;
         x[0] = stage_in[idx];  x[1] = stage_in[idx+1];
         x[2] = stage_in[idx+2]; x[3] = stage_in[idx+3];
         let twid = getTwiddle(stage, fromInteger(i));
         let y = bfly4(twid, x);
         stage_temp[idx]   = y[0]; stage_temp[idx+1] = y[1];
         stage_temp[idx+2] = y[2]; stage_temp[idx+3] = y[3];
      end
   //Permutation
   for (Integer i = 0; i < 64; i = i + 1)
      stage_out[i] = stage_temp[permute[i]];
return(stage_out);
endfunction

BSV Code for stage_f

twid's are mathematically derivable constants
Permute

- `permute[i]` specifies the destination index for each source index
- Even though the permute is known at compile time, the BSV compiler takes too long to inline array indices
- A better way to supply the permute function

```hs
function Integer permute (Integer dst, Integer points);
    Integer src = ?;
    if (dst < points/2) src = dst*2;
    else src = (dst - points/2)*2 + 1;
    return src;
endfunction
```