Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 13, 2013
http://csg.csail.mit.edu/6.375

Combinational circuits are acyclic interconnections of gates

- And, Or, Not
- Nand, Nor, Xor
- ...

February 13, 2013
http://csg.csail.mit.edu/6.375
Arithmetic-Logic Unit (ALU)

ALU performs all the arithmetic and logical functions

Each individual function can be described as a combinational circuit

Full Adder: A one-bit adder

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

Structural code - only specifies interconnection between boxes
Full Adder: A one-bit adder

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

How big is `{c_out, s}`?

Types

- A type is a grouping of values:
  - Integer: 1, 2, 3, ...
  - Bool: True, False
  - Bit: 0, 1
  - A pair of Integers: Tuple2#(Integer, Integer)
  - A function `fname` from Integers to Integers:
    ```verilog
    function Integer fname (Integer arg)
    ```

- Every expression and variable in a BSV 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

The type of each expression is unique
Type declaration versus deduction

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

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

2-bit Ripple-Carry Adder

```verilog
function Bit#(3) add(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
```

The “let” syntax avoids having to write down types explicitly.

fa can be used as a black-box long as we understand its type signature.
“let” syntax

- The “let” syntax: avoids having to write down types explicitly
  - let cs0 = fa(x[0], y[0], c[0]);
  - Bits#(2) cs0 = fa(x[0], y[0], c[0]);

The same

Parameterized types: #

- A type declaration itself can be parameterized by other types
- Parameters are indicated by 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), …
An w-bit Ripple-Carry Adder

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

Unfold the loop to get the wiring diagram

```
        x[0]       y[0]       c[0]
        |           |           |
        v           v           v
        c[1] fa     s[0]        fa
        |           |           |
        v           v           v
        c[2] fa     s[1]        ...
        |           |           |
        v           v           v
        c[w-1] fa   s[w-1]      c[w]
```

Instantiating the parametric Adder

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

Define add32, add3 ... using addN

```vhdl
// 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);
```

February 13, 2013
http://csg.csail.mit.edu/6.375
valueOf(w) versus w

- Each expression has a type and a value and these come from two 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 allows us to do that
  - Thus
    - i<w is not type correct
    - i<valueOf(w) is type correct

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: Add, Mul, Log
- We define a few special operators in the types world for such operations
  - Examples: TAdd#(m, n), TMul#(m, n), ...
A w-bit Ripple-Carry Adder

```plaintext
corrected

function Bit#(TAdd#(w,1)) addN(Bit#(w) x, Bit#(w) y, Bit#(1) c0);
    Bit#(w) s; Bit#(TAdd#(w,1)) c = 0; 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
Static Elaboration phase

When BSV programs are compiled, first type checking is done and then the compiler gets rid of many constructs which have no direct hardware meaning, like Integers, loops

```
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
```

In mathematics integers are unbounded but in computer systems integers always have a fixed size

BSV allows us to express both types of integers, though unbounded integers are used only as a programming convenience

```
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
```

Integer versus Int#(32)

In mathematics integers are unbounded but in computer systems integers always have a fixed size

BSV allows us to express both types of integers, though unbounded integers are used only as a programming convenience

```
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
```
Type synonyms

typedef bit [7:0] Byte;

typedef Bit#(8) Byte;

typedef Bit#(32) Word;

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

typedef Int#(n) MyInt#(type n);

typedef Int#(n) MyInt#(numeric type n);

The same

Complex combinational circuits

IFFT
Combinational IFFT

All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

4-way Butterfly Node

$t$’s (twiddle coefficients) are mathematically derivable constants for each bfly4 and depend upon the position of bfly4 the in the network.
BSV code: 4-way Butterfly

```hs
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

Note: Vector does not mean storage; 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):

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

- This will usually result in introduction of muxes in a circuit as the following example illustrates:

```hs
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**
  - \( z_R = x_R \cdot y_R - x_I \cdot y_I \)
  - \( z_I = x_R \cdot y_I + x_I \cdot y_R \)

Representing complex numbers as a struct

```plaintext
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 \( x.i \).
BSV code for Addition

```haskell
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 +, *

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

dehendfunction

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

endinstance

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

Combinational IFFT

in0 in1 in2 in3 in4 in63
Bfly4 Bfly4 Bfly4 x16 Bfly4 Bfly4
Permute Permute Permute Permute
Bfly4 Bfly4 Bfly4...
Permute
Bfly4 Bfly4 Bfly4...
Permute
Bfly4 Bfly4 Bfly4...

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);

repeat stage_f three times

function to be continued
BSV Code: Combinational IFFT

```hsaskell
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 inlined during static elaboration

Note: no notion of loops or procedures during execution

BSV Code for stage_f

```hsaskell
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
```