Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 9, 2011 L03-1
http://csg.csail.mit.edu/6.375

Bluespec: Two-Level Compilation

Bluespec (Objects, Types, Higher-order functions)

Level 1 compilation

Rules and Actions (Term Rewriting System)

Level 2 synthesis

Object code (Verilog/C)

Lennart Augustsson
@Sandburst 2000-2002

Now we call this Guarded Atomic Actions

• Type checking
• Massive partial evaluation and static elaboration

James Hoe & Arvind
@MIT 1997-2000

• Rule conflict analysis
• Rule scheduling
Static Elaboration

At compile time
- Inline function calls and unroll loops
- Instantiate modules with specific parameters
- Resolve polymorphism/overloading, perform most data structure operations

Software Toolflow:
- source
  - compile
  - run w/ params

Hardware Toolflow:
- source
  - elaborate w/params
  - run w/ params

Combinational IFFT

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

February 9, 2011 L03-3
http://csg.csail.mit.edu/6.375

February 9, 2011 L03-4
http://csg.csail.mit.edu/6.375
BSV code: 4-way Butterfly

```vhdl
function Vector#(4,Complex) bfly4 (Vector#(4,Complex) t,  Vector#(4,Complex) k);
    Vector#(4,Complex) m, y, z;
    m[0] = k[0] * t[0]; m[1] = k[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
Complex Arithmetic

- **Addition**
  - \( z_R = x_R + y_R \)
  - \( z_I = x_I + y_I \)

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

The actual arithmetic for FFT is different because we use a non-standard fixed point representation.

BSV code for Addition

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

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
```

What is the type of this +?
Combinational IFFT

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

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

BSV Code: Combinational IFFT

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

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

The for-loop is unfolded and stage_f is inlined during static elaboration

Note: no notion of loops or procedures during execution
BSV Code: Combinational IFFT- Unfolded

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

//Declare vectors
    Vector#{4,Vector#{64, Complex}} stage_data;
    stage_data[0] = in_data;
    for (Integer stage = 0; stage < 3; stage = stage + 1)
        stage_data[stage+1] = stage_f(stage,stage_data[stage]);
    return(stage_data[3]);
```

Bluespec Code for stage_f

```haskell
function Vector#(64, Complex) stage_f
    (Bit#{2} stage, Vector#{64, Complex} stage_in);
begin
    for (Integer i = 0; i < 16; i = i + 1)
        begin
            Integer idx = i * 4;
            let twid = getTwiddle(stage, fromInteger(i));
            let y = bfly4(twid, stage_in[idx:idx+3]);
            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);
```
Higher-order functions:
Stage functions f1, f2 and f3

function f0(x);
    return (stage_f(0, x));
endfunction

function f1(x);
    return (stage_f(1, x));
endfunction

function f2(x);
    return (stage_f(2, x));
endfunction

What is the type of f0(x)?

Suppose we want to reuse some part of the circuit ...

Reuse the same circuit three times to reduce area

But why?
Architectural Exploration:
Area-Performance tradeoff in 802.11a Transmitter

802.11a Transmitter Overview

24 Uncoded bits

Controller Scrambler Encoder
Interleaver Mapper

IFFT

Cyclic Extend

Must produce one OFDM symbol every 4 µsec

IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers
Preliminary results

[MEMOCODE 2006] Dave, Gerding, Pellauer, Arvind

<table>
<thead>
<tr>
<th>Design Block</th>
<th>Lines of Code (BSV)</th>
<th>Relative Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Scrambler</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Conv. Encoder</td>
<td>113</td>
<td></td>
</tr>
<tr>
<td>Interleaver</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>Mapper</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>IFFT</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>Cyc. Extender</td>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

Complex arithmetic libraries constitute another 200 lines of code

Combinational IFFT

Reuse the same circuit three times to reduce area
Design Alternatives

Reuse a block over multiple cycles

we expect:
Throughput to
Area to

Circular pipeline: Reusing the Pipeline Stage
Superfolded circular pipeline:
Just one Bfly-4 node!

Pipelining a block
Inelastic pipeline

Stage functions f1, f2 and f3
Problem: What about pipeline bubbles?

```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f0(inQ.first());
  sReg2 <= f1(sReg1);
  outQ.enq(f2(sReg2));
endrule
```

February 9, 2011  http://csg.csail.mit.edu/6.375  L03-25

The Maybe type data in the pipeline

```
typedef union tagged {
  void Invalid;
  data_T Valid;
} Maybe#(type data_T);
```

```
rule sync-pipeline (True);
  if (inQ.notEmpty())
  begin
    sReg1 <= tagged Valid f0(inQ.first()); inQ.deq();
  end
  else  sReg1 <= tagged Invalid;
  case (sReg1) matches
    tagged Valid {sx1: sReg2 <= tagged Valid f1(sx1)};
    tagged Invalid: sReg2 <= tagged Invalid; endcase
  case (sReg2) matches
    tagged Valid {sx2: outQ.enq(f2(sx2))};
  endcase
endrule
```

February 9, 2011  http://csg.csail.mit.edu/6.375  L03-26
When is this rule enabled?

```verilog
rule sync-pipeline (True);
if (inQ.notEmpty())
begin
sReg1 <= tagged Valid f0(inQ.first()); inQ.deq();
end
else
sReg1 <= tagged Invalid;

begin
  case (sReg1.matches)
    tagged Valid .sx1: sReg2 <= tagged Valid f1(sx1);
    tagged Invalid: sReg2 <= tagged Invalid;
  endcase
  endcase
endrule
```

<table>
<thead>
<tr>
<th>inQ</th>
<th>sReg1</th>
<th>sReg2</th>
<th>outQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>V</td>
<td>V</td>
<td>NF</td>
</tr>
<tr>
<td>NE</td>
<td>V</td>
<td>V</td>
<td>F</td>
</tr>
<tr>
<td>NE</td>
<td>V</td>
<td>I</td>
<td>NF</td>
</tr>
<tr>
<td>NE</td>
<td>I</td>
<td>V</td>
<td>NF</td>
</tr>
<tr>
<td>NE</td>
<td>I</td>
<td>I</td>
<td>NF</td>
</tr>
<tr>
<td>NE</td>
<td>I</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inQ</th>
<th>sReg1</th>
<th>sReg2</th>
<th>outQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>V</td>
<td>V</td>
<td>NF</td>
</tr>
<tr>
<td>E</td>
<td>V</td>
<td>V</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td>V</td>
<td>I</td>
<td>NF</td>
</tr>
<tr>
<td>E</td>
<td>I</td>
<td>V</td>
<td>NF</td>
</tr>
<tr>
<td>E</td>
<td>I</td>
<td>I</td>
<td>NF</td>
</tr>
<tr>
<td>E</td>
<td>I</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Next lecture

Folded pipeline for FFT