Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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

- Type checking
- Massive partial evaluation and static elaboration

Now we call this Guarded Atomic Actions

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:
- compile
- run w/ params

Hardware Toolflow:
- 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, ...
BSV has a very strong notion of types
- Every expression has a type. Either it is declared by the user or automatically deduced by the compiler
- The compiler verifies that the type declarations are compatible

BSV code: 4-way Butterfly

```plaintext
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
```
Complex Arithmetic

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

- **Multiplication**
  - $z_R = x_R * y_R - x_I * y_I$
  - $z_R = x_R * y_I + x_I * y_R$

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

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
```
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

```hs
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]);
```

Stage_f can be inlined now; it could have been inlined before loop unfolding also.

Does the order matter?

Bluespec Code for stage_f

```hs
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]];
end
return(stage_out);
```
twid's are mathematically derivable constants
Higher-order functions:
Stage functions $f_1$, $f_2$ and $f_3$

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

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

function f3(x);
    return (stage_f(3,x));
endfunction
```

What is the type of $f_1(x)$?

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

- Controller
- Scrambler
- Encoder
- Interleaver
- Mapper
- IFFT
- Cyclic Extend

Must produce one OFDM symbol every 4 μsec

Depending upon the transmission rate, consumes 1, 2 or 4 tokens to produce one OFDM symbol

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

One OFDM symbol (64 Complex Numbers) accounts for 85% area
## 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>0%</td>
</tr>
<tr>
<td>Scrambler</td>
<td>40</td>
<td>0%</td>
</tr>
<tr>
<td>Conv. Encoder</td>
<td>113</td>
<td>0%</td>
</tr>
<tr>
<td>Interleaver</td>
<td>76</td>
<td>1%</td>
</tr>
<tr>
<td>Mapper</td>
<td>112</td>
<td>11%</td>
</tr>
<tr>
<td>IFFT</td>
<td>95</td>
<td>85%</td>
</tr>
<tr>
<td>Cyc. Extender</td>
<td>23</td>
<td>3%</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
Superfolded circular pipeline: Just one Bfly-4 node!

Pipelining a block
**Synchronous pipeline**

Stage functions $f_1$, $f_2$ and $f_3$

- **Rule**
  ```
  rule sync-pipeline (True);
      inQ.deq();
      sReg1 <= f1(inQ.first());
      sReg2 <= f2(sReg1);
      outQ.enq(f3(sReg2));
  endrule
  ```

- **Function**
  ```
  function f1(x);
      return (stage_f(1,x));
  endfunction
  ```

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

  ```
  function f3(x);
      return (stage_f(3,x));
  endfunction
  ```

*This rule can fire only if...*
Problem: What about pipeline bubbles?

```
rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f1(inQ.first());
  sReg2 <= f2(sReg1);
  outQ.enq(f3(sReg2));
endrule
```

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 <= Valid f1(inQ.first()); inQ.deq(); end
else sReg1 <= Invalid;
case (sReg1) matches
tagged Valid .sx1: sReg2 <= Valid f2(sx1);
tagged Invalid: sReg2 <= Invalid;
case (sReg2) matches
tagged Valid .sx2: outQ.enq(f3(sx2));
endrule
```

sx1 will get bound to the appropriate part of sReg1
Next lecture

Code for folded pipelined FFT