Architectural Exploration:
Area-Power tradeoff in 802.11a transmitter design

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 16, 2007
http://csg.csail.mit.edu/6.375/

This lecture has two purposes

- Illustrate how area-power tradeoff can be studied at a high-level for a realistic design
  - Example: 802.11a transmitter
- Illustrate some features of BSV
  - Static elaboration
  - Combinational circuits
  - Simple synchronous pipelines
  - Valid bits as the Maybe type in BSV

No prior understanding of 802.11a is necessary to follow this lecture
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)

- Type checking
- Massive partial evaluation and static elaboration

Now we call this Guarded Atomic Actions

- Rule conflict analysis
- Rule scheduling

Lennart Augustsson  
@Sandburst 2000-2002

James Hoe & Arvind  
@MIT 1997-2000

Static Elaboration

At compile time
- Inline function calls and datatypes
- Instantiate modules with specific parameters
- Resolve polymorphism/overloading

Software Toolflow:
source  
compile  
.run with params  
.exe  
run1  
run2  
run3

Hardware Toolflow:
source  
.elaborate w/params  
.run w/params  
.design  
.run1  
run1.1  
run2.1  
run3.1
802.11a Transmitter Overview

- **Controller**
- **Scrambler**
- **Encoder**
- **Interleaver**
- **Mapper**
- **IFFT**
- **Cyclic Extender**

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

Must produce one OFDM symbol every 4 μsec.

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

**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

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

4-way Butterfly Node

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

```
function Vector#(4,Complex) Bfly4 (Vector#(4,Complex) t, Vector#(4,Complex) k);
    Vector#(4,Complex) m = newVector(),
    y = newVector(),
    z = newVector();
    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

Combinational IFFT

```
stage_f function repeat it three times
```

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

Note: Vector does not mean storage
BSV Code: Combinational IFFT

```
function SVector#(64, Complex) ifft (SVector#(64, Complex) in_data);
    //Declare vectors
    SVector#(4,SVector#(64, Complex)) stage_data = replicate(newSVector);
    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

```
function SVector#(64, Complex) ifft (SVector#(64, Complex) in_data);
    //Declare vectors
    SVector#(4,SVector#(64, Complex)) stage_data = replicate(newSVector);
    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 \texttt{stage\_f}

```plaintext
function SVector\#(64, Complex) \texttt{stage\_f} (Bit\#(2) stage, SVector\#(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 = \texttt{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);
```

Architectural Exploration
Design Alternatives

Reuse a block over multiple cycles

we expect:
Throughput to
Area to

Combinational IFFT
Opportunity for reuse

Reuse the same circuit three times
Circular pipeline: Reusing the Pipeline Stage

16 Bfly4s can be shared but not the three permutations. Hence the need for muxes

Superfolded circular pipeline: Just one Bfly-4 node!
1. All the three permutations can be made identical
⇒ more saving in area in the folded case

2. One multiplication can be removed from Bfly-4

Area improvements because of change in Algorithm

<table>
<thead>
<tr>
<th>Design</th>
<th>Old Area (mm²)</th>
<th>New Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combinational</td>
<td>4.69</td>
<td>4.91</td>
</tr>
<tr>
<td>Simple Pipe</td>
<td>5.14</td>
<td>5.25</td>
</tr>
<tr>
<td>Folded Pipe</td>
<td>5.89</td>
<td>3.97</td>
</tr>
</tbody>
</table>
Which design consumes the least energy to transmit a symbol?

- Can we quickly code up all the alternatives?
  - single source with parameters?

5-minute break to stretch your legs

Pipelining a block

Clock? Area? Throughput?
Synchronous pipeline

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

This rule can fire only if

Atomicity: Either all or none of the state elements inQ, outQ, sReg1 and sReg2 will be updated

Stage functions f1, f2 and f3

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

The stage_f function is given on slide 12
Problem: What about pipeline bubbles?

Red and Green tokens must move even if there is nothing in the inQ!
Also if there is no token in sReg2 then nothing should be enqueued in the outQ.

Modify the rule to deal with these conditions.

The Maybe type data in the pipeline.

Registers contain Maybe type values.
Folded pipeline

The same code will work for superfolded pipelines by changing n and stage function f.

```verilog
rule folded-pipeline (True);
if (stage==1)
begin
sxIn= inQ.first(); inQ.deq();
end
else
sxIn= sReg;
sxOut = f(stage, sxIn);
if (stage==n) outQ.enq(sxOut);
else
sReg <= sxOut;
stage <= (stage==n)? 1 : stage+1;
endrule
```

802.11a Transmitter Synthesis results (Only the IFFT block is changing)

<table>
<thead>
<tr>
<th>IFFT Design</th>
<th>Area (mm²)</th>
<th>Throughput Latency (CLKs/sym)</th>
<th>Min. Freq Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelined</td>
<td>5.25</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Combinational</td>
<td>4.91</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Folded (16 Bfly-4s)</td>
<td>3.97</td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Super-Folded (8 Bfly-4s)</td>
<td>3.69</td>
<td>06</td>
<td>1.5 MHz</td>
</tr>
<tr>
<td>SF(4 Bfly-4s)</td>
<td>2.45</td>
<td>12</td>
<td>3.0 MHz</td>
</tr>
<tr>
<td>SF(2 Bfly-4s)</td>
<td>1.84</td>
<td>24</td>
<td>6.0 MHz</td>
</tr>
<tr>
<td>SF (1 Bfly4)</td>
<td>1.52</td>
<td>48</td>
<td>12 MHz</td>
</tr>
</tbody>
</table>

TSMC .18 micron; numbers reported are before place and route.
Why are the areas so similar

Folding should have given a 3x improvement in IFFT area

BUT a constant twiddle allows low-level optimization on a Bfly-4 block
  ■ a 2.5x area reduction!

Summary

It is essential to do architectural exploration for better (area, power, performance, ...) designs.

It is possible to do so with new design tools and methodologies, i.e., Bluespec

Better and faster tools for estimating area, timing and power would dramatically increase our capability to do architectural exploration.
Bluespec Learnings

- How to write highly parameterized combinational codes
- How to write rules for simple synchronous pipelines
- Effect of dynamic vs static values on generated circuits
- Using Maybe types to express valid/invalid data

Thanks