Combinational Circuits and Simple Synchronous Pipelines

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:
- source
- compile
- .exe
- run
- run w/ params

Hardware Toolflow:
- source
- elaborate w/params
- design
- run
- 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, ...

* * *

* * *

* * *

* * *

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
</tr>
</tbody>
</table>
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

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

Note: Vector does not mean storage
Add a slide on arithmetic, compile time costants, ...

Arvind, 4/28/2007
Combinational IFFT

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

``` pública
def 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]);
end
```

Bluespec Code for stage_f

``` pública
def 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);
end
```
Suppose we want to reuse some part of the circuit ...

Reuse the same circuit three times to reduce area

Architectural Exploration:
Area-Performance tradeoff in 802.11a Transmitter
802.11a Transmitter Overview

- Headers
- Data
- Scrambler
- Encoder
- Interleaver
- Mapper
- IFFT
- Cyclic Extend

- IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers
- One OFDM symbol (64 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

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

Combinational

Pipeline

Folded Pipeline

Clock?  Area?  Throughput?

Synchronous pipeline

This rule can fire only if

rule sync-pipeline (True);
inQ.deq();
sReg1 <= f1(inQ.first());
sReg2 <= f2(sReg1);
outQ.enq(f3(sReg2));
endrule
Stage functions \( f_1, f_2 \) and \( f_3 \)

\[
\begin{align*}
\text{function } f_1(x); \\
&\quad \text{return } (\text{stage}_f(1,x)); \\
\text{endfunction}
\end{align*}
\[
\begin{align*}
\text{function } f_2(x); \\
&\quad \text{return } (\text{stage}_f(2,x)); \\
\text{endfunction}
\end{align*}
\[
\begin{align*}
\text{function } f_3(x); \\
&\quad \text{return } (\text{stage}_f(3,x)); \\
\text{endfunction}
\end{align*}
\]

The \( \text{stage}_f \) function was given earlier.

Problem: What about pipeline bubbles?

\[
\begin{align*}
\text{rule } \text{sync-pipeline} (\text{True}); \\
&\quad \text{inQ}.\text{deq}(); \\
&\quad \text{sReg}1 \leftarrow f_1(\text{inQ}.\text{first}()); \\
&\quad \text{sReg}2 \leftarrow f_2(\text{sReg}1); \\
&\quad \text{outQ}.\text{enq}(f_3(\text{sReg}2)); \\
\text{endrule}
\end{align*}
\]

Red and Green tokens must move even if there is nothing in the \text{inQ}!

Also if there is no token in \text{sReg}2 then nothing should be enqueued in the \text{outQ}.
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

Folded pipeline

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

rule folded-pipeline (True);
   if (stage==0)
      begin sxIn= inQ.first(); inQ.deq(); end
   else sxIn= sReg;
   sxOut = f(stage,sxIn);
   if (stage==n-1) outQ.enq(sxOut);
   else sReg <= sxOut;
   stage <= (stage==n-1)? 0 : stage+1;
endrule

Need type declarations for sxIn and sxOut
**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><strong>4.91</strong></td>
<td>04</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Folded (16 Bfly-4s)</td>
<td><strong>3.97</strong></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>

All these designs were done in less than 24 hours!

The same source code was used for all these designs.

---

**Why are the areas so similar?**

*Folding should have given a 3x improvement in IFFT area*

---

TSMC .18 micron; numbers reported are before place and route.
Language notes

- Pattern matching syntax
- Vector syntax
- Implicit conditions

Pattern-matching: A convenient way to extract datastructure components

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

if (m matches (Valid .x) &&& (x > 10))
```

- The &&& is a conjunction, and allows pattern-variables to come into scope from left to right.
Syntax: Vector of Registers

- Register
  - suppose \( x \) and \( y \) are both of type Reg. Then
    \[ x \leq y \] means \( x._{\text{write}}(y._{\text{read}}) \)

- Vector of Int
  - \( x[i] \) means \( \text{sel}(x,i) \)
  - \( x[i] = y[j] \) means \( x = \text{update}(x, i, \text{sel}(y,j)) \)

- Vector of Registers
  - \( x[i] \leq y[j] \) does not work. The parser thinks it means
    \( (\text{sel}(x,i)._{\text{read}})._{\text{write}}(\text{sel}(y,j)._{\text{read}}) \), which will not type check
  - \( (x[i]) \leq y[j] \) parses as
    \( \text{sel}(x,i)._{\text{write}}(\text{sel}(y,j)._{\text{read}}) \), and works correctly

Don’t ask me why

Making guards explicit

```verilog
rule recirculate (True);
  if (p) fifo.enq(8);
  r <= 7;
endrule

rule recirculate ((p && fifo.enq) || !p);
  if (p) fifo.enq(8);
  r <= 7;
endrule
```

Effectively, all implicit conditions (guards) are lifted and conjoined to the rule guard
Implicit guards (conditions)

Rule

\[
\text{rule} \ <\text{name}> \ (<\text{guard}>); \ <\text{action}>; \ \text{endrule}
\]

where

\[
<\text{action}> ::= r \leq <\text{exp}> \quad \text{m.g}<\text{exp}> > \quad \text{m.g}<\text{exp}> > \quad \text{m.g}<\text{exp}> > \quad \text{if} (<\text{exp}>_<\text{action}> _\text{endif}
\]

make implicit guards explicit

Guards vs If’s

- A guard on one action of a parallel group of actions affects every action within the group
  
  \[
  (a1 \text{ when } p1); \ (a2 \text{ when } p2) \Rightarrow (a1; a2) \text{ when } (p1 \&\& p2)
  \]

- A condition of a Conditional action only affects the actions within the scope of the conditional action
  
  \[
  (\text{if} \ (p1) \ a1); \ a2
  \]
  
  p1 has no effect on a2 ...

- Mixing ifs and whens
  
  \[
  (\text{if} \ (p) \ (a1 \text{ when } q)) \ ; \ a2
  \]
  
  \[
  = ((\text{if} \ (p) \ a1); \ a2) \text{ when } ((p \&\& q) \ | \ !p)
  \]