Expressing designs for IFFT in Bluespec (BSV)

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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

function Vector#(4,Complex) Bfly4 (Vector#(4,Complex) t, Vector#(4,Complex) k);

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

```
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

```haskell
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]);
```

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

Does the order matter?

Bluespec Code for stage_f

```haskell
function SVector#(64, Complex) 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 = 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);
```
Design Alternatives

Reuse a block over multiple cycles

we expect:
Throughput to decrease – less parallelism
Energy/unit work to increase - due to extra HW
Area to decrease – reusing a block
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>

Coding various pipelined versions in BSV
Pipelining a block

Clock: $C < P \approx FP$
Area: $FP < C < P$
Throughput: $FP < C < P$

Synchronous pipeline

This rule can fire only if:
- $inQ$ has an element
- $outQ$ has space

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

This is real IFFT code; just replace $f1$, $f2$ and $f3$ with stage_f code
Stage functions $f_1$, $f_2$ and $f_3$

```plaintext
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

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

Modify the rule to deal with these conditions
The Maybe type data in the pipeline

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

Registers contain Maybe type values

```
rule sync-pipeline (True);
if (inQ.notEmpty())
begin
    sReg1 <= Valid f1(inQ.first()); inq.deq();
else
    sReg1 <= Invalid;
endcase (sReg1) matches
    tagged Valid .sx1: sReg2 <= Valid f2(sx1);
    tagged Invalid: sReg2 <= Invalid;
endrule
```

Folded pipeline

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

The same code will work for superfolded pipelines by changing n and stage function f
Function f for the folded pipeline is the same stage_f function but ...

```固体
function SVector#(64, Complex) 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 = 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);
```

notice this is no longer a static parameter! ⇒ will cause a mux to be generated

Folded pipeline: stage function f

The rest of stage_f, i.e. Bfly-4s and permutations (shared)
Function $f$ for the Superfolded pipeline (One Bfly-4 case)

- $f$ will be invoked for 48 dynamic values of stage
  - each invocation will modify 4 numbers in sReg
  - after 16 invocations a permutation would be done on the whole sReg

Code for the Superfolded pipeline stage function

```haskell
function SVector#(64, Complex) f
  (Bit#(6) stage, SVector#(64, Complex) stage_in);
begin
  let idx = stage `mod` 16;
  let twid = getTwiddle(stage `div` 16, idx);
  let y = bfly4(twid, stage_in[idx:idx+3]);

  stage_temp = stage_in;
  stage_temp[idx] = y[0];
  stage_temp[idx+1] = y[1];
  stage_temp[idx+2] = y[2];
  stage_temp[idx+3] = y[3];

  for (Integer i = 0; i < 64; i = i + 1)
    stage_out[i] = stage_temp[permute[i]];
end
return((idx == 15) ? stage_out: stage_temp);
```

One Bfly-4 case
802.11a Transmitter Synthesis results (Only the IFFT block is changing)

<table>
<thead>
<tr>
<th>IFFT Design</th>
<th>Area (mm²)</th>
<th>Symbol Latency (CLKs)</th>
<th>Throughput Latency (CLKs/sym)</th>
<th>Min. Freq Required</th>
<th>Average Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipelined</td>
<td>5.25</td>
<td>12</td>
<td>04</td>
<td>1.0 MHz</td>
<td>4.92</td>
</tr>
<tr>
<td>Combinational</td>
<td><strong>4.91</strong></td>
<td>10</td>
<td>04</td>
<td>1.0 MHz</td>
<td>3.99</td>
</tr>
<tr>
<td>Folded (16 Bfly-4s)</td>
<td><strong>3.97</strong></td>
<td>12</td>
<td>04</td>
<td>1.0 MHz</td>
<td>7.27</td>
</tr>
<tr>
<td>Super-Folded (8 Bfly-4s)</td>
<td>3.69</td>
<td>15</td>
<td>06</td>
<td>1.5 MHz</td>
<td>10.9</td>
</tr>
<tr>
<td>SF(4 Bfly-4s)</td>
<td>2.45</td>
<td>21</td>
<td>12</td>
<td>3.0 MHz</td>
<td>14.4</td>
</tr>
<tr>
<td>SF(2 Bfly-4s)</td>
<td>1.84</td>
<td>33</td>
<td>24</td>
<td>6.0 MHz</td>
<td>21.1</td>
</tr>
<tr>
<td>SF (1 Bfly4)</td>
<td>1.52</td>
<td>57</td>
<td>48</td>
<td>12 MHZ</td>
<td>34.6</td>
</tr>
</tbody>
</table>

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!