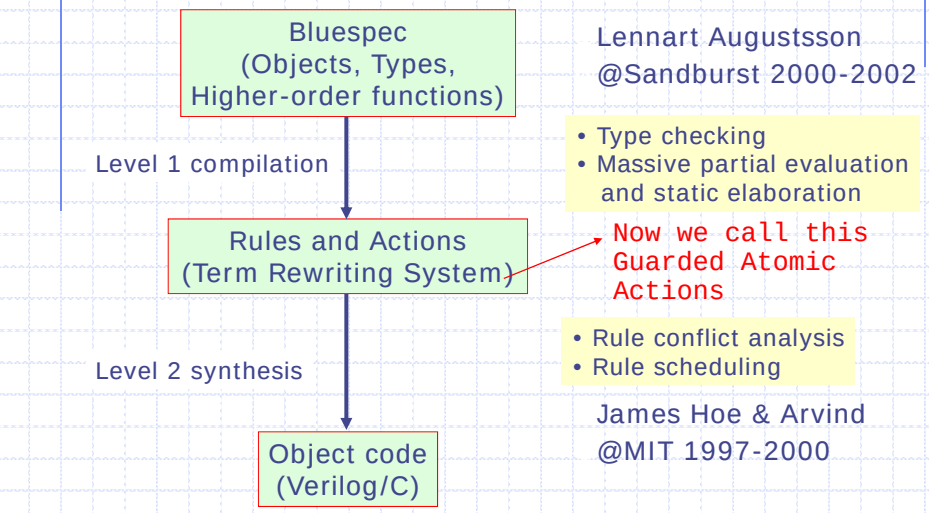
Combinational Circuits in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Bluespec: Two-Level Compilation



Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

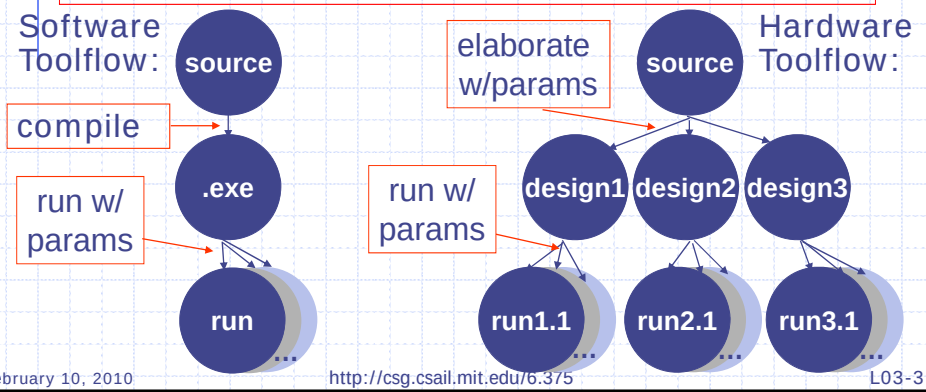
Now we call this Guarded Atomic Actions

- Rule conflict analysis
- Rule scheduling

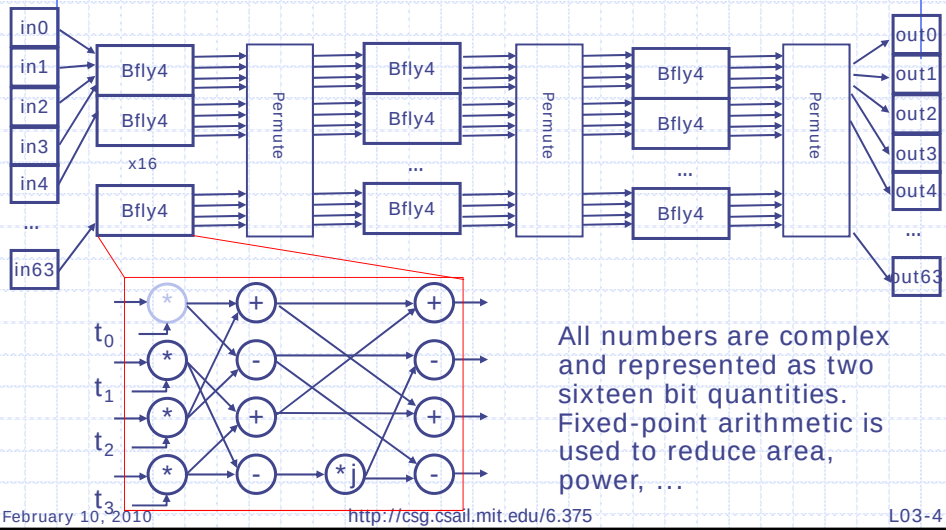
James Hoe & Arvind
@MIT 1997-2000

Static Elaboration

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

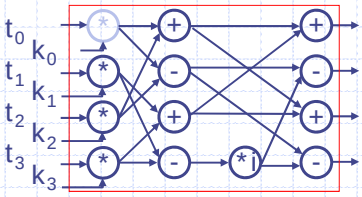


Combinational IFFT



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

4-way Butterfly Node

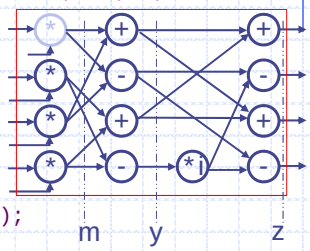



```
function Vector#(4,Complex) bfly4
          (Vector#(4,Complex) t,  Vector#(4,Complex) k);
```

- 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, y, z;

  m[0] = k[0] * t[0]; m[1] = k[1] * t[1];
  m[2] = k[2] * t[2]; m[3] = k[3] * t[3];

  y[0] = m[0] + m[2]; y[1] = m[0] - m[2];
  y[2] = m[1] + m[3]; y[3] = i*(m[1] - m[3]);

  z[0] = y[0] + y[2]; z[1] = y[1] + y[3];
  z[2] = y[0] - y[2]; z[3] = y[1] - y[3];

  return(z);
endfunction
```



Complex Arithmetic

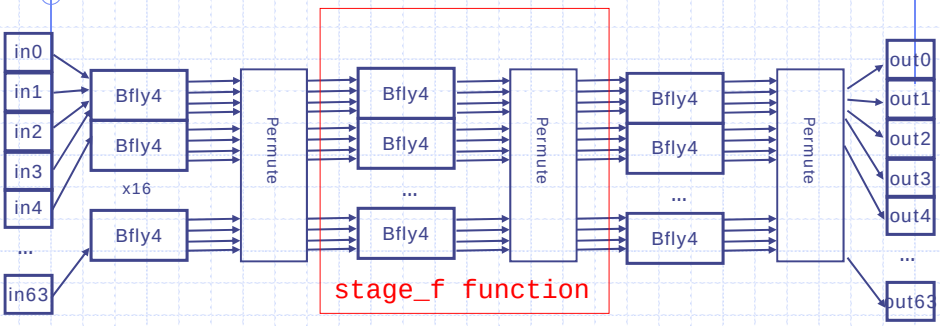
- Addition
  - $z_R = x_R + y_R$
  - $z_I = x_I + y_I$
- Multiplication
  - $z_R = x_R * y_R - x_I * y_I$
  - $z_R = x_R * y_I + x_I * y_R$

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

BSV code for Addition

```
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);
```

repeat stage_f three times

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

```
function Vector#(64, Complex) ifft
                    (Vector#(64, Complex) in_data);
//Declare vectors
   Vector#(4,Vector#(64, Complex)) stage_data;

   stage_data[0] = in_data;
   stage_data[1] = stage_f(0,stage_data[0]);
   stage_data[2] = stage_f(1,stage_data[1]);
   stage_data[3] = stage_f(2,stage_data[2]);

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

```
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);
```

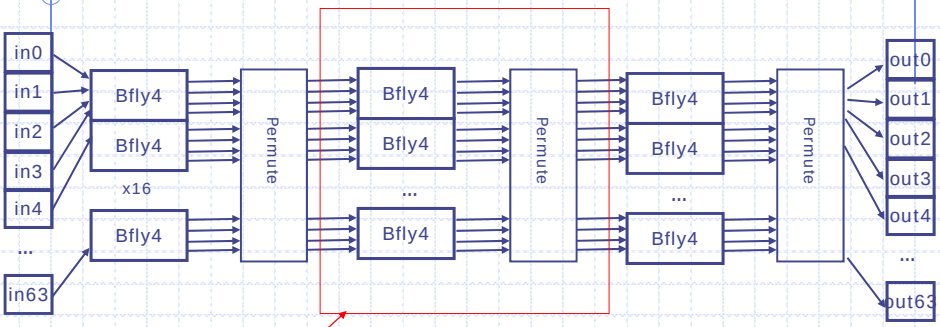
twid's are mathematically derivable constants

Higher-order functions: Stage functions 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
```

What is the type of `f1(x)` ?

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



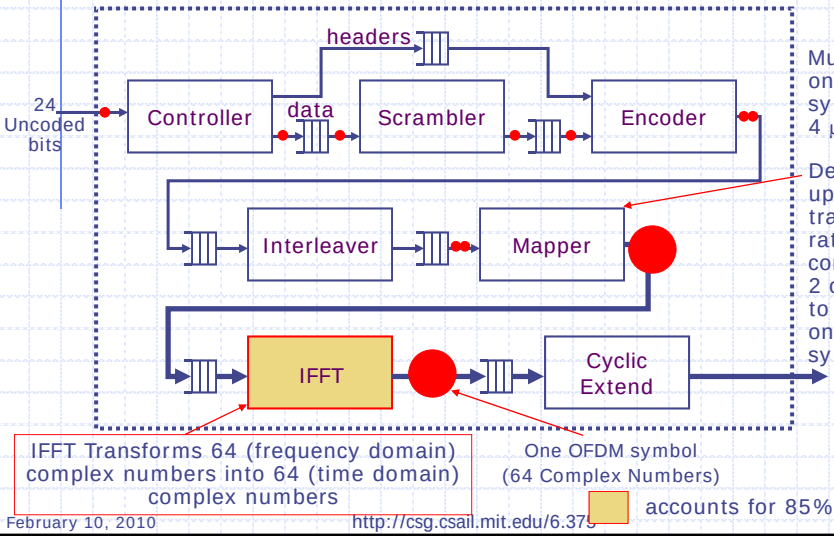
Reuse the same circuit three times to reduce area

But why?

Architectural Exploration:

Area-Performance tradeoff in 802.11a Transmitter

802.11a Transmitter Overview



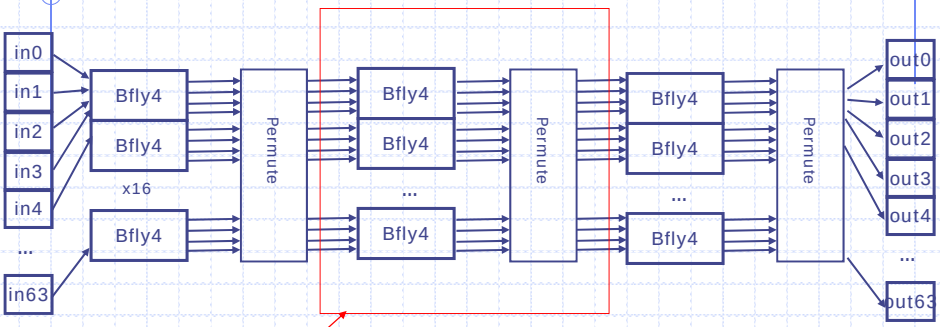
Preliminary results

[MEMOCODE 2006] Dave, Gerding, Pellauer, Arvind

Design Block	Lines of Code (BSV)	Relative Area
Controller	49	0%
Scrambler	40	0%
Conv. Encoder	113	0%
Interleaver	76	1%
Mapper	112	11%
IFFT	95	85%
Cyc. Extender	23	3%

Complex arithmetic libraries constitute another 200 lines of code

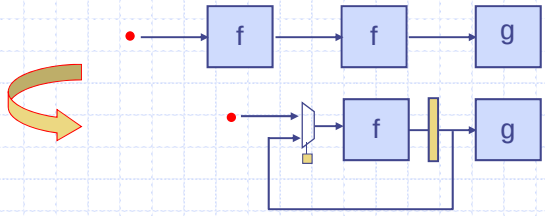
Combinational IFFT



Reuse the same circuit three times to reduce area

Design Alternatives

Reuse a block over multiple cycles

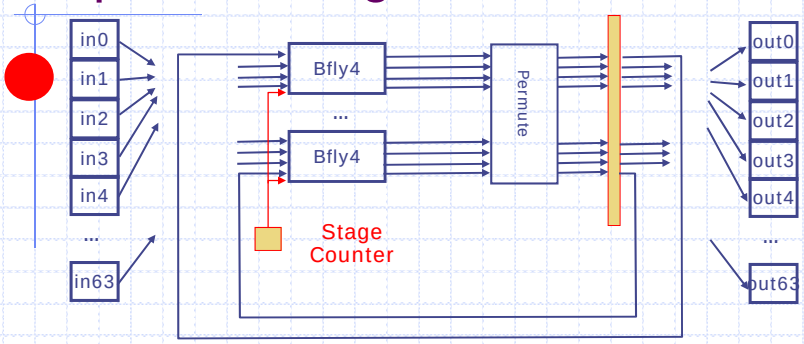


we expect:

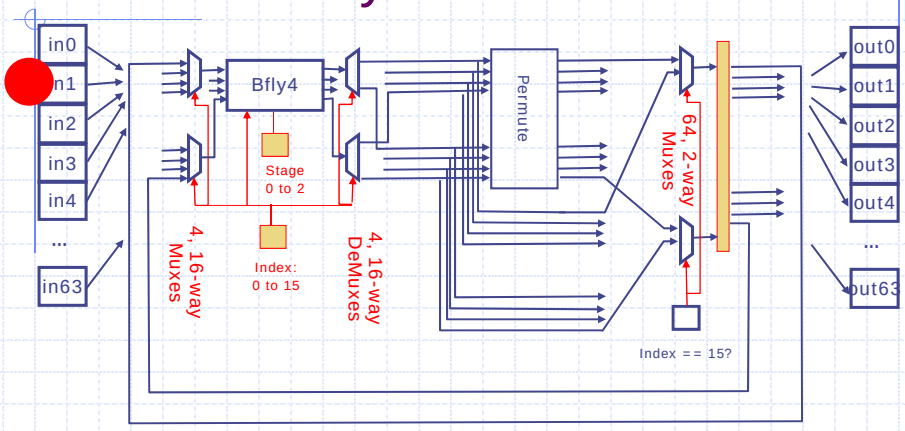
Throughput to

Area to

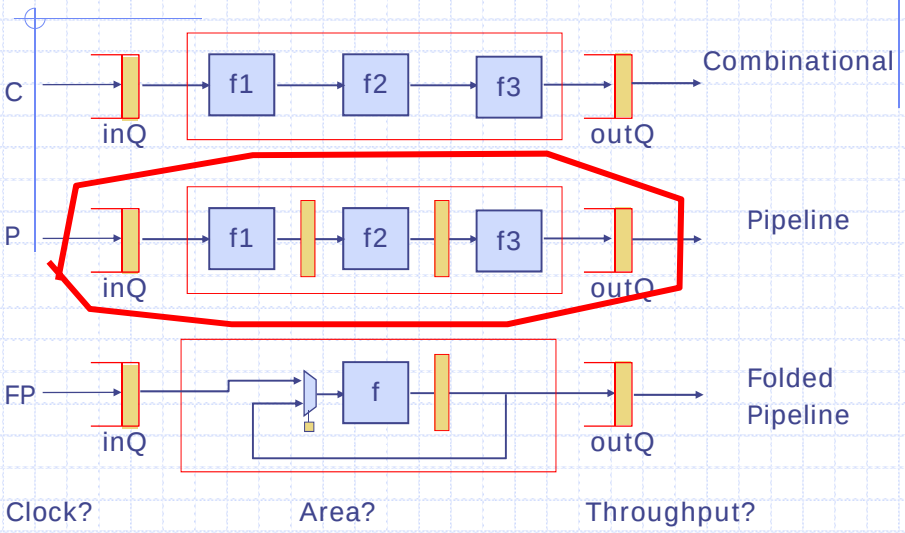
Circular pipeline: Reusing the Pipeline Stage



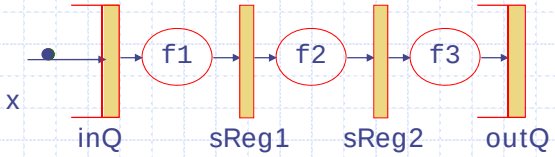
Superfolded circular pipeline: Just one Bfly-4 node!



Pipelining a block



Synchronous pipeline




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

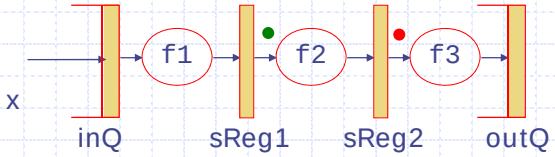
This rule can fire only if

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 was given earlier

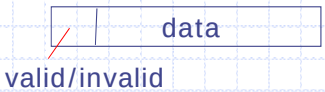
Problem: What about pipeline bubbles?




```
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);
```



Registers contain Maybe type values

```
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