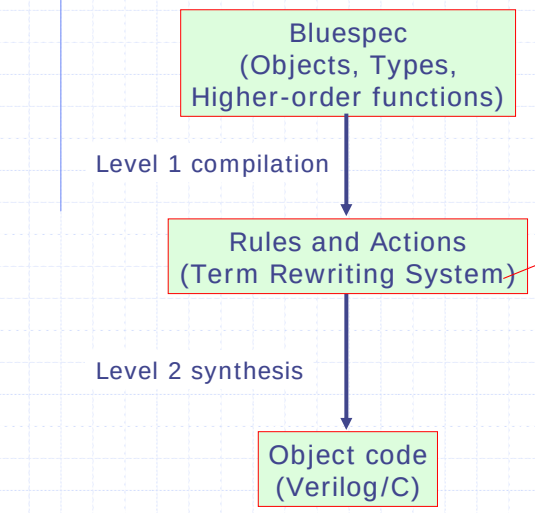
Combinational Circuits and Simple Synchronous Pipelines

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

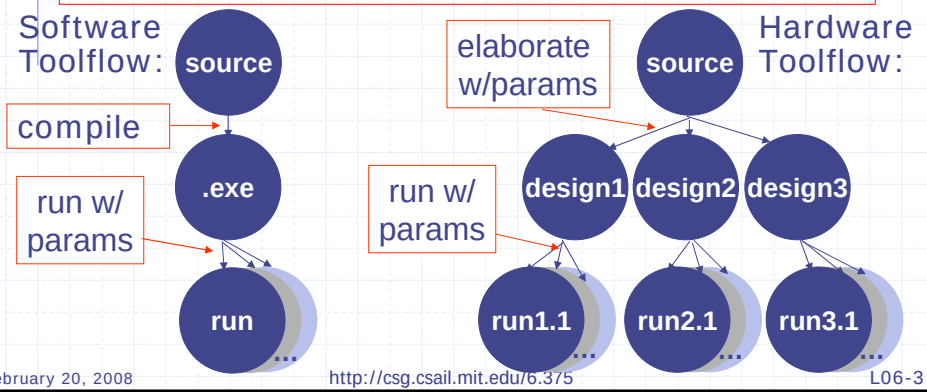
Now we call this Guarded Atomic Actions

- Rule conflict analysis
- Rule scheduling

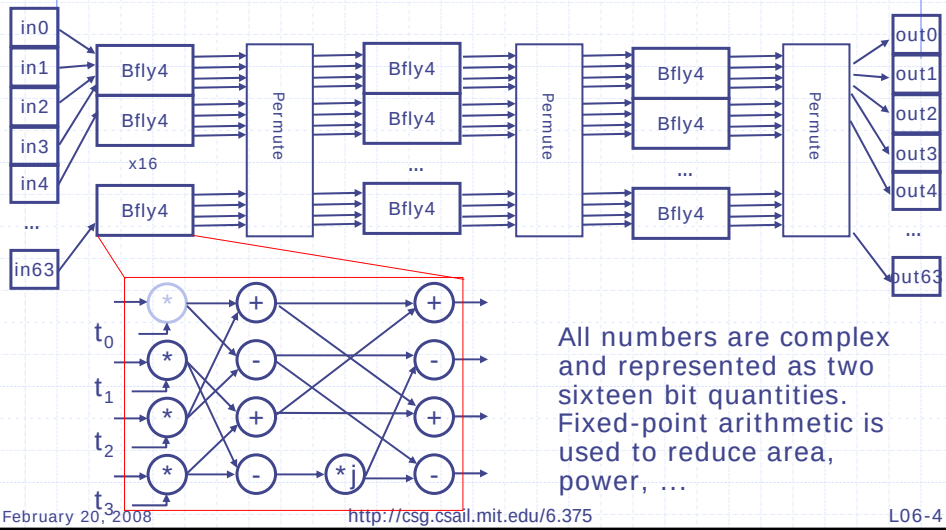
James Hoe & Arvind
@MIT 1997-2000

Static Elaboration

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

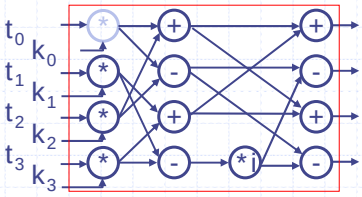


Combinational IFFT



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

4-way Butterfly Node



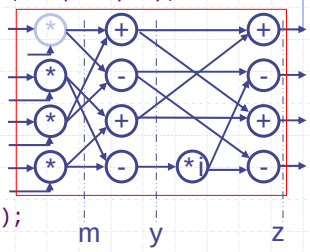

```
function Vector#(4,Complex) bfly4
          (Vector#(4,Complex) t,  Vector#(4,Complex) k);
```

- 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

A1

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
```



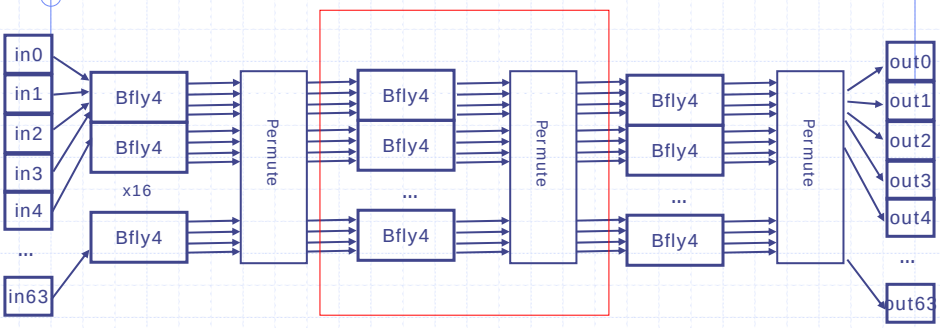
Polymorphic code: works on any type of numbers for which *, + and - have been defined

Note: Vector does not mean storage

Slide 6

A1 Add a slide on arithmetic, compile time costants, ...
Arvind, 4/28/2007

Combinational IFFT



stage_f function

repeat it 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

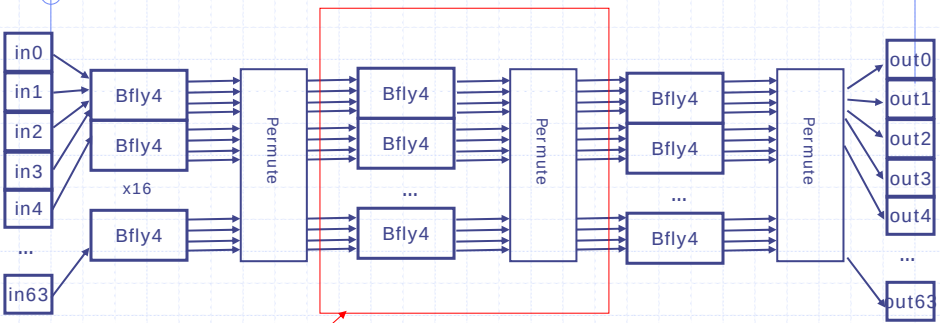
BSV Code: Combinational IFFT- Unfolded

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

   stage_data[0] = in_data;
   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 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);
```

Suppose we want to reuse some part of the circuit …

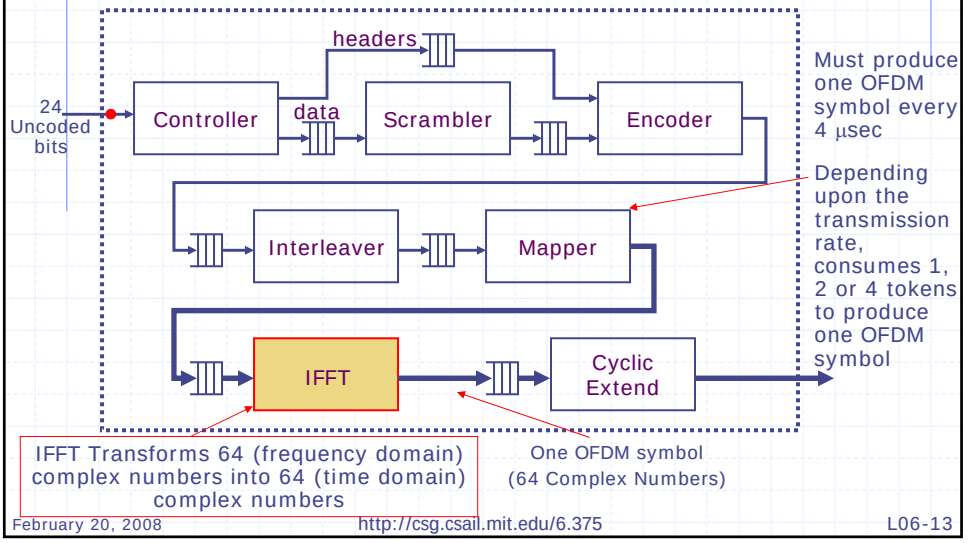


Reuse the same circuit three times to reduce area

Architectural Exploration:

Area-Performance tradeoff in 802.11a Transmitter

802.11a Transmitter Overview



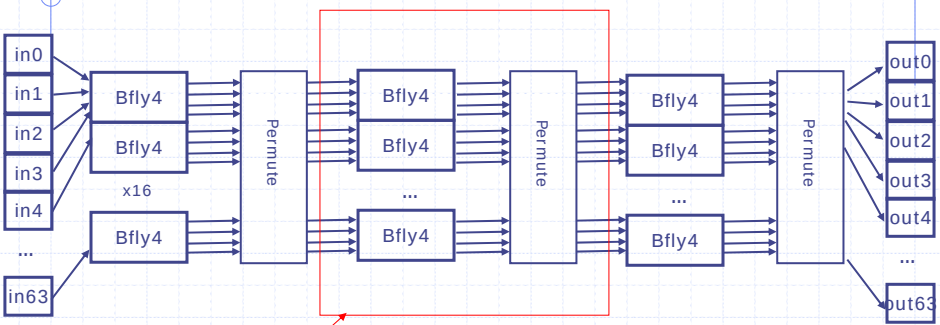
Preliminary results

[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

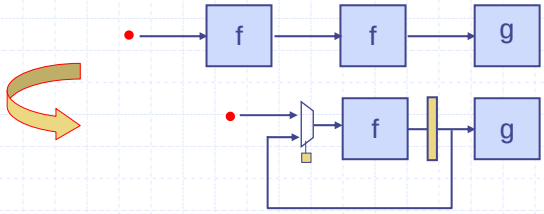
Combinational IFFT



Reuse the same circuit three times to reduce area

Design Alternatives

Reuse a block over multiple cycles

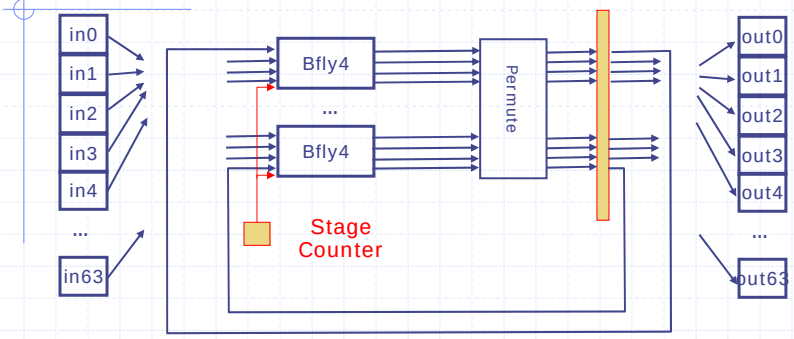


we expect:

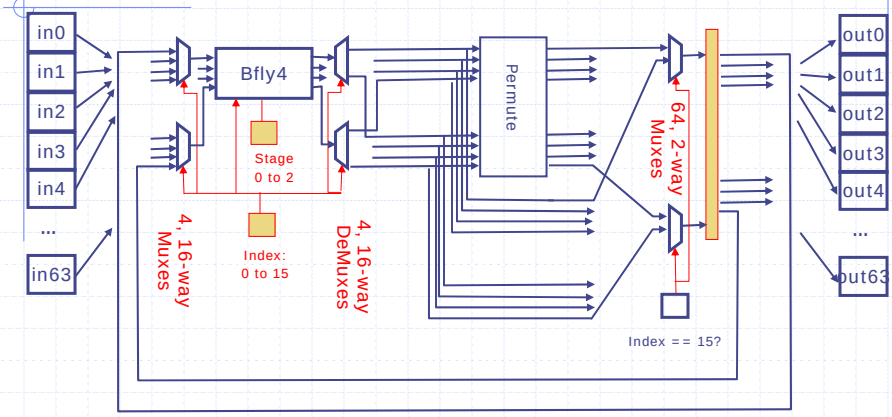
Throughput to

Area to

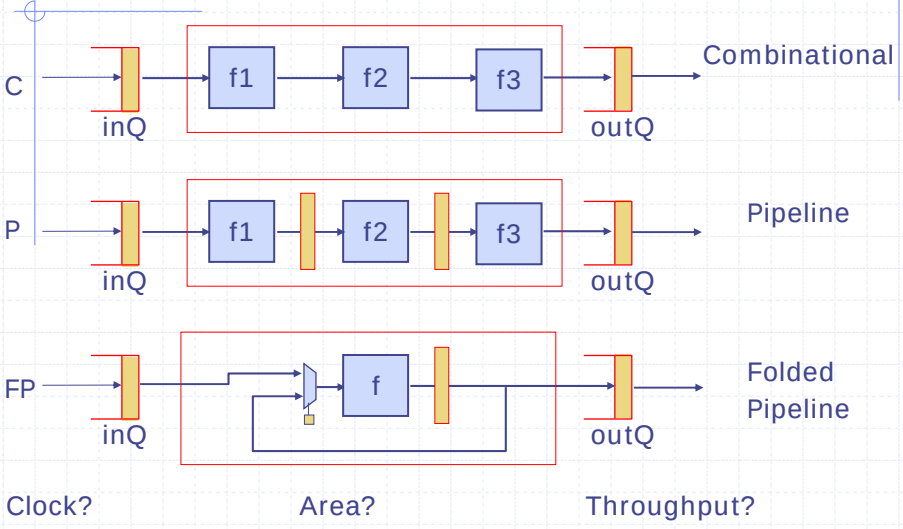
Circular pipeline: Reusing the Pipeline Stage



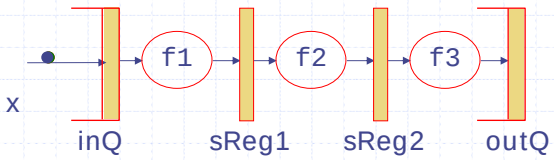
Superfolded circular pipeline: Just one Bfly-4 node!



Pipelining a block



Synchronous pipeline




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

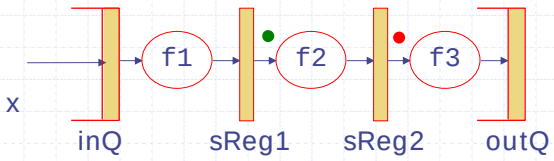
This rule can fire only if

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
```

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

The Maybe type data in the pipeline

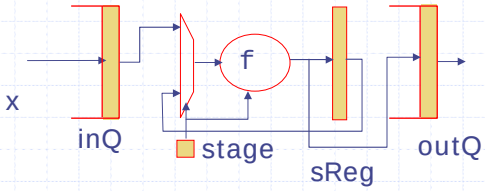
```
typedef union tagged {
  void Invalid;
  data_T Valid;
} Maybe#(type data_T);
```

valid/invalid

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
```


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)

IFFT Design	Area (mm^2)	Throughput Latency (CLKs/sym)	Min. Freq Required
Pipelined	5.25	04	1.0 MHz
Combinational	**4.91**	04	1.0 MHz
Folded (16 Bfly-4s)	**3.97**	04	1.0 MHz
Super-Folded (8 Bfly-4s)	3.69	06	1.5 MHz
SF(4 Bfly-4s)	2.45	12	3.0 MHz
SF(2 Bfly-4s)	1.84	24	6.0 MHz
SF (1 Bfly4)	1.52	48	12 MHZ

The same source code

All these designs were done in less than 24 hours!

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

Language notes

- Pattern matching syntax
- Vector syntax
- Implicit conditions

Pattern-matching: A convenient way to extract datastructure components

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

```
case (m) matches
  tagged Invalid  : return 0;
  tagged Valid .x : return x;
endcase
```

x will get bound to the appropriate part of m

```
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 <= y` means `x._write(y._read())`
- Vector of `Int`
  - `x[i]` means `sel(x,i)`
  - `x[i] = y[j]` means `x = update(x,i, sel(y,j))`
- Vector of Registers
  - `x[i] <= y[j]` does not work. The parser thinks it means `(sel(x,i)._read)._write(sel(y,j)._read)`, which will not type check
  - `(x[i]) <= y[j]` parses as `sel(x,i)._write(sel(y,j)._read)`, and works correctly

Don't ask me why

Making guards explicit

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

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

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

Implicit guards (conditions)

- Rule

  rule <name> (<guard>); <action>; **endrule**

  where

  <action> ::= r <= <exp>

  | ~~m.g(<exp>)~~

  | **if** (<exp>) <action> **endif**

  | <action> ; <action>

$m.g_B(\text{<exp>})$ **when** $m.g_G$

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 when p1); (a2 when p2)

  ==> (a1; a2) when (p1 && p2)

- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if (p1) a1); a2

  p1 has no effect on a2 ...

- Mixing ifs and whens

  (if (p) (a1 when q)) ; a2

  ≡ ((if (p) a1); a2) when ((p && q) | !p)