Architectural Exploration:

Area-Power tradeoff in 802.11a transmitter design

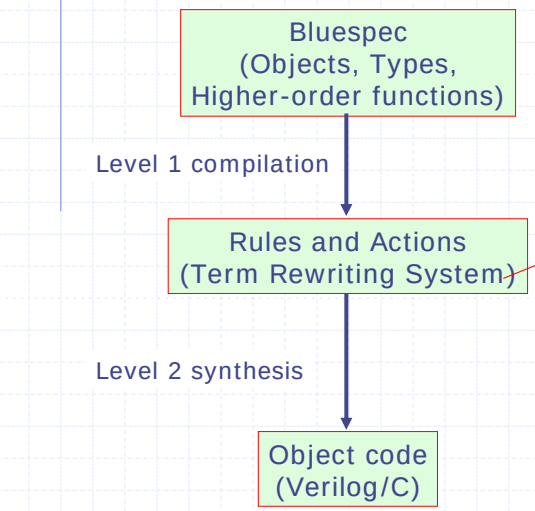
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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



Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

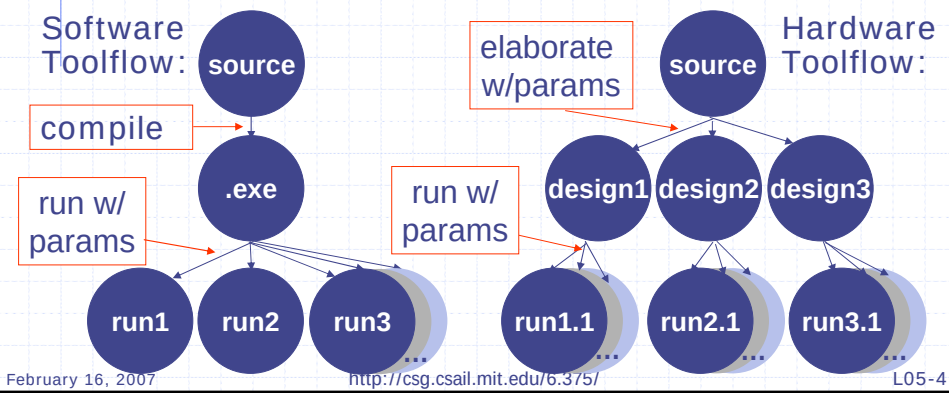
Now we call this Guarded Atomic Actions

- Rule conflict analysis
- Rule scheduling

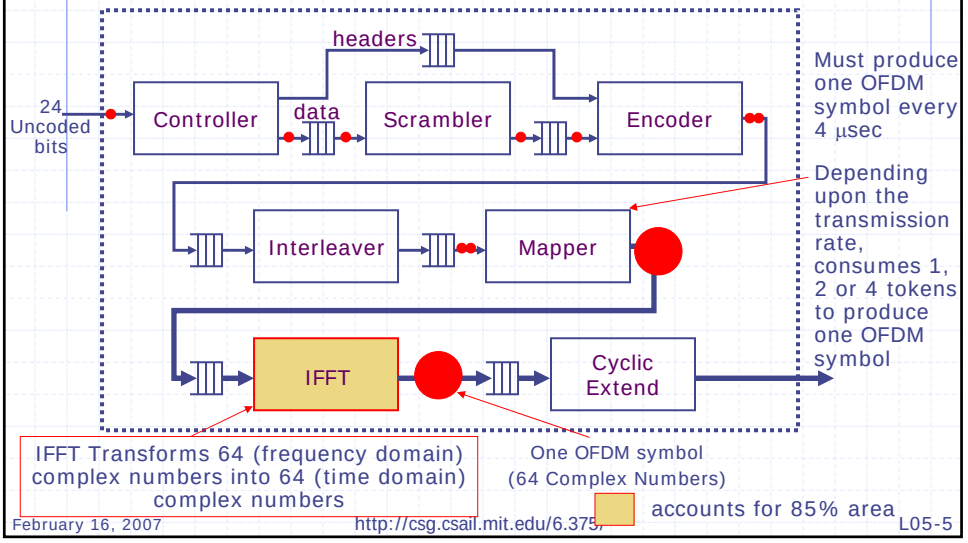
James Hoe & Arvind
@MIT 1997-2000

Static Elaboration

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



802.11a Transmitter Overview



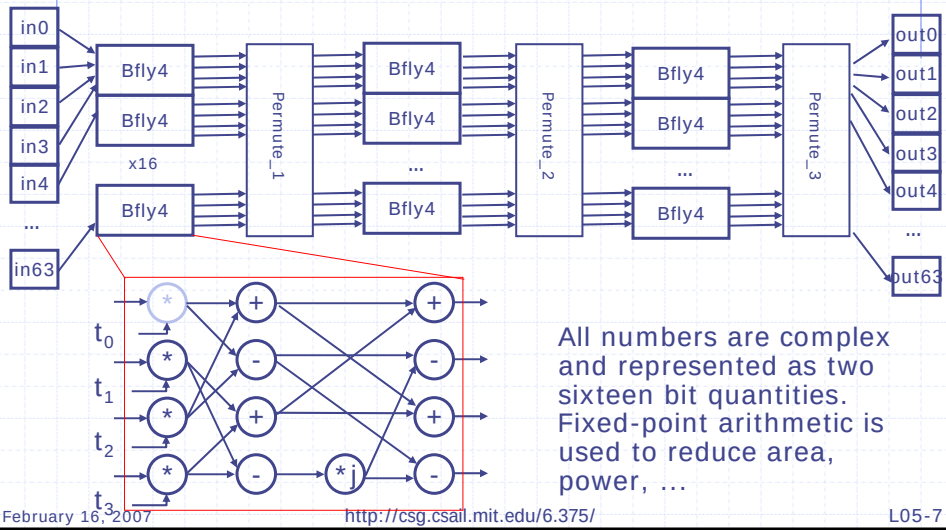
Preliminary results

[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%

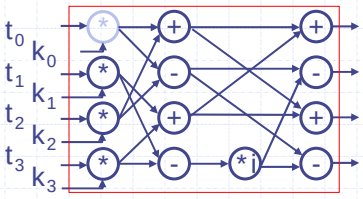
Complex arithmetic libraries constitute another 200 lines of code

Combinational IFFT



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

4-way Butterfly Node

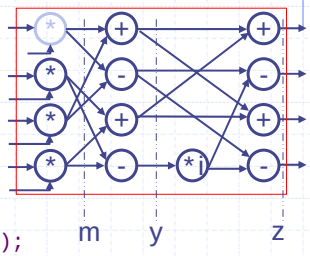



```
function Vector#(4,Complex) Bfly4
         (Vector#(4,Complex) t,  Vector#(4,Complex) k);
```

- 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

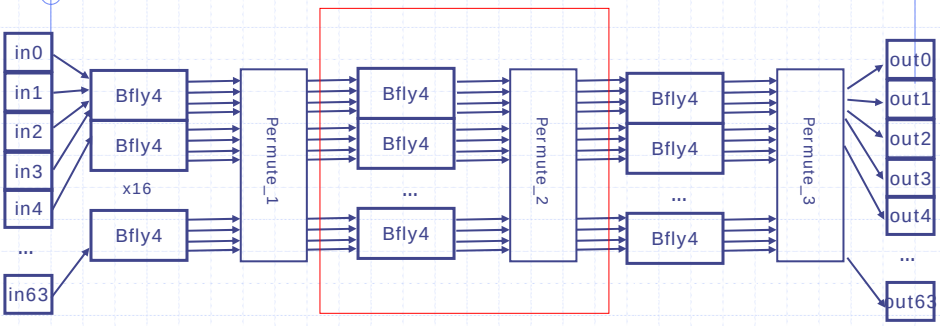
```
function Vector#(4,Complex) Bfly4
          (Vector#(4,Complex) t,  Vector#(4,Complex) k);

  Vector#(4,Complex) m = newVector(),
                     y = newVector(),
                     z = newVector();

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

Combinational IFFT



stage_f function

repeat it three times

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]);
```

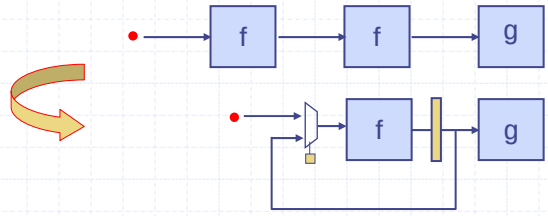
Bluespec Code for stage_f

```
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);
```


Architectural Exploration

Design Alternatives

Reuse a block over multiple cycles



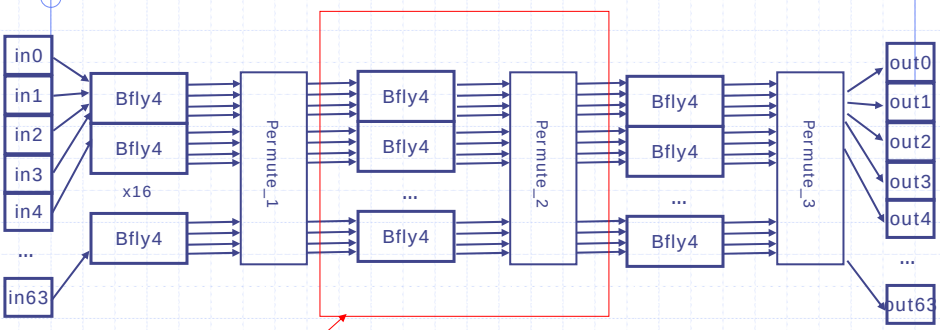
we expect:

Throughput to

Area to

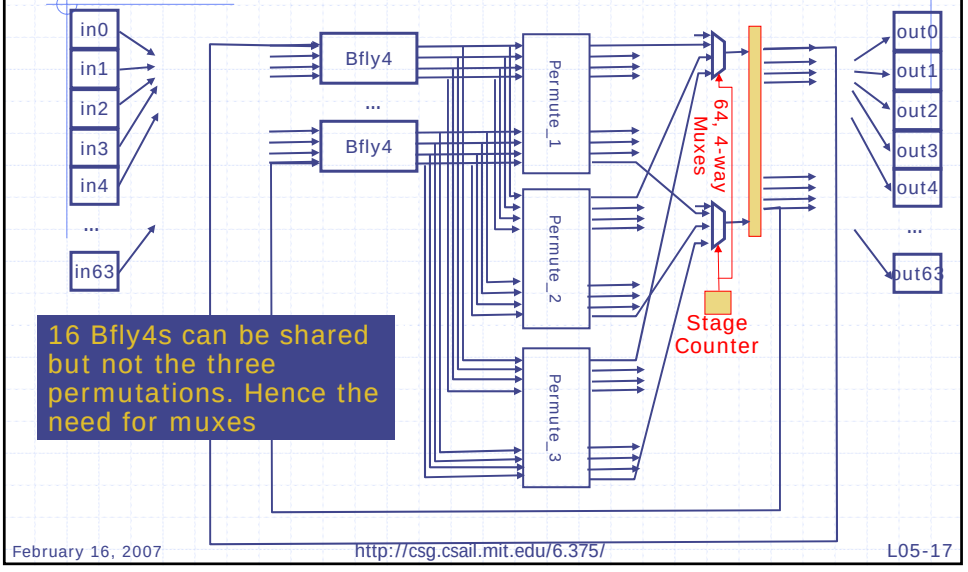
Combinational IFFT

Opportunity for reuse

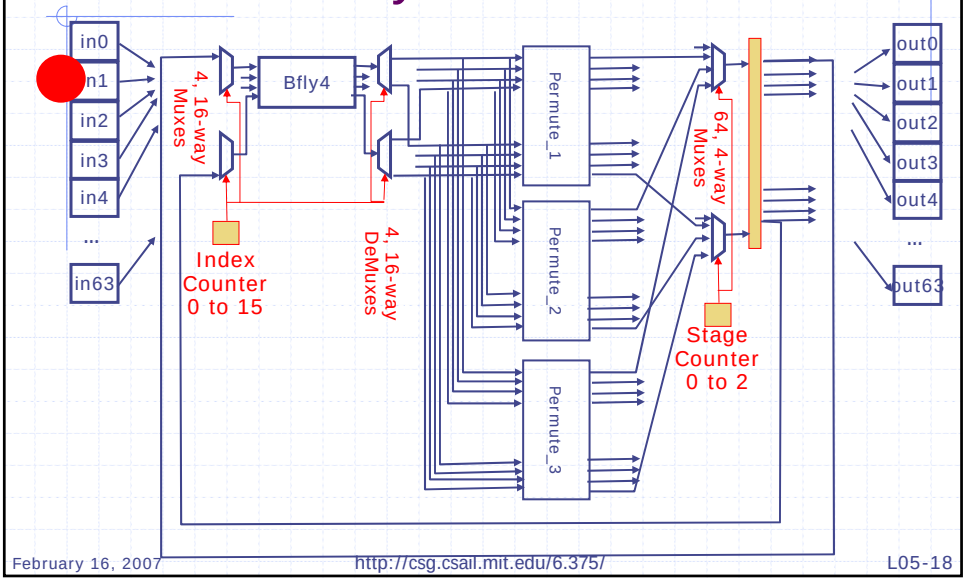


Reuse the same circuit three times

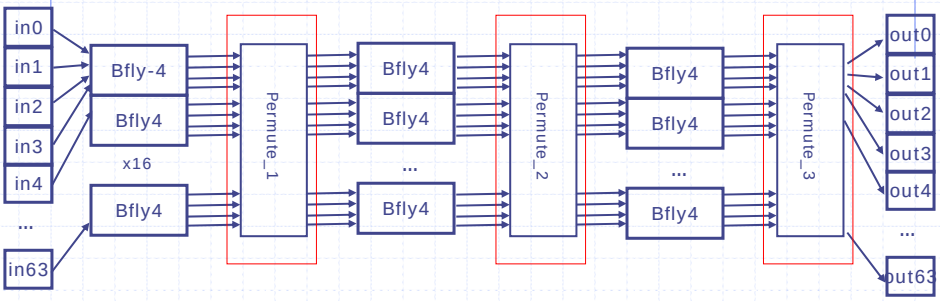
Circular pipeline: Reusing the Pipeline Stage



Superfolded circular pipeline: Just one Bfly-4 node!



Algorithmic Improvements



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

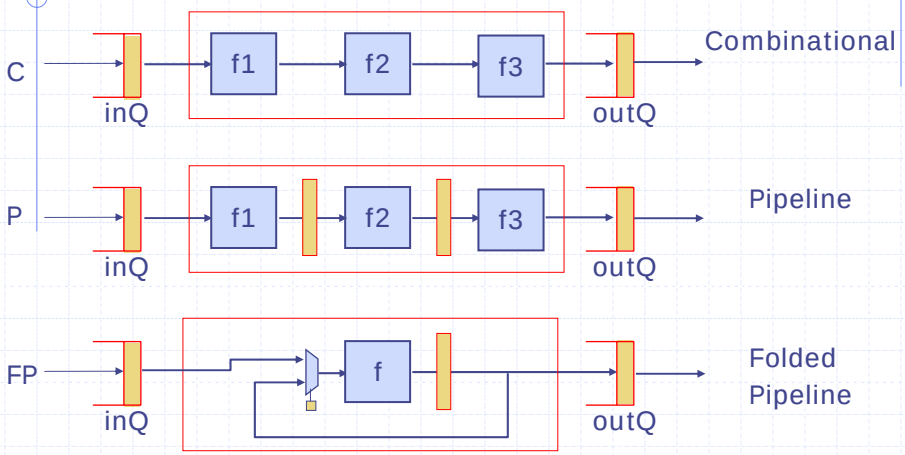
Design	Old Area (mm^2)	New Area (mm^2)
Combinational	4.69	4.91
Simple Pipe	5.14	5.25
Folded Pipe	5.89	3.97

Which design consumes the least energy to transmit a symbol?

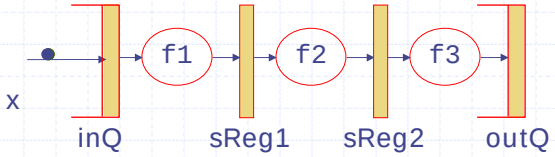
- Can we quickly code up all the alternatives?
  - single source with parameters?

5-minute break to stretch you legs

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

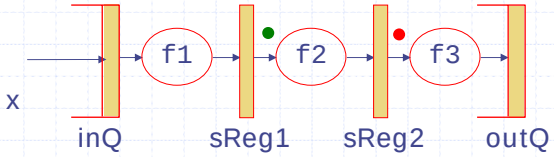
Atomicity: Either *all* or *none* of the state elements inQ, outQ, sReg1 and sReg2 will be updated

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 fucntion is given on slide 12

Problem: What about pipeline bubbles?




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

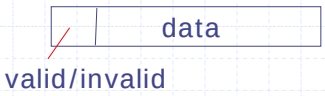
Red and Green tokens must move even if there is nothing in the inQ!

Also if there is no token in sReg2 then nothing should be enqueued in the outQ

Modify the rule to deal with these conditions

The Maybe type data in the pipeline

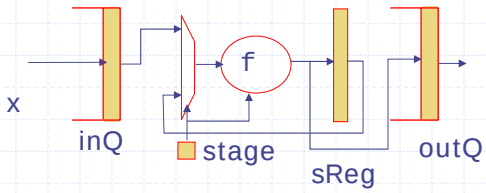
```
typedef union tagged {
  void Invalid;
  data_T Valid;
} Maybe#(type data_T);
```



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
```

Folded pipeline



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

```
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)

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

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

Backup slides