

Combinational Circuits in Bluespec

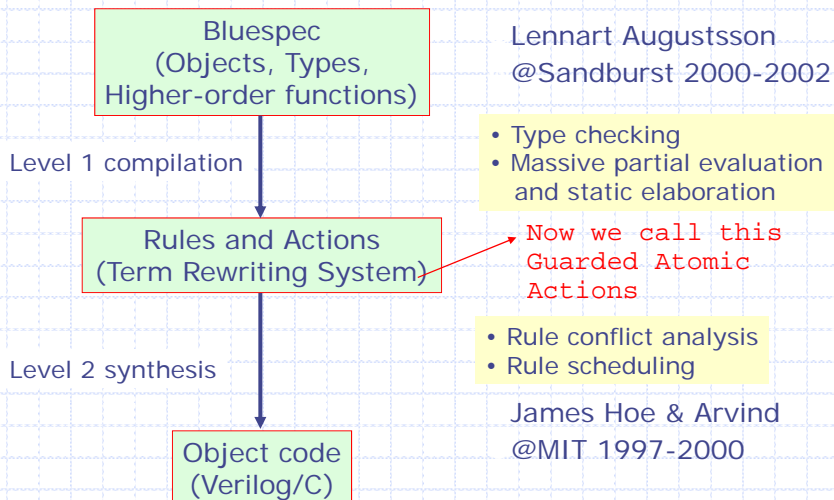
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-1

Bluespec: Two-Level Compilation



February 9, 2011

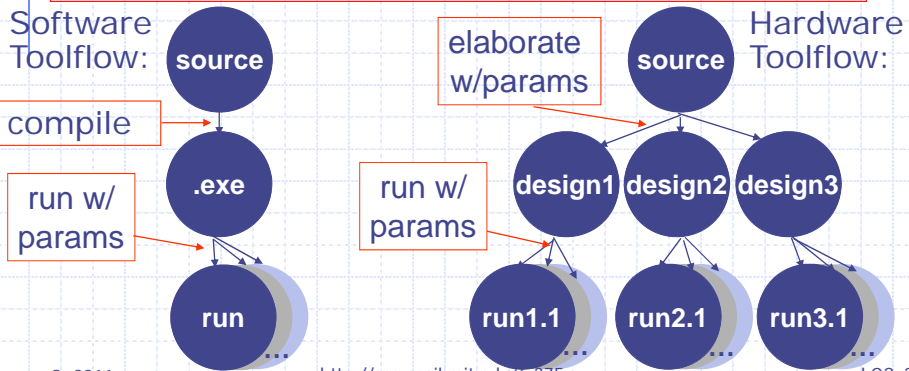
<http://csg.csail.mit.edu/6.375>

L03-2

Static Elaboration

At compile time

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

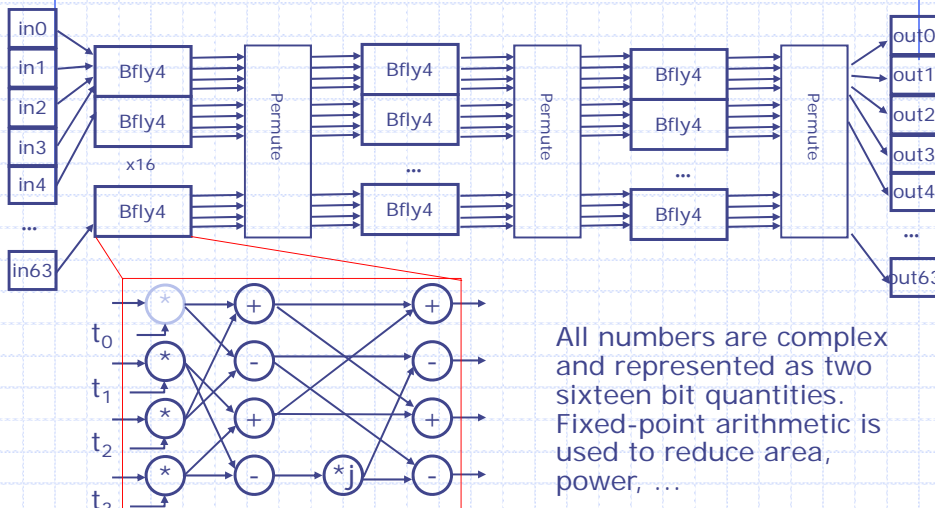


February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-3

Combinational IFFT



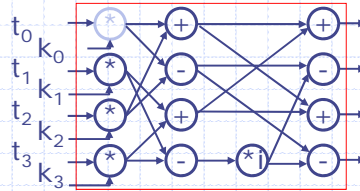
All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-4

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);
```

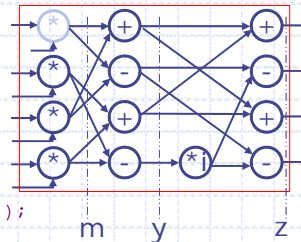
```
Vector#(4,Complex) m, y, z;
```

```
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
```



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

Note: Vector does not mean storage

Complex Arithmetic

◆ Addition

- $Z_R = X_R + Y_R$
- $Z_I = X_I + Y_I$

◆ Multiplication

- $Z_R = X_R * Y_R - X_I * Y_I$
- $Z_I = X_R * Y_I + X_I * Y_R$

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

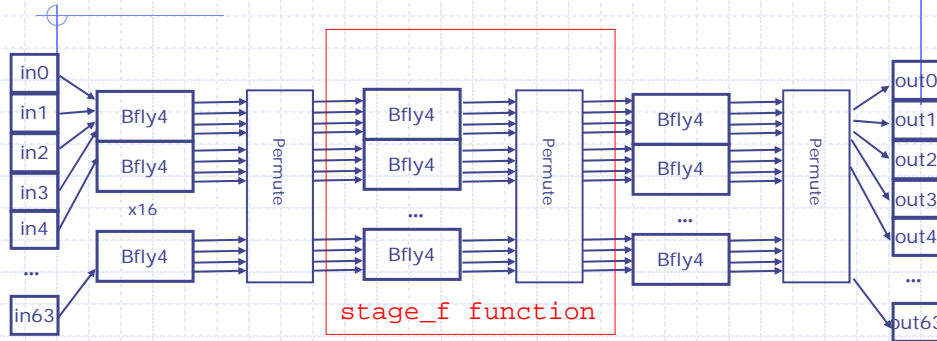
BSV code for Addition

```
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
```

What is the type of this + ?

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

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-9

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

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-10

BSV Code: Combinational IFFT- Unfolded

```
function Vector#(64, Complex) iff
  (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]);
```

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-11

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);
```

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-12

Higher-order functions: Stage functions f1, f2 and f3

```
function f0(x);  
    return (stage_f(0,x));  
endfunction  
  
function f1(x);  
    return (stage_f(1,x));  
endfunction  
  
function f2(x);  
    return (stage_f(2,x));  
endfunction
```

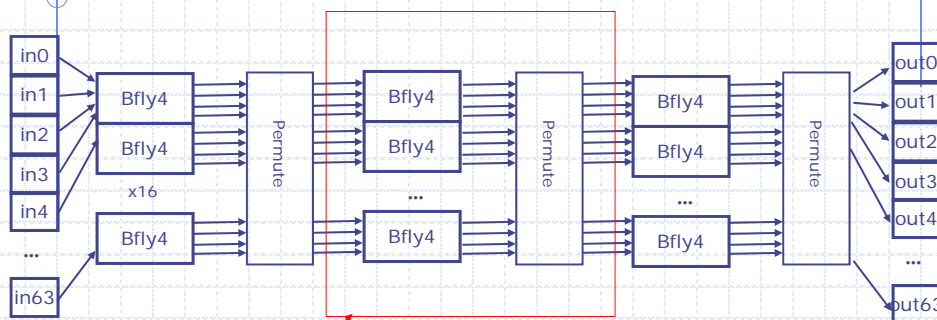
What is the type of $f0(x)$?

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-13

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



Reuse the same circuit three times
to reduce area

But why?

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-14

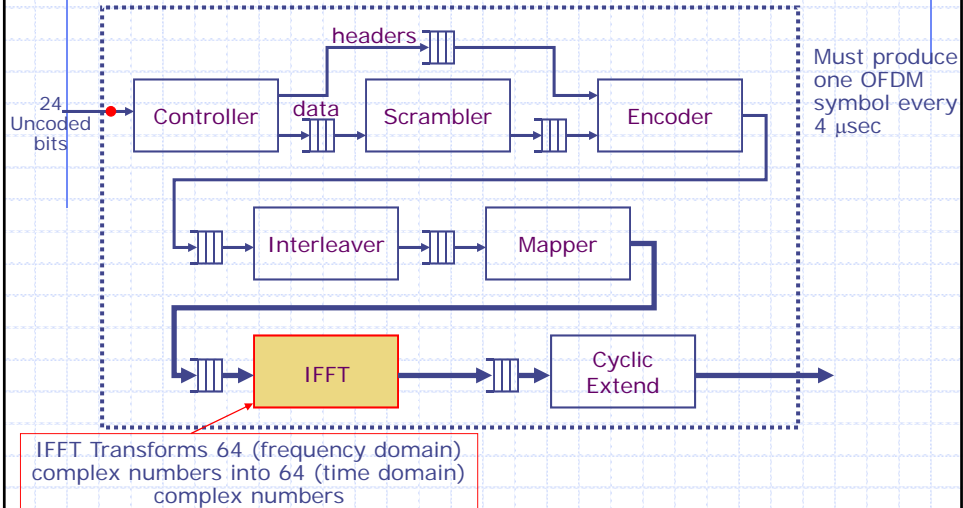
Architectural Exploration: Area-Performance tradeoff in 802.11a Transmitter

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-15

802.11a Transmitter Overview



February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-16

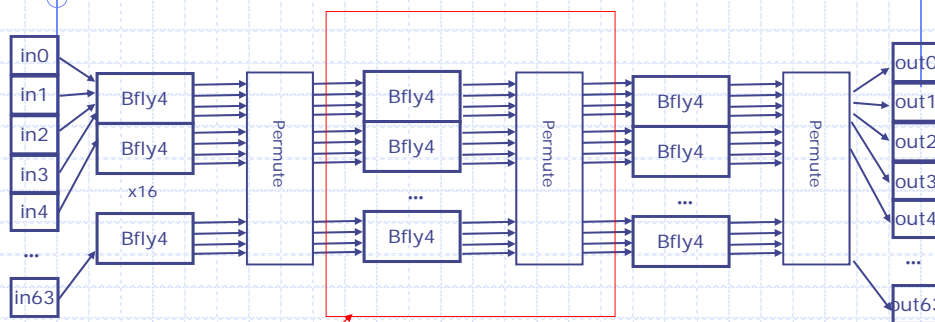
Preliminary results

[MEMOCODE 2006] Dave, Gerding, Pellauer, Arvind

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

Complex arithmetic libraries constitute another 200 lines of code

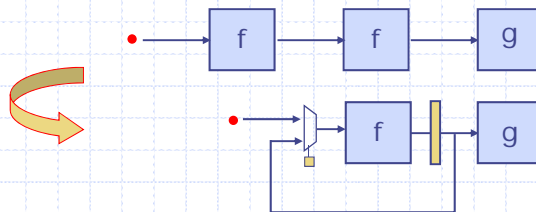
Combinational IFFT



Reuse the same circuit three times to reduce area

Design Alternatives

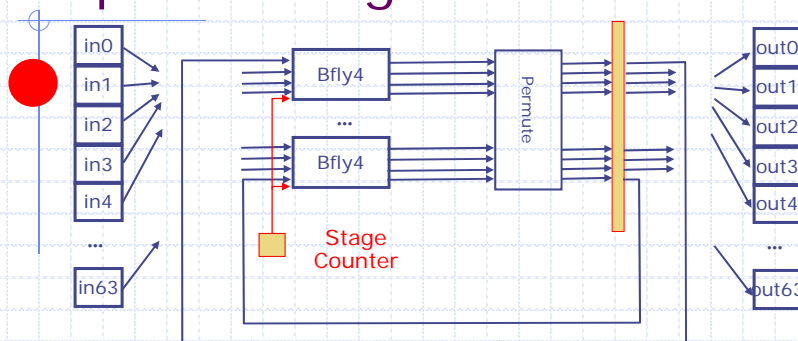
Reuse a block over multiple cycles



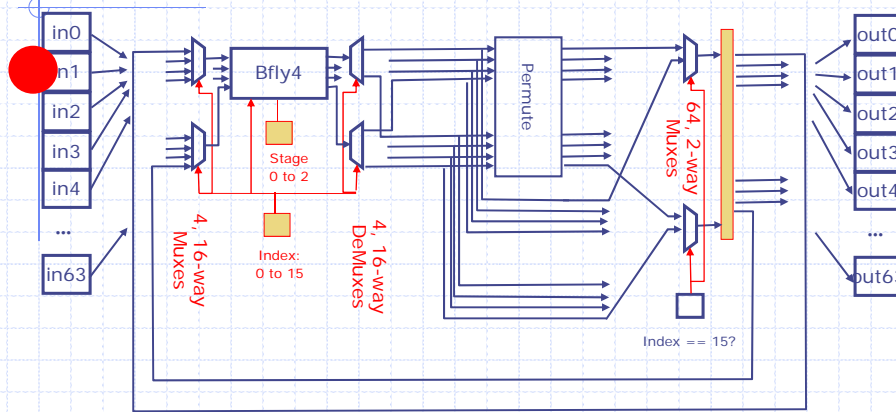
we expect:

Throughput to
Area to

Circular pipeline: Reusing the Pipeline Stage



Superfolded circular pipeline: Just one Bfly-4 node!

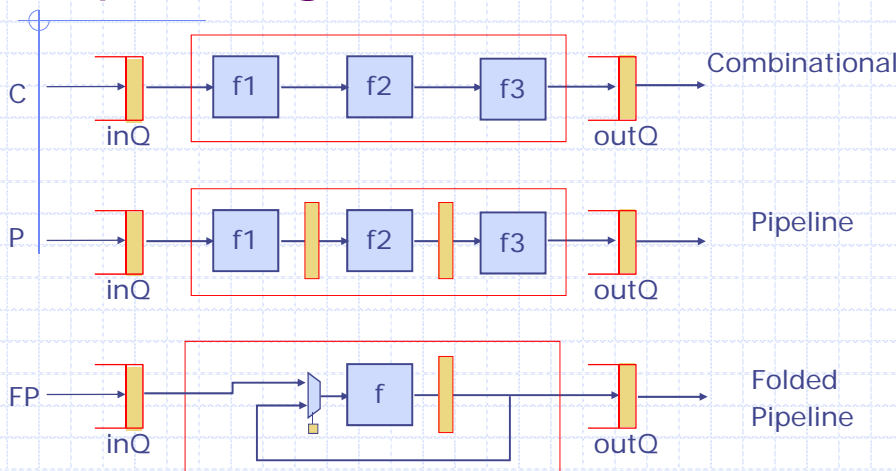


February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-21

Pipelining a block



Clock?

Area?

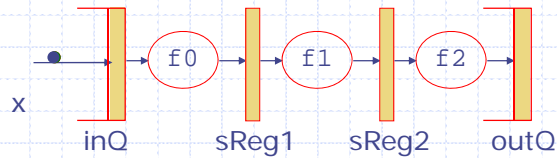
Throughput?

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-22

Inelastic pipeline



```
rule sync-pipeline (True);  
  inQ.deq();  
  sReg1 <= f0(inQ.first());  
  sReg2 <= f1(sReg1);  
  outQ.enq(f2(sReg2));  
endrule
```

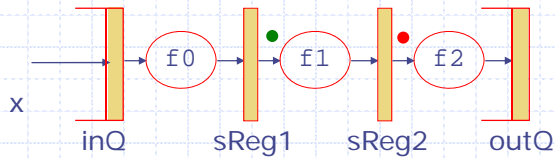
This rule can fire only if

Stage functions f1, f2 and f3

```
function f0(x);  
  return (stage_f(0,x));  
endfunction  
  
function f1(x);  
  return (stage_f(1,x));  
endfunction  
  
function f2(x);  
  return (stage_f(2,x));  
endfunction
```

The stage_f
function
was given
earlier

Problem: What about pipeline bubbles?



```

rule sync-pipeline (True);
  inQ.deq();
  sReg1 <= f0(inQ.first());
  sReg2 <= f1(sReg1);
  outQ.enq(f2(sReg2));
endrule

```

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-25

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 <= tagged Valid f0(inQ.first()); inQ.deq();
    end
  else sReg1 <= tagged Invalid;
  case (sReg1) matches
    tagged Valid .sx1: sReg2 <= tagged Valid f1(sx1);
    tagged Invalid: sReg2 <= tagged Invalid; endcase
  case (sReg2) matches
    tagged Valid .sx2: outQ.enq(f2(sx2));
  endcase
endrule

```

sx1 will get bound to the appropriate part of sReg1

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-26

When is this rule enabled?

```

rule sync-pipeline (True);
if (inQ.notEmpty())
begin sReg1 <= tagged Valid f0(inQ.first()); inQ.deq(); end
else sReg1 <= tagged Invalid;
case (sReg1) matches
tagged Valid .sx1: sReg2 <= tagged Valid f1(sx1);
tagged Invalid: sReg2 <= tagged Invalid; endcase
case (sReg2) matches
tagged Valid .sx2: outQ.enq(f2(sx2));
endcase
endrule
    
```

inQ	sReg1	sReg2	outQ		inQ	sReg1	sReg2	outQ	
NE	V	V	NF	yes	E	V	V	NF	yes
NE	V	V	F	No	E	V	V	F	No
NE	V	I	NF		E	V	I	NF	
NE	V	I	F		E	V	I	F	
NE	I	V	NF		E	I	V	NF	
NE	I	V	F		E	I	V	F	
NE	I	I	NF		E	I	I	NF	
NE	I	I	F		E	I	I	F	

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-27

Next lecture

Folded pipeline for FFT

February 9, 2011

<http://csg.csail.mit.edu/6.375>

L03-28