

Expressing designs for IFFT in Bluespec (BSV)

Arvind

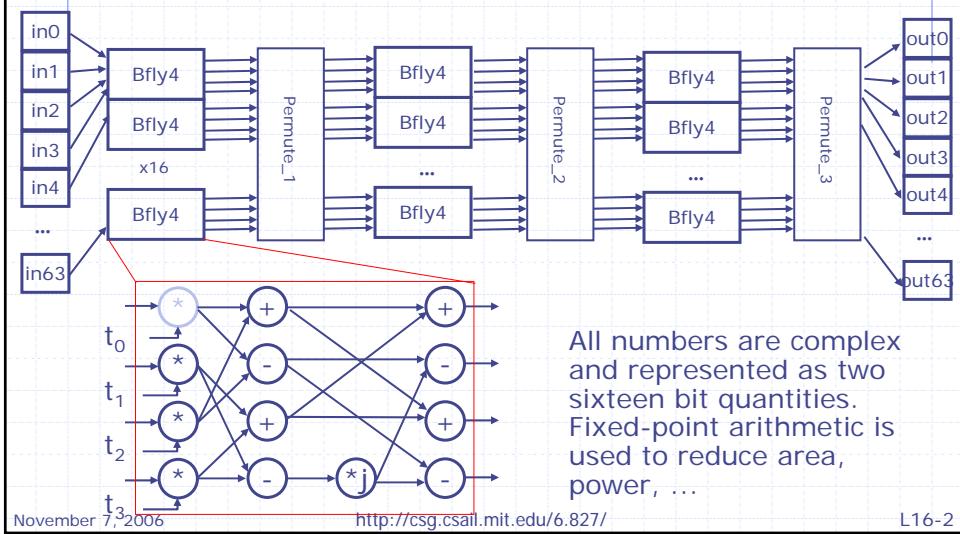
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

November 7, 2006

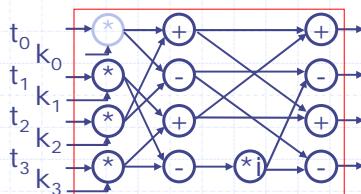
<http://csg.csail.mit.edu/6.827/>

L16-1

Combinational IFFT



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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-3

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();

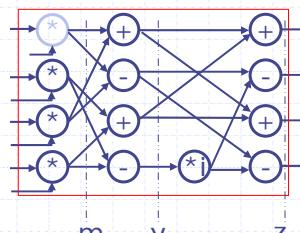
    m[0] = k[0] * t[0]; m[1] = k[1] * t[1];
    m[2] = k[2] * t[2]; m[3] = k[3] * t[3];

    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
```

Note: Vector ≠ does not mean storage

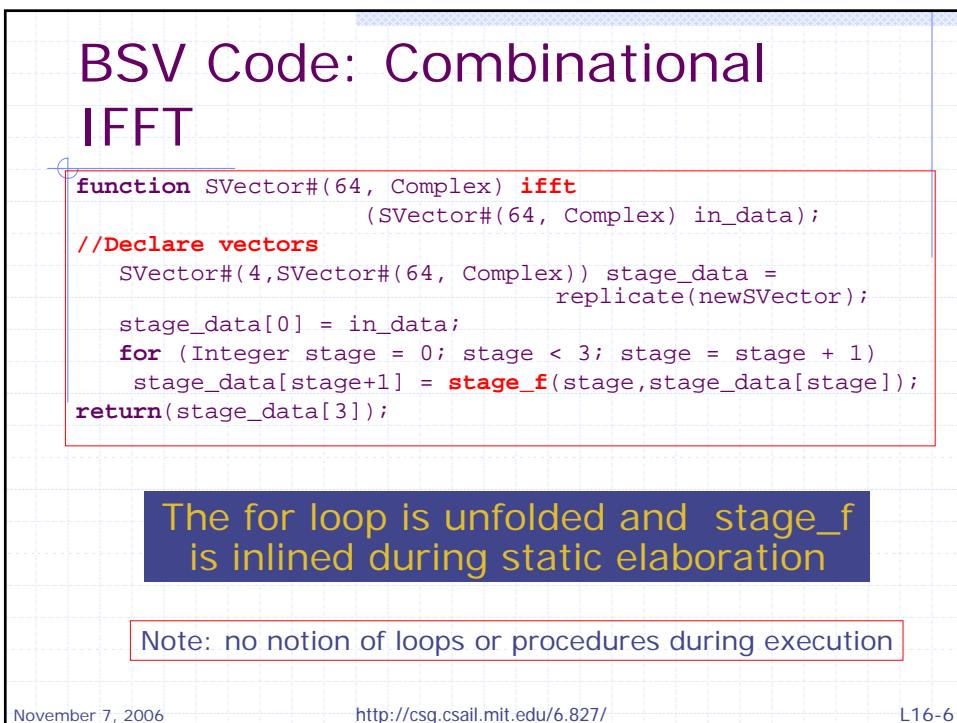
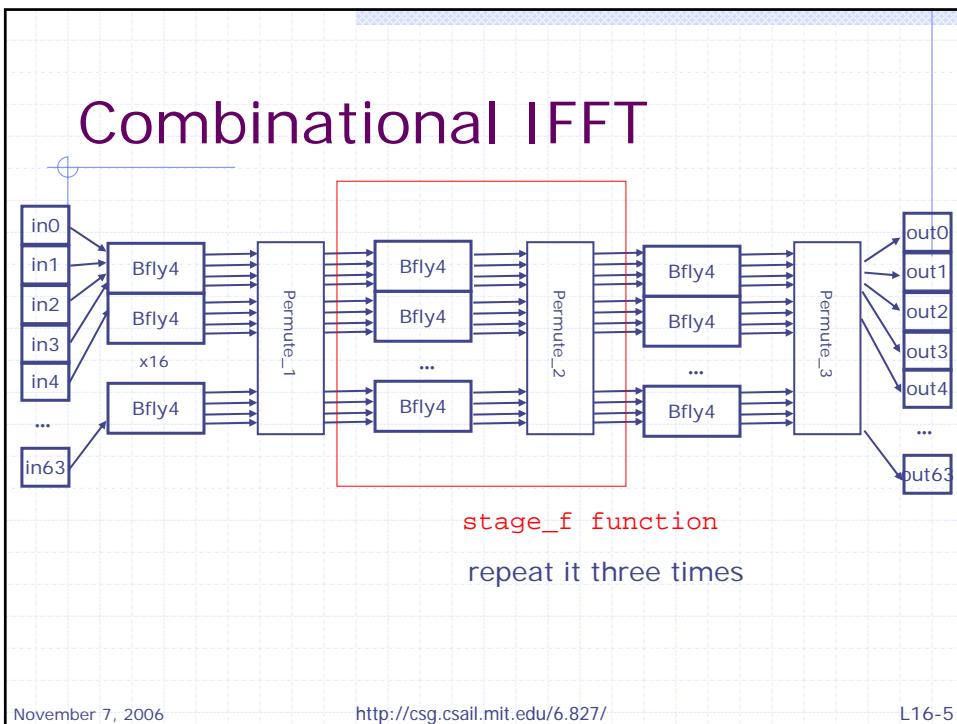


Polymorphic code:
works on any type
of numbers for
which *, + and -
have been defined

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-4



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[1] = stage_f(0,stage_data[0]);  lata[stage]);
        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?

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-7

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 twiddle = getTwiddle(stage, fromInteger(i));
            let y = bfly4(twiddle, 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);
```

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-8

Architectural Exploration

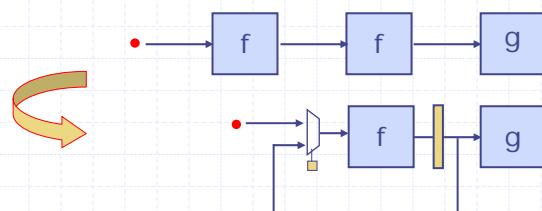
November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-9

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

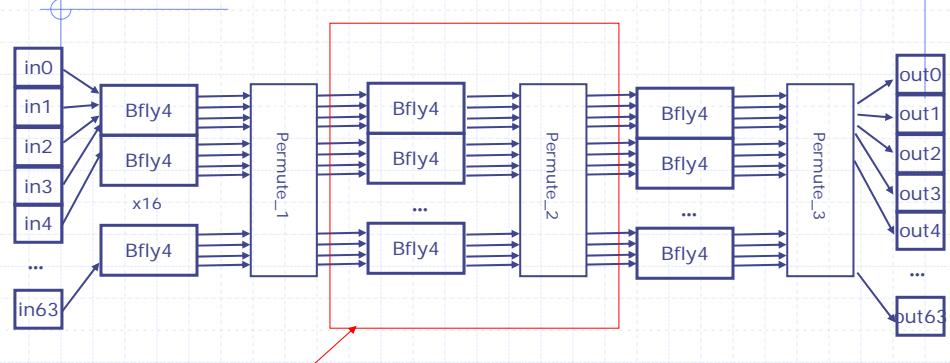
November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-10

Combinational IFFT

Opportunity for reuse

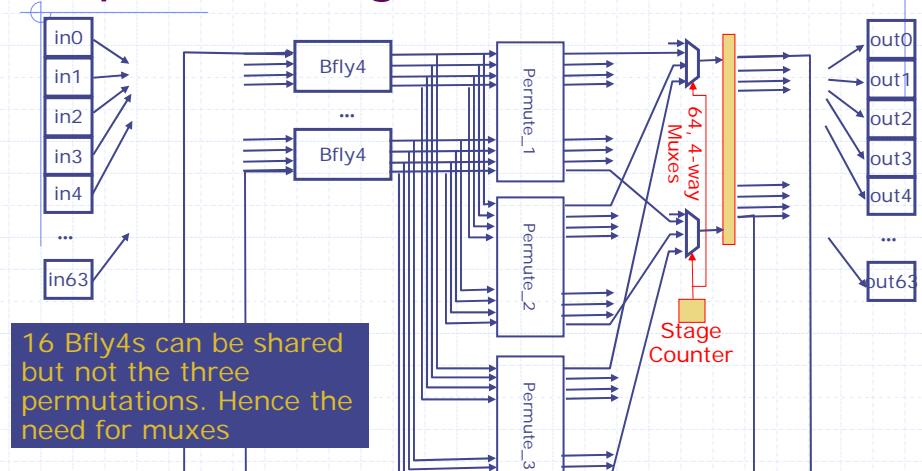


November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-11

Circular pipeline: Reusing the Pipeline Stage

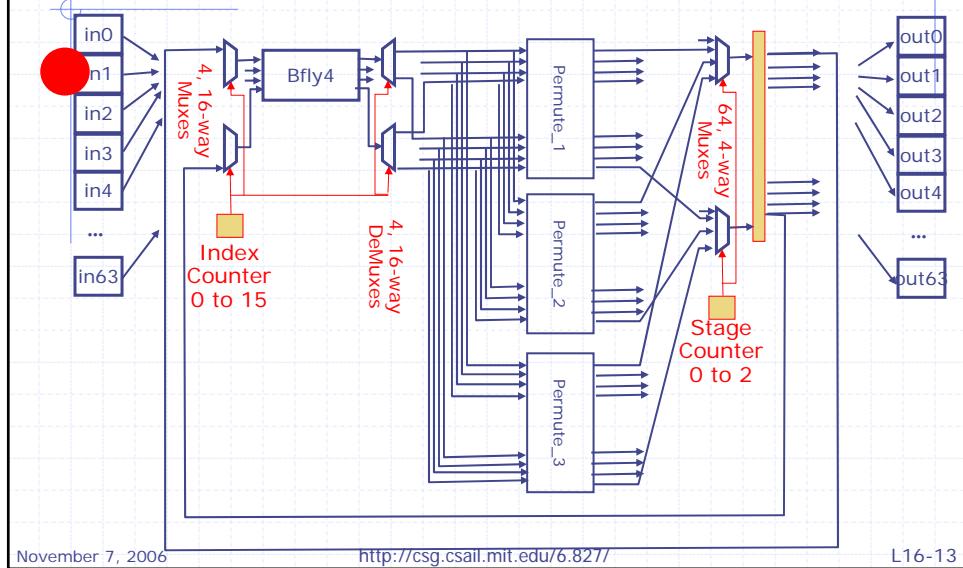


November 7, 2006

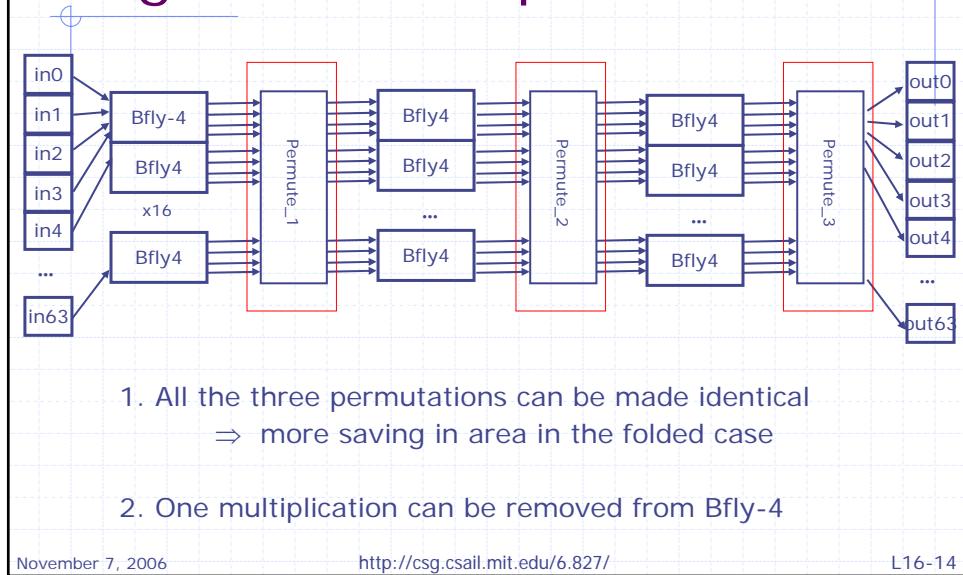
<http://csg.csail.mit.edu/6.827/>

L16-12

Superfolded circular pipeline: Just one Bfly-4 node!



Algorithmic Improvements



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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-15

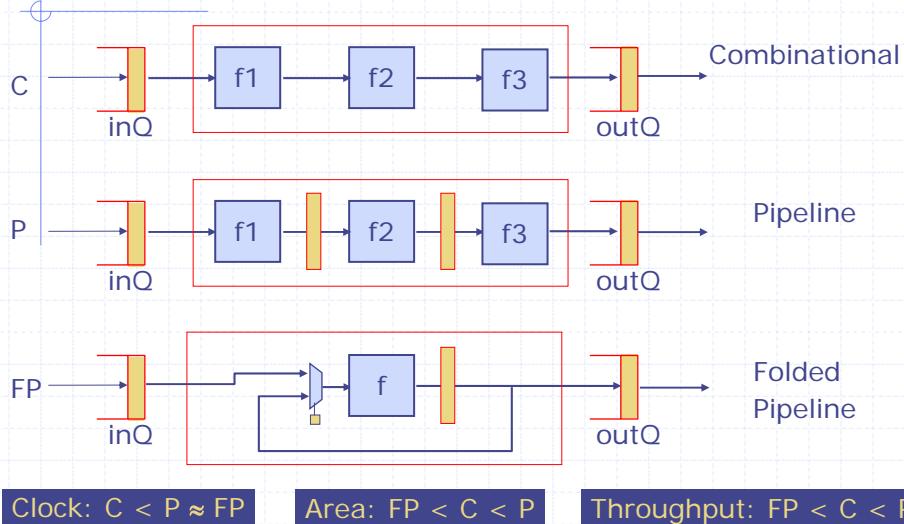
Coding various pipelined versions in BSV

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-16

Pipelining a block

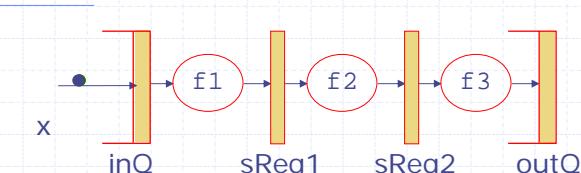


November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-17

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
- 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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-18

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
```

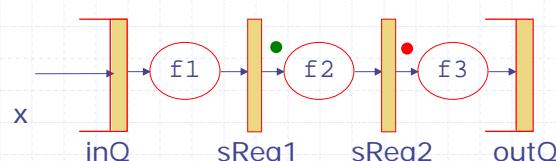
The stage_f function is given on slide 12

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-19

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

Valid bits or the Maybe type

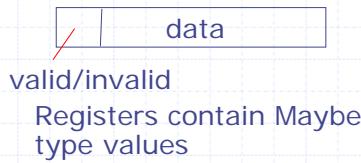
November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-20

The Maybe type data in the pipeline

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



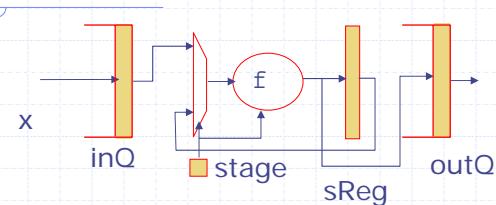
```
rule sync-pipeline (True);
    if (inQ.notEmpty())
        begin sReg1 <= Valid f1(inQ.first()); inQ.deq(); end
    else sReg1 <= Invalid;
    case (sReg1) matches
        tagged Valid .sx1: sReg2 <= Valid f2(sx1);
        tagged Invalid:     sReg2 <= Invalid;
    endcase
    case (sReg2) matches
        tagged Valid .sx2: outQ.enq(f3(sx2));
    endrule
```

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-21

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
```

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-22

Function f for the folded pipeline is the same stage_f function but ...

```
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);
end
```

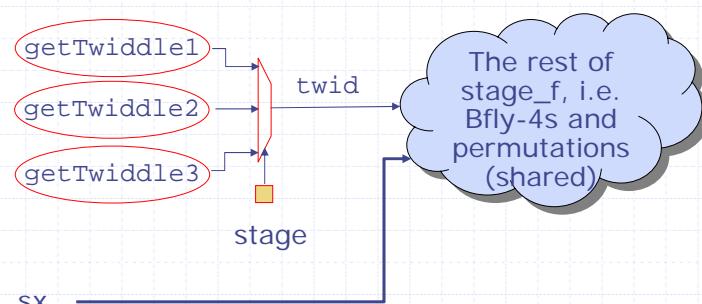
notice this is
no longer a
static
parameter!
⇒ will cause a
mux to be
generated

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-23

Folded pipeline:
stage function f



November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-24

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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-25

Code for the Superfolded pipeline stage function

```
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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-26

802.11a Transmitter Synthesis results (Only the IFFT block is changing)

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

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-27

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!

November 7, 2006

<http://csg.csail.mit.edu/6.827/>

L16-28