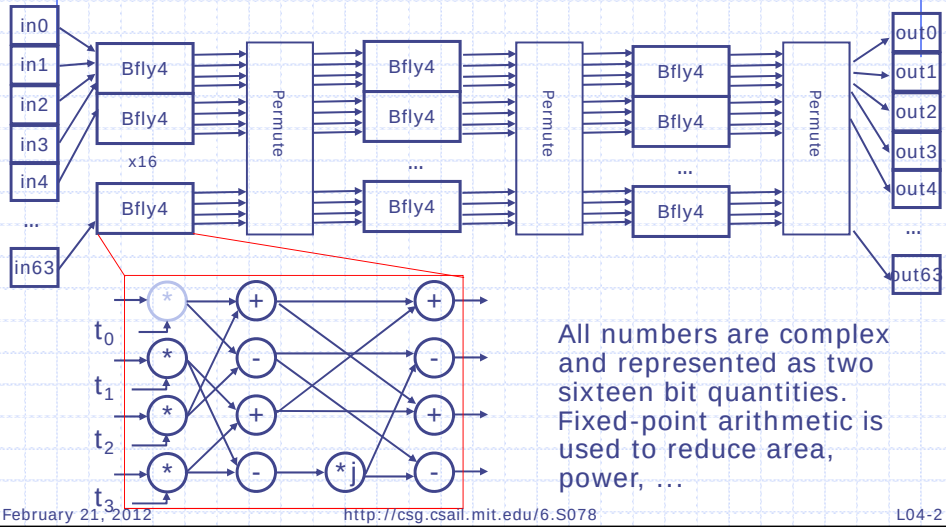
Folding complex combinational circuits to save area

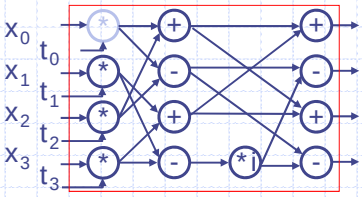
Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

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

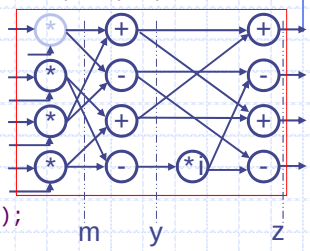



```
function Vector#(4,Complex) bfly4
          (Vector#(4,Complex) t,  Vector#(4,Complex) x);
```

- t's are mathematically derivable constants for each bfly4 and depend upon the position of bfly4 the in the network
- 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) x);

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

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

Complex Arithmetic

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

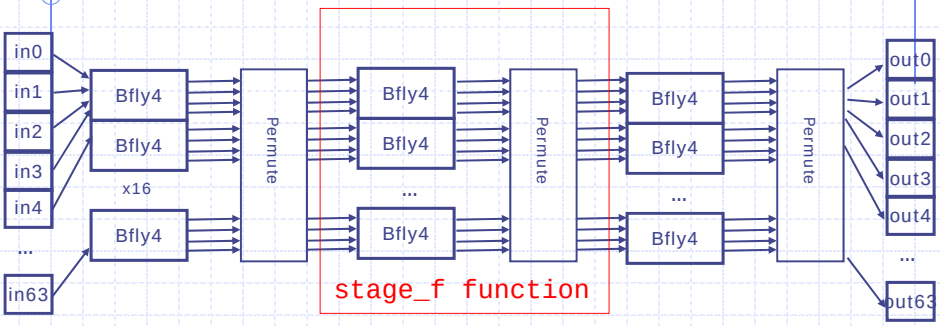
The actual arithmetic for FFT is different because we use a non-standard fixed point representation

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

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

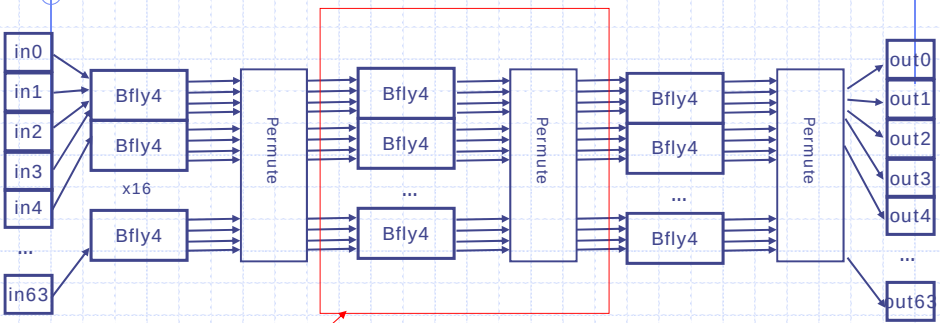
BSV Code: Combinational IFFT- Unfolded

```
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

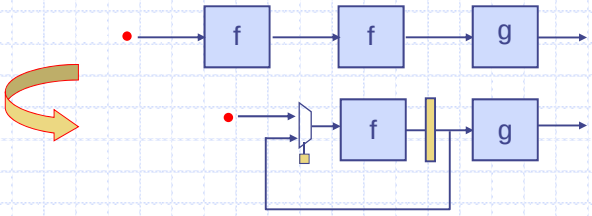
```
function Vector#(64, Complex) stage_f
        (Bit#(2) stage, Vector#(64, Complex) stage_in);
  begin
   for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid = getTwiddle(stage, fromInteger(i));
      let y = bfly4(twid, stage_in[idx:idx+3]);
      stage_temp[idx]   = 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 area of the circuit



Reuse the same circuit three times to reduce area

Reusing a combinational block



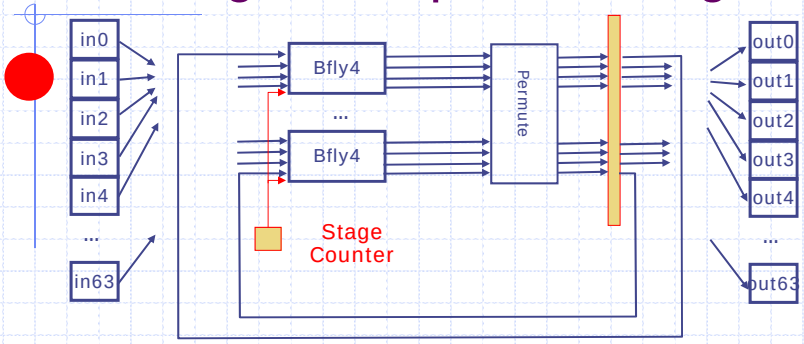
Introduce state elements to hold intermediate values

we expect:

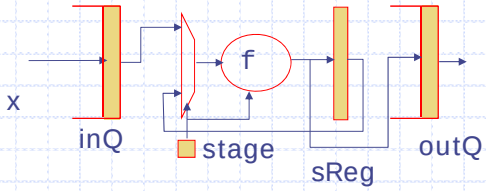
Throughput to

Area to

Circular or folded pipeline: Reusing the Pipeline Stage



Folded pipeline



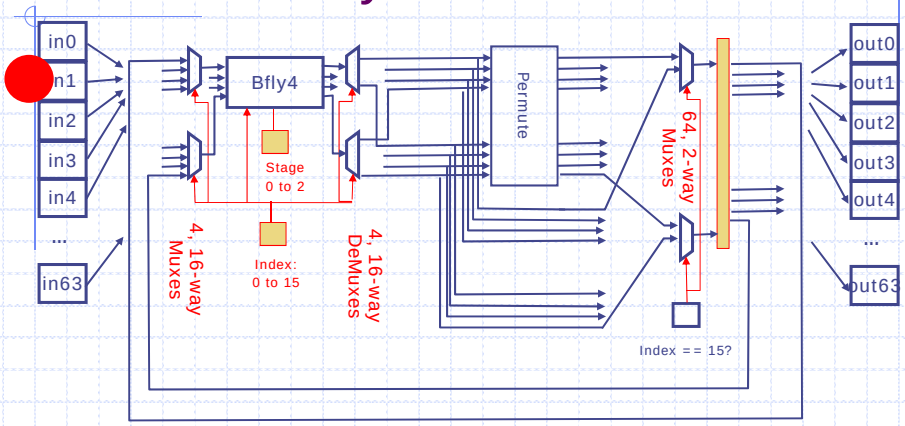

```
rule folded-pipeline (True);
 if (stage==0)
   begin sxIn= inQ.first(); inQ.deq(); end
 else    sxIn= sReg;
 sxOut = f(stage,sxIn);
 if (stage==n-1) outQ.enq(sxOut);
 else sReg <= sxOut;
 stage <= (stage==n-1)? 0 : stage+1;
endrule
```

no for-loop

Need type declarations for sxIn and sxOut

Extras: may be useful for Lab 3

Superfolded circular pipeline: Just one Bfly-4 node!



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

Superfolded pipeline: stage function f

Bit#(2+4) (stage,i)

```
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 Bit#(2) stage
        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);
```

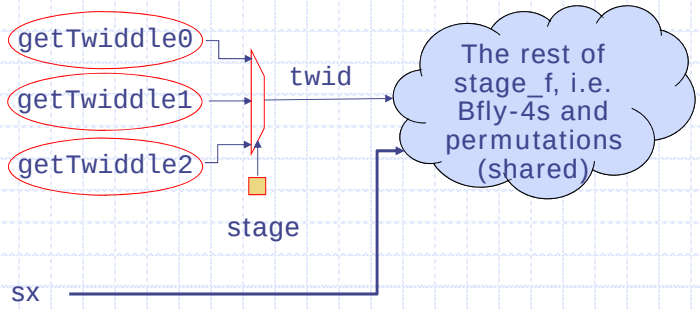
should be done only when i=15

Code for the Superfolded pipeline stage function

```
Function Vector#(64, Complex) f
        (Bit#(6) stagei, Vector#(64, Complex) stage_in);
   let i = stagei `mod` 16;
   let twid = getTwiddle(stagei `div` 16, i);
   let y = bfly4(twid, stage_in[i:i+3]);

   let stage_temp = stage_in;
   stage_temp[i]   = y[0];
   stage_temp[i+1] = y[1];
   stage_temp[i+2] = y[2];
   stage_temp[i+3] = y[3];

   let stage_out = stage_temp;
   if (i == 15)
     for (Integer i = 0; i < 64; i = i + 1)
       stage_out[i] = stage_temp[permute[i]];
   return(stage_out);
endfunction
```

One Bfly-4 case

Folded pipeline: stage function f



- The Twiddle constants can be expressed in a table or in a case expression