Lecture from 6.s195 taught in Fall 2103

Constructive Computer Architecture

FFT: An example of complex combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

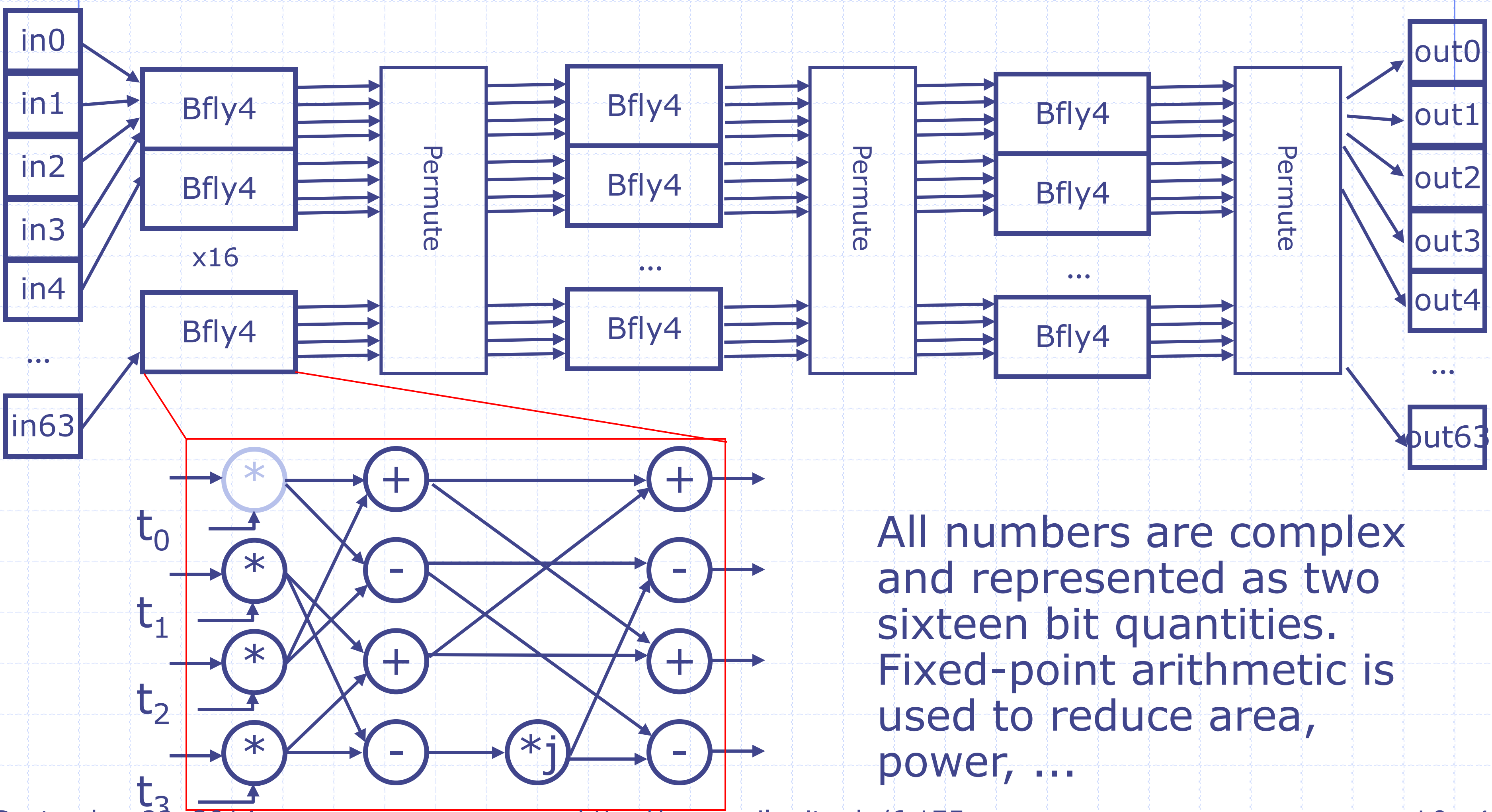
Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion

Contents

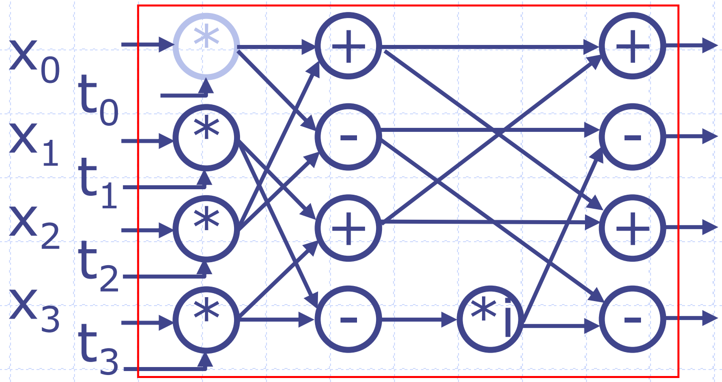
- FFT and IFFT: Another complex combinational circuit and its folded implementations
  - FFT: Converts signals from time domain to frequency domain
  - IFFT: Converts signals from frequency domain to time domain
  - Two calculations are identical- the same hardware can be used
- New BSV concepts
  - structure type
  - overloading

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

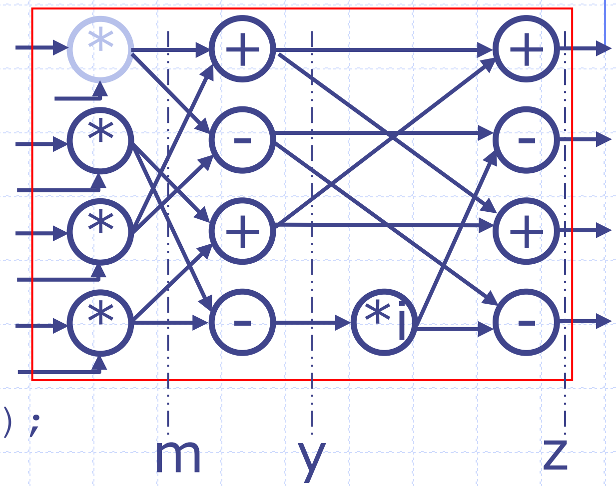



```
function Vector#(4,Complex) bfly4
                (Vector#(4,Complex) t,  Vector#(4,Complex) x);
```

- t's (twiddle coefficients) are mathematically derivable constants for each bfly4 and depend upon the position of bfly4 the in the network
- FFT and IFFT calculations differ only in the use of Twiddle coefficients in various butterfly nodes

BSV code: 4-way Butterfly

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

  Vector#(4,Complex#(s)) 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
```



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

Note: Vector does not mean storage; just a group of wires with names

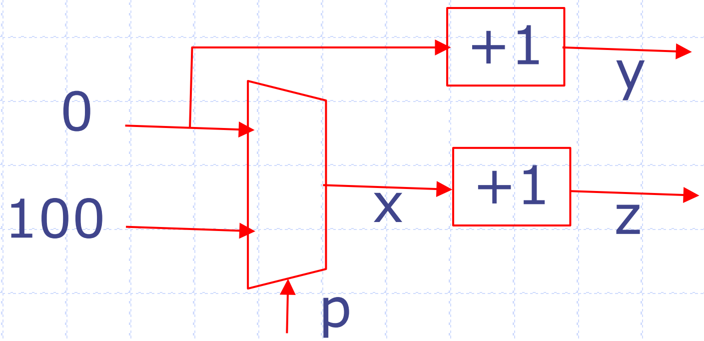
Language notes: Sequential assignments

- Sometimes it is convenient to reassign a variable (x is zero every where except in bits 4 and 8):

```
Bit#(32) x = 0;
x[4] = 1; x[8] = 1;
```

- This will usually result in introduction of muxes in a circuit as the following example illustrates:

```
Bit#(32) x = 0;
let y = x+1;
if(p) x = 100;
let z = x+1;
```



Complex Arithmetic

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

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

Representing complex numbers as a `struct`

```
typedef struct{
   Int#(t) r;
   Int#(t) i;
} Complex#(numeric type t) deriving (Eq,Bits);
```

Notice the Complex type is parameterized by the size of Int chosen to represent its real and imaginary parts

If x is a struct then its fields can be selected by writing x.r and x.i

BSV code for Addition

```
typedef struct{
  Int#(t) r;
  Int#(t) i;
} Complex#(numeric type t) deriving (Eq,Bits);

function Complex#(t) cAdd
             (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 + ?

Overloading (Type classes)

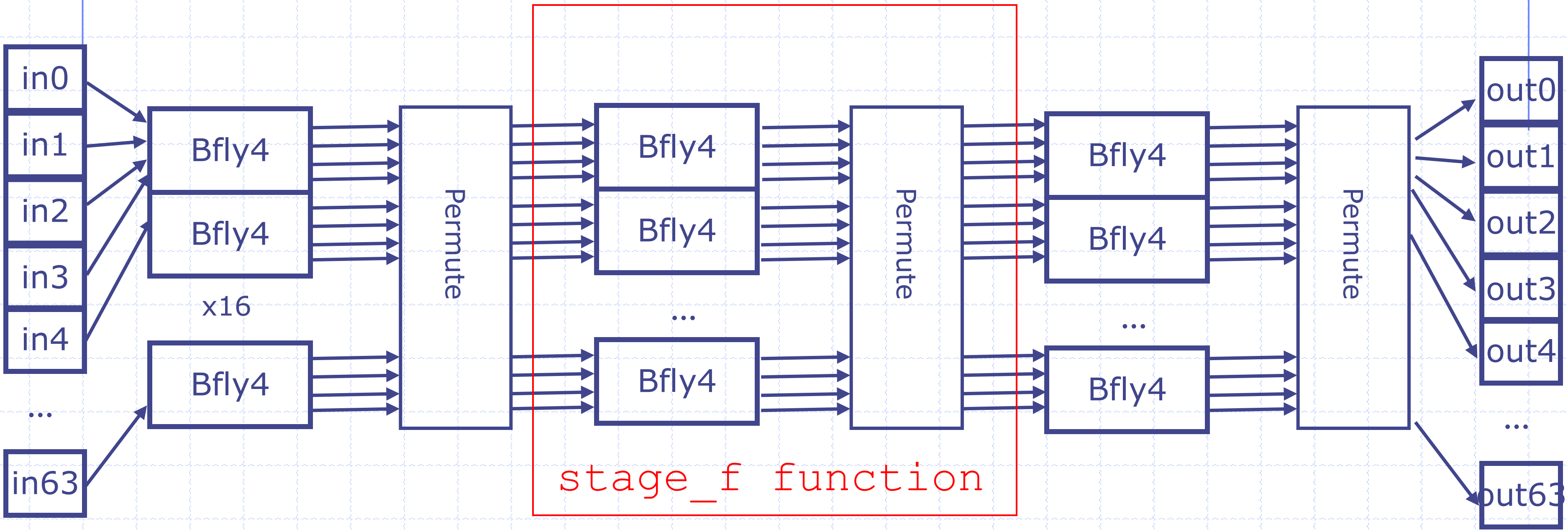
- The same symbol can be used to represent different but related operators using Type classes
- A type class groups a bunch of types with similarly named operations. For example, the type class Arith requires that each type belonging to this type class has operators +,-, *, / etc. defined
- We can declare Complex type to be an instance of Arith type class

Overloading +, *

```
instance Arith#(Complex#(t));
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

function Complex#(t) \*
                    (Complex#(t) x, Complex#(t) y);
   Int#(t) real = x.r*y.r - x.i*y.i;
   Int#(t) imag = x.r*y.i + x.i*y.r;
   return(Complex{r:real, i:imag});
endfunction
…
endinstance
```

The context allows the compiler to pick the appropriate definition of an operator

Combinational IFFT




```
function Vector#(64, Complex#(n)) stage_f
      (Bit#(2) stage, Vector#(64, Complex#(n)) stage_in);
```

```
function Vector#(64, Complex#(n)) ifft
      (Vector#(64, Complex#(n)) in_data);
```

repeat `stage_f` three times

BSV Code: Combinational IFFT

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

   stage_data[0] = in_data;
   for (Bit#(2) stage = 0; stage < 3; stage = stage + 1)
    stage_data[stage+1] = stage_f(stage,stage_data[stage]);
return(stage_data[3]);
endfunction
```

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

   stage_data[0] = in_data;
    stage_data[1] = stage_f(0,stage_data[0]); stage + 1)
    stage_data[2] = stage_f(1,stage_data[1]); data[stage]);
    stage_data[3] = stage_f(2,stage_data[2]);
return(stage_data[3]);
endfunction
```

Stage_f can be inlined now; it could have been inlined before loop unfolding also.

Does the order matter?

BSV Code for `stage_f`

```
function Vector#(64, Complex#(n)) stage_f
        (Bit#(2) stage, Vector#(64, Complex#(n)) stage_in);
Vector#(64, Complex#(n)) stage_temp, stage_out;
   for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      Vector#(4, Complex#(n)) x;
      x[0] = stage_in[idx];   x[1] = stage_in[idx+1];
      x[2] = stage_in[idx+2]; x[3] = stage_in[idx+3];
      let twid = getTwiddle(stage, fromInteger(i));
      let y = bfly4(twid, x);
      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]];
  return(stage_out);
endfunction
```

twid's are mathematically derivable constants

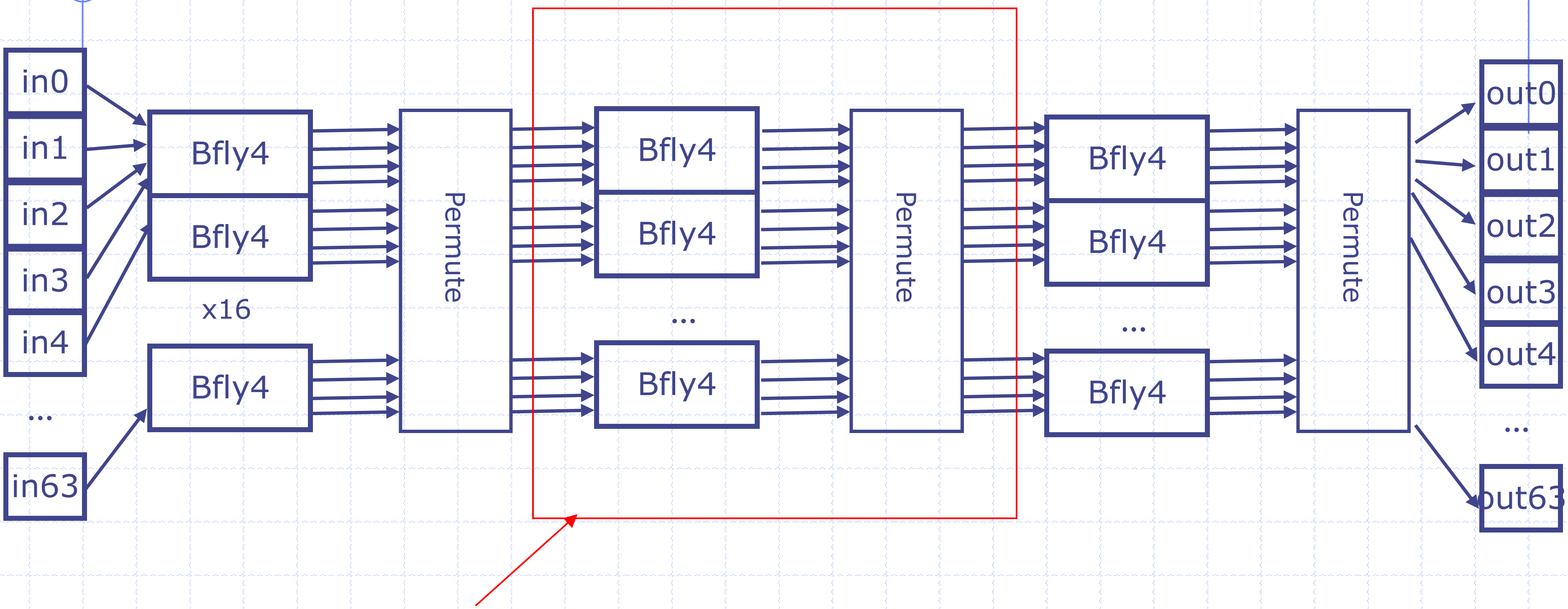
Higher-order functions: Stage functions f1, f2 and f3

```
function f0(x)=  stage_f(0,x);

function f1(x)=  stage_f(1,x);

function f2(x)=  stage_f(2,x);
```

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

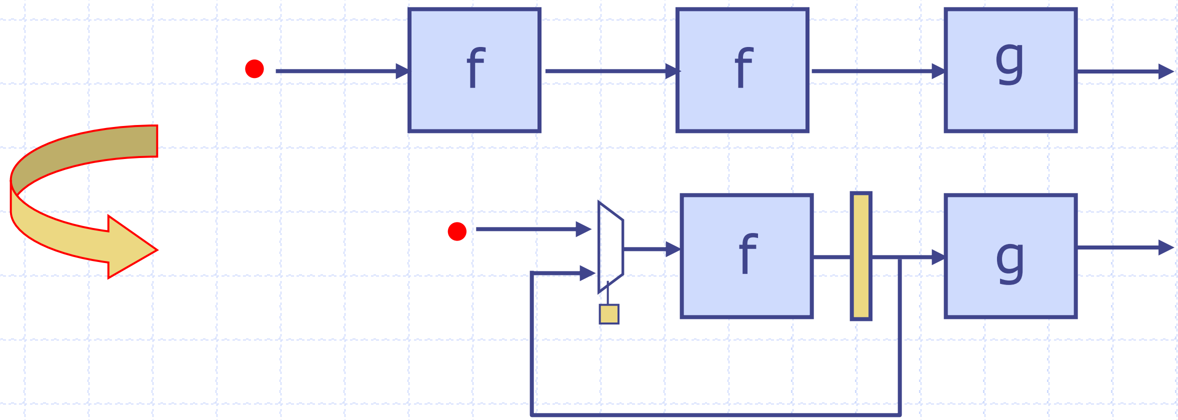
```
function Vector#(64, Complex) f0
        (Vector#(64, Complex) x);
```

Suppose we want to reduce the area of the circuit



Reuse the same circuit three times to reduce area

Reusing a combinational block



Introduce state elements to hold intermediate values

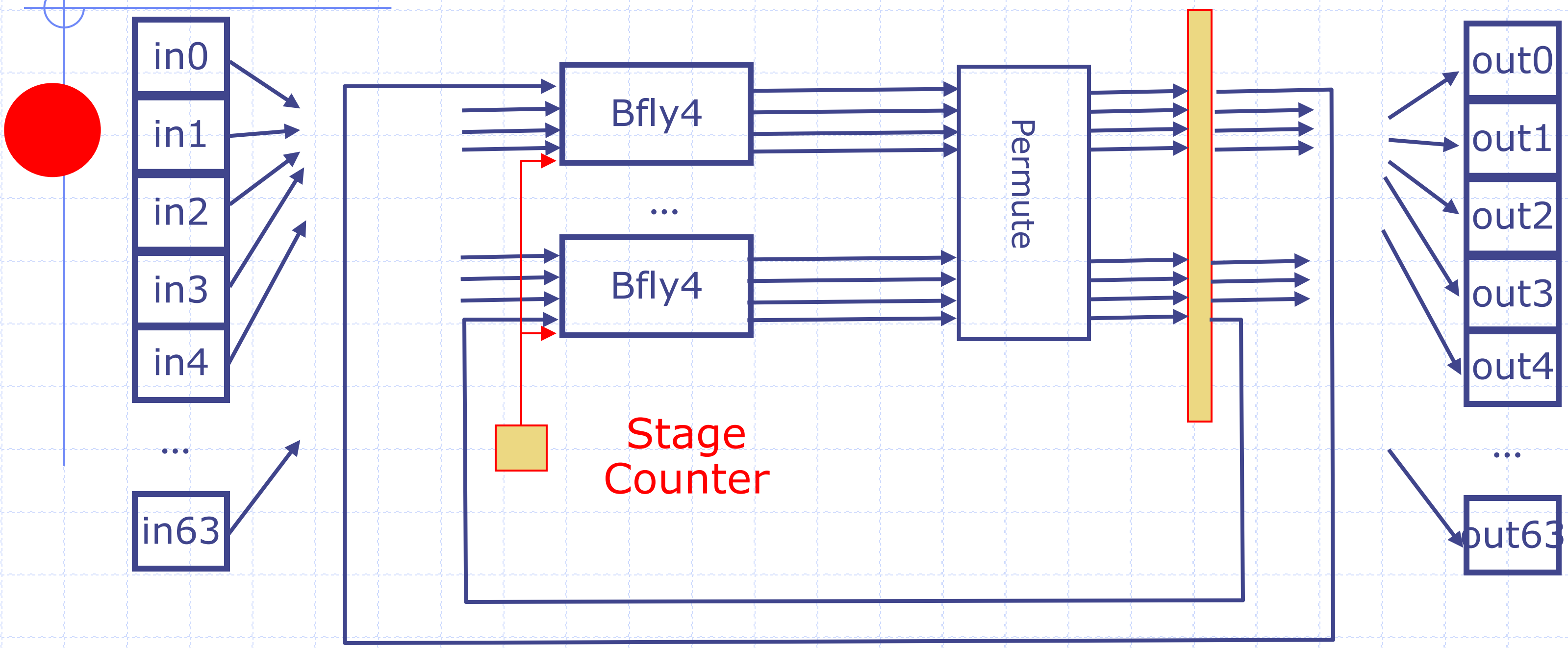
we expect:

Throughput to decrease – less parallelism

Area to decrease – reusing a block

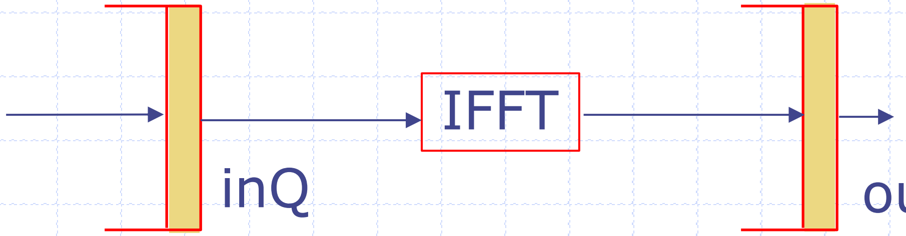
The clock needs to run faster for the same throughput

Folded IFFT: Reusing the stage combinational circuit



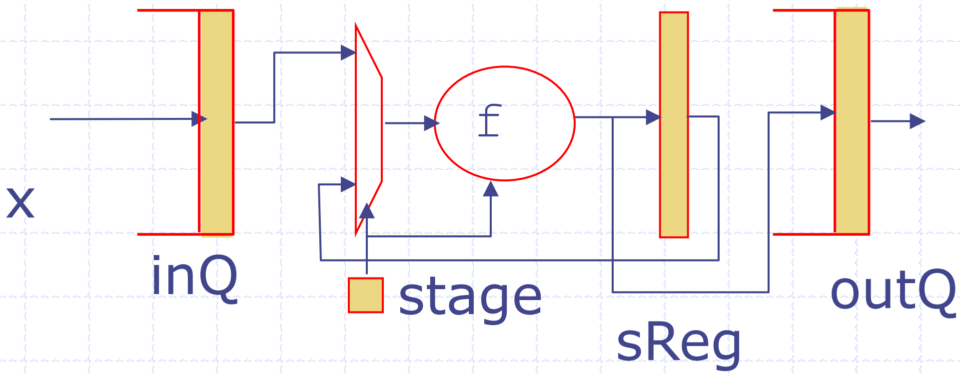
Input and Output FIFOs

- If IFFT is implemented as a sequential circuit it may take several cycles to process an input
- Sometimes it is convenient to think of input and output of a combinational function being connected to FIFOs



- FIFO operations:
  - enq – when the FIFO is not full
  - deq, first – when the FIFO is not empty
  - These operations can be performed only when the guard condition is satisfied

Folded implementation rules



Each rule has some additional implicit guard conditions associated with FIFO operations

Disjoint firing conditions

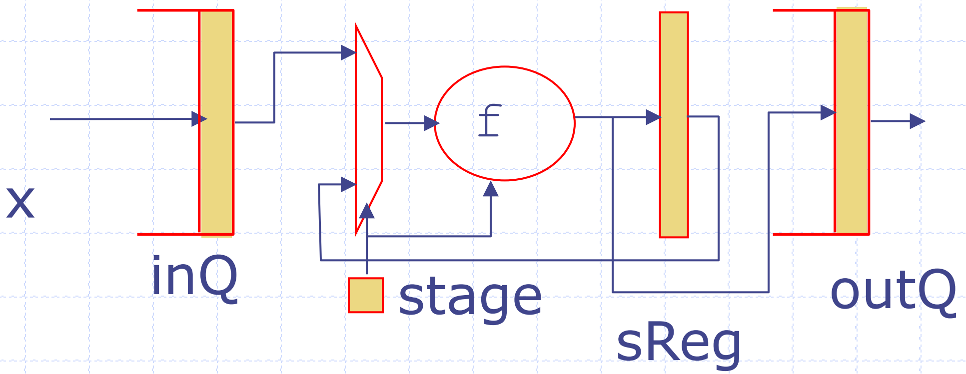
```
rule foldedEntry if (stage==0);
   sReg <= f(stage, inQ.first()); stage <= stage+1;
   inQ.deq();
endrule
rule foldedCirculate if (stage!=0)&(stage<(n-1));
   sReg <= f(stage, sReg);  stage <= stage+1;
endrule
rule foldedExit if (stage==n-1);
   outQ.enq(f(stage, sReg));  stage <= 0;
endrule
```

notice stage is a dynamic parameter now!

no for-loop

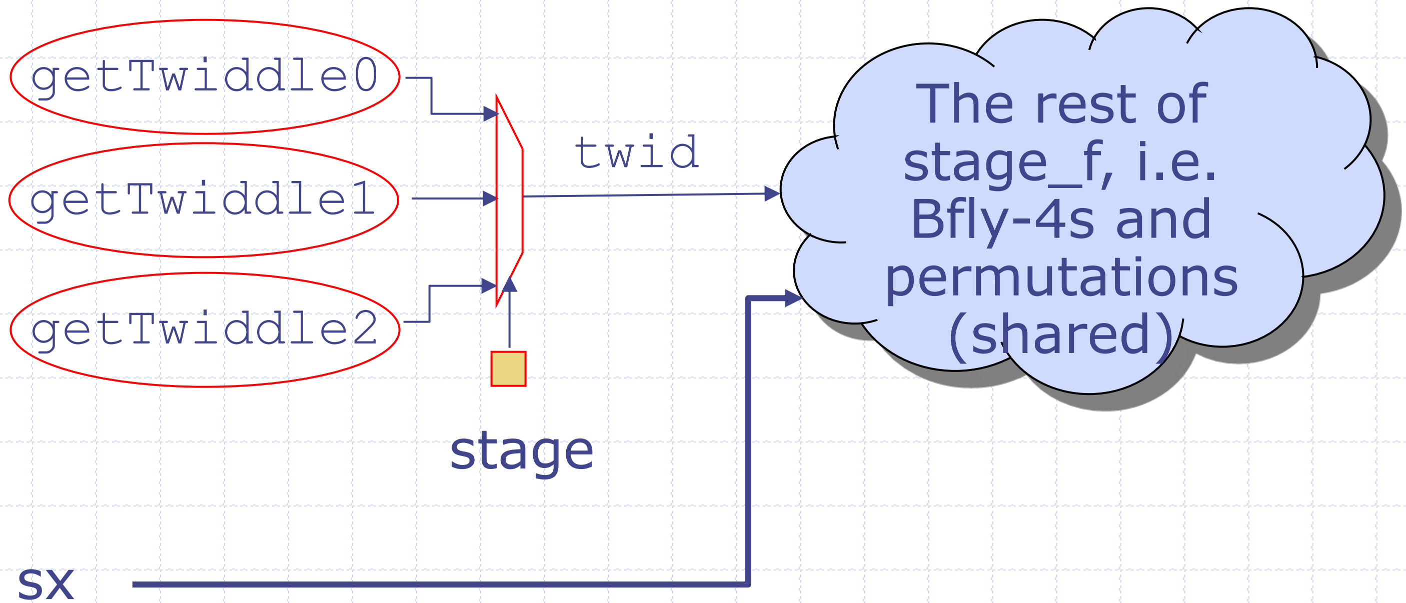
Folded implementation

expressed as a single rule



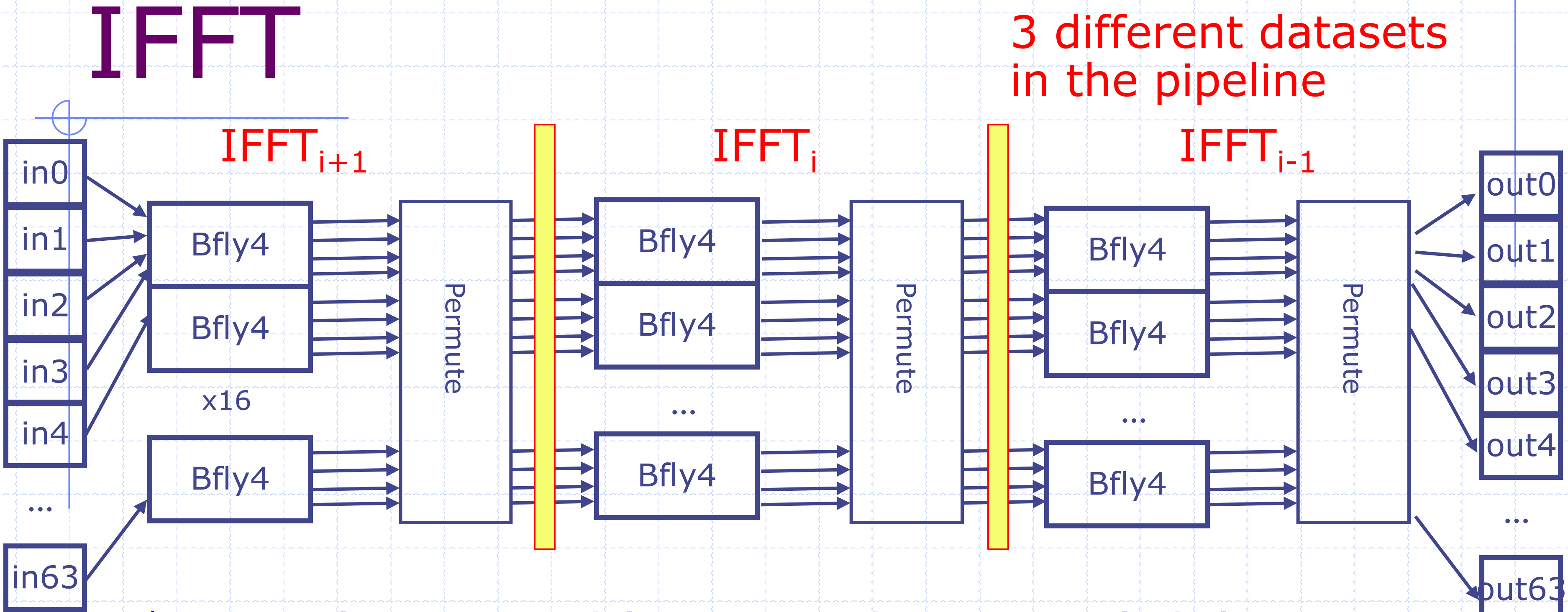

```
rule folded-pipeline (True);
 let sxIn = ?;
 if (stage==0)
   begin sxIn= inQ.first(); inQ.deq(); end
 else     sxIn= sReg;
 let sxOut = f(stage,sxIn);
 if (stage==n-1) outQ.enq(sxOut);
 else sReg <= sxOut;
 stage <= (stage==n-1)? 0 : stage+1;
endrule
```

Shared Circuit



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

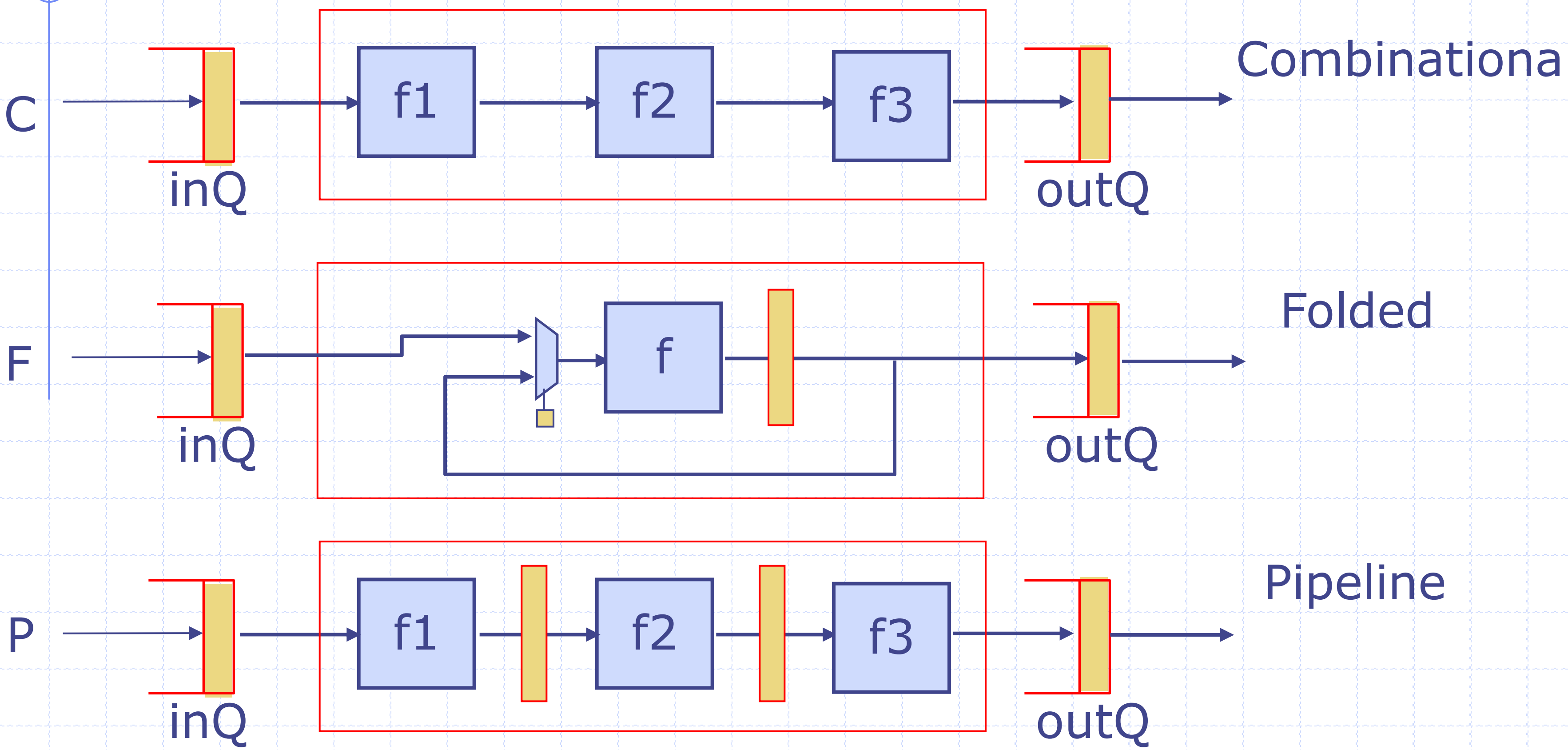
Pipelining Combinational IFFT



- Lot of area and long combinational delay
- Folded or multi-cycle version can save area and reduce the combinational delay but throughput per clock cycle gets worse
- Pipelining: a method to increase the circuit throughput by evaluating multiple IFFTs

Next lecture

Design comparison



Clock: C < P ≈ F

Area: F < C < P

Throughput: F < C < P

Area estimates

Tool: Synopsys Design Compiler

- Comb. FFT
  - Combinational area: 16536
  - Noncombinational area: 9279
- Folded FFT
  - Combinational area: 29330
  - Noncombinational area: 11603
- Pipelined FFT
  - Combinational area: 20610
  - Noncombinational area: 18558

Are the results surprising?

Why is folded implementation not smaller?

Explanation: Because of constant propagation optimization, each bfly4 gets reduced by 60% when twiddle factors are specified. Folded design disallows this optimization because of the sharing of bfly4's

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))`

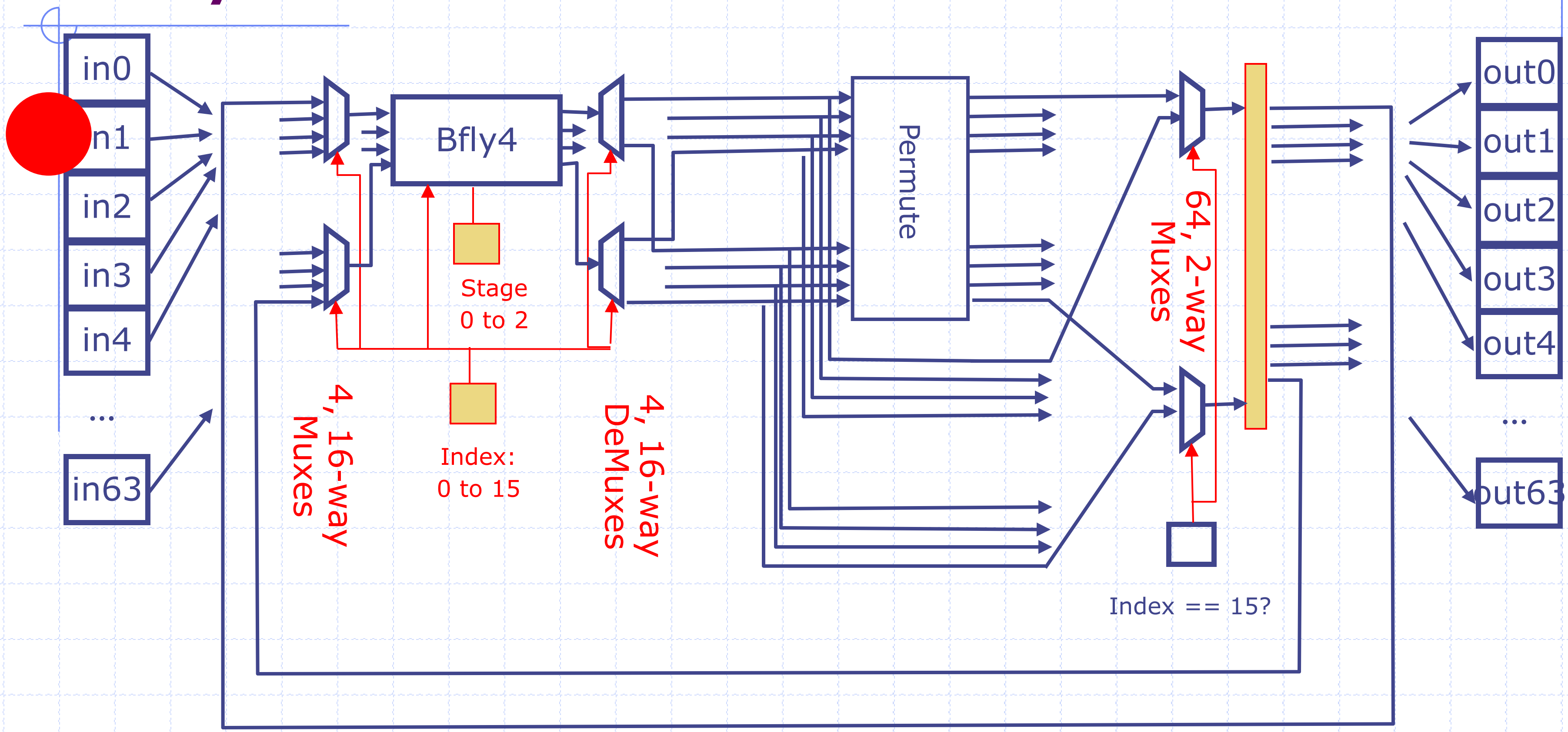
- Vector of Registers
  - `x[i] <= y[j]` does not work. The parser thinks it means `(sel(x,i)._read)._write(sel(y,j)._read)`, which will not type check
  - `(x[i]) <= y[j]` parses as `sel(x,i)._write(sel(y,j)._read)`, and works correctly

Don't ask me why

Optional: Superfolded FFT

Superfolded IFFT: Just one Bfly-4 node!

Optional



- 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 IFFT: stage function f

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

```
function Vector#(64, Complex) stage_f
        (Bit#(2) stage, Vector#(64, Complex) stage_in);
    Vector#(64, Complex#(n)) stage_temp, stage_out;
    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]];
 return(stage_out);
endfunction
```

should be done only when i=15

Code for the Superfolded 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