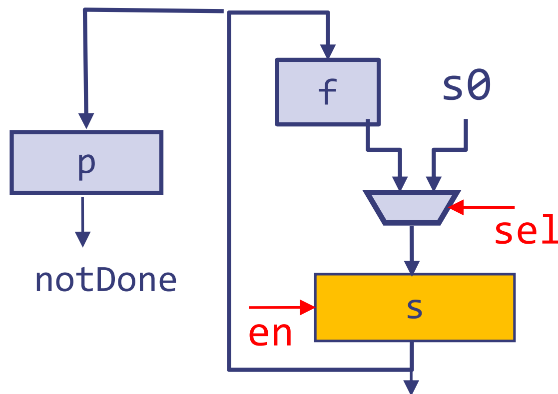
Folded and Pipelined Sequential circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Expressing a loop using registers

```
int s = s0;
while (p(s)) {
    s = f(s);
  }
return s;
```
C-code



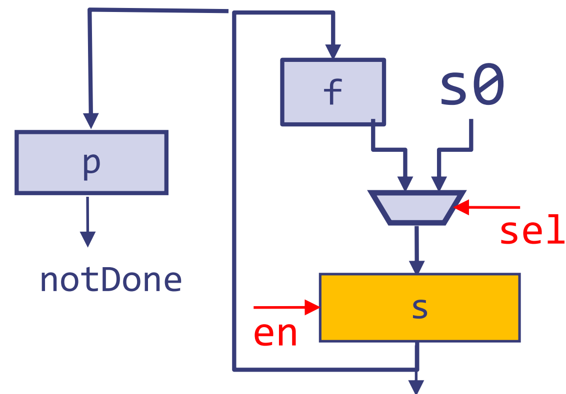
- Such a loop cannot be implemented by unfolding because the number of iterations is input-data dependent
- A register is needed to hold the value of s from one iteration to the next
- s has to be initialized when the computation starts, and updated every cycle until the computation terminates

```
sel = start
en  = start | notDone
```

Expressing a loop in BSV

- When a rule executes:
  - the register s is read at the beginning of a clock cycle
  - computations to evaluate the next value of the register and the s_{en} are performed
  - If s_{en} is True then s is updated at the end of the clock cycle
- A mux is needed to initialize the register

```
Reg#(n) s <- mkRegU();
rule step;
  if (p(s)) s <= f(s);
endrule
```

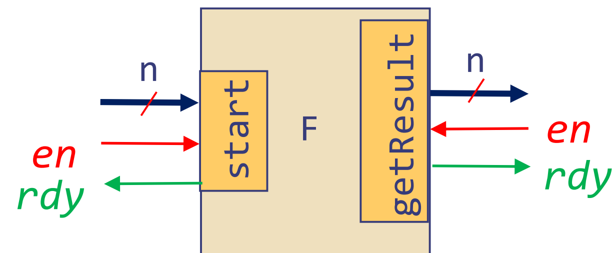



```
sel = start
en  = start | notDone
```

How should this circuit be packaged for proper use?

Packaging a computation as a Latency-Insensitive Module

```
interface GFMI#(n);
  method Action start (Bit#(n) x);
  method ActionValue#(Bit#(n)) getResult;
endinterface
```




```
module mkF (GFMI#(n));
  Reg#(n) s <- mkRegU();
  Reg#(Bool) busy <- mkReg(False);
  rule step if (p(s))&&busy;
     s <= f(s);
  endrule
  method Action start(Bit#(n) x) if (!busy);
   s <= x; busy <= True;
  endmethod
  method ActionValue Bit#(n) getResult if (!p(s) && busy);
   busy <= False; return s;
  endmethod
endmodule
```

The user of this module does not know how long it takes to execute the loop

Combinational 32-bit multiply

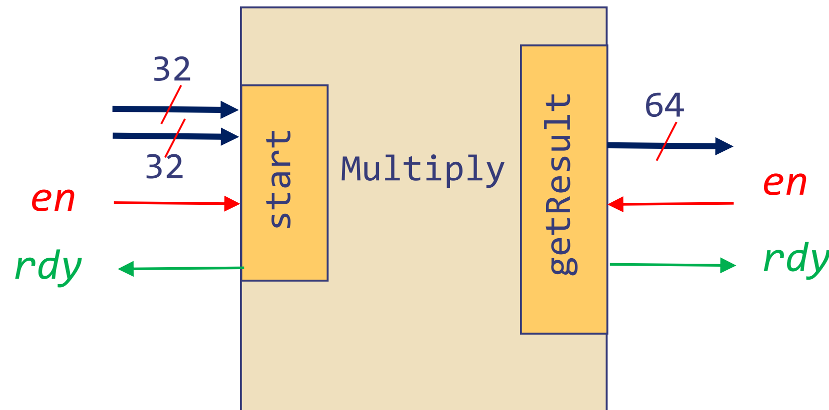
```
function Bit#(64) mul32(Bit#(32) a, Bit#(32) b);
  Bit#(32) tp = 0;
  Bit#(32) prod = 0;
  for(Integer i = 0; i < 32; i = i+1)
  begin
     Bit#(32) m   = (a[i]==0)? 0 : b;
     Bit#(33) sum = add32(m,tp,0);
     prod[i]      = sum[0];
     tp           = sum[32:1];
  end
  return {tp,prod};
endfunction
```

This circuit uses 32 add32 circuits

Lot of gates!

- We can reuse the same `add32` circuit if we store the partial results, e.g., `sum`, in a *register*
- Need registers to hold a, b, tp, prod and i
- Update the registers every cycle until we are done

Packaging Multiply as a Latency-Insensitive Module



Interface with guards

```
interface Multiply;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(64)) getResult;
endinterface
```

The user of this module does not know how long it takes to execute a particular multiply

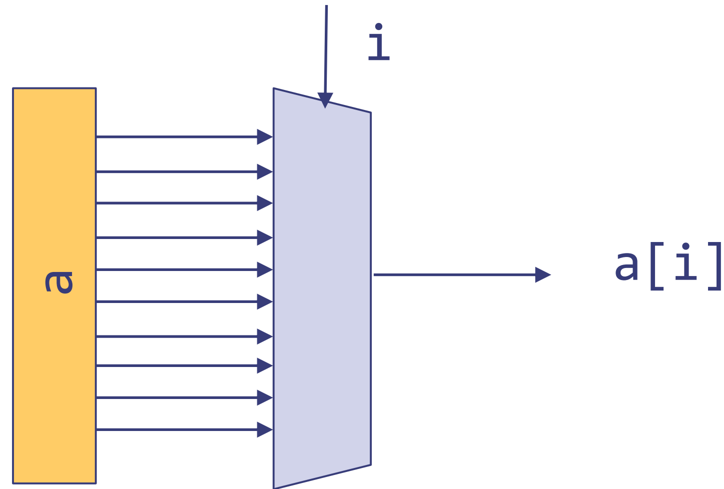
Multiply Module

```
Module mkMultiply (Multiply);
    Reg#(Bit#(32)) a<-mkRegU(); Reg#(Bit#(32)) b<-mkRegU();
    Reg#(Bit#(32)) prod <-mkRegU(); Reg#(Bit#(32)) tp <- mkReg(0);
    Reg#(Bit#(6))  i <- mkReg(32);
    Reg#(Bool) busy <- mkReg(False);

rule mulStep if (i < 32);
      Bit#(32) m = (a[i]==0)? 0 : b;
      Bit#(33) sum = add32(m,tp,0);
      prod[i] <= sum[0];
      tp <= sum[32:1];
      i <= i+1;
endrule;

method Action start(Bit#(32) x, Bit#(32) y) if (!busy);
  a <= x; b <= y; busy <= True; i <= 0;  endmethod
method ActionValue Bit#(64) getResult if ((i==32) && busy);
  busy <= False; return {tp,prod}; endmethod
```

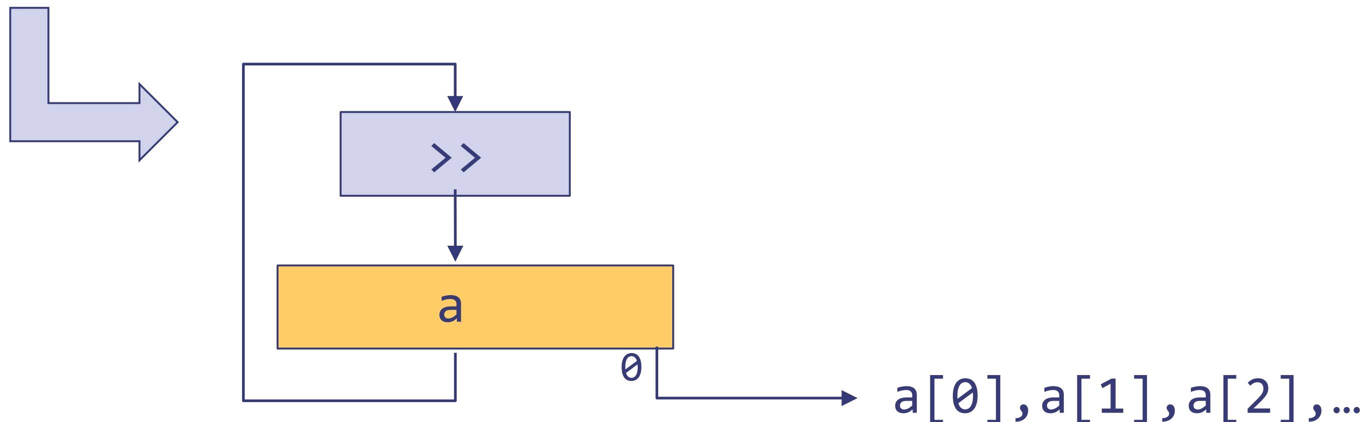
like the loop body in the combinational version

We use a 6-bit i register and initialize it to 32 to make sure that the rule has no effect until i is set to some value < 32

Dynamic selection requires a mux



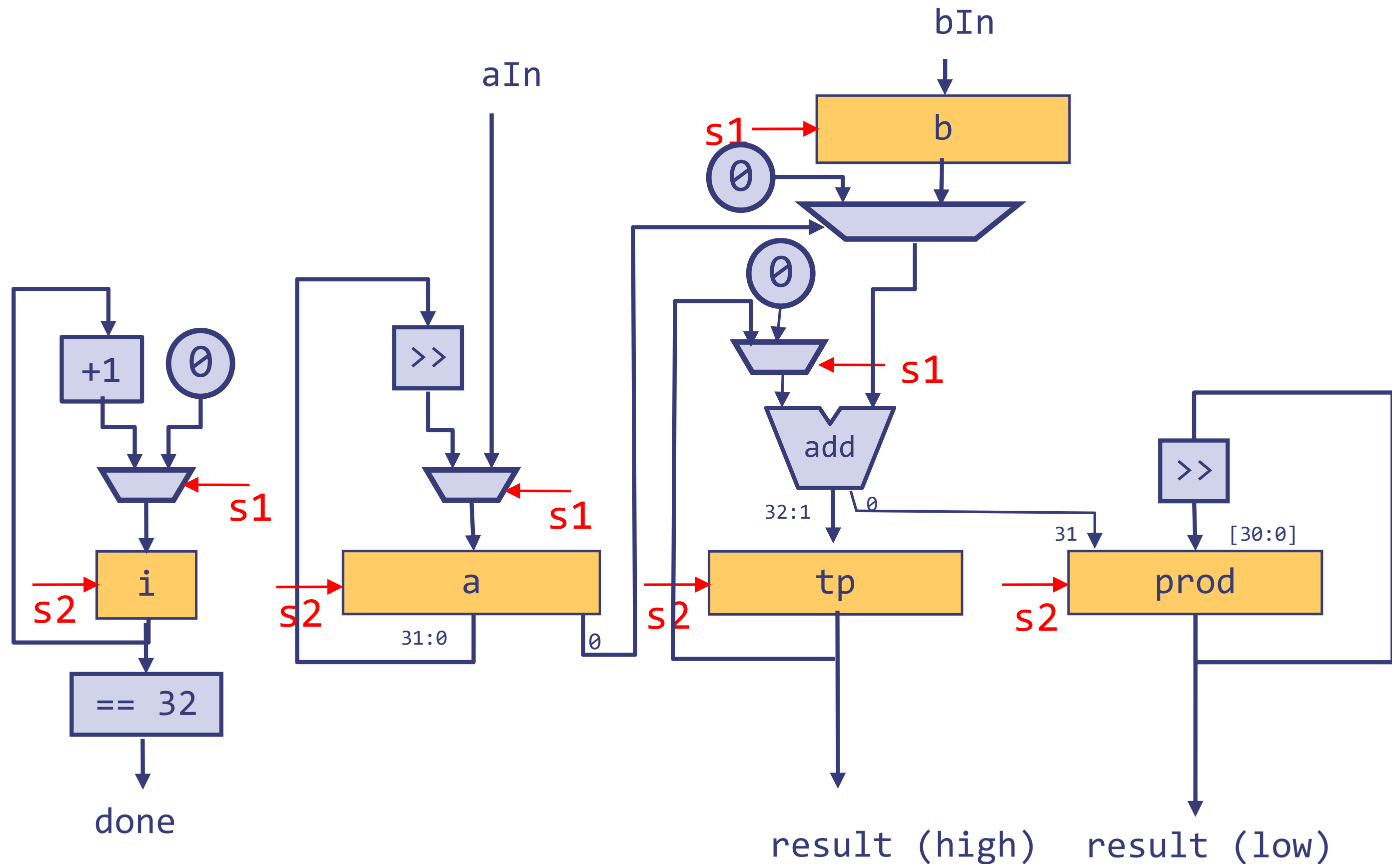
When the selection indices follow a regular pattern then it is cheaper to use a shift operator (no gates!) instead of a mux



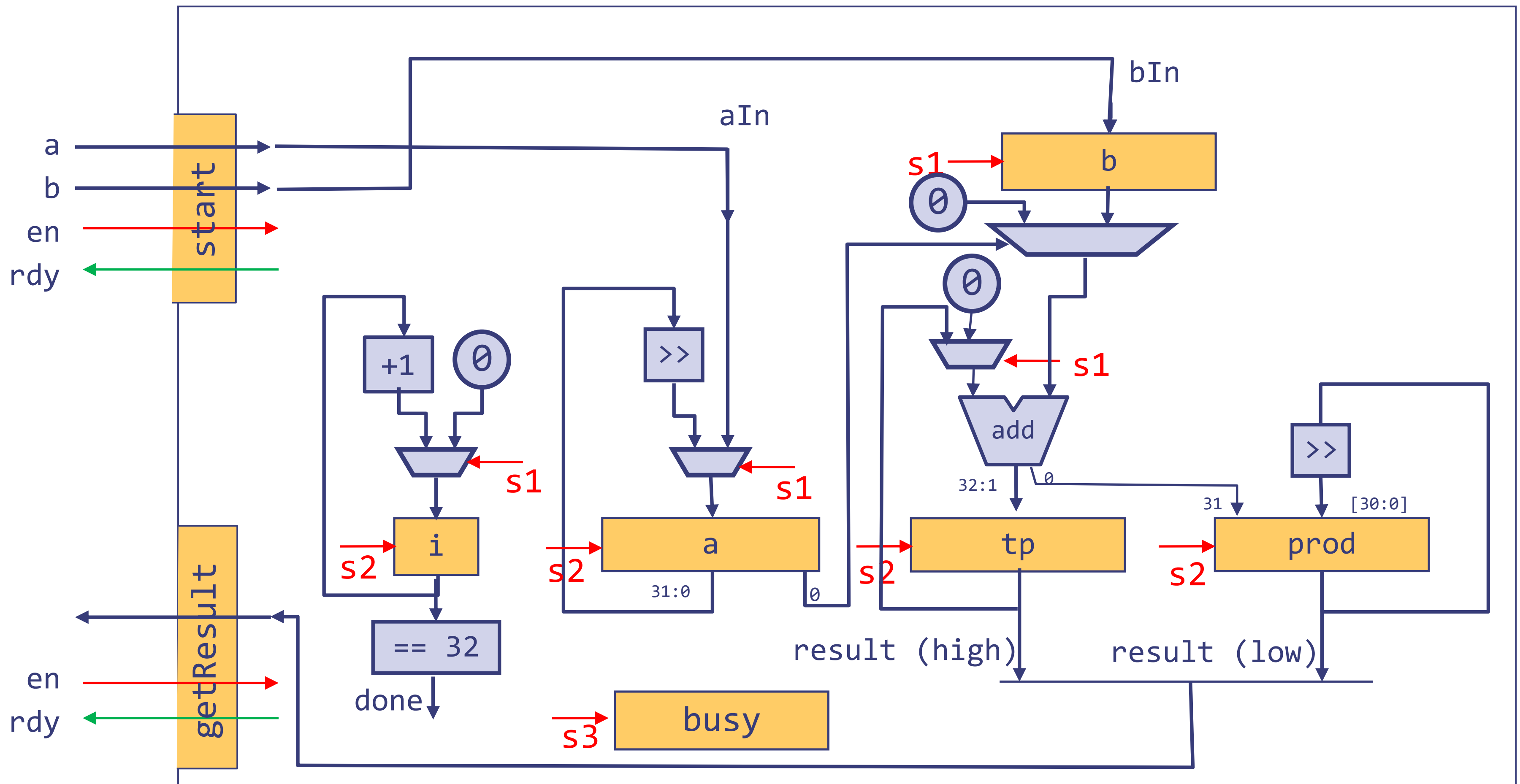
Replacing repeated selections by shifts

```
rule mulStep if (i < 32);
     Bit#(32) m = (a[0]==0)? 0 : b;
     a <= a >> 1;
     Bit#(33) sum = add32(m,tp,0);
     prod <= {sum[0], prod[31:1]};
     tp <= sum[32:1];
     i <= i+1;
endrule
```

Circuit for Sequential Multiply



Multiply Module



start_rdy = !busy
getResult_rdy = busy & done

s1 = start_en
s2 = start_en | !done

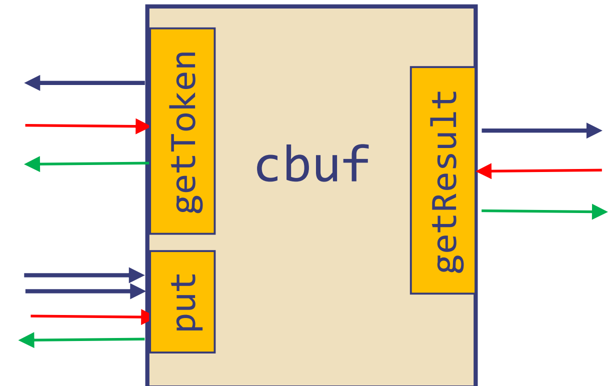
s3 = start_en | getResult_en

Sequential Multiply analysis

- Number of add32 circuits has been reduced from 31 to one, though some registers and muxes have been added
- The longest combinational path has been reduced from 62 FAs to one add32 plus a few muxes
- The sequential circuit will take 31 clock cycles to compute an answer

Completion buffer

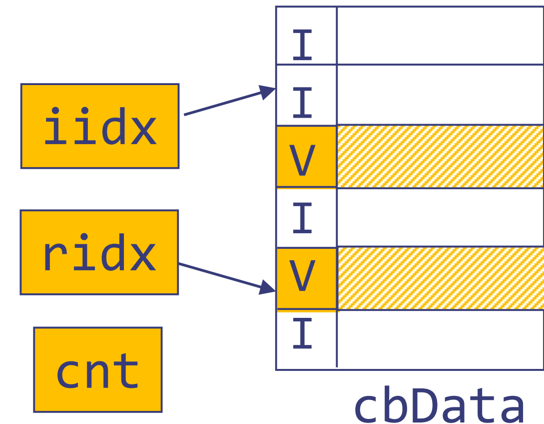
```
interface CBuffer#(type t);
  method ActionValue#(Token) getToken;
  method Action put(Token tok, t d);
  method ActionValue#(t) getResult;
endinterface
```



- Suppose a module is required to process inputs in the FIFO order
- Internally the module processes several inputs simultaneously and different inputs take different amount of time
- Completion buffer is used to restore the order in which the processing of inputs was done

Completion buffer: Implementation

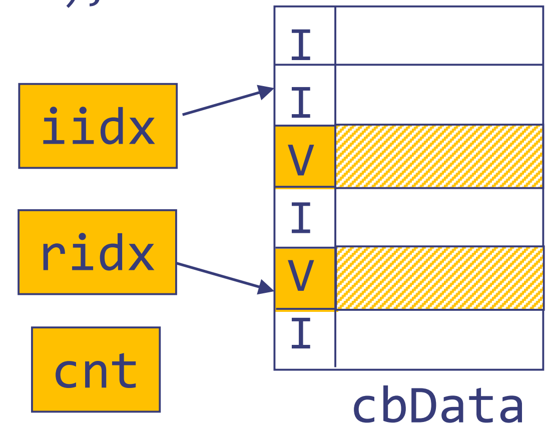
- A circular buffer with two pointers iidx and ridx, and a counter cnt
- Each data element has a valid bit associated with it



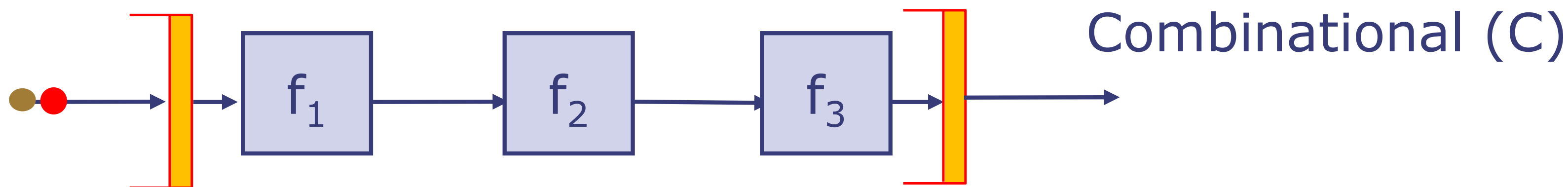

```
module mkCompletionBuffer(CompletionBuffer#(t));
  Vector#(32, Reg#(Bool)) cbv <- replicateM(mkReg(False));
  Vector#(32, Reg#(t)) cbData <- replicateM(mkRegU());
  Reg#(Bit#(5))   iidx <- mkReg(0);
  Reg#(Bit#(5))   ridx <- mkReg(0);
  Reg#(Bit#(6))    cnt <- mkReg(0);

       rules and methods...
endmodule
```

Completion Buffer *cont*

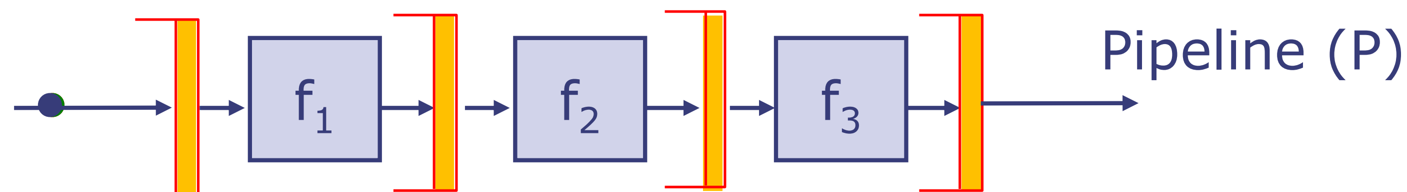
```
method ActionValue#(Bit#(5)) getToken() if (cnt < 32);
  cbv[iidx] <= False;
  iidx <= (iidx==31) ? 0 : iidx + 1;
  cnt <= cnt + 1;
  return iidx;
endmethod

method Action put(Token idx, t data);
  cbData[idx] <= data;
  cbv[idx] <= True;
endmethod

method ActionValue#(t) getResult() if ((cnt > 0)&&(cbv[ridx]));
  cbv[ridx] <= False;
  ridx <= (ridx==31) ? 0 : ridx + 1;
  cnt <= cnt - 1;
  return cbData[ridx];
endmethod
```



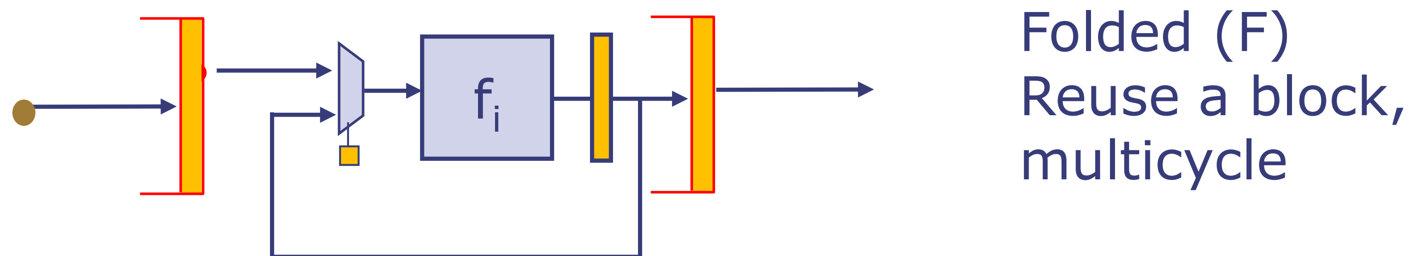
Design Alternatives



Combinational (C)



Pipeline (P)



Folded (F)
Reuse a block, multicycle

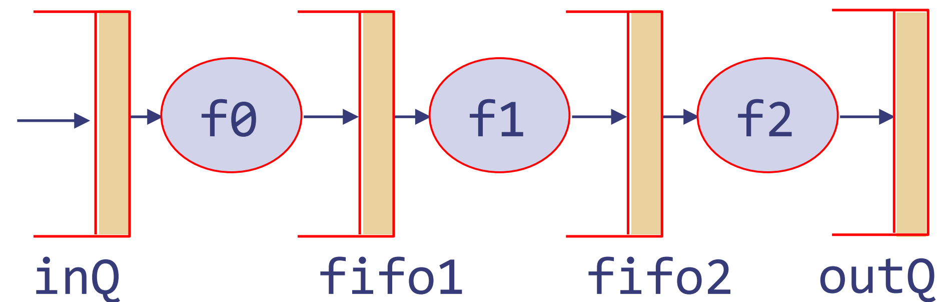
Clock: C < P ≈ F

Area: F < C < P

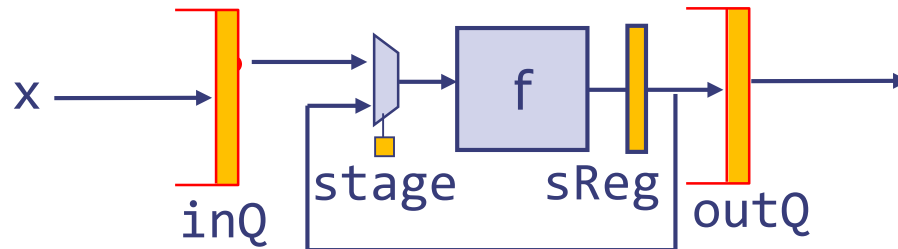
Throughput: F < C < P

Rules for Pipeline

```
rule stage1;
  fifo1.enq(f0(inQ.first));
  inQ.deq;
endrule

rule stage2;
  fifo2.enq(f1(fifo1.first));
  fifo1.deq;
endrule

rule stage3;
  outQ.enq(f2(fifo2.first));
  fifo2.deq;
endrule
```



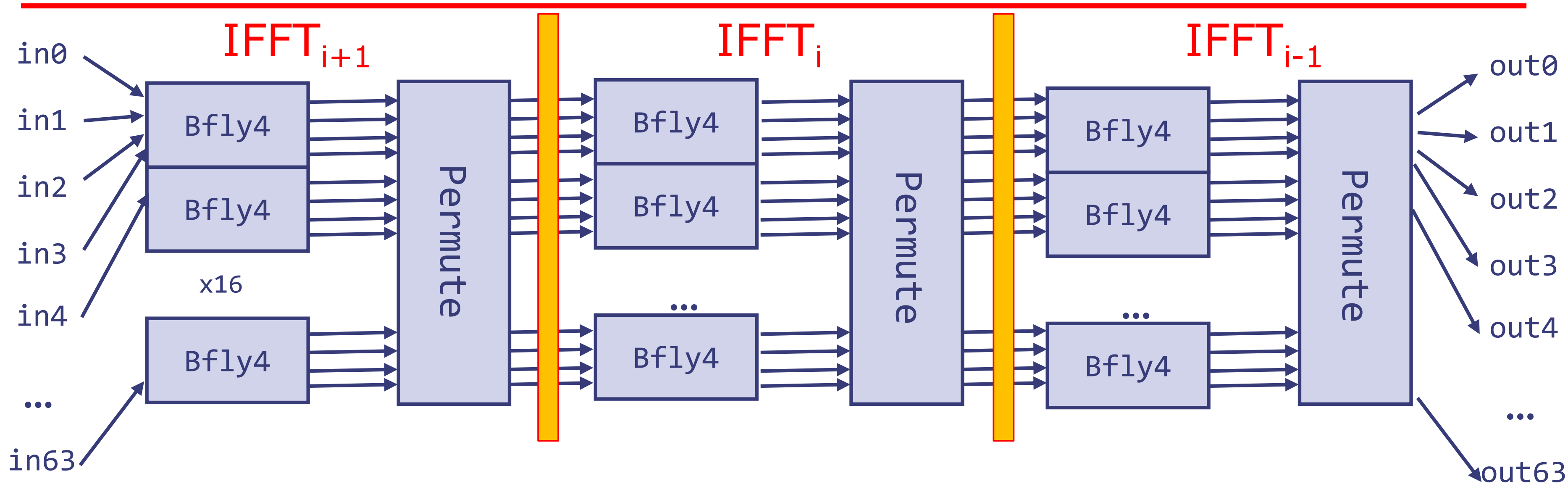
Rule for the Folded Circuit




```
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
```

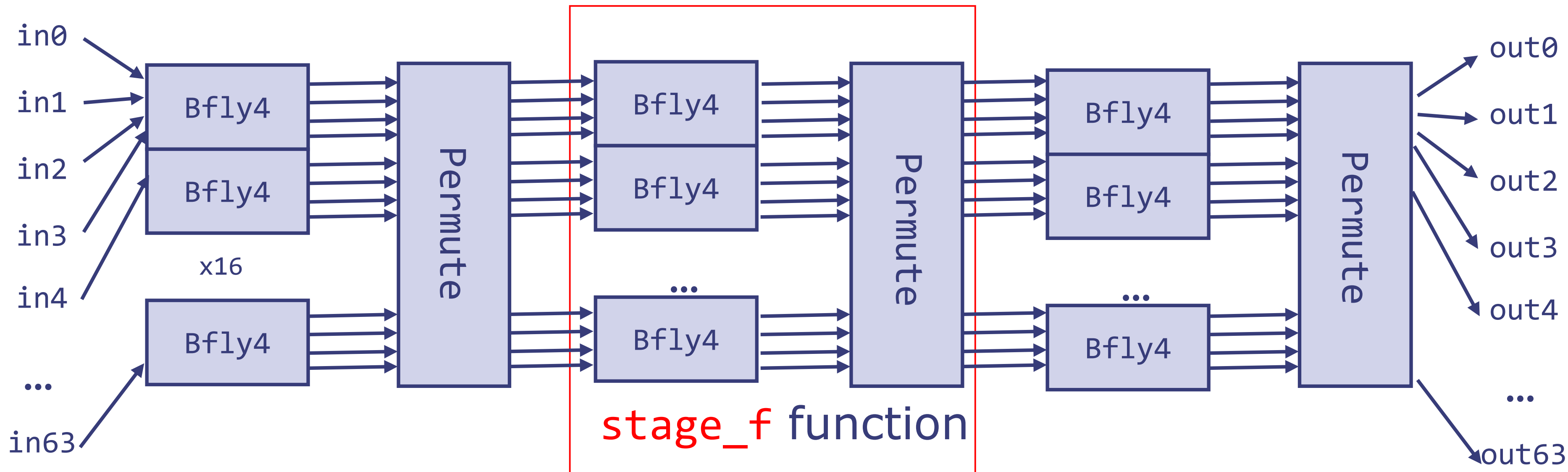
notice stage is a dynamic parameter now!

Combinational IFFT



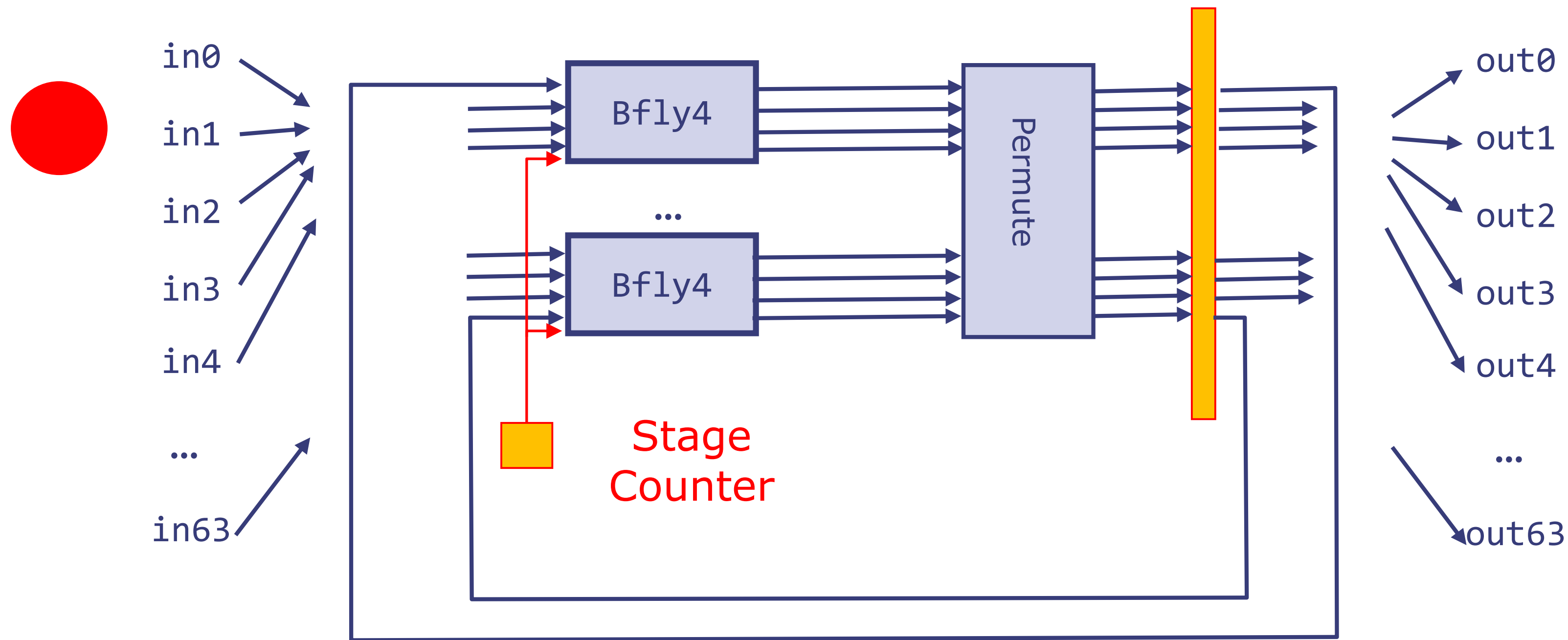
- Lot of area and long combinational delay
- *Pipelining:* Reduces critical-path delay. Increases throughput by evaluating multiple items in parallel

Combinational IFFT



- Lot of area and long combinational delay
- *Pipelining:* Reduces critical-path delay. Increases throughput by evaluating multiple items in parallel
- *Folded or multi-cycle Circuit:* Save area and reduces the combinational delay but makes throughput per clock cycle worse

Folded IFFT: Reusing the Stage computation



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
```

saw this in L03

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

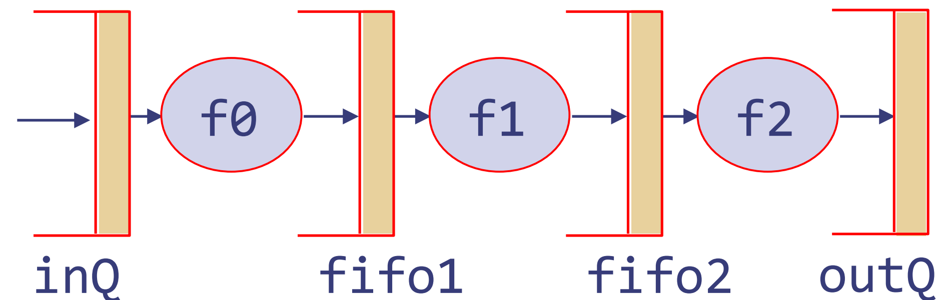
```
function f0(x)=  stage_f(0,x);

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);
```

Code for Pipelined IFFT

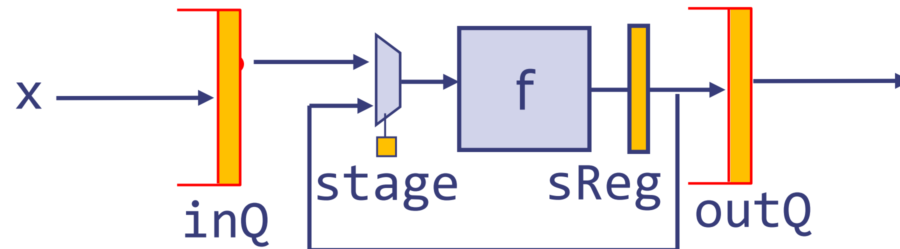
```
rule stage1;
  fifo1.enq(f0(inQ.first));
  inQ.deq;
endrule

rule stage2;
  fifo2.enq(f1(fifo1.first));
  fifo1.deq;
endrule

rule stage3;
  outQ.enq(f2(fifo2.first));
  fifo2.deq;
endrule
```



This is the code for IFFT if you use stage_f function for functions for f0, f1 and f2!

from slide 17

Code for Folded IFFT

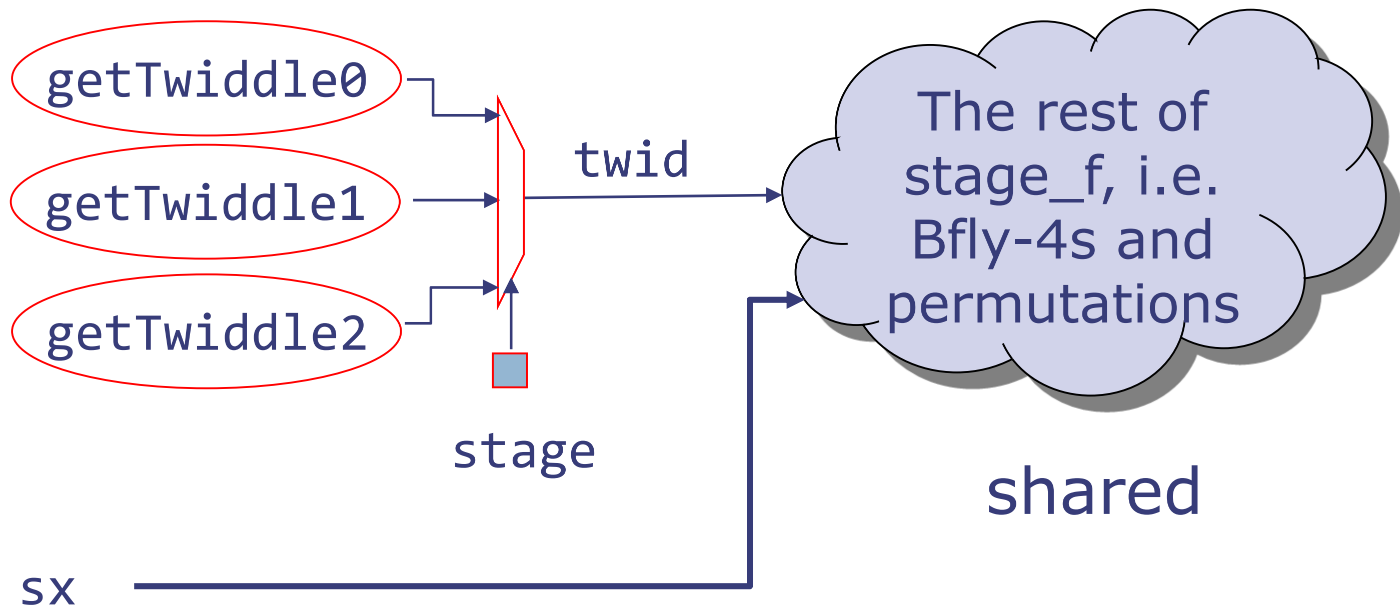



```
rule folded-pipeline (True);
 let sxIn = ?;
 if (stage==0)
   begin sxIn= inQ.first(); inQ.deq(); end
 else     sxIn= sReg;
 let sxOut = f(stage,sxIn);
 if (stage==n-1) outQ.enq(sxOut);
 else sReg <= sxOut;
 stage <= (stage==n-1)? 0 : stage+1;
endrule
```

from slide 18

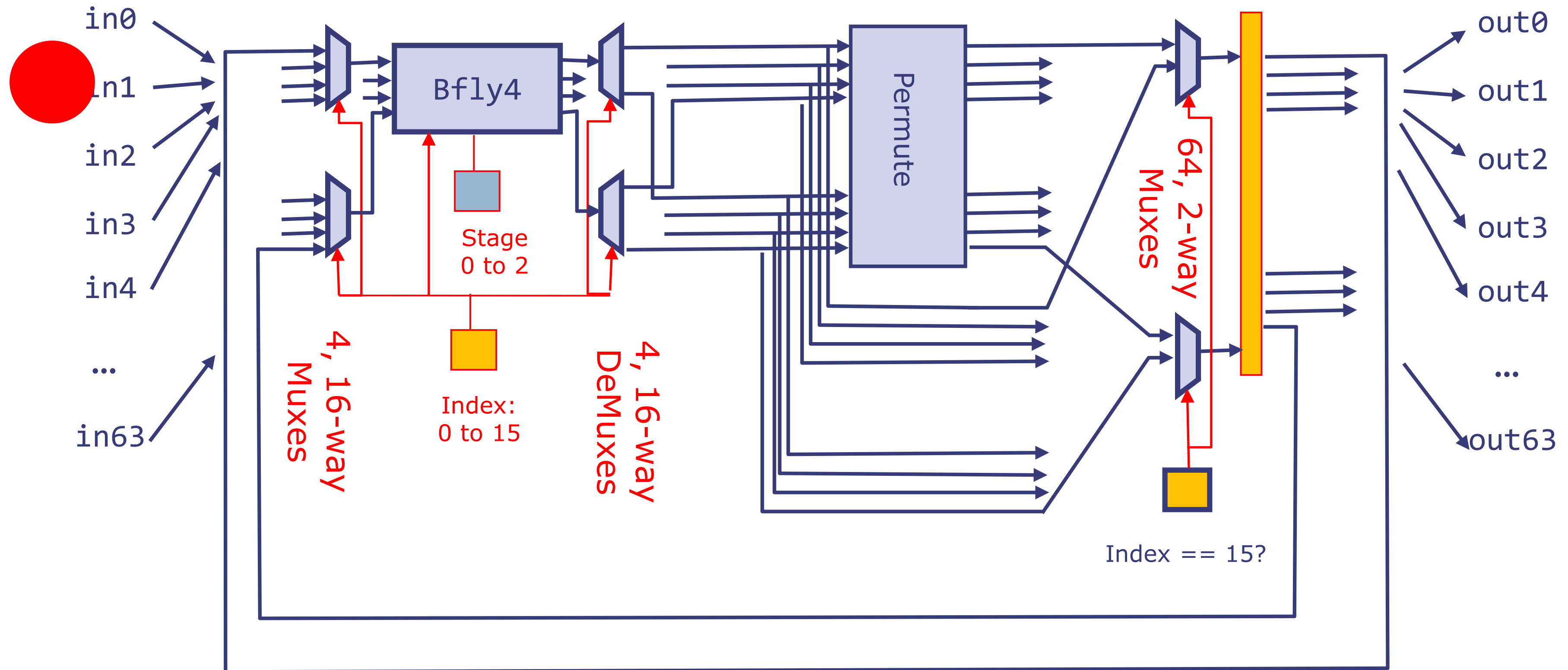
This is the code for IFFT if you use stage_f function for functions for f!

Shared Circuit



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

Superfolded IFFT: Just one Bfly-4 node!



- 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

Superfolding 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
```

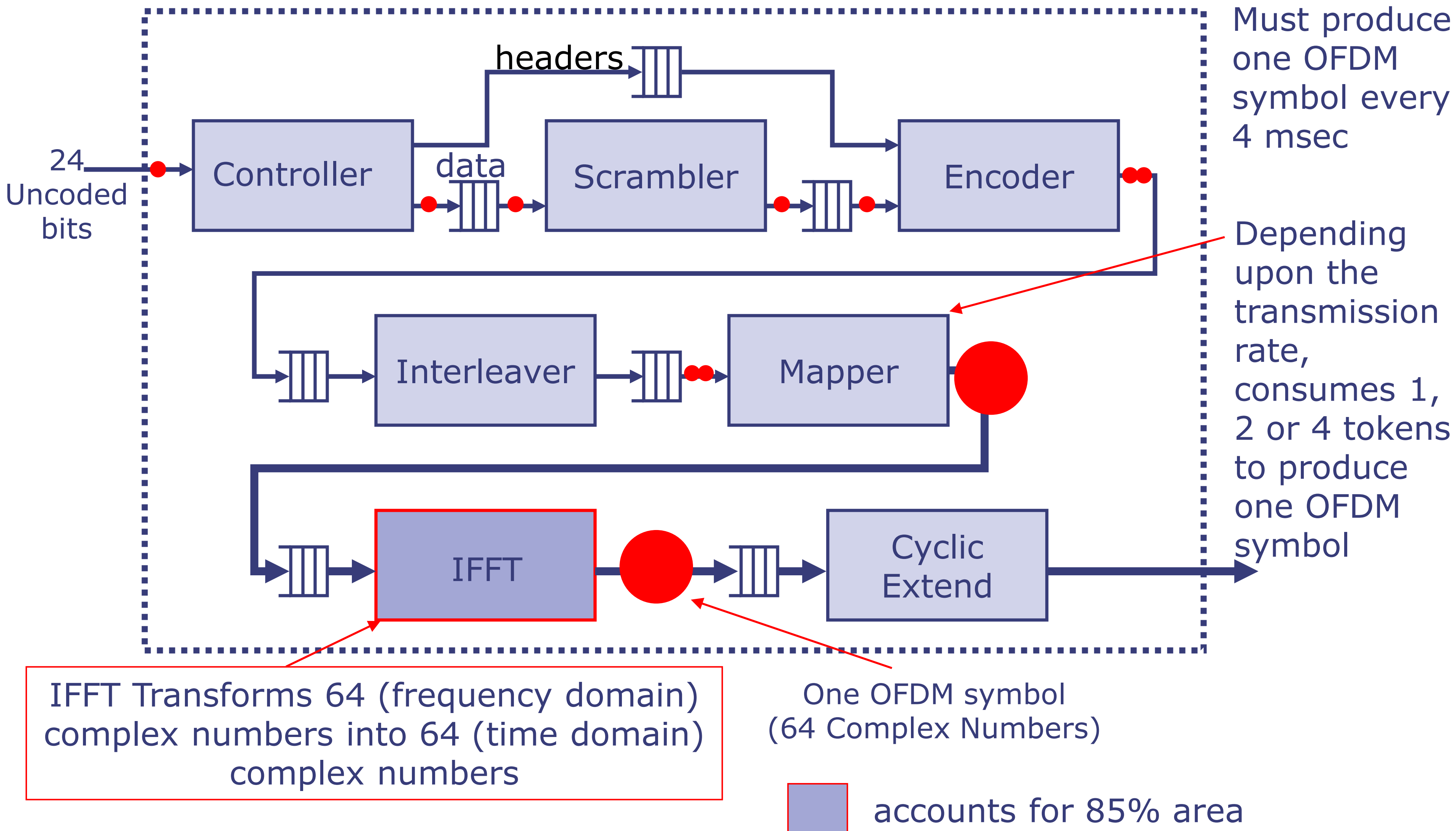
should be done only when i=15

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 idx = i*4;
   let y = bfly4(twid, stage_in[idx:idx+3]);
   let 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];

   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

802.11a Transmitter Overview



802.11a Transmitter

[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

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

The same source code

All these designs were done in less than 24 hours!

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!

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