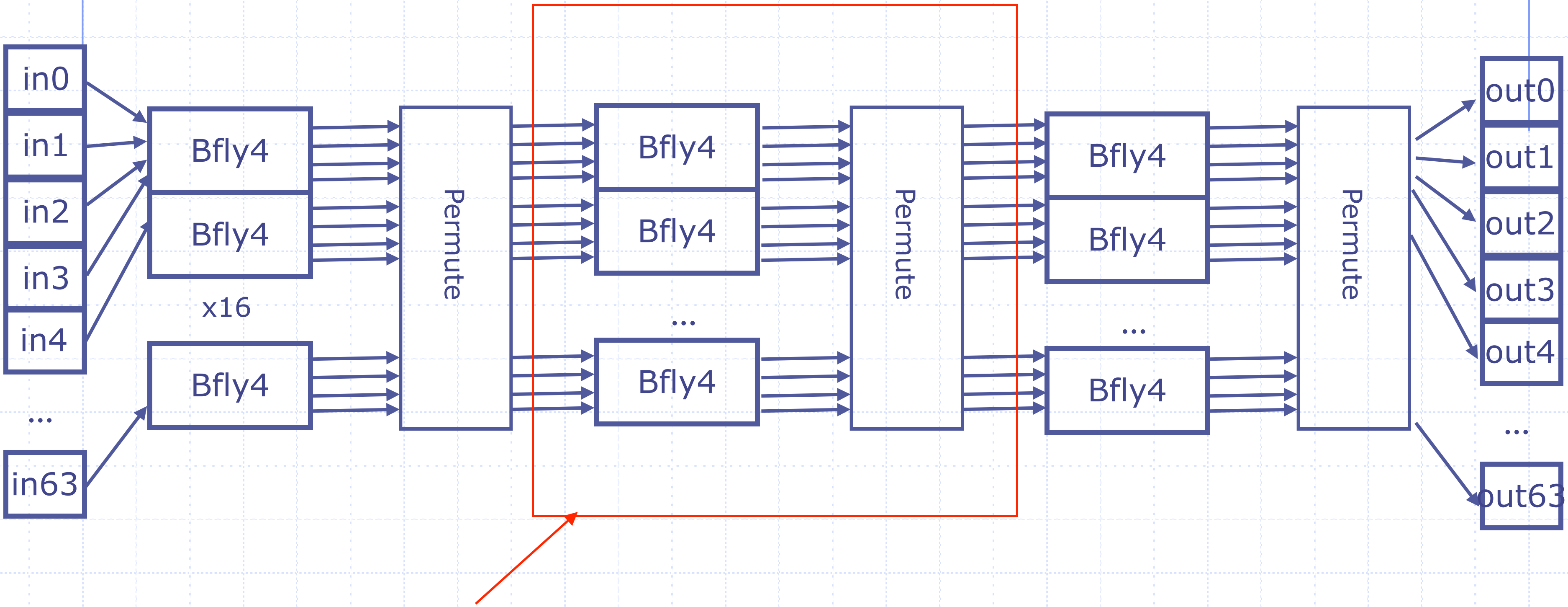
Folding and Pipelining complex combinational circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Combinational IFFT:

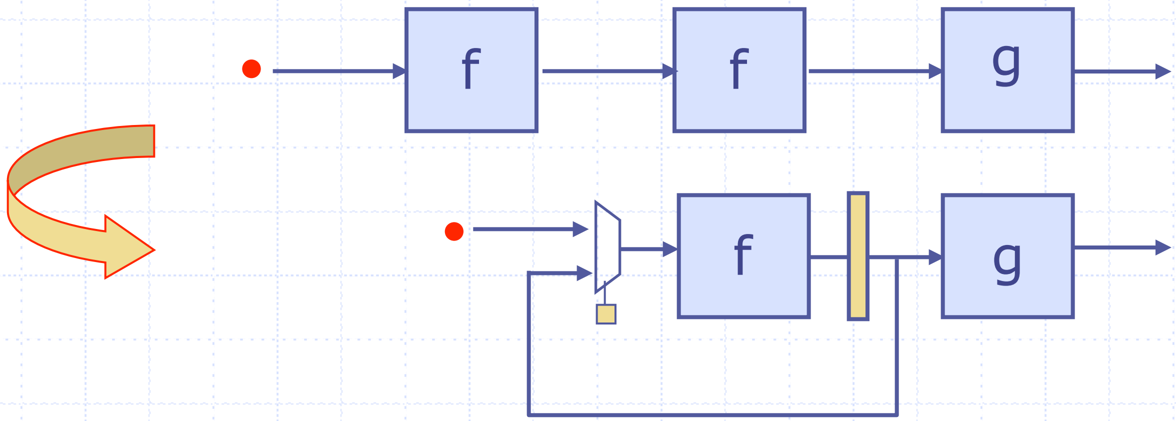
Suppose we want to reduce the area of the circuit



Reuse the same circuit three times to reduce area

Folding

Reusing a combinational block



Introduce state elements to hold intermediate values

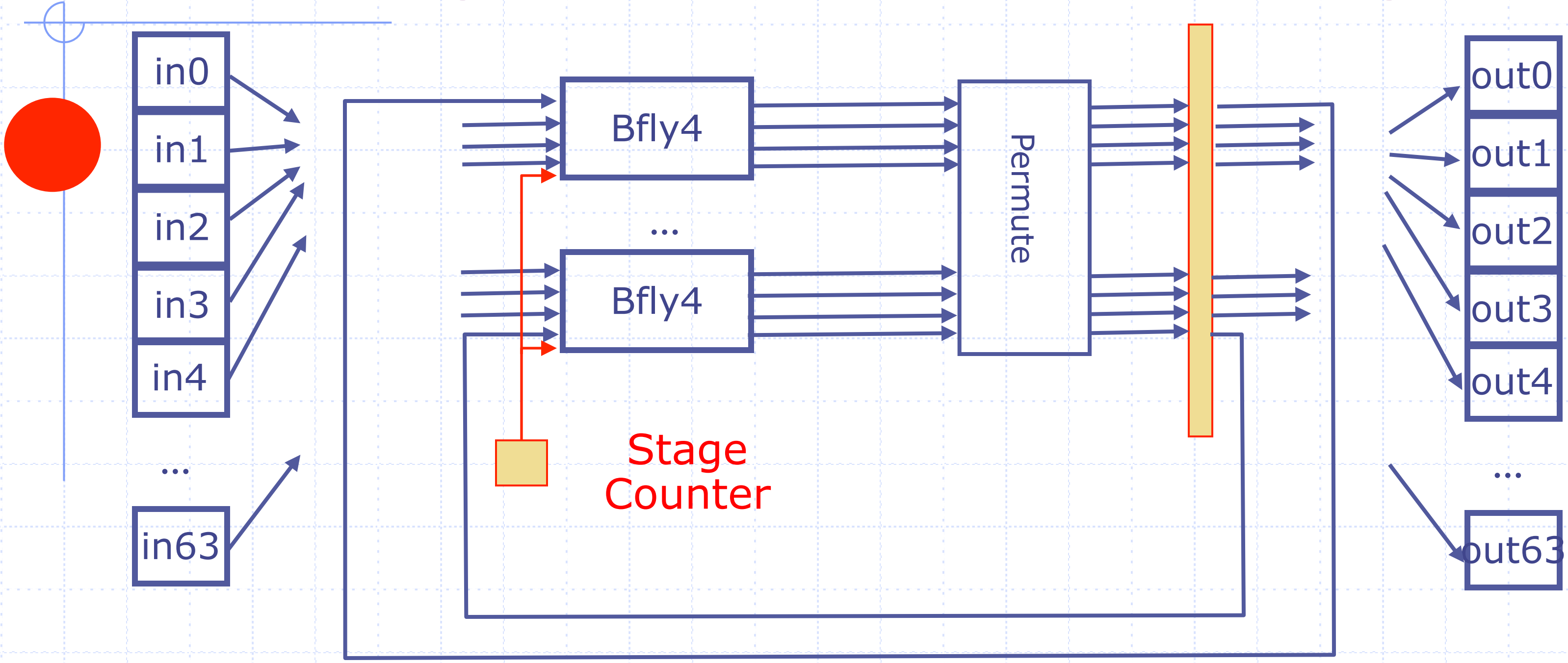
we expect:

Throughput to decrease – less parallelism

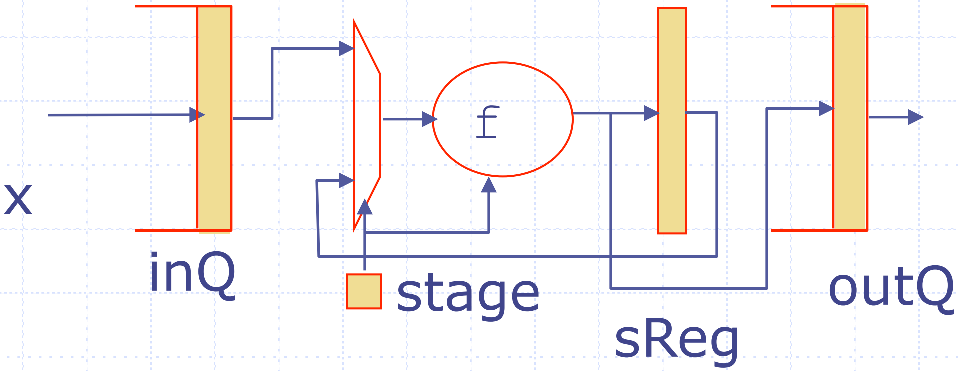
Area to decrease – reusing a block

The clock needs to run faster for the same throughput ⇒ hyper-linear increase in energy

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

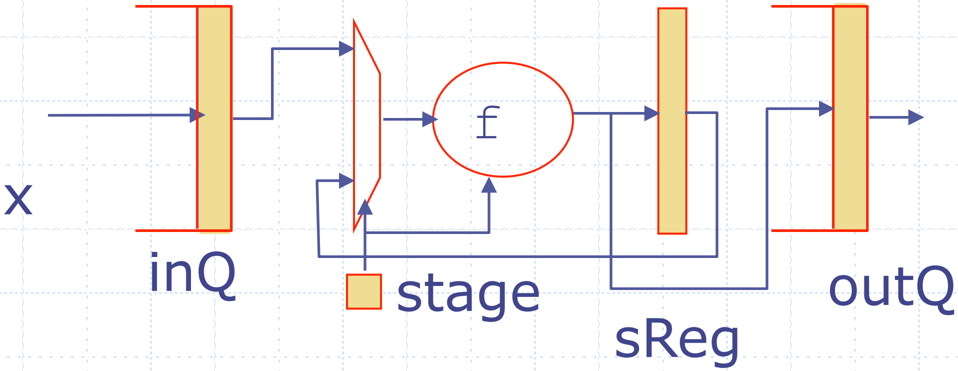
Need type declarations for sxIn and sxOut

BSV 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);
```

Folded pipeline-multiple rules

another way of expressing the same computation

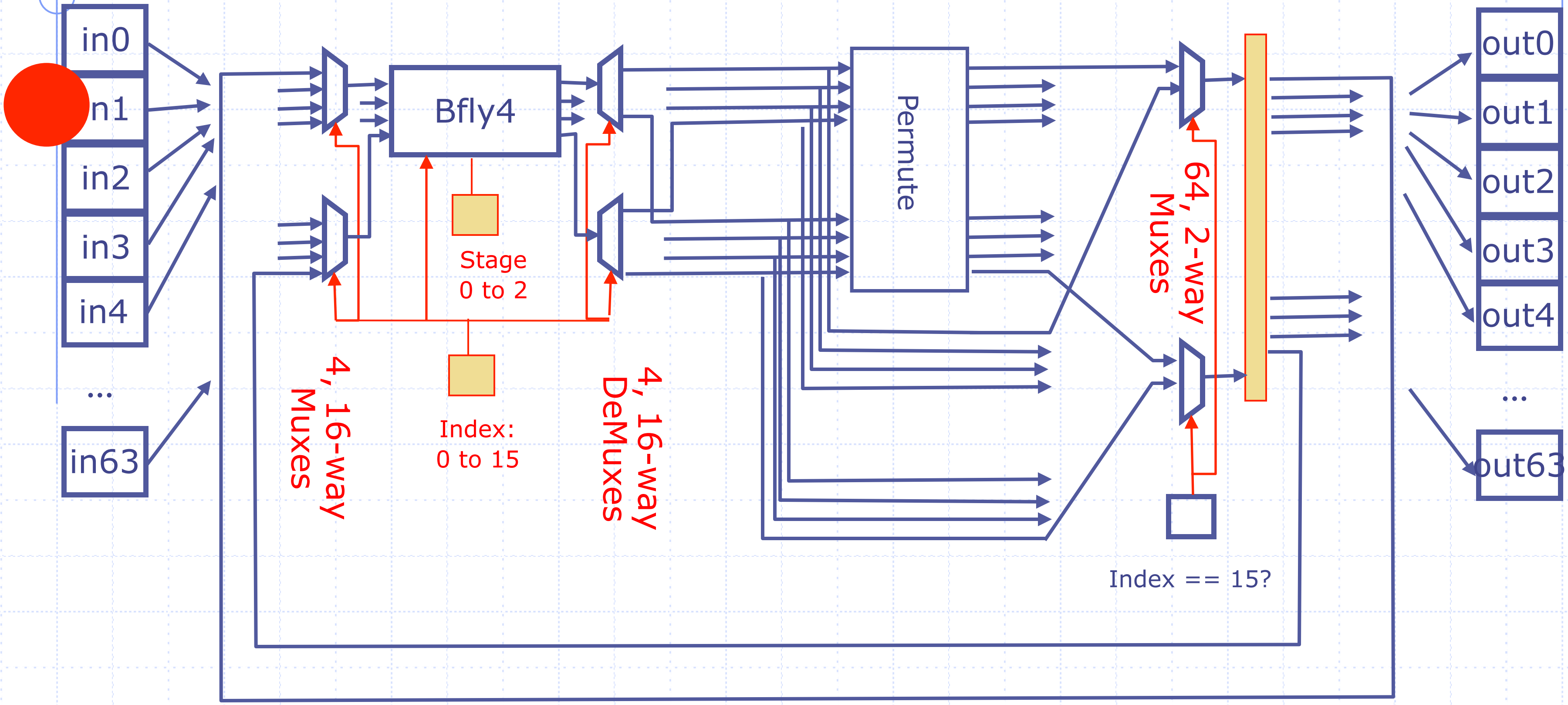


Disjoint firing conditions

```
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
```

Superfolded circular pipeline:

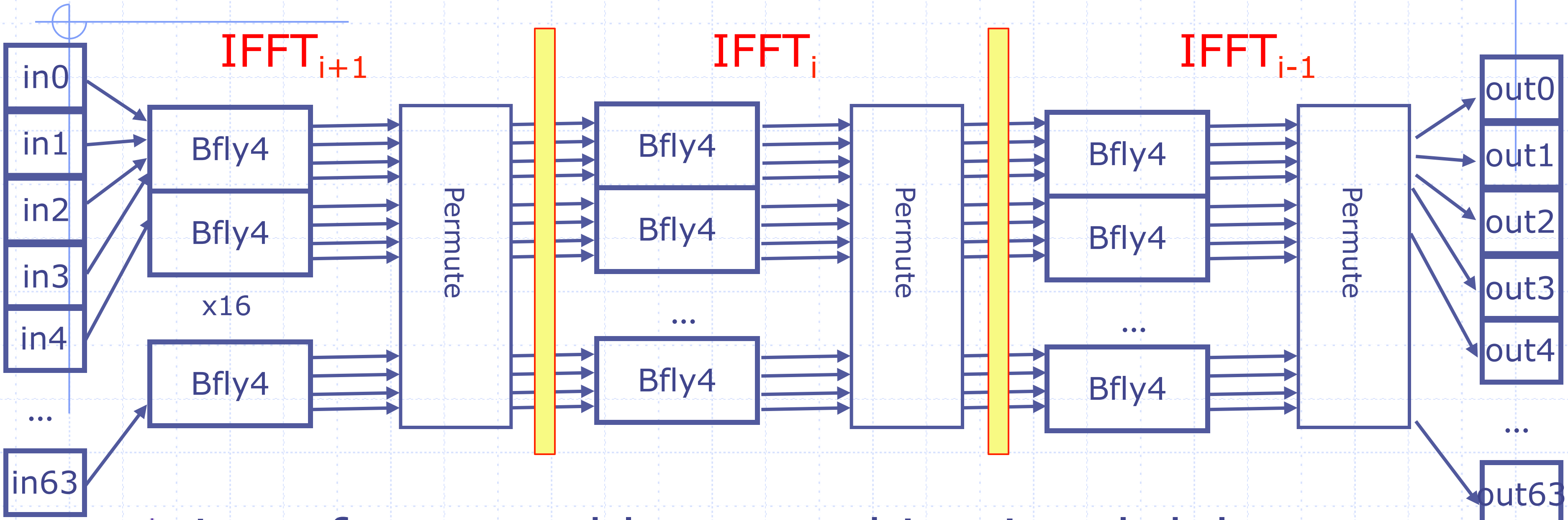
use just one Bfly-4 node!



Lab 3

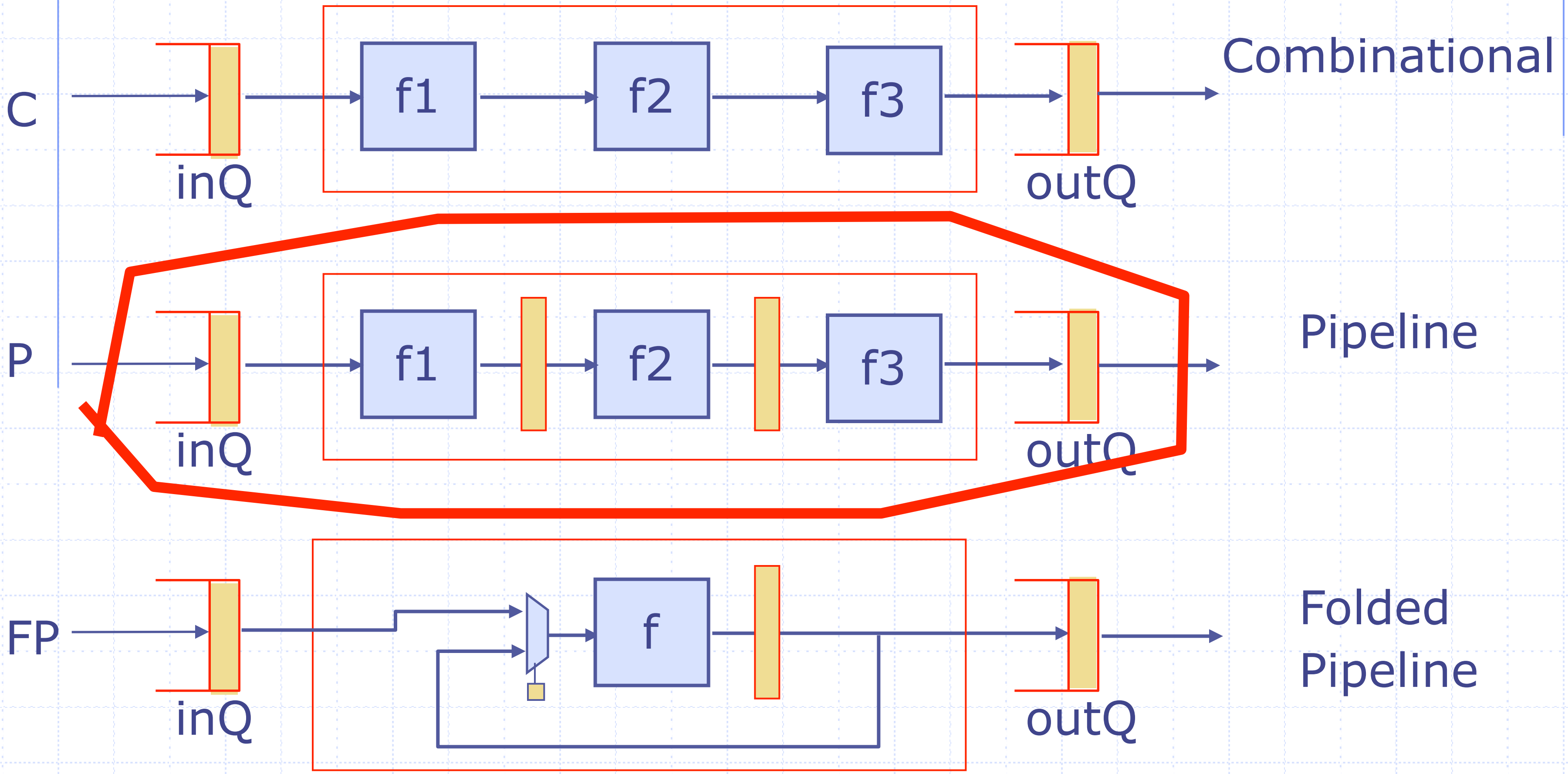
Combinational IFFT

3 different datasets in the pipeline



- 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 to evaluate multiple IFFTs

Pipelining a block

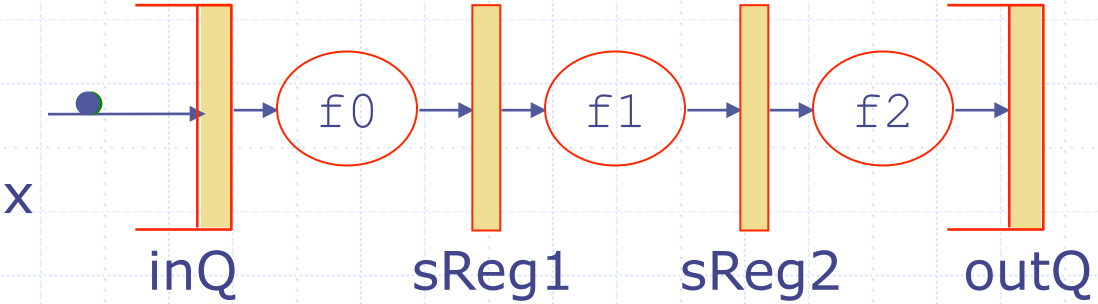


Clock: C < P ≈ FP

Area: FP < C < P

Throughput: FP < C < P

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
- 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 f0, f1 and f2 with stage_f code

Stage functions f1, f2 and f3

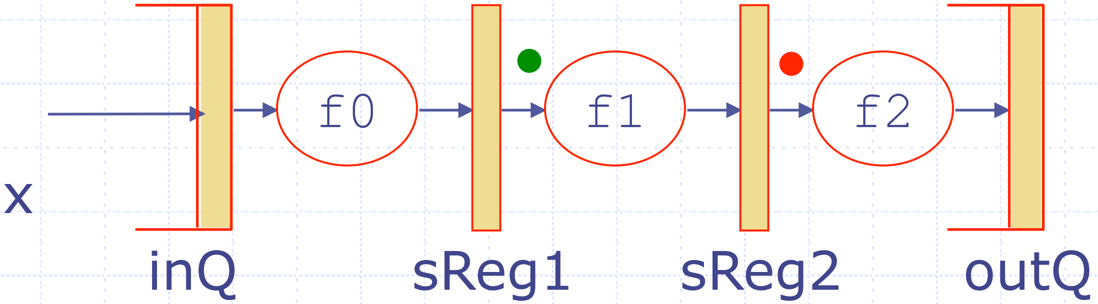
```
function f0(x);
        return (stage_f(0,x));
endfunction

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
```

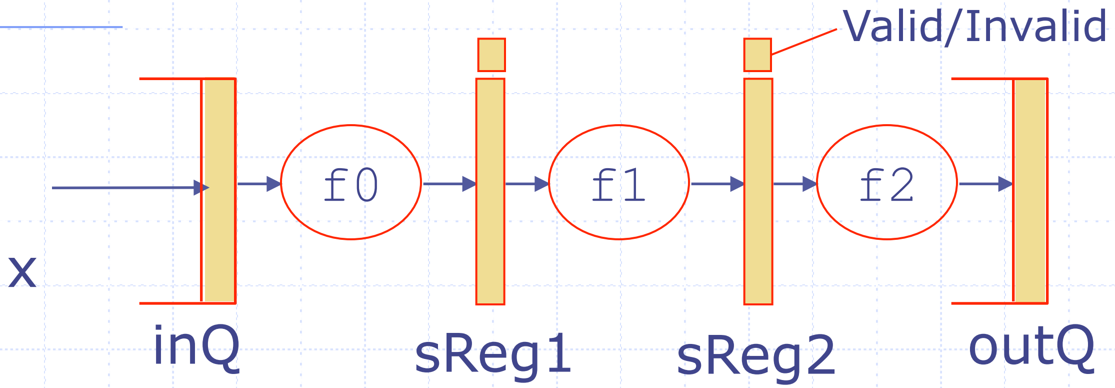
Red and Green tokens must move even if there is nothing in the inQ!

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

Explicit encoding of Valid/ Invalid data



```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg1 <= f0(inQ.first()); inQ.deq();
        sReg1f <= Valid end
  else  sReg1f <= Invalid;
  sReg2 <= f1(sReg1); sReg2f <= sReg1f;
 if (sReg2f == Valid) outQ.enq(f2(sReg2));
endrule
```

When is this rule enabled?

```
rule sync-pipeline (True);
 if (inQ.notEmpty())
  begin sReg1 <= f0(inQ.first()); inQ.deq();
        sReg1f <= Valid end
  else  sReg1f <= Invalid;
  sReg2 <= f1(sReg1); sReg2f <= sReg1f;
 if (sReg2f == Valid) outQ.enq(f2(sReg2));
endrule
```

inQ	sReg1f	sReg2f	outQ	
NE	V	V	NF	yes
NE	V	V	F	No
NE	V	I	NF	Yes
NE	V	I	F	Yes
NE	I	V	NF	Yes
NE	I	V	F	No
NE	I	I	NF	Yes
NE	I	I	F	yes

inQ	sReg1f	sReg2f	outQ	
E	V	V	NF	yes
E	V	V	F	No
E	V	I	NF	Yes
E	V	I	F	Yes
E	I	V	NF	Yes
E	I	V	F	No
E	I	I	NF	Yes1
E	I	I	F	yes

Yes1 = yes but
no change

Area estimates

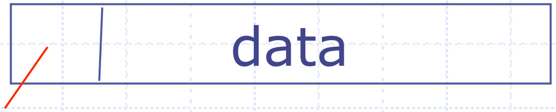
Tool: Synopsys Design Compiler

- Comb. FFT
  - Combinational area: 16536
  - Noncombinational area: 9279
- Linear FFT
  - Combinational area: 20610
  - Noncombinational area: 18558
- Circular FFT
  - Combinational area: 29330
  - Noncombinational area: 11603

Surprising? Explanation?

The Maybe type data in the pipeline

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



valid/invalid

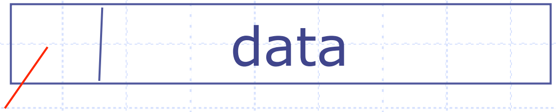
- Registers contain Maybe type values

```
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

The Maybe type data in the pipeline – another style

```
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;
  sReg2 <= isValid(sReg1)? Valid f1(unJust(sReg1)) :
Invalid;
 if isValid(sReg2) outQ.enq(f2(unJust(sReg2)));
endrule
```