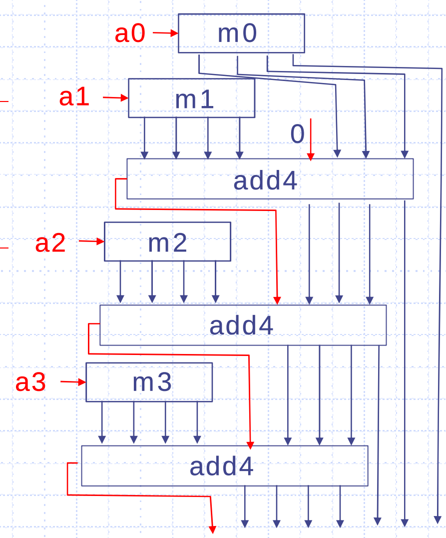
Folded Combinational Circuits as an example of Sequential Circuits

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Multiplication by repeated addition

```
b Multiplicand   1101   (13)
a Muliplier   *  1011   (11)
                 1101
          +     1101
          +    0000
          +   1101
             10001111   (143)
```




```
mi = (a[i]==0)? 0 : b;
```

Combinational 32-bit multiply

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

Combinational circuit uses 31 add32 circuits

Design issues with combinational multiply

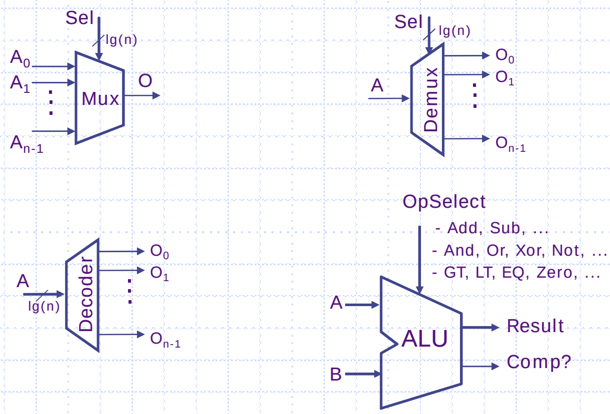
- Lot of hardware
  - 32-bit multiply uses 31 add32 circuits
- Long chains of gates
  - 32-bit ripple carry adder has a 31-long chain of gates
  - 32-bit multiply has 31 ripple carry adders in sequence!

The speed of a combinational circuit is determined by its longest input-to-output path

Can we do better?

We can reuse the same add32 circuit if we can store the partial results in some storage device, e.g., *register*

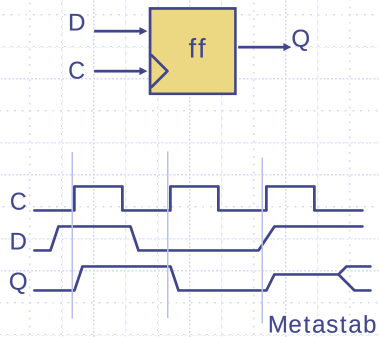
Combinational circuits



Such circuits have no cycles (feedback) or state elements

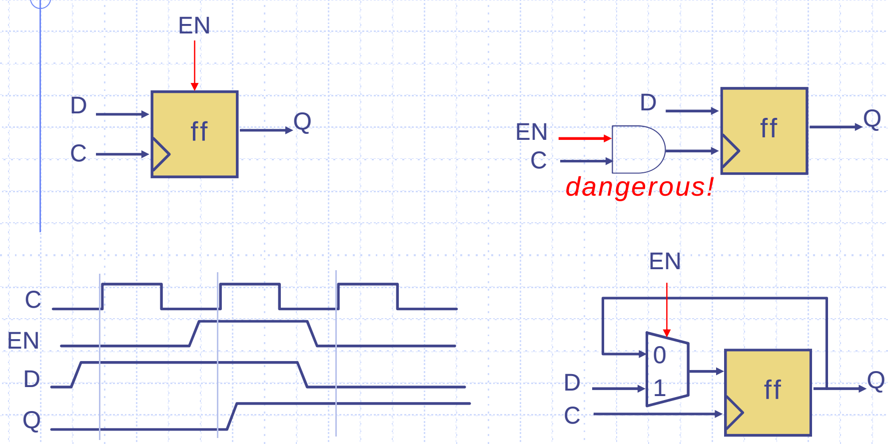
A simple synchronous state element

Edge-Triggered Flip-flop



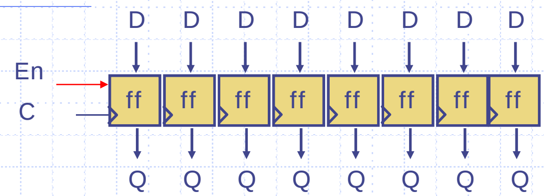
Data is sampled at the rising edge of the clock

Flip-flops with Write Enables



Data is captured only if EN is on

Registers



Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, input and output port(s)

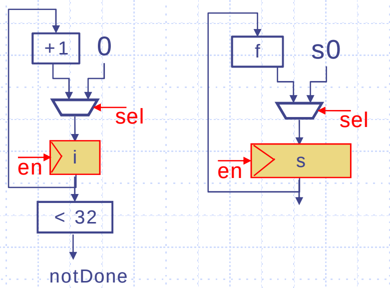
We can build useful and compact circuits using registers

Circuits containing state elements are called *sequential circuits*

Expressing a loop using registers

```
int s = s0;
for (int i = 0; i < 32; i = i+1) {
    s = f(s);
  }
return s;
```

C-code



We need two registers to hold s and i values from one iteration to the next.

These registers are initialized when the computation starts and updated every cycle until the computation terminates

sel = start
en = start | notDone

Expressing sequential circuits in BSV

- Sequential circuits, unlike combinational circuits, are not expressed structurally (as wiring diagrams) in BSV
- For sequential circuits a designer defines:
  - *State elements* by instantiating modules

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);
```

make a 32-bit register which is uninitialized

make a 6-bit register with initial value 32

  - *Rules* which define how state is to be transformed atomically

```
rule step if (i < 32);
  s <= f(s);
  i <= i+1;
endrule
```

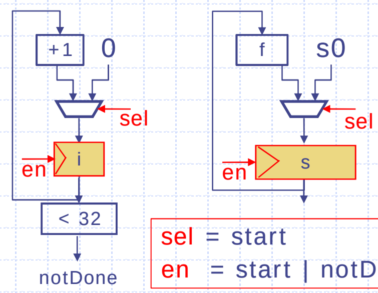
the rule can execute only when its guard is true

actions to be performed when the rule executes

Rule Execution

- When a rule executes:
  - all the registers are read at the beginning of a clock cycle
  - the guard and computations to evaluate the next value of the registers are performed
  - at the end of the clock cycle registers are updated iff the guard is true
- Muxes are need to initialize the registers

```
Reg#(Bit#(32)) s <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);

rule step if (i < 32);
     s <= f(s);
     i <= i+1;
endrule
```



sel = start
en = start | notDone

Multiply using registers

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

Combinational version

Need registers to hold a, b, tp, prod and i

Update the registers every cycle until we are done

Sequential multiply

```
Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <-mkRegU();
Reg#(Bit#(32)) tp <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);

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

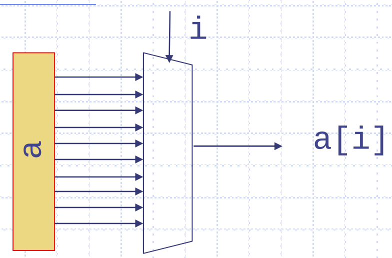
state elements

a rule to describe the dynamic behavior

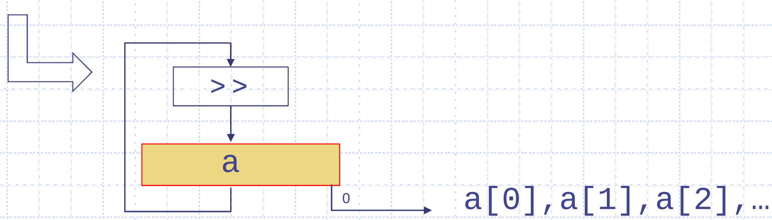
similar to the loop body in the combinational version

So that the rule won't fire until i is set to some other value

Dynamic selection requires a mux



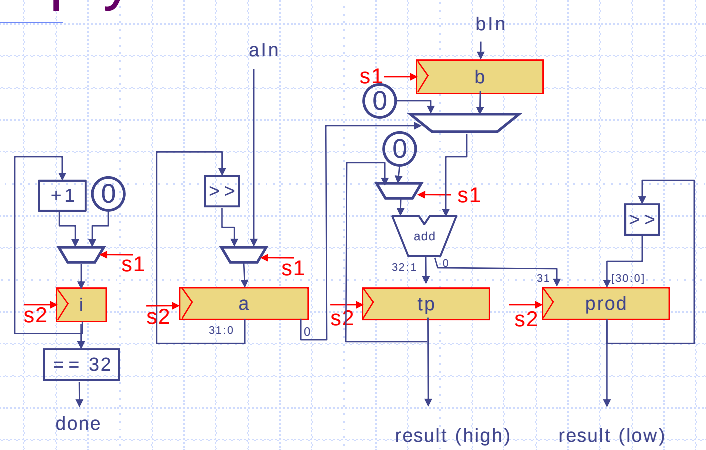
when the selection indices are regular then it is better to use a shift operator (no gates!)



Replacing repeated selections by shifts

```
Reg#(Bit#(32)) a <- mkRegU();
Reg#(Bit#(32)) b <- mkRegU();
Reg#(Bit#(32)) prod <-mkRegU();
Reg#(Bit#(32)) tp <- mkRegU();
Reg#(Bit#(6))  i <- mkReg(32);

rule mulStep if (i < 32);
   let m = (a[0]==0)? 0 : b;
   a <= a >> 1;
   let sum = add32(m,tp,0);
   prod <= {sum[0], (prod >> 1)[30:0]};
   tp <= sum[32:1];
   i <= i+1;
endrule
```


Circuit for Sequential Multiply

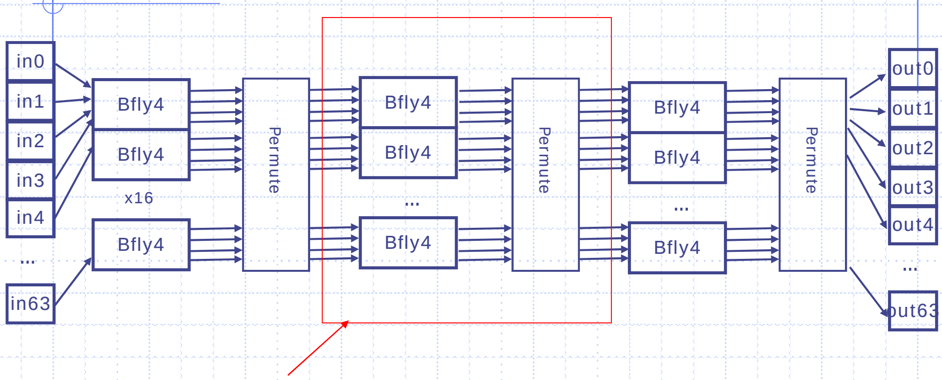


s1 = start_en
s2 = start_en | !done

Circuit 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 31 serial add32's to one add32 plus a few muxes
- The sequential circuit will take 31 clock cycles to compute an answer

Combinational IFFT



Reuse the same circuit three times to reduce area

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

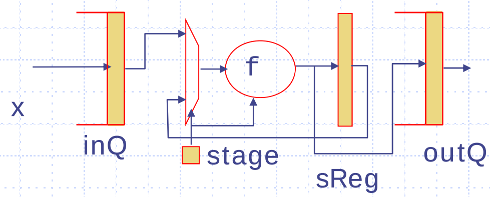
Higher-order functions: Stage functions f1, f2 and f3

```
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);
```

Folded Combinational Ckts

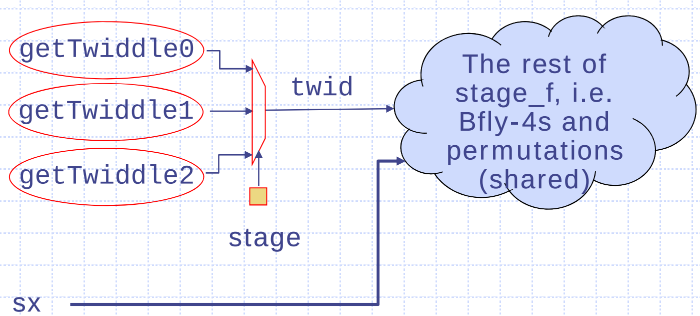



```
rule folded-pipeline (True);
 let sxIn = ?;
 if (stage==0)
   begin sxIn= inQ.first(); inQ.deq(); end
 else     sxIn= sReg;
 let sxOut = f(stage,sxIn);
 if (stage==n-1) outQ.enq(sxOut);
 else sReg <= sxOut;
 stage <= (stage==n-1)? 0 : stage+1;
endrule
```

notice stage is a dynamic parameter now!

no for-loop

Shared Circuit



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

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 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

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!