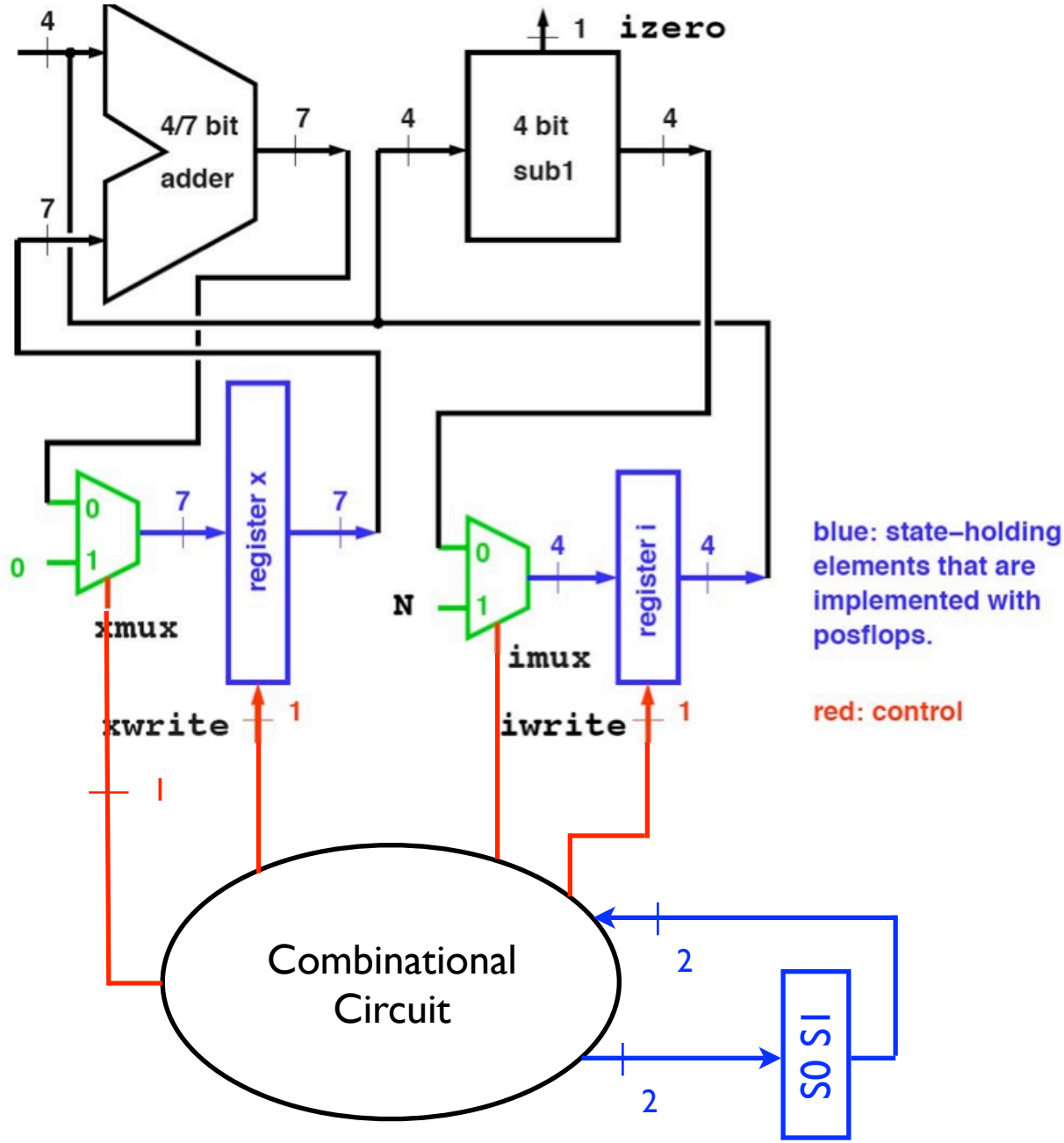
The Design So Far ...



The Logic Equations ...

- xdone = s0
- xwrite = $\overline{s0}$
- imux = $\overline{s0} \cdot \overline{s1}$
- xmux = $\overline{s0} \cdot \overline{s1}$
- iwrite = 1
- News0 = $\overline{s0} \cdot s1 \cdot izero \cdot \overline{Reset}$
- News1 = $(\overline{s0} \cdot \overline{s1} + s1 \cdot \overline{izero}) \cdot \overline{Reset}$

CSL

The CAST Language

A Circuit Description Language
(looks a lot like a programming language)

CAST objects:

NODES: connection points / wires
(look a lot like variables)

BLOCKS: gates connected together through nodes
(look a lot like procedures)

CSL

The Initial Parts List ...

node CLK, _CLK;	global clock
node Reset;	global reset
node Vdd, GND;	supply and ground
node TRUE = Vdd;	source of logical 1
node FALSE = GND;	source of logical 0
Nand2( )(node a, b; node out)	2-input NAND gate
Nor2( )(node a, b; node out)	2-input NOR gate
Inv( )(node in; node out)	Inverter
posFLOP( )(node in, out)	pos edge-triggered Flip-Flop
negFLOP( )(node in, out)	neg edge-triggered Flip-Flop (clock is implicit) (no complemented output)

A CAST Header

Looks like a C function prototype:

```
define Negater (int N) (node[N] in, out; node sel) {
    ...
}
```

numeric parameters
control size of circuit
i.e. replication of parts

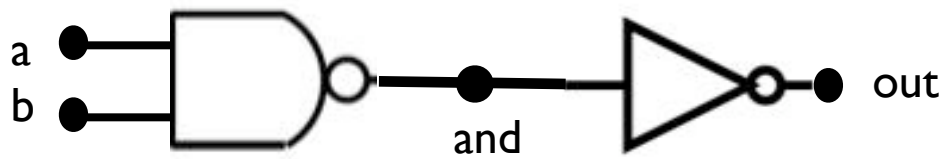
interface -- nodes and arrays of nodes that connect this block to the rest of the circuit

definition of the Negater block

CSL

Defining a Logic Block

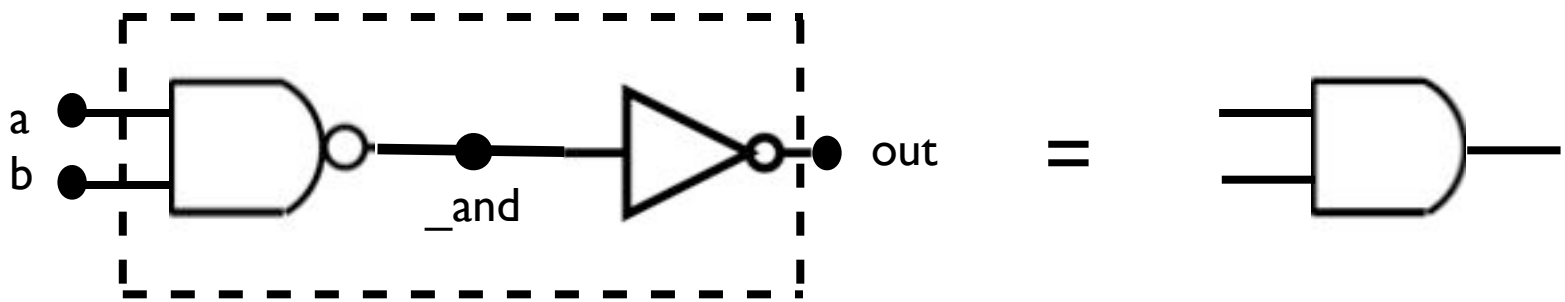
```
define And2( ) (node a, b; node out)
{
  node _and;

  Nand2() (a,b,_and);
  Inv() (_and,out);
}
```



CSL

The Header defines the Interface ...

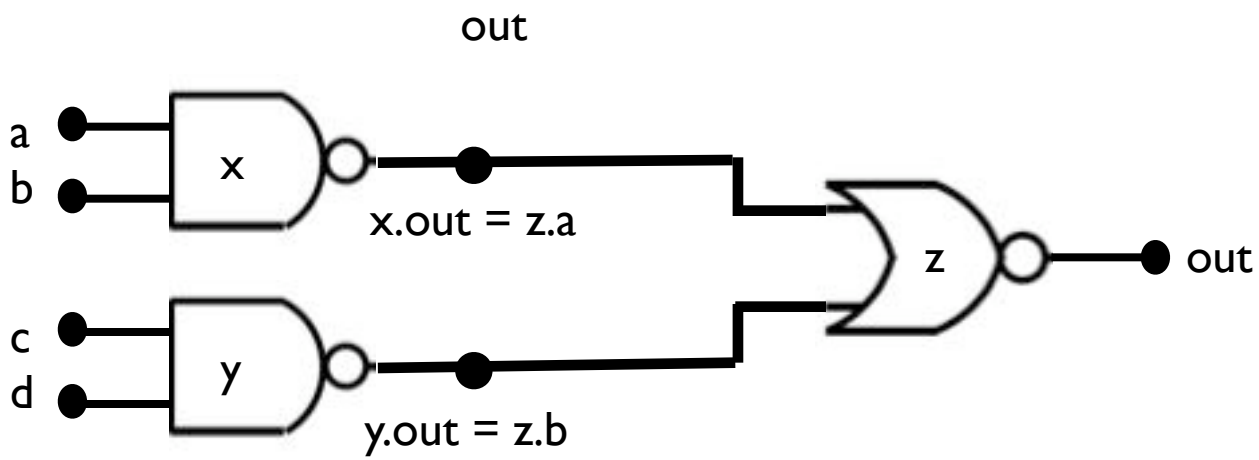
```
define And2( ) (node a, b; node out)
{
   . . .
}
```



The structure inside is hidden.

Another Style ...

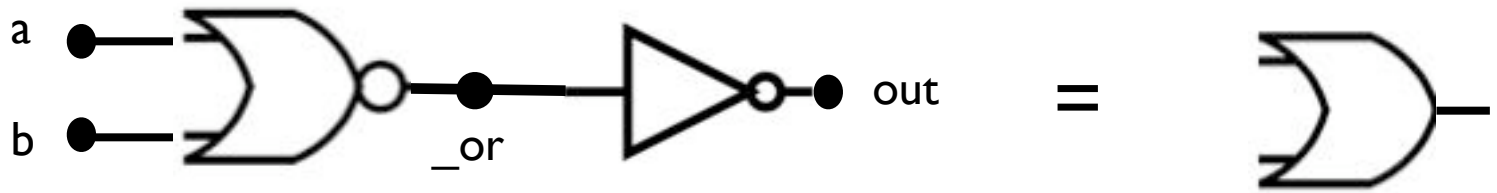
```
define And4( ) (node a,b,c,d; node out)
{
  Nand2 x(a,b);
  Nand2 y(c,d);
  Nor2 z(x.out,y.out,out);
}
```



CSL

Again with OR ...

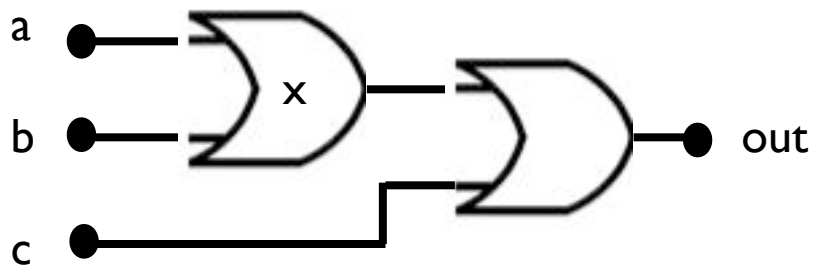
```
define Or2() (node a, b; node out)
{
  node _or;
  Nor2() (a,b,_or);
  Inv () (_or, out);
}
```



CSL

Nested Instantiations ...

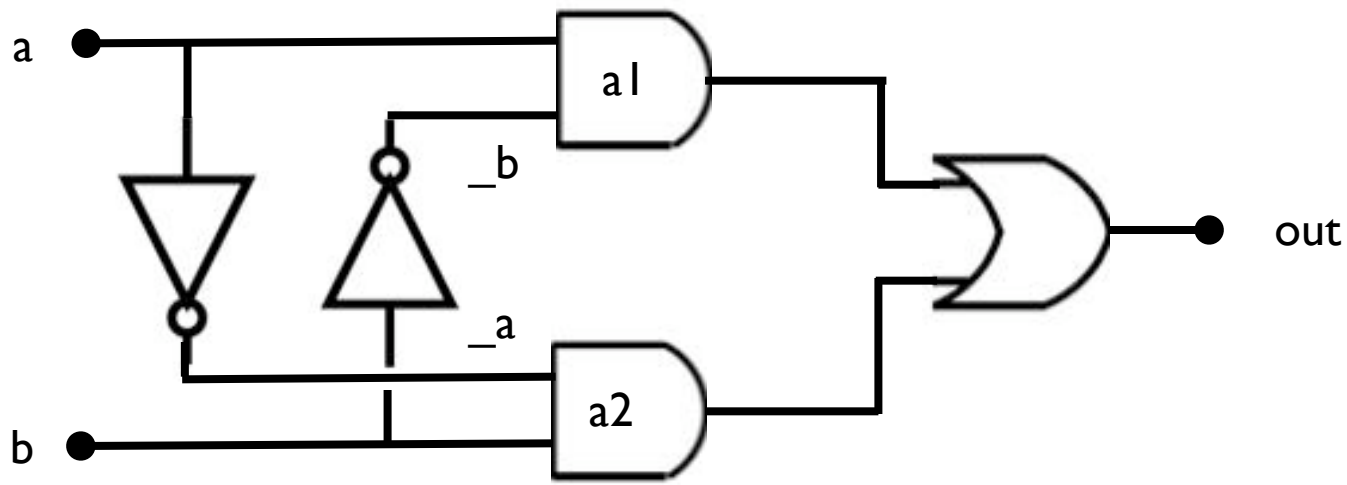
```
define Or3() (node a, b, c; node out)
{
  Or2 x(a,b);
  Or2() (x.out,c,out);
}
```



CSL

Nested Instantiations Again ...

```
define Xor2() (node a, b; node out)
{
  node _a, _b;
  Inv() (a,_a);
  Inv() (b,_b);
 And2 a1(a,_b);
 And2 a2(b,_a);
  Or2() (a1.out,a2.out,out);
}
```



CSL

The 1-bit Full Adder

```
define onebitadd() (node a, b, cin; node sum, cout)
{
  node x0,x1,x2;

  And2() (a,b,x0);
  And2() (a,cin,x1);
  And2() (b,cin,x2);
  Or3() (x0,x1,x2,cout);

  Xor2 x(a,b);
  Xor2() (x.out,cin,sum);
}
```

Verify these match the FA we defined earlier ...

CSL

Arrays of Nodes & Parts and Iteration

```
define Adder(int N) (node[N] a,b; node cin;
                     node[N] sum; node cout)
{
  onebitadd[N] o;

  <i:N: o[i].a=a[i]; o[i].b=b[i]; o[i].sum=sum[i];>

  <i:N-1:o[i].cout=o[i+1].cin;>

  o[0].cin=cin;
  o[N-1].cout=cout;
}
```

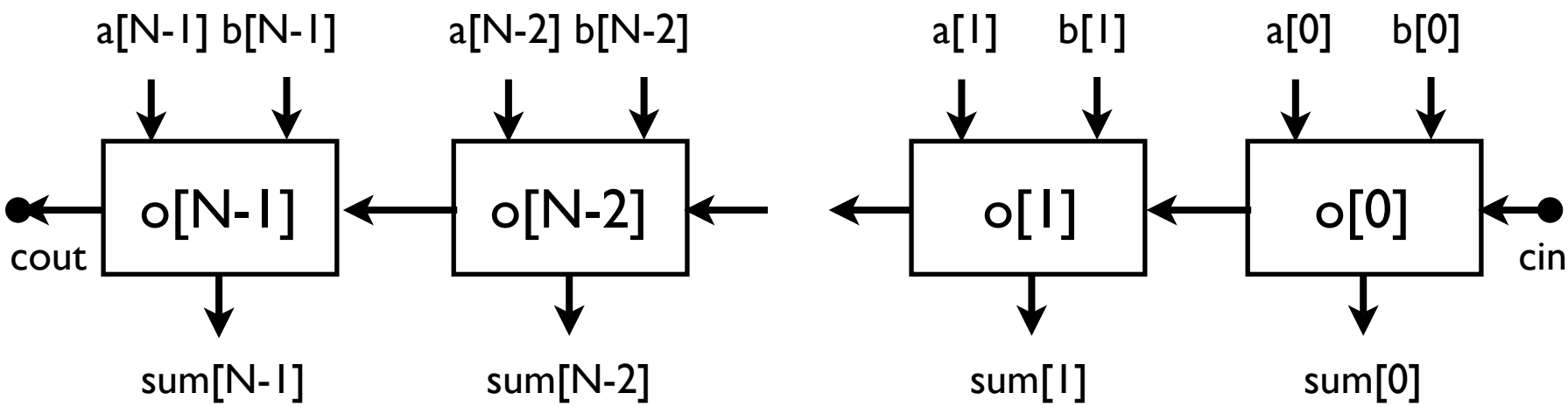
Array of FA parts

Loops

Simple connections

CSL

Here's the Diagram



A (Wasteful) 4 + 7 bit Adder

```
define Add47() (node[4] a; node[7] b; node[7] out)
{
  Adder(7) x;

  x.b=b;
  x.cin=GND;
  x.sum=out;

  x.a[0..3]=a;
  x.a[4]=GND;
  x.a[5]=GND;
  x.a[6]=GND;
}
```

Array assignment connection

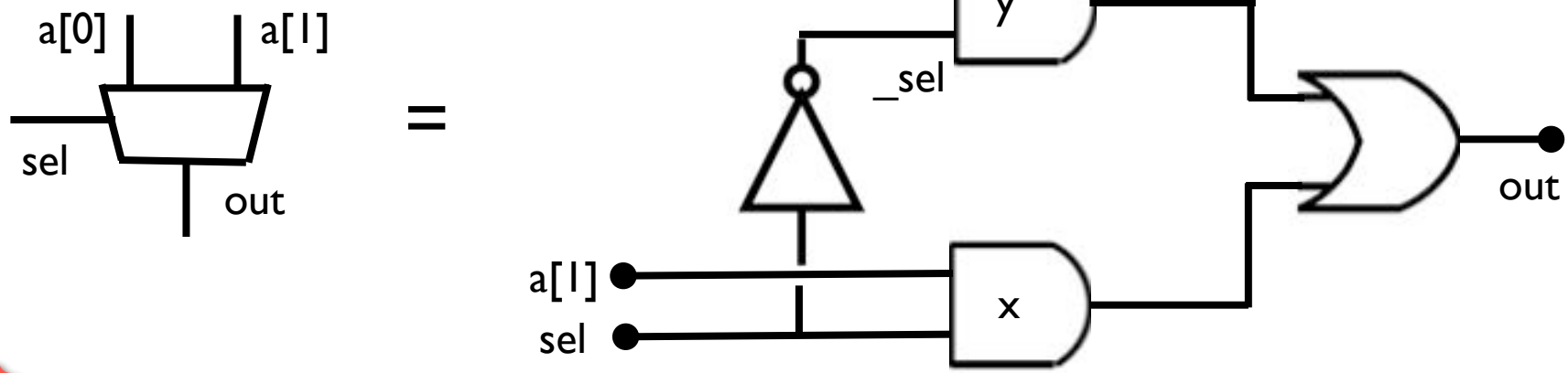
Just connect the unused inputs to 0.

CSL

The 1-bit Mux

```
define Mux() (node[2] a; node sel; node out)
{
 node _sel;
 Inv () (sel,_sel);
 And2 x(a[1],sel);
 And2 y(a[0],_sel);
 Or2() (x.out,y.out,out);
}
```

Note specification of two inputs as node[2] array rather than two scalar nodes

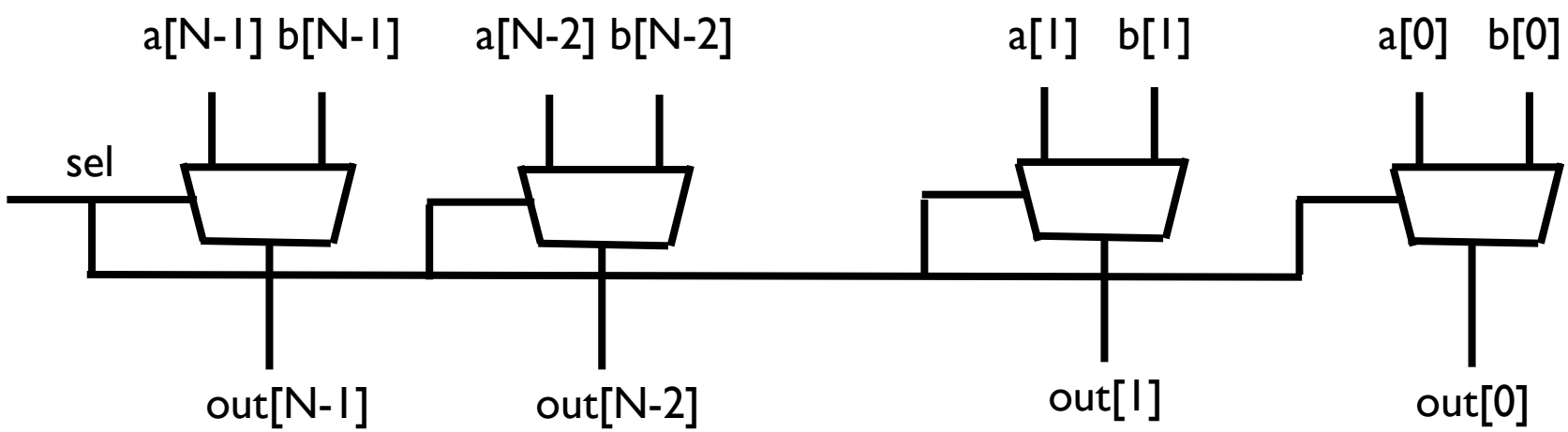


CSL

An N-bit Mux

```
define MuxN(int N) (node[N] a, b; node sel; node[N] out)
{
  <i:N:Mux() ({a[i],b[i]},sel,out[i]);>
}
```

Note use of {a[i],b[i]}
to construct a 2-element array



CSL

Subtract-1 Unit

```
define sub1() (node[4] in, out; node izero)
{
  Adder(4) a;

  a.a = in;
  a.b[0] = GND;
  <i:1..3:a.b[i]=Vdd;>
  a.cin=Vdd;
  out=a.sum;
  Or2 x1(out[0],out[1]);
  Or2 x2(out[2],out[3]);
  Or2 x(x1.out,x2.out);
  Inv() (x.out,izero);
}
```

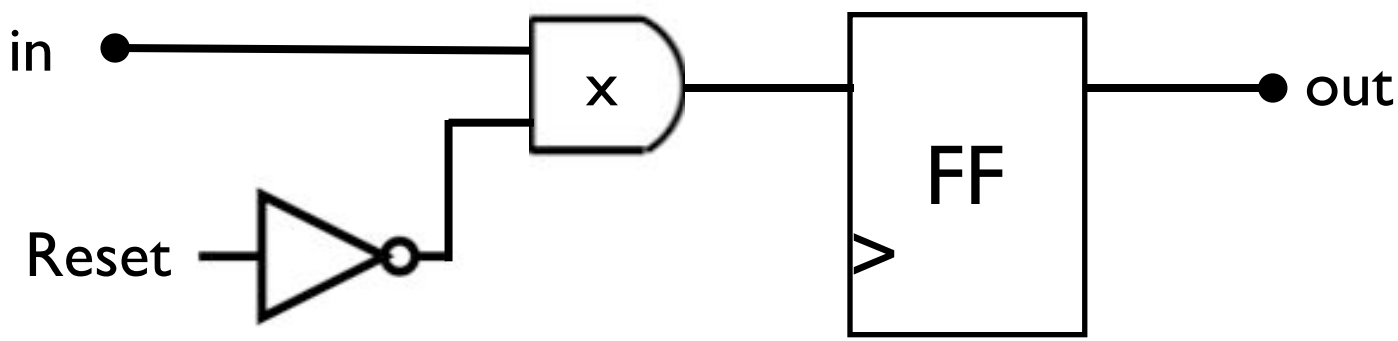
Another correct but wasteful part!

CSL

A FLOP that is Cleared on Reset

```
define rposFLOP() (node in, out)
{
  node _r;
  Inv () (Reset,_r);
  And2  x(_r,in);
  posFLOP() (x.out,out);
}
```

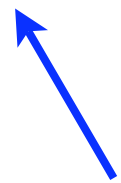
Reset will be high for several clock periods



CSL

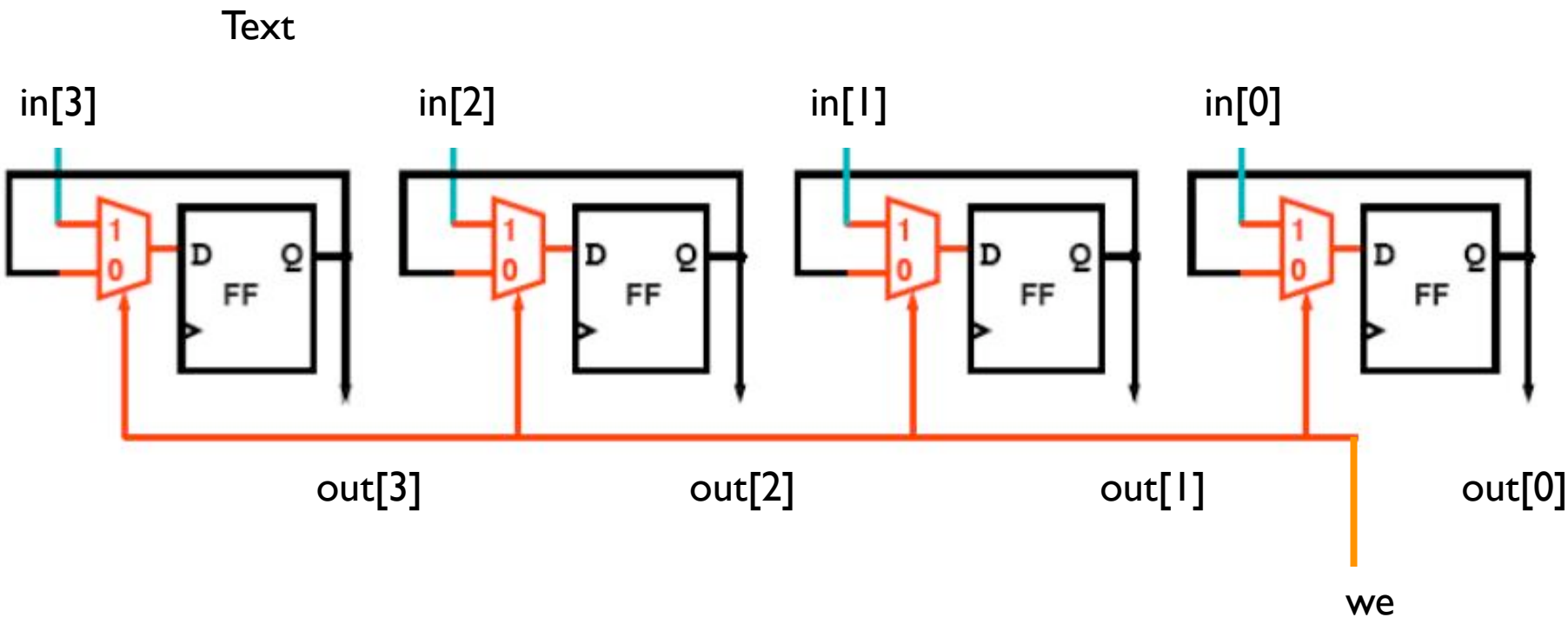
N-bit Register with Write Enable

```
define WReg(int N) (node[N] in, out; node we)
{
 Mux[N] m;

 <i:N:rposFLOP() (m[i].out,out[i]);>

 <i:N:m[i].sel = we;>
 <i:N:m[i].a[0] = out[i];>
 <i:N:m[i].a[1] = in[i];>
}
```



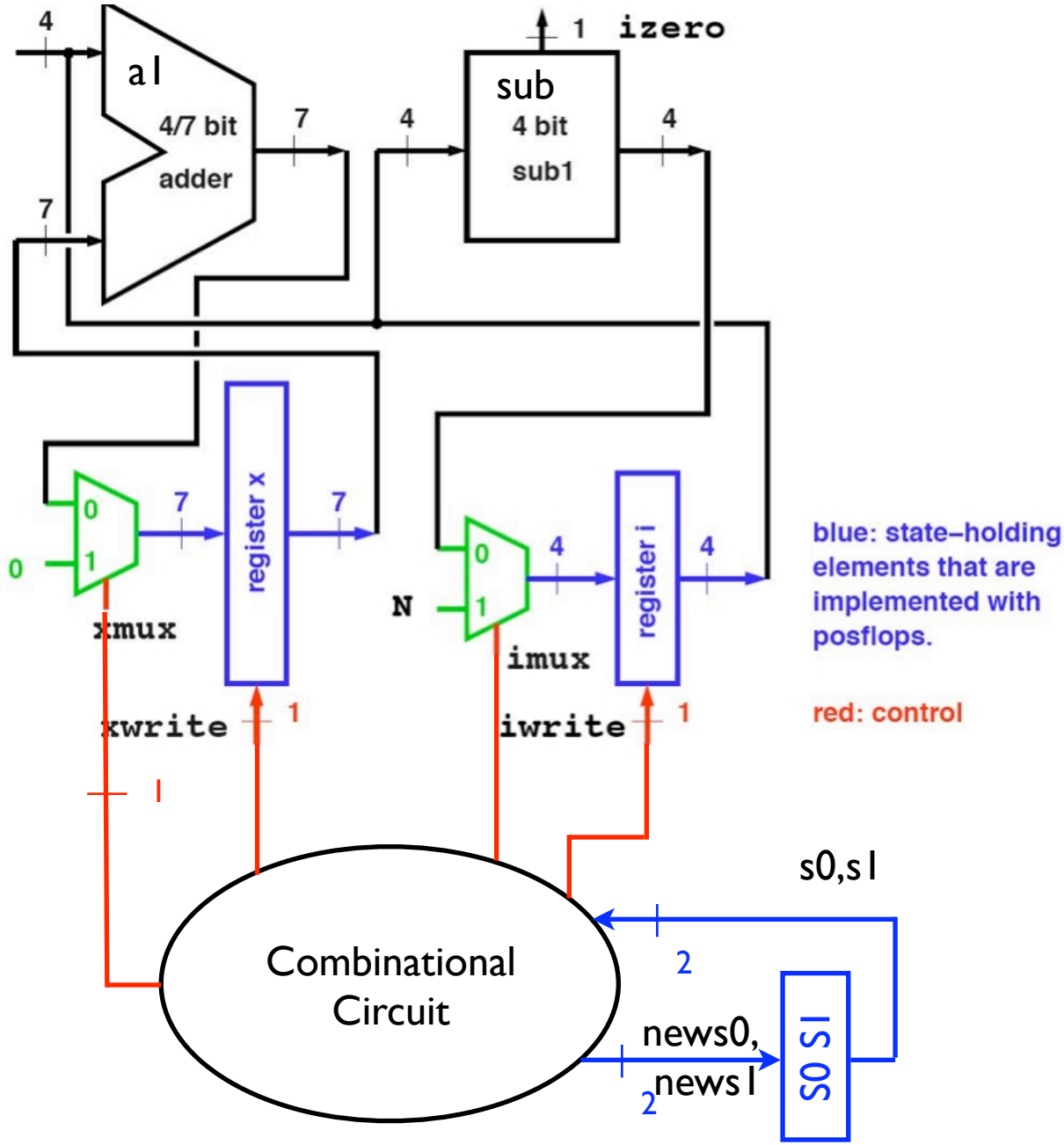
Create the FLOPs inside loop rather than as array

Here's the Diagram ...



CSL

The Design So Far ...



Text

blue: state–holding elements that are implemented with posflops.

red: control

Declaring the Signals ...

```
define myckt() (node[4] N; node[7] X; node xdone)
{
  node[7] xval;
  node[4] ival;
  node xmux, xwrite, imux, iwrite, izero;
  node s0,s1,news0,news1;
  MuxN(7) mx;
  MuxN(4) mi;
 WReg(7) xreg(mx.out, xval, xwrite);
 WReg(4) ireg(mi.out, ival, iwrite);
 WReg(2) sreg({news0,news1},{s0,s1},Vdd);
 Add47 a1(ival,xval,mx.a);
  sub1 sub(ival,mi.a,izero);
```

CSL

Remaining Datapath Connections ...

```
mi.sel = imux;
mx.sel = xmux;
xval = X;
<i:7:mx.b[i]=GND;>
mi.b=N;
```

State Transitions ...

```
node _s0, _r, _izero;
Inv() (Reset,_r);
Inv() (s0,_s0);
Inv() (izero,_izero);

And4() (_r,_s0,s1,izero,news0);

Nor2 n1(s0,s1);
And2 n2(_izero,s1);
Or2 o(n1.out,n2.out);
And2 () (_r,o.out,news1);
```

- $\text{News0} = \overline{s0} \cdot s1 \cdot \text{izero} \cdot \overline{\text{Reset}}$
- $\text{News1} = (\overline{s0} \cdot \overline{s1} + s1 \cdot \overline{\text{izero}}) \cdot \overline{\text{Reset}}$

CSL

Control ...

```
  xdone = s0;
  Inv() (s0,xwrite);
  Nor2() (s0,s1,imux);
  imux=xmux;
  iwrite=Vdd;
}
```

- $xdone = s0$
- $xwrite = \overline{s0}$
- $imux = \overline{s0} \cdot \overline{s1}$
- $xmux = \overline{s0} \cdot \overline{s1}$
- $iwrite = 1$

CSL

Finally, Instantiate the Circuit

... using the myckt block we just defined

```
node[4] N;
node[7] X;
node xdone;

myckt m(N,X,xdone);
```