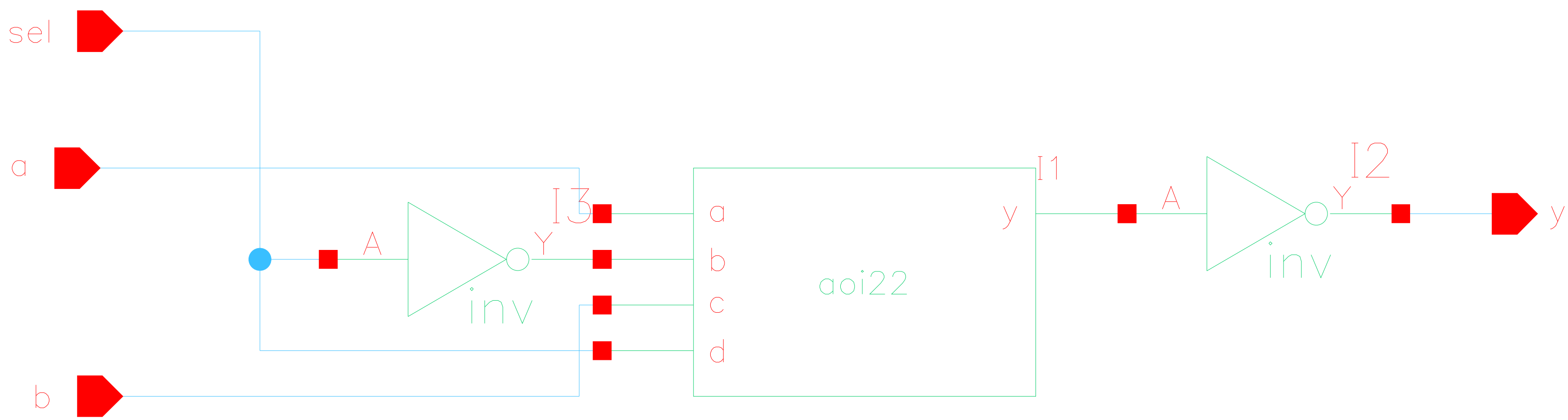
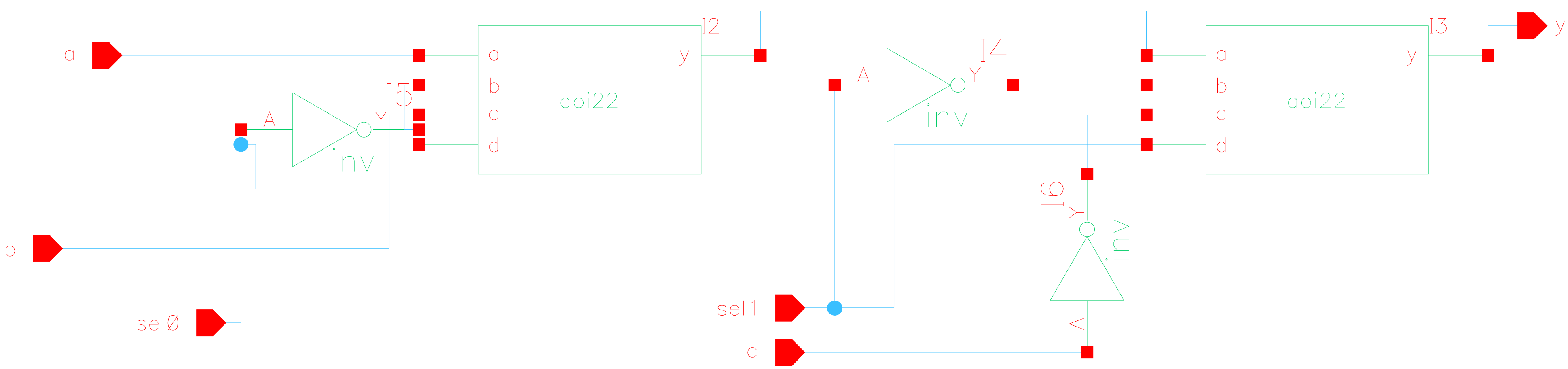
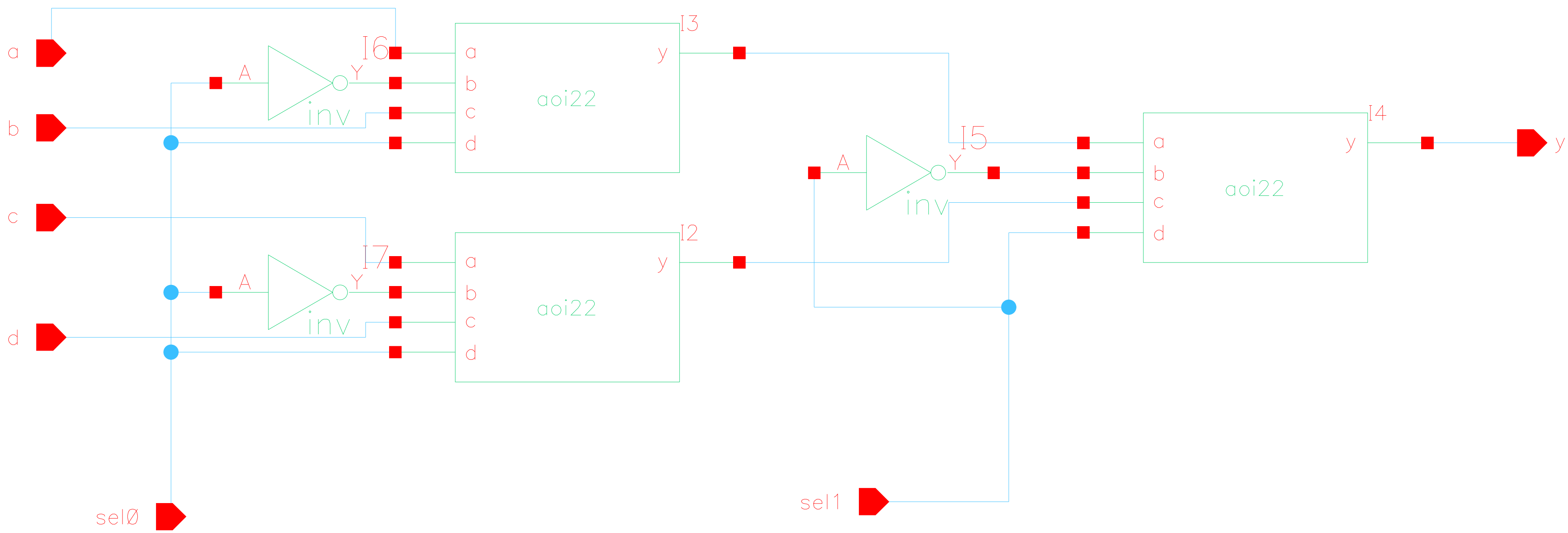
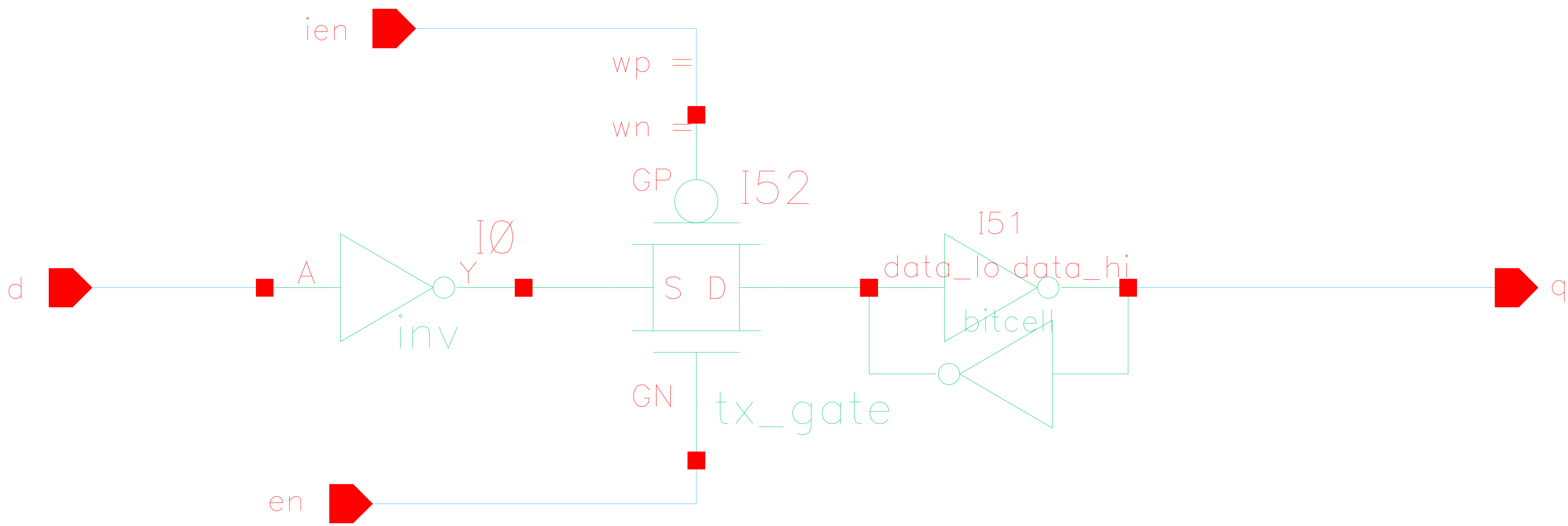

sel
a
b
A
inv
Y
I3
a
b
c
d
aoi22
y
I1
A
inv
Y
I2
y

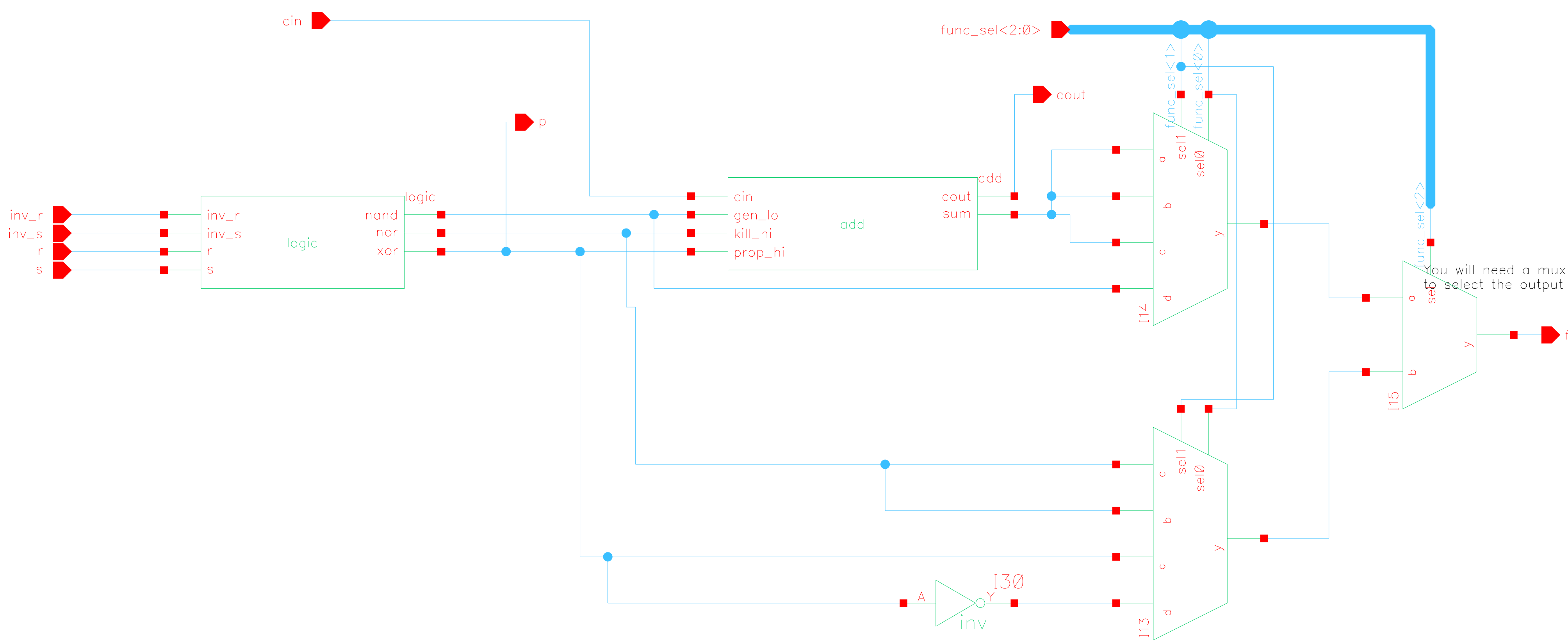


a
b
sel0
I5
A
Y
inv
I2
a
b
c
d
y
aoi22
I4
A
Y
inv
sel1
c
I6
Y
A
inv
I3
a
b
c
d
y
aoi22
y

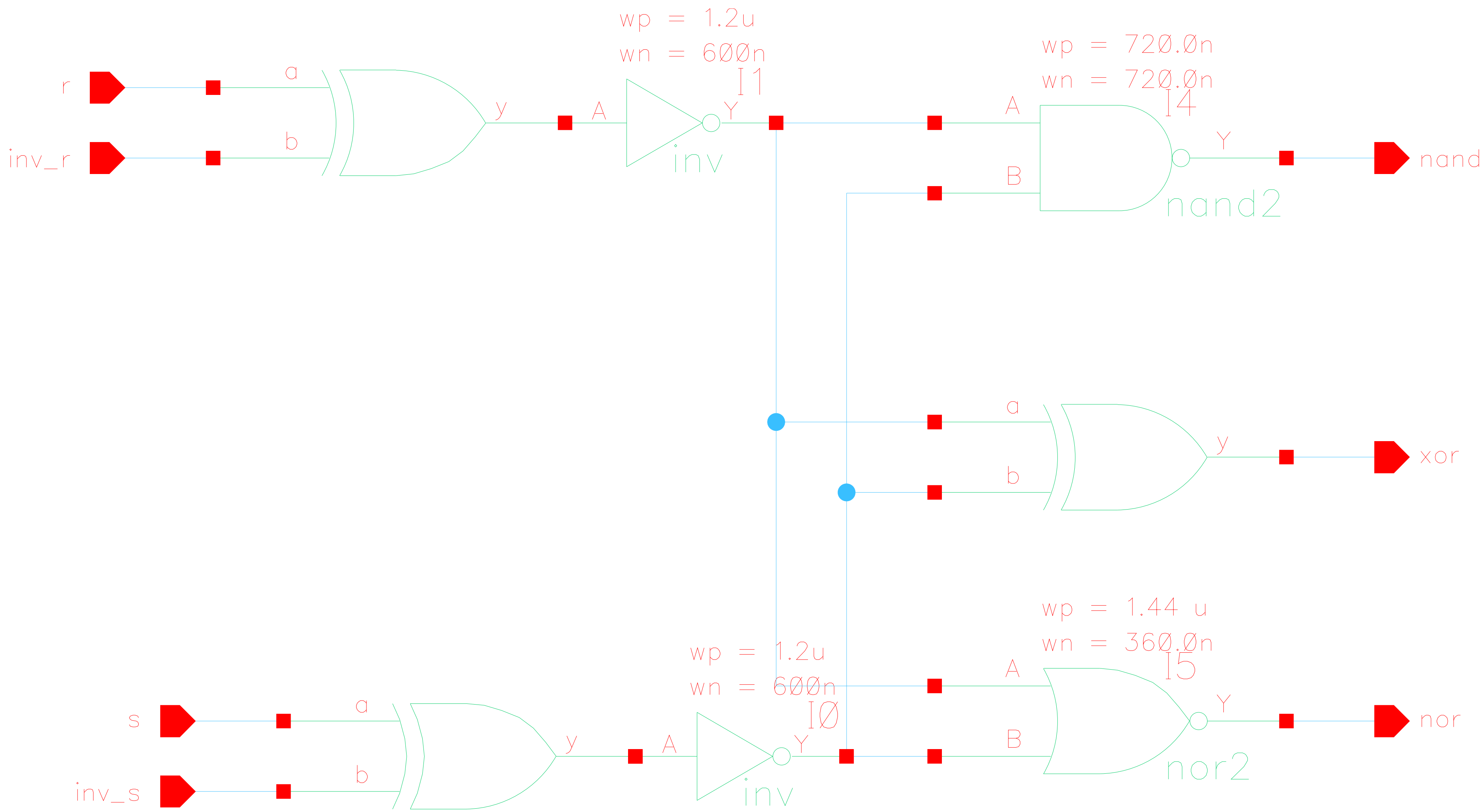

a
b
c
d
sel0
sel1
y
I6
I7
I5
I3
I2
I4
A
Y
inv
A
Y
inv
A
Y
inv
aoi22
aoi22
aoi22
a
b
c
d
y
a
b
c
d
y
a
b
c
d
y



ien
wp =
wn =
GP
I52
I0
A
Y
inv
S
D
GN
tx_gate
I51
data_lo
data_hi
bitcell
d
q
en



inv_r
inv_s
r
s
logic
nand
nor
xor
cin
p
add
gen_lo
kill_hi
prop_hi
cout
sum
func_sel<2:Ø>
func_sel<1>
func_sel<Ø>
func_sel<2>
sel1
selØ
sel
a
b
c
d
y
I14
I13
I15
A
Y
I3Ø
inv
You will need a mux
to select the output
f


r
inv_r
a
b
y
A
wp = 1.2u
wn = 600n
I1
Y
inv
wp = 720.0n
wn = 720.0n
I4
A
B
Y
nand2
nand
a
b
y
xor
s
inv_s
a
b
y
A
wp = 1.2u
wn = 600n
I0
Y
inv
wp = 1.44 u
wn = 360.0n
I5
A
B
Y
nor2
nor

ALU Functions:

1. ADD:

   Implemented by using the adder given. inv_r and in_s are both set to high because they are active low. We want to add two positive numbers.

2. SUBR

   Uses the adder from ADD but we have inv_r = 0 and inv_s = 1 and C_{in} = 1, giving us S + (-R), where (-R) is just the two's complement of R.

3. SUBS

   Uses the adder from ADD but we have inv_r = 0 and inv_s = 1 and C_{in} = 1, giving us S + (-R), where (-R) is just the two's complement of R.

4. OR

   We use the *logic* in our ALU to get the OR value. We have inv_r = 0 and inv_s = 0. We use DeMorgan's law to get R OR S from ~R NAND ~S.

5. AND

   We use the *logic* in our ALU to get the AND value. We have inv_r = 0 and inv_s = 0. We use DeMorgan's law to get R AND S from ~R NOR ~S.

6. NOTRS

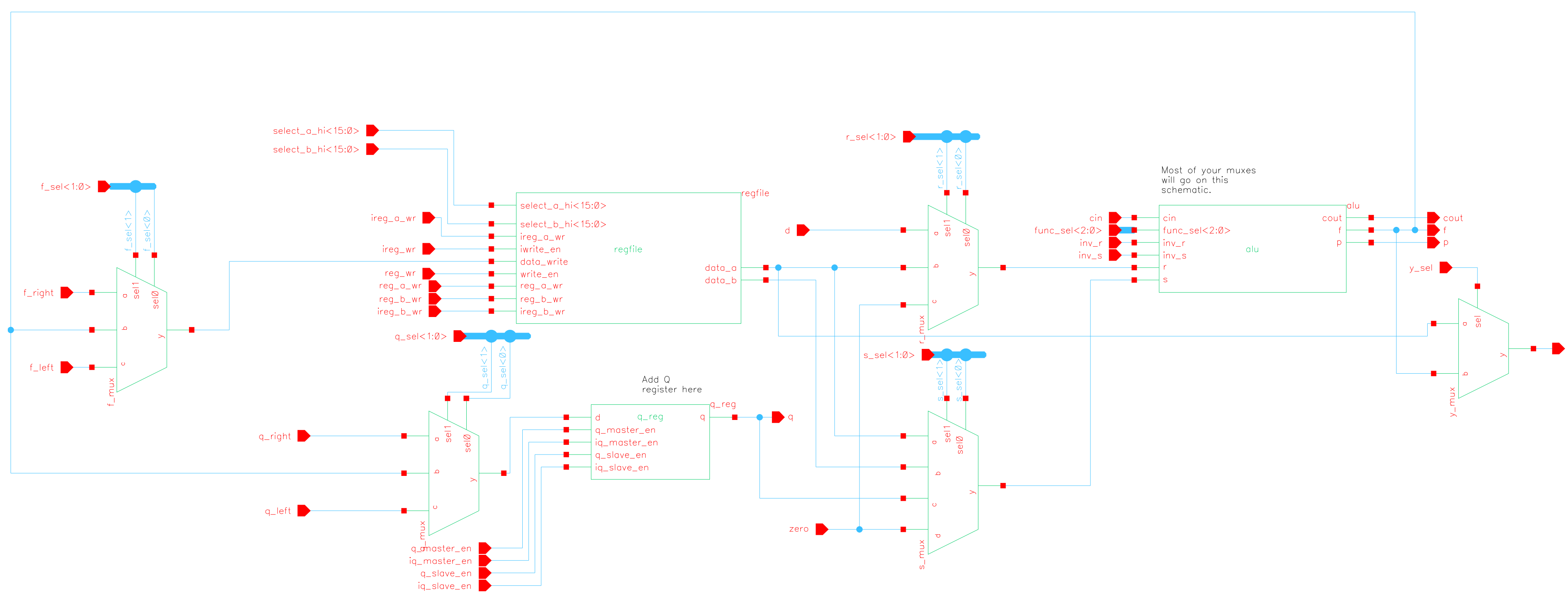
   We use the same exact logic as the AND function but we set inv_r = 1 and keep inv_s = 0. This gives us R NOTRS S from R NOR ~S

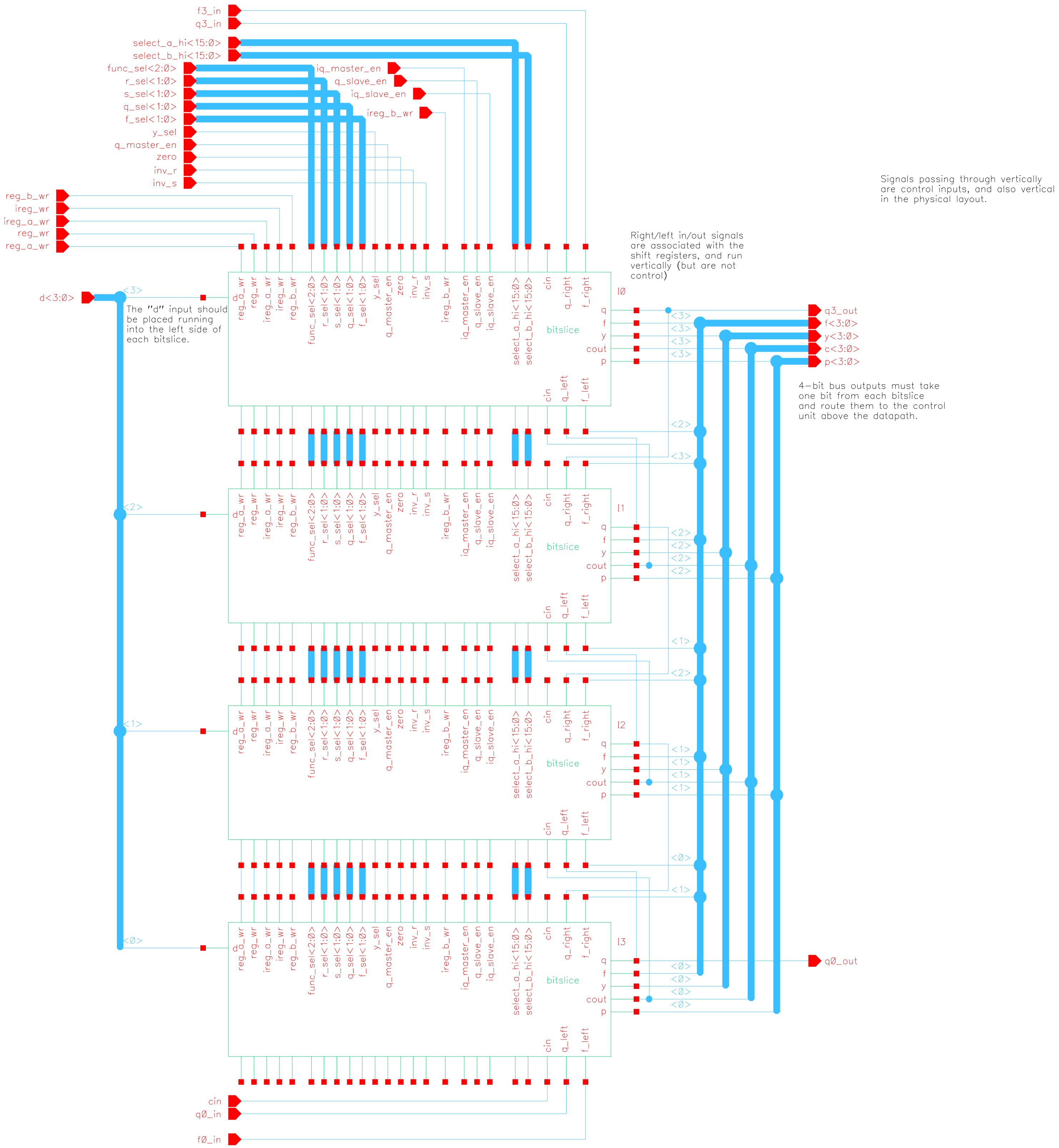
7. XOR

   We use the XOR gate in the ALU logic to produce the XOR function. We set inv_r = 1 and inv_s = 1 to make sure our logic uses R and S.

8. XNOR

The XNOR function uses the exact same signals as XOR but we invert the output to get XNOR.


Most of your muxes
will go on this
schematic.
Add Q
register here
regfile
alu
q_reg
f_mux
q_mux
r_mux
s_mux
y_mux
select_a_hi<15:0>
select_b_hi<15:0>
ireg_a_wr
iwrite_en
data_write
write_en
reg_a_wr
reg_b_wr
ireg_b_wr
ireg_wr
reg_wr
data_a
data_b
cin
func_sel<2:0>
inv_r
inv_s
cout
f
p
y_sel
y
d
zero
q
q_master_en
iq_master_en
q_slave_en
iq_slave_en
f_sel<1:0>
q_sel<1:0>
r_sel<1:0>
s_sel<1:0>
f_right
f_left
q_right
q_left



f3_in
q3_in
select_a_hi<15:0>
select_b_hi<15:0>
func_sel<2:0>
r_sel<1:0>
s_sel<1:0>
q_sel<1:0>
f_sel<1:0>
y_sel
q_master_en
zero
inv_r
inv_s
iq_master_en
q_slave_en
iq_slave_en
ireg_b_wr
reg_b_wr
ireg_wr
ireg_a_wr
reg_wr
reg_a_wr
d<3:0>
The "d" input should be placed running into the left side of each bitslice.
Right/left in/out signals are associated with the shift registers, and run vertically (but are not control)
Signals passing through vertically are control inputs, and also vertical in the physical layout.
4-bit bus outputs must take one bit from each bitslice and route them to the control unit above the datapath.
I0
I1
I2
I3
bitslice
d
cin
q_right
f_right
q_left
f_left
q
f
y
cout
p
q3_out
f<3:0>
y<3:0>
c<3:0>
p<3:0>
q0_out
cin
q0_in
f0_in

Waveform 1 - SimVision