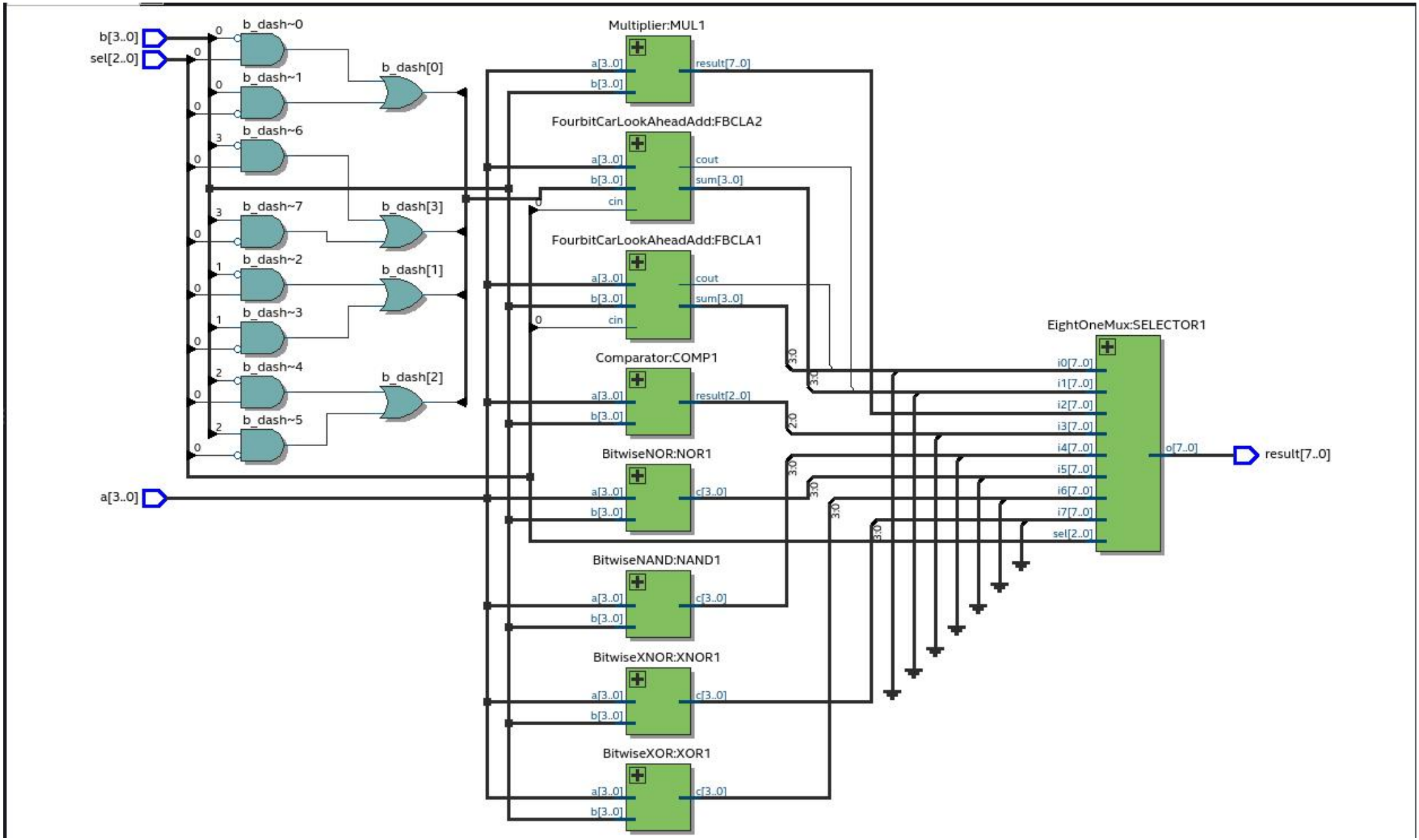
Four Bit ALU


b[3..0]
sel[2..0]
a[3..0]
b_dash~0
b_dash~1
b_dash~6
b_dash~7
b_dash~2
b_dash~3
b_dash~4
b_dash~5
b_dash[0]
b_dash[3]
b_dash[1]
b_dash[2]
Multiplier:MUL1
a[3..0]
b[3..0]
result[7..0]
FourbitCarLookAheadAdd:FBCLA2
cout
sum[3..0]
cin
FourbitCarLookAheadAdd:FBCLA1
Comparator:COMP1
result[2..0]
BitwiseNOR:NOR1
c[3..0]
BitwiseNAND:NAND1
BitwiseXNOR:XNOR1
BitwiseXOR:XOR1
EightOneMux:SELECTOR1
i0[7..0]
i1[7..0]
i2[7..0]
i3[7..0]
i4[7..0]
i5[7..0]
i6[7..0]
i7[7..0]
sel[2..0]
o[7..0]
result[7..0]

Four bit ALU

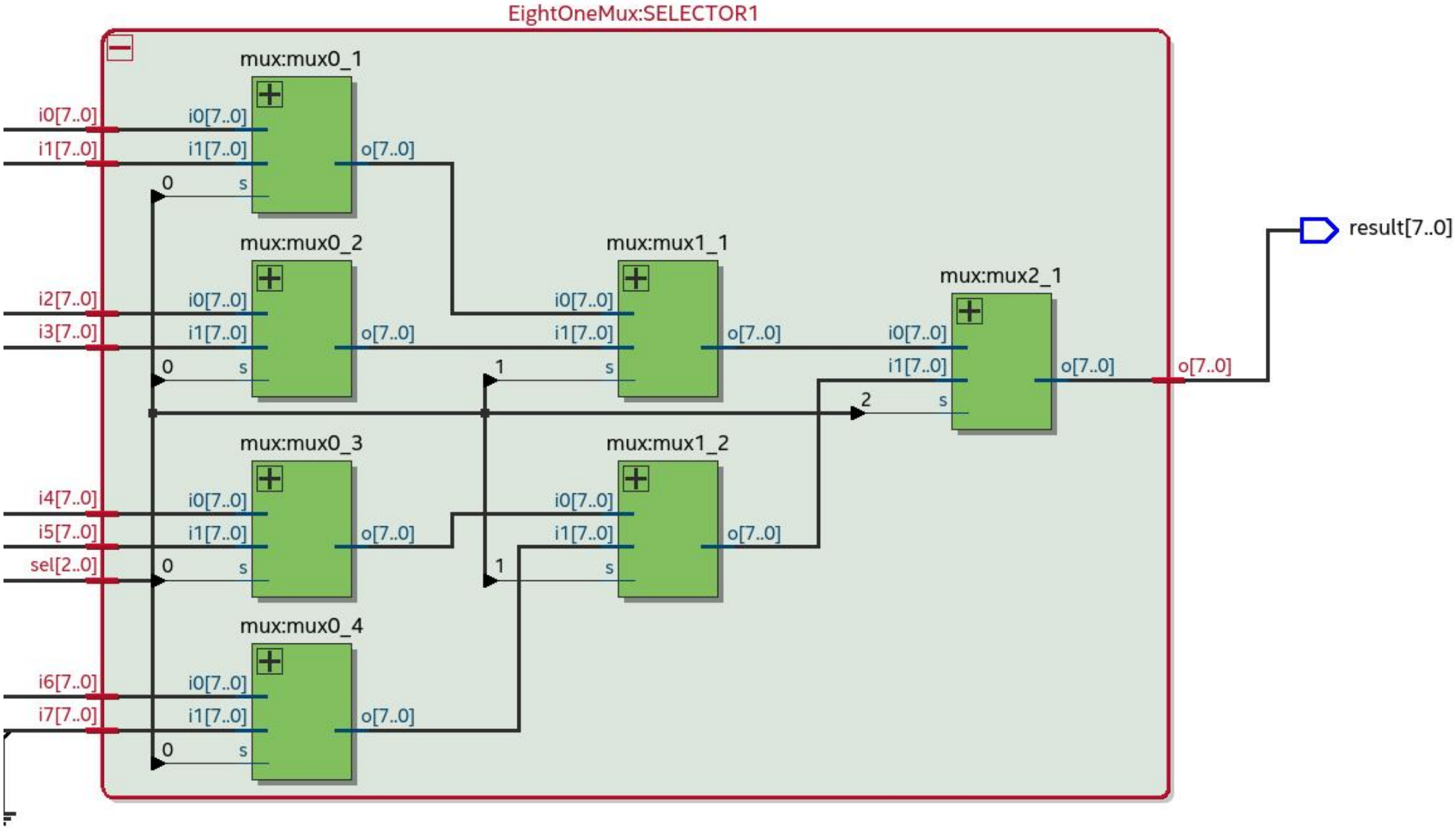
Selector

To create the selector, I planned to use a 8x1 MUX, which takes in a 3 bit input along with 8 different result 8bit - logic vectors and based on the 3 bit vector (selector), one of the 8 vectors is chosen correspondingly and given as output.

To make an 8x1 MUX, I used seven 2x1 MUXES connected such that the i^{th} result is selected when 'i' in binary is given as inputs. I also modified the MUXES to take in two 8 bit logic vectors as inputs and give out the selected result. (refer to mux.vhd)

The selectors(2x1) are connected such that at every step based on the values of 'sel', one of the input vectors the number of results will be filtered to half and after 3 rounds, the expected result from the expected arithmetic or logical unit is obtained.

(See below image for idea)


EightOneMux:SELECTOR1
mux:mux0_1
mux:mux0_2
mux:mux0_3
mux:mux0_4
mux:mux1_1
mux:mux1_2
mux:mux2_1
i0[7..0]
i1[7..0]
i2[7..0]
i3[7..0]
i4[7..0]
i5[7..0]
i6[7..0]
i7[7..0]
sel[2..0]
o[7..0]
s
0
1
2
result[7..0]

Fourbit Carry Look Ahead adder (000 : addition , 001 : subtraction)

Used logic and design discussed in class to implement the adder which does addition in $\log_2 n$ time. Made use of a partial full-adder component to find p_i, g_i and s_i's. finally found the carries and the cout.

To perform subtraction, we use the method of 2's complement by performing XOR between digits of 'b' and '1'. Then we add to get the difference 'a-b'. The answers are 5 bits long and the upper 3 bits are set to dont care.

(see figures below)

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i \cdot C_i$$

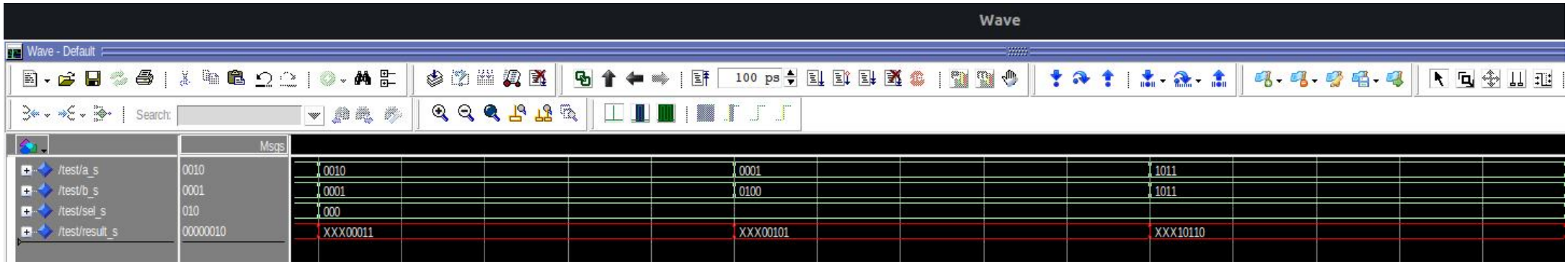
$$C_1 = G_0 + P_0 C_{in}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$

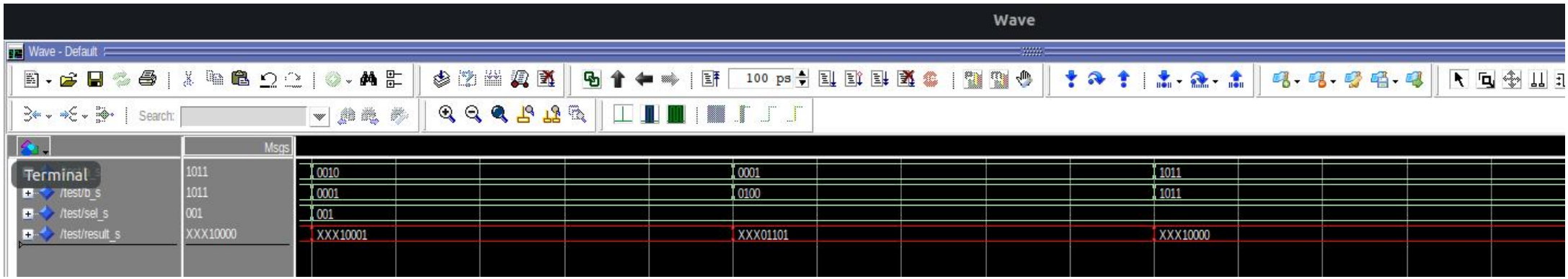
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$

$$C_{out} = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

Addition


Wave
Wave - Default
100 ps
Search:
Msgs
/test/a_s
/test/b_s
/test/sel_s
/test/result_s
0010
0001
010
00000010
0010
0001
000
XXX00011
0001
0100
XXX00101
1011
1011
XXX10110

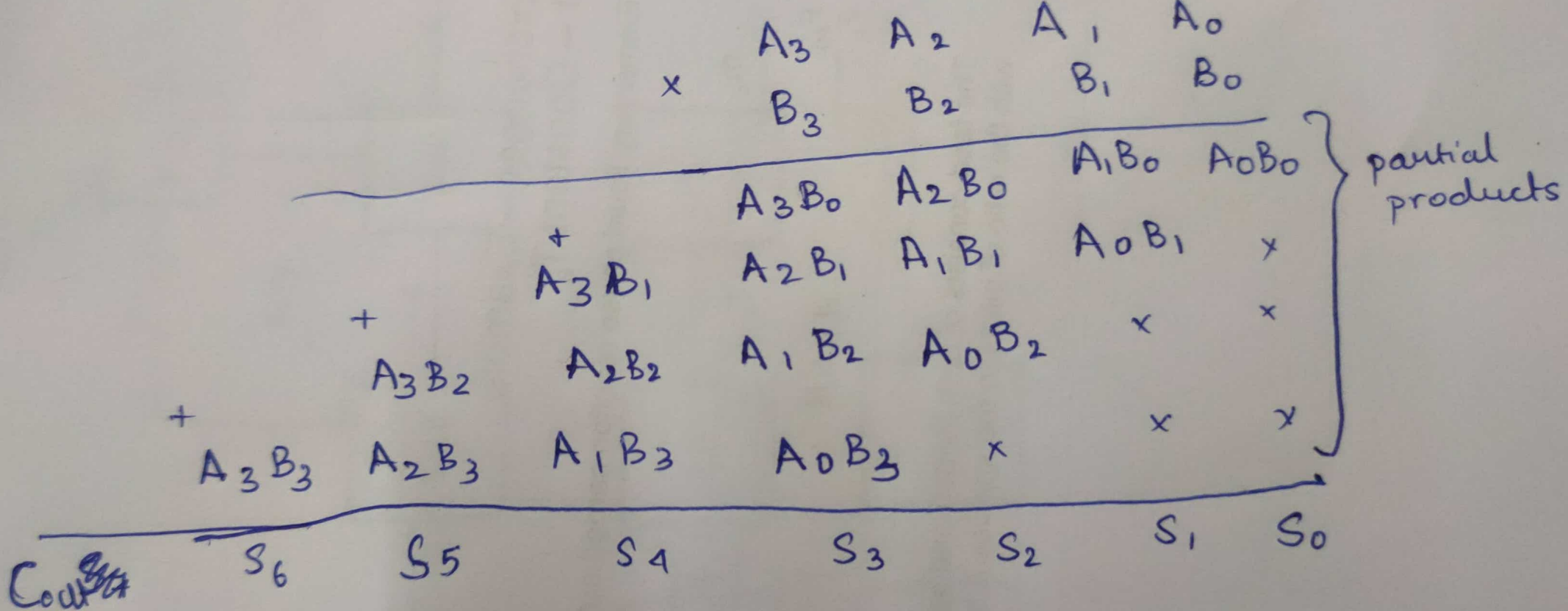
Subtraction



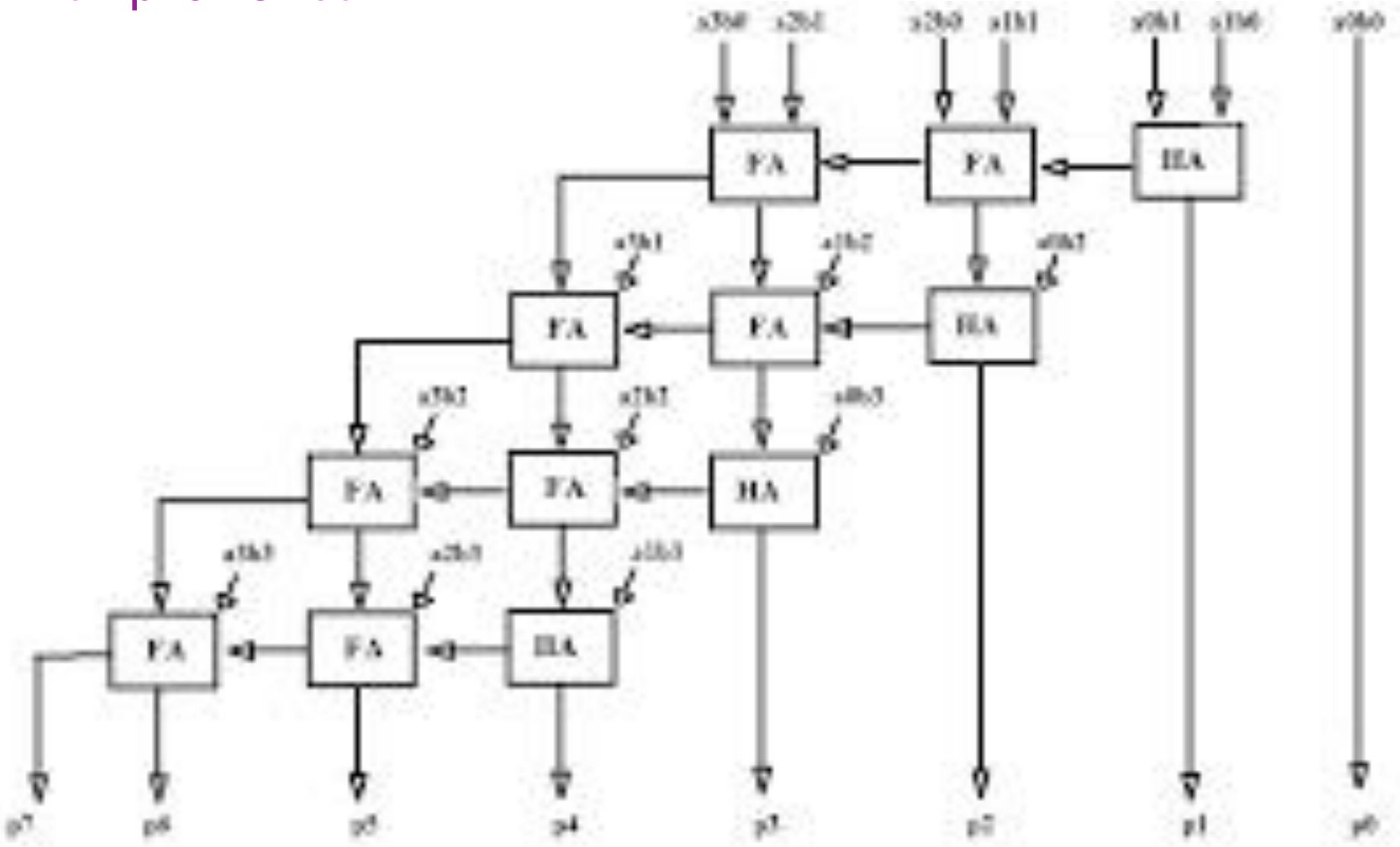
Multiplier (010)

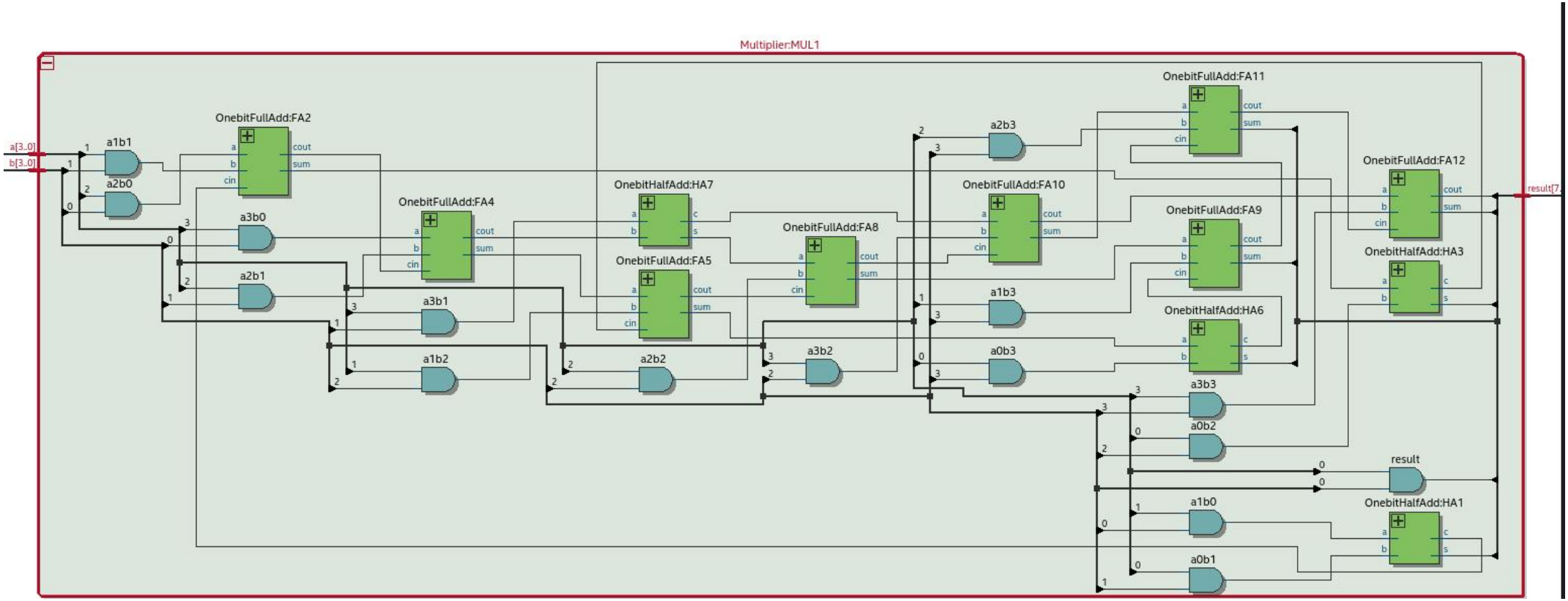
- Array multiplier circuit (shown below) is used. It mimics the multiplication process (see fig) in a direct way using and gates, half adders and full adders to obtain the result which is 8 bit std_logic_vector.

unsigned multiplication

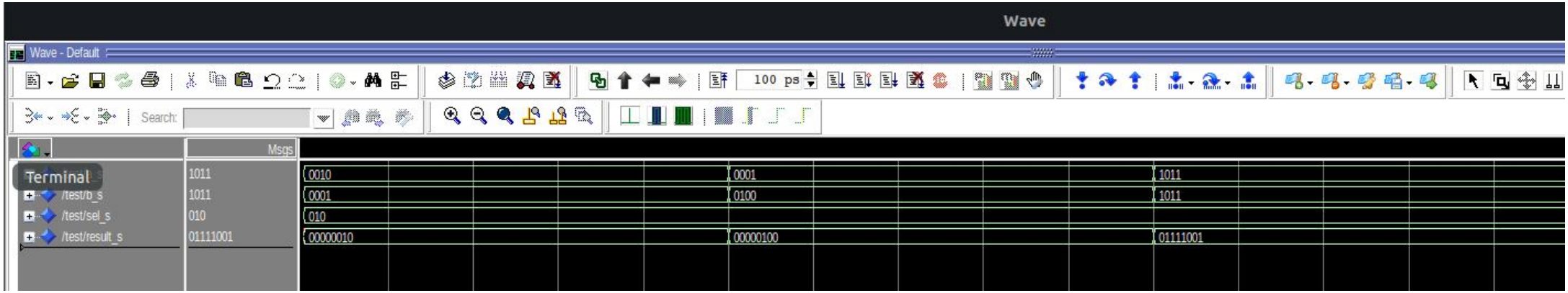


Array Multiplier Circuit




Multiplier:MUL1
a[3..0]
b[3..0]
a1b1
a2b0
a3b0
a2b1
a3b1
a1b2
a2b2
a3b2
a2b3
a1b3
a0b3
a3b3
a0b2
a1b0
a0b1
result
result[7..
OnebitFullAdd:FA2
OnebitFullAdd:FA4
OnebitHalfAdd:HA7
OnebitFullAdd:FA5
OnebitFullAdd:FA8
OnebitFullAdd:FA10
OnebitFullAdd:FA11
OnebitFullAdd:FA9
OnebitHalfAdd:HA6
OnebitFullAdd:FA12
OnebitHalfAdd:HA3
OnebitHalfAdd:HA1
a
b
cin
cout
sum
c
s

Multiplication



Comparator (011)

I used my own logic (atleast that what I think cause I got the idea) to implement this. I use a full-adder (4bit carry-lookahead) and obtain the difference 'a-b'. if the sign_bit of a-b is 1 then a-b ≥ 0. if it is 0 then a-b < 0 ⇒ a < b. if a-b is '10000' (sign bit | diff), then a = b. So first I check equality and get the middle bit of 'result', the 3 bit output vector.

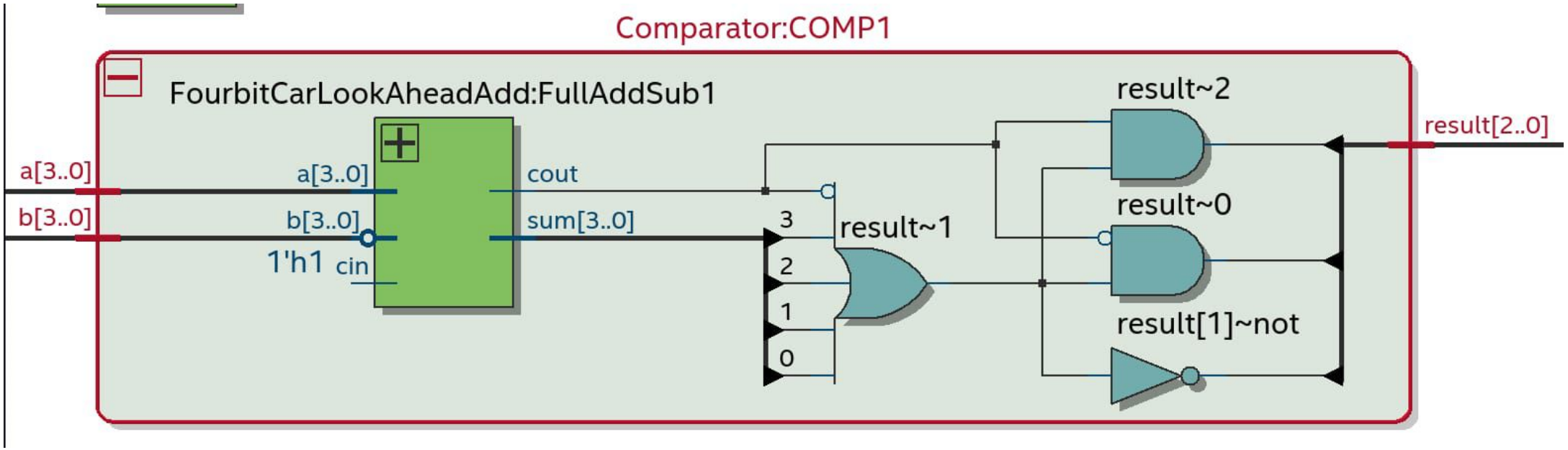
result(1) = !(!(signbit) ∨ (diff(0)) ∨ diff(1) ∨ diff(2) ∨ diff(3))

↑ will be 1 only if the above '10000' case present.

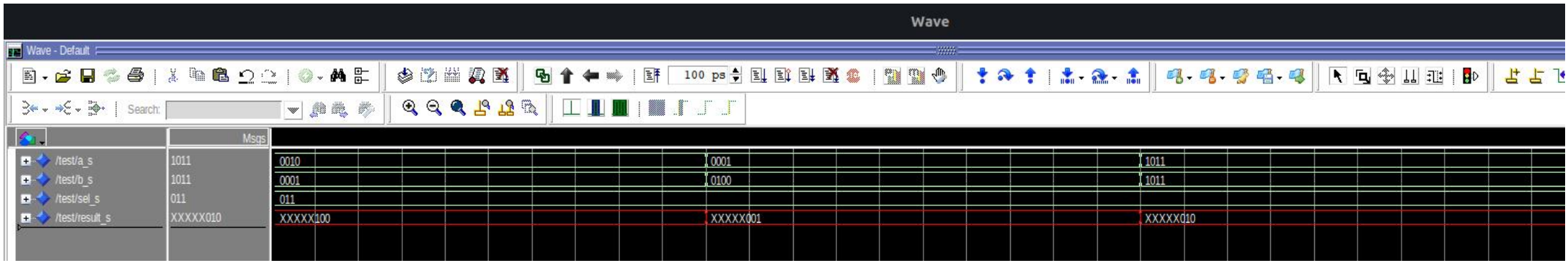
to get the other 2 bits,

result(0) = (! result(1)) ∧ (! sign_bit) ← will be 1 only when a < b

result(2) = (! result(1)) ∧ (sign_bit) ← will be 1 only when a > b


Comparator:COMP1
FourbitCarLookAheadAdd:FullAddSub1
a[3..0]
b[3..0]
1'h1
a[3..0]
b[3..0]
cin
cout
sum[3..0]
3
2
1
0
result~1
result~2
result~0
result[1]~not
result[2..0]

Comparison


Wave
Wave - Default
100 ps
Search:
Msgs
/test/a_s
/test/b_s
/test/sel_s
/test/result_s
1011
1011
011
XXXXX010
0010
0001
011
XXXXX100
0001
0100
XXXXX001
1011
1011
XXXXX010

Bitwise operators

Quite straightforward.

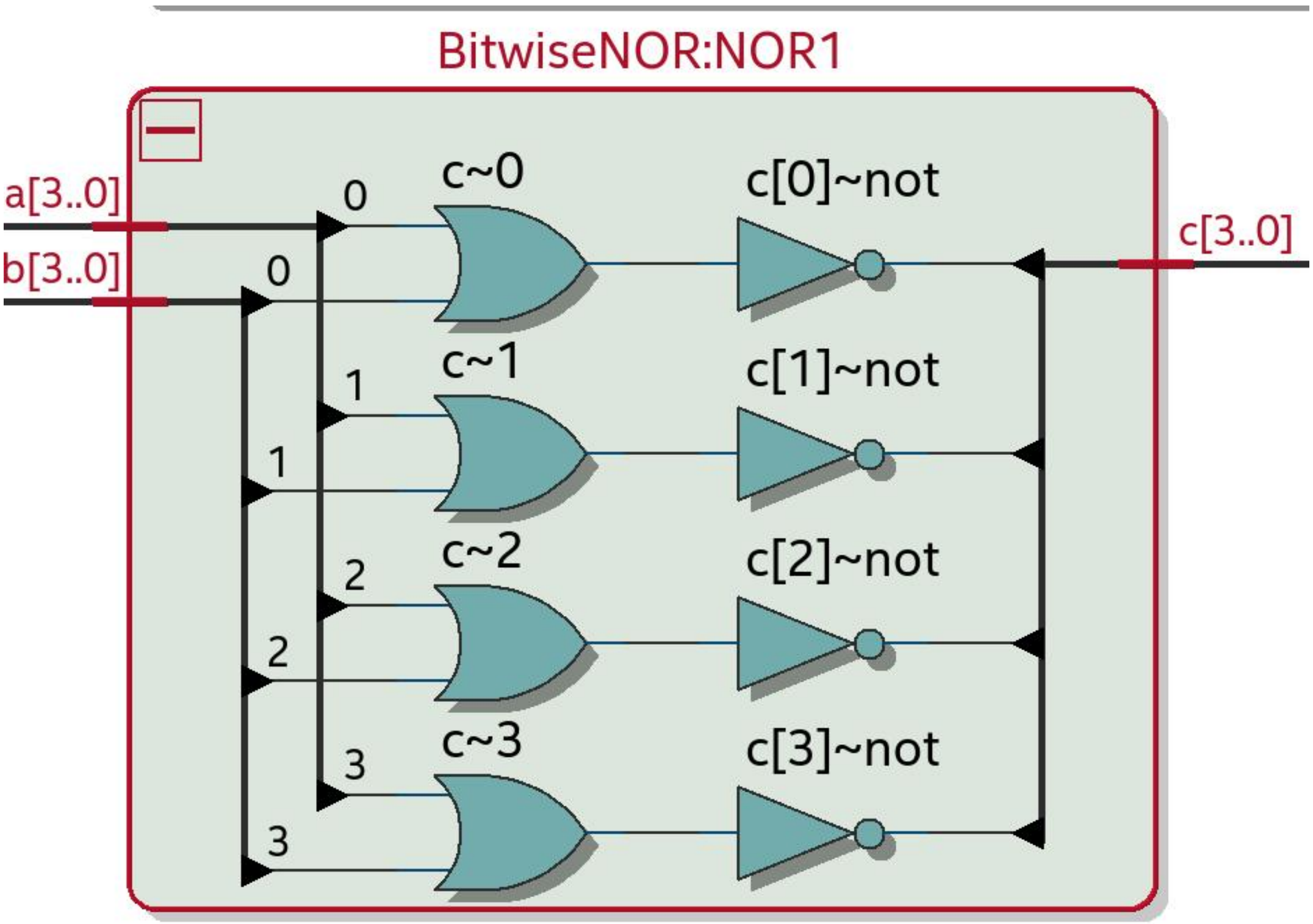
We apply the NAND, NOR, XOR, XNOR operators bitwise on corresponding bits of a and b to get resulting 'c'.

The definitions of the operators are in 'and', 'or' and 'not' gates.

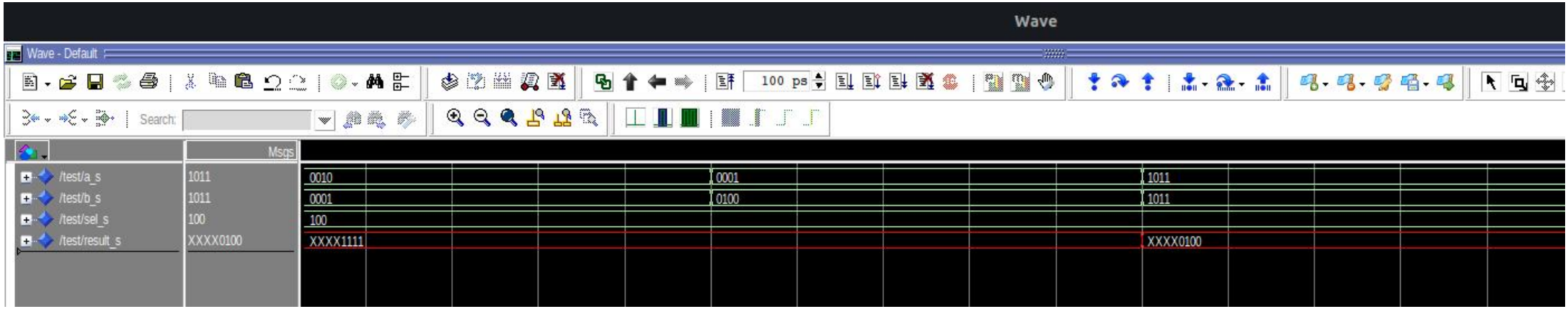
eg. XOR

c(0) ⇐ (a(0) and not (b(0))) or (not(a(0)) and b(0))

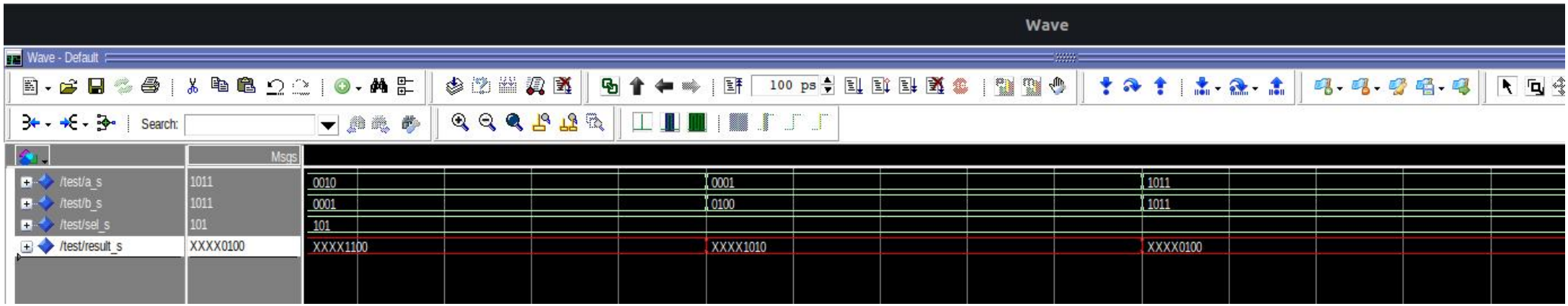
(see pic below and waveforms).


BitwiseNOR:NOR1
a[3..0]
b[3..0]
c[3..0]
c~0
c~1
c~2
c~3
c[0]~not
c[1]~not
c[2]~not
c[3]~not
0
0
1
1
2
2
3
3

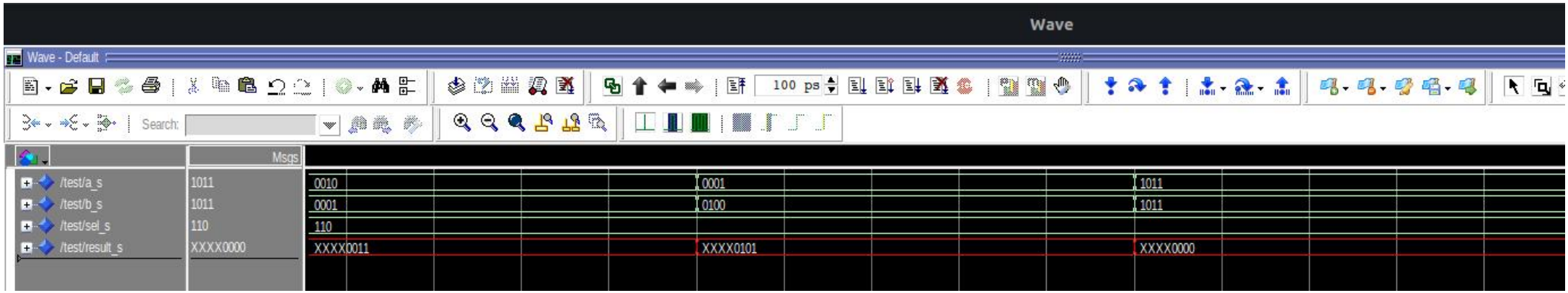
Bitwise NAND



Bitwise NOR


Wave
Wave - Default
100 ps
Search:
Msgs
/test/a_s
/test/b_s
/test/sel_s
/test/result_s
1011
1011
101
XXXX0100
0010
0001
101
XXXX1100
0001
0100
XXXX1010
1011
1011
XXXX0100

Bitwise XOR


Wave
Wave - Default
100 ps
Search:
Msgs
/test/a_s
/test/b_s
/test/sel_s
/test/result_s
1011
1011
110
XXXX0000
0010
0001
XXXX0011
0001
0100
XXXX0101
1011
1011
XXXX0000

Bitwise XNOR