Lec-6-HW-3-ALUarithmetic-SOLN

Reading, PP, Chp 2:
2.1 (Bits and datatypes)
2.2 (signed and unsigned integers)
2.3 (2's complement)
2.4 (positional notation)
2.5 (int. add/sub, signed/unsigned overflow)
2.6 (logic: AND, OR, NOT, XOR)
2.7 (bit vectors, floating point, ascii, hex)

Problems, PP, Chp 2:
2.2 (#bits for 26 char)
2.4 (unsigned range of n-bit)
2.5 (5-bit 2's comp., signed mag., and 1's comp.)
2.6 (6-bit 2's comp.)
2.8 (8-bit and n-bit 2's comp. ranges)
2.9 (bits per decimal digit in fp. format)
2.10ab (convert 2's comp. to dec.)
2.11ac (convert dec. to 2's comp.)
2.13 (convert k-bit 2's comp. to 8-bit)
2.15 (what op == shift right?)
2.17ab (i-bit + k-bit 2's comp.)
2.18ab (i-bit + k-bit unsigned)
2.20 (4-bit 2's comp. overflow)
2.34b (AND-OR-NOT)
2.36 (bit masking: BUSYNESS vector)
2.37 (alg. for detecting 4-bit 2's comp. overflow)
2.39a (dec. to fp format)
2.40ab (fp format to dec.)
2.44 (convert bin. to ascii)
2.45ab (convert bin. to hex.)
2.49ab (add in hex. notation)
2.50a (logic in hex. notation)
2.52 [only col. 1](32-bit hex. to unsigned, 1's-2's comp., fp, ascii)
2.56 (define 8-bit fp format)

PP, Chp 2 [Notation: x^y means x raised to the power y.]

2.2 (#bits for 26 char? for upper and lower case?)
26 chars is less than 32 = 2^5, 5 bits is enough.
52 chars is less than 64 = 2^6, 6 bits is enough.

2.4 (unsigned range of n-bit)
0 - (2^n - 1), that is, 2^n numbers.

2.5 (7 and -7 in 5-bit 2's comp., signed mag., and 1's comp.)

```
 7       -7
-----   -----
00111   11001   2's complement
00111   10111   signed mag.
00111   11000   1's complement
```

2.6 (6-bit 2's comp. -32)

```
011111 2sCompRepresentaion( +31 )
100000 flip bits
    +1 add one
100001 gives 2sCompRepresentaion( -31 )
    -1 subtract 1 from -31 should give us
100000 2sCompRepresentaion( -32 )
```

2.8 (8-bit and n-bit 2's comp. ranges)

(a) max pos. 8-bit: 01111111 = 127 = (2^7 - 1)
(c) max pos. n-bit: (2^(n-1) - 1)
(b) max mag. 8-bit: 10000000 = -128 = -(2^7)
(d) max mag. n-bit: -2^(n-1)

2.9 (bits for 2's comp rep. of 6.02 X 10^23)

[This asks for the 2's complement representation, not floating point.]
6.02 X 10^23 = 602 X 10^21
gives 602 followed by 21 zeros: 24 dec. digits

999 ~= 1024 = 2^10, about 10 bits per 3 dec. digits
(24 dec. digits) X (10/3 bits per dec. digit) = 80 bits
but in 2's comp. we need sign, which gives us about 81 bits.

To be a little more exact, 80 bits magnitude gives us,
2^80 = (2^10)^8 = 1024^8
= (1.024*1000)^8 = (1.024)^8 X (1000)^8 = 1.024^8 X (10^3)^8
= 1.024^8 X 10^24 which is about
~= 1.2089 X 10^24 = 12.089 X 10^23

We don't need quite that large a number, if we use one fewer bit, we
divide by 2, which is about
~= 6.04 X 10**23
So, we can get by with 79 bits for magnitude + 1 sign bit = 80 bits.

2.10ab (convert 2's comp. to dec.)
(a) 1010 is negative; so take 2sComp to find magnitude: 0101+1 = 0110 = 6; so 1010 is -6.
(b) 01011010 = 64 + 16 + 8 + 2 = 90

2.11ac (convert dec. to 2's comp.)
(a) 102 = 64 + 32 + 4 + 2 gives us 01100110 (base 2)
(b) 33 = 32 + 1 gives us 00100001 (base 2)

2.13 (sign extension, 2's complement)
(a) -6, sign extend leftward 4 bits: 1111 1010
(b) 16+8+1 = 25, sign extend left: 0001 1001
(c) -8, chop sign bits from left: 1111 1000
(d) 1, sign extend left: 0000 0001

2.15 (what op == shift right?)

0110 shifted right gives 0011, which is 3; so, 6 right shifted once is 3, which is 6/2.
0111 shifted right gives 0011, is also 3; so 7 right shifted once is 3, which is 7/2 without remainder.
This is integer division by 2 with truncation of remainder.

2.17
(a) 0001 + 1011 = 1100 or 1 + -5 = -4
(b) 11111111 + 01010101 = 01010100 or -1 + (64+16+4+1) = 84

2.18ab (i-bit + k-bit unsigned)
(a) 1011 + 01 = 1100; which is 8+2+1 + 1 = 12
(b)

```
01010101
+     11
---------
01011000 which is 64+16+8 = 88
```

2.20 (4-bit 2's comp. overflow)

1100 + 0011 = 1111
which is -4 + 3 = -1
No overflow: opposite signed values cannot overflow addition.

1100 + 0100 = [1]0000
which is -4 + 4 = 0
No overflow: opposite signed values cannot overflow addition, carries ignored.

0111 + 0001 = 1000
which is 7 + 1 = -8
Overflow: addition of two positive values gives a negative result.

1000 + 1111 = [1]0111
which is -8 + -1 = 7
Overflow: addition of two negative values gives a positive result.

0111 + 1001 = [1]0000
which is 7 + -7 = 0
No overflow: opposite signed values cannot overflow addition, carries ignored.

2.34b (bitwise logic)
NOT(1000 AND (1100 OR 0101))

1100 + 0101 == 1101

1000 * 1101 == 1000

-1000 == 0111

2.36 (bit masking: BUSYNESS vector)
(a) AND 11111011
(b) OR 01000100
(c) AND 00000000
(d) OR 11111111
(e) AND 00000100 (to get bit); then ADD five times to shift left.

2.37 (alg. for detecting 4-bit 2's comp. overflow)
The job here is to write a series of ALU operations that can detect whether an ALU operation overflowed. All you have is the result of the operation, s, and the two operands, m and n. You must use the result to find if there was overflow. There is overflow if m and n have the same sign but the result, s, has a different sign. A boolean (AND-OR-NOT) expression for this would be:

OVERFLOW = (-S * (M * N)) + (S * (-M * -N))

where, "*" = AND, "+" = OR, and "-" = NOT. Below, take these to be bitwise ALU logical instructions.
First, we mask to get the sign bits of the three quantities:

```
M <== m * 1000
N <==  n * 1000
S <==  s * 1000
```

Assume we have three 4-bit variables, L and R and V.

L gets the value of left term, (-S * (M * N)):

```
L <== -S
L <==  M * L
L <==  N * L
```

R gets the value of the right term, (S * (-M * -N)):

```
S <==   S
M <== -M
N <==  -N
R <==   M * N
R <==   S * R
```

Finally, V gets the value of the entire expression:

```
V <== L + R
```

If the high-bit of V is 1, there was overflow.

2.39 (convert to IEEE 32-bit float)
(a)
3.75 = 3 + (1/2 + 1/4)
= 11.11 (in binary)
= 1.111 X 2^+1 (normalized)

which gives us,

1-bit sign (+):
0
8-bit exponent: +1 + 127 = 128 =
10000000
23-bit fractional part:
111 0000 0000 0000 0000 0000

complete 32-bit representation:
0 10000000 111 0000 0000 0000 0000 0000 0000

2.40ab (fp format to dec.)

```
  (a)
sign : exponent : fractional part
0 : 10000000 : 0000000000000000000
+ :   128    :    0
+ : 128-127  :    0
+ :    1     :    0
which is  + 1.0 X 2^1 = 2

  (b)
sign : exponent : fractional part
1 : 10000011 : 0001000000000000000
- :   128+3  : .0001
- : 128+3-127: .0001
- :    4     : .0001
which is  - 1.0001 X 2^4 = -(1 + 1/16)*16 = -(16+1) = -17
```

2.44 (convert bin. to ascii)

00000100, value is x04, add x30 = x34, is code for ascii digit 4 (ascii 0 is coded x30 and code for successive digits increases by 1.)

2.45ab (convert bin. to hex.)

```
 (a)
1101 0001 1010 1111
13     1   10   15
D      1    A    F

 (b)
0001 1111
  1   15
  1   F
```

2.49ab (add in hex. notation)

```
 (a)
0    2    5    B
2    6    D    E
```

converting to decimal for each digit,

```
   0    2     5     11
+ 2    6    13     14
--------------------------
   2    8   16+2  16+9
```

Now, convert back to hex w/ carries:

```
2   8+1   2+1   9  hex w/ hex carries,
2    9       3    9   hex w/o carries.
```

```
 (b)
7  D  9  6
F  0  A  0
```

becomes, after conversion to decimal,

```
7  13  9  6
15  0 10  0
+ ------------------
16+6  13  16+3  6
1   6   13+1  3   6 w/ hex carries
1   6    E    3   6 in hex
```

2.50a (logic in hex. notation)

```
5    4    7    8
F    D    E    A
AND    which gives

0101 0100 0111 1000
1111 1101 1110 1010
AND == ------------------------
0101 0100 0110 1000
  5    4    6    8  in hex
```

2.52 [do only col. 1] (32-bit hex. to unsigned, 1's-2's comp., fp, ascii)

```
4      3    4    F    4    D    5    0
0100 0011 0100 1111 0100 1101 0101 0000
```

which gives as a summation:

```
   0100 0000 0000 0000 0000 0000 0000 0000 = 2^30 = 1,073,741,824
 + 0000 0010 0000 0000 0000 0000 0000 0000 = 2^25 = 33,554,432
 + 0000 0001 0000 0000 0000 0000 0000 0000 = 2^24 = 16,777,216
 + 0000 0000 0100 0000 0000 0000 0000 0000 = 2^22 = 4,194,304
 + 0000 0000 0000 1000 0000 0000 0000 0000 = 2^19 = 524,288
 + 0000 0000 0000 0100 0000 0000 0000 0000 = 2^18 = 262,144
 + 0000 0000 0000 0010 0000 0000 0000 0000 = 2^17 = 131,072
 + 0000 0000 0000 0001 0000 0000 0000 0000 = 2^16 = 65,536
 + 0000 0000 0000 0000 0100 0000 0000 0000 = 2^14 = 16,384
 + 0000 0000 0000 0000 0000 1000 0000 0000 = 2^11 = 2048
 + 0000 0000 0000 0000 0000 0100 0000 0000 = 2^10 = 1024
 + 0000 0000 0000 0000 0000 0001 0000 0000 = 2^8 = 256
 + 0000 0000 0000 0000 0000 0000 0100 0000 = 2^6 = 64
 + 0000 0000 0000 0000 0000 0000 0001 0000 = 2^4 = 16
 = 1,129,270,608 (unsigned)
 = 1,129,270,608 (1's comp.)
 = 1,129,270,608 (2's comp.)

 as floating point,
0 (10000110)             (100 1111 0100 1101 0101 0000)
0 (10000110 - 01111111)  (1.5197136)
0 (00000111)             (1.5197136)
  which is,
+2^7 X 1.5197136 = +194.5233408

  = COMP (as ascii, but it might be PMOC.)
```

2.56 (convert from hex to 8-bit float)

```
  E       5
1110 0101
1 1100 101

sign: -1
exponent: 1100 - 7 = (8+4) - 7  =  5
fraction: 1.101
```

the complete value is, using left-shift is equivalent to multiply-by-2:

```
  -(1.101) (2^5) = -110100 =   -(32+16+4)  =  -52
```

In your LC3, implement the ALU, muxes, and the "plus1" circuit.

(1.) Implement muxes using basic gates (AND, NOT, OR). There are several muxes in the design, but we will only concern ourselves with the PCMUX, and the muxes in the RegFile and in the ALU.

The PCMUX is a 16-bit, 4-input, 1-output mux: inputs and output are 16-bits each; there is a 2-bit select input. It is mux16_4x1 in parts.jelib.

RegFile has two copies of mux16_8x1, one for the SR1out, the other for the SR2out. It is implemented using two mux16_4x1s and a mux16_2x1.

The ALU has a single mux16_4x1.

There are only two parts to modify: mux16_2x1 and mux16_4x1 in parts.jelib. You are to remove the verilog code and blackboxes from these cells and build each part using basic gates. (See notes below for building arrays of elements in Electric.)

(2.) Implement function units in the ALU and the plus1 using basic gates. The ALU contains three 16-bit functional subparts: AND16, ADD16, and NOT16. AND and NOT bitwise operations are very simple.

For ADD16, you will implement a 16-bit full adder. To do that, create a 1-bit full adder, FA1, from basic gates. Then use that as a base element to implement ADD16.

The plus1 part can simply be an ADD16 with appropriate inputs; eg., one 16-bit input will always be 0 and the low-bit carry-in will always be 1.

(3) EXTRA-CREDIT: plus1 can be made using many fewer gates than using ADD16 to implement it. Analyze its boolean logic and implement plus1 directly. Use logic circuit minimization techniques we have learned.

NOTES

(1) Arrays of gates/devices. It is easy to create a 16-bit circuit from 1-bit elements using Electric's Edit.Array command. First, build a 1-bit element; eg., a 1-bit AND with short wires attached to inputs and outputs. Name the wires as bus elements; eg., inA[0], inB[0], and out[0]. Now, select the entire 1-bit device by dragging a select-frame over it. Then, use Edit.Array, and fill in the appropriate number of repetitions of the device (16 in this case). Decide if you want them laid out vertically or horizontally. Recall that Edit.undo is ctl-z in MS Windows and Apple-z in Mac. The wires will be automagically numbered Ain[0], Ain[1], and so forth. All that's left to do is check that the cell's bus ports are still working. If you were careful not to delete the ports and you used the same naming scheme for wires/busses used originally, they should be ok. (But, when you have the same signal going to all instances in your array, as in MUX selects, you will have to manually rename them since they will be numbered as if they form a bus.) If you have to create an entire cell from scratch, remember to set all port's input/output characteristics appropriately, and do View.MakeIconView. You can edit the icon view, "{ic}", to move text around and resize it, add text to inputs and outputs, and move the location of ports. To move ports you may have to Edit.Rotate them and their attached wire/bus (select the pin, not the wire/bus for moving; select both for rotation.)

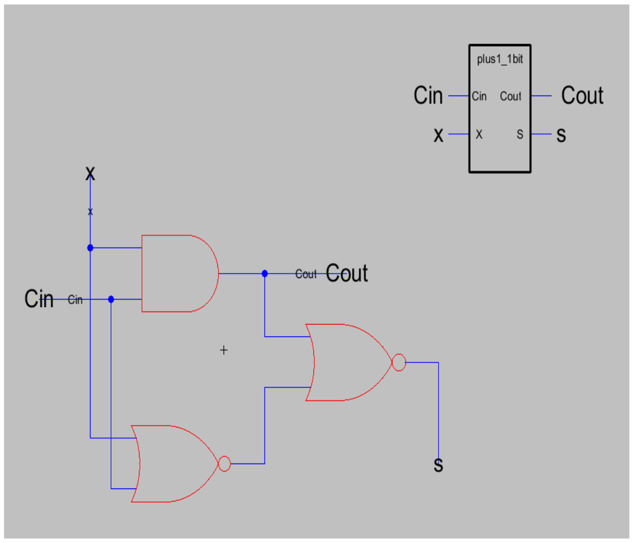
(2) WARNING: In editing pre-existing cells, be careful not to delete the exports when removing the old implementation. When you edit or delete cells, either schematic or icon cells, you may find your connections to its icon in higher-level cells break, or your exports disappear or get automagically renamed in the schematic. Visually check connections: (1) in your schematic cell, select the export name or its icon's export crosshair, all connected wire/bus nets will be highlighted; (2) in a higher-level cell, select the icon's export crosshairs. In the worst case, you may need to delete the old icon from the cells in which there are instances (including its own schematic cell), make a new icon view, and put the new icon in where the old one was and reconnect it. If you are only editing an existing schematic, you can delete the icon in the cell's schematic without deleting its icon view "{ic}" and then recreate exports that match the cell's icon view's exports.

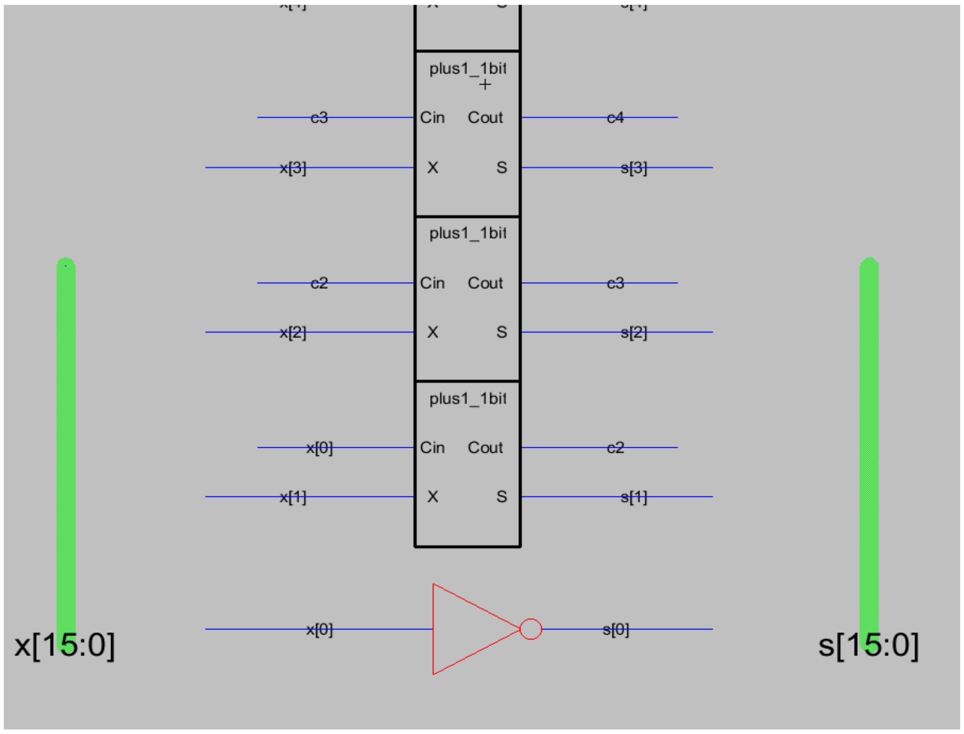
EXTRA CREDIT. After fiddling with 4-bit examples, using Karnaugh maps, and trying various rearrangements of logic, here's what I came up with (S0 is sum bit 0, X0 is input bit 0, Xi is the i-th input bit, Ci is the i-th carry bit, and so forth),

```
   S0    =  -X0
   C1    =   X0
   Si    =  NOR( Xi * Ci, NOR( Xi, Ci ) )
   Ci+1  =  Xi * Ci
```

where "-" means NOT, and "*" means AND. Note the common term (Xi * Ci). Per input bit, this circuit uses 2 NORs and 1 AND, except bit 0 uses only NOT.

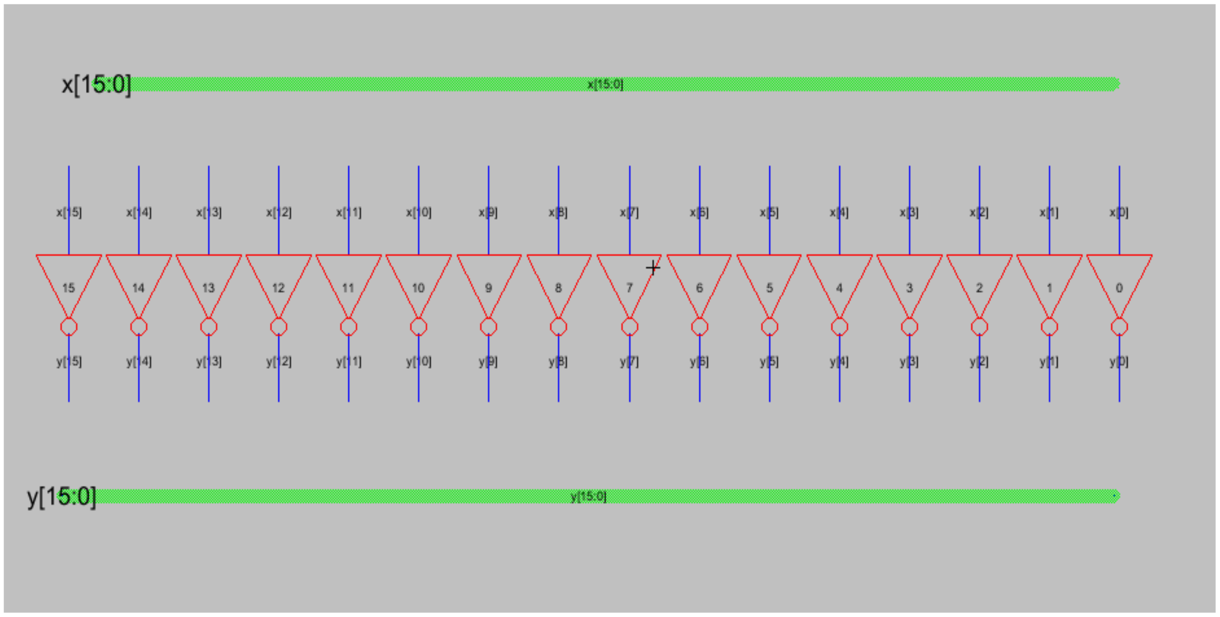
Here's plus1's base component, plus1_1bit.

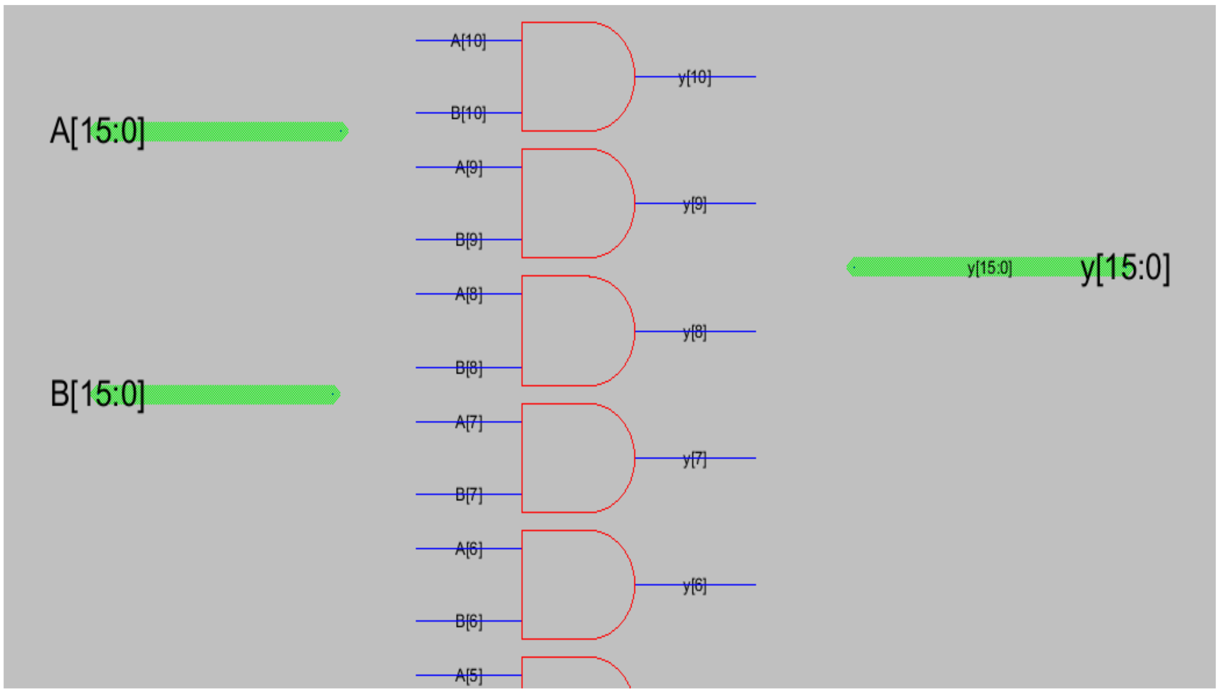




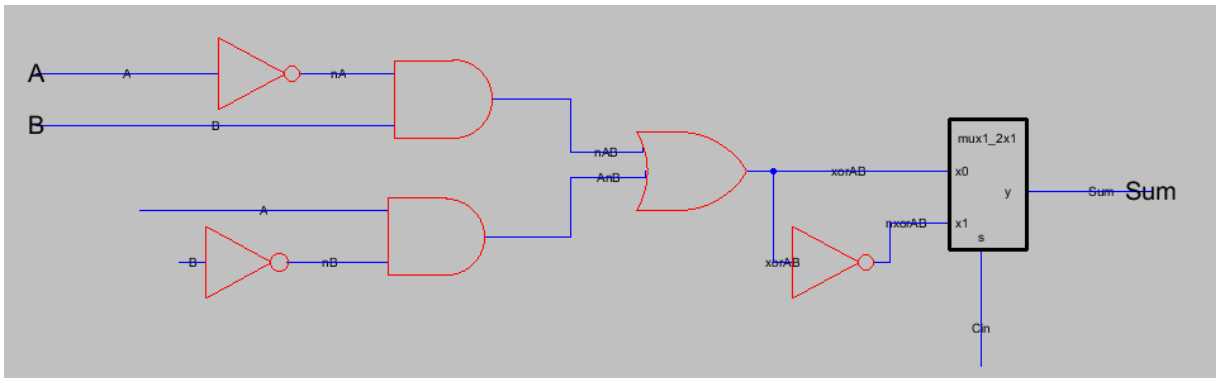
Here's plus1, showing just part of the array of plus1_1bit devices. Note the carries are chained to ripple up. The array continues up to bit 15. c16 is ignored. "x" is the input export, "s" is the output export.

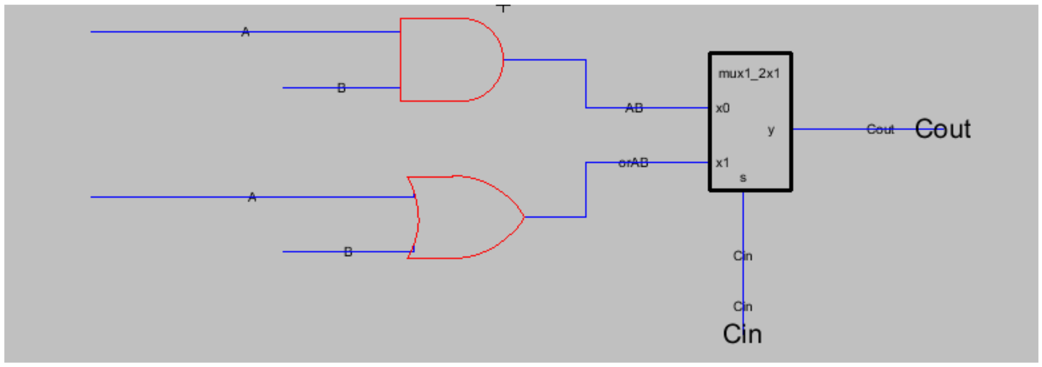
Here's NOT16
and part of
AND16


x[15:0]
y[15:0]
x[15] x[14] x[13] x[12] x[11] x[10] x[9] x[8] x[7] x[6] x[5] x[4] x[3] x[2] x[1] x[0]
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
y[15] y[14] y[13] y[12] y[11] y[10] y[9] y[8] y[7] y[6] y[5] y[4] y[3] y[2] y[1] y[0]

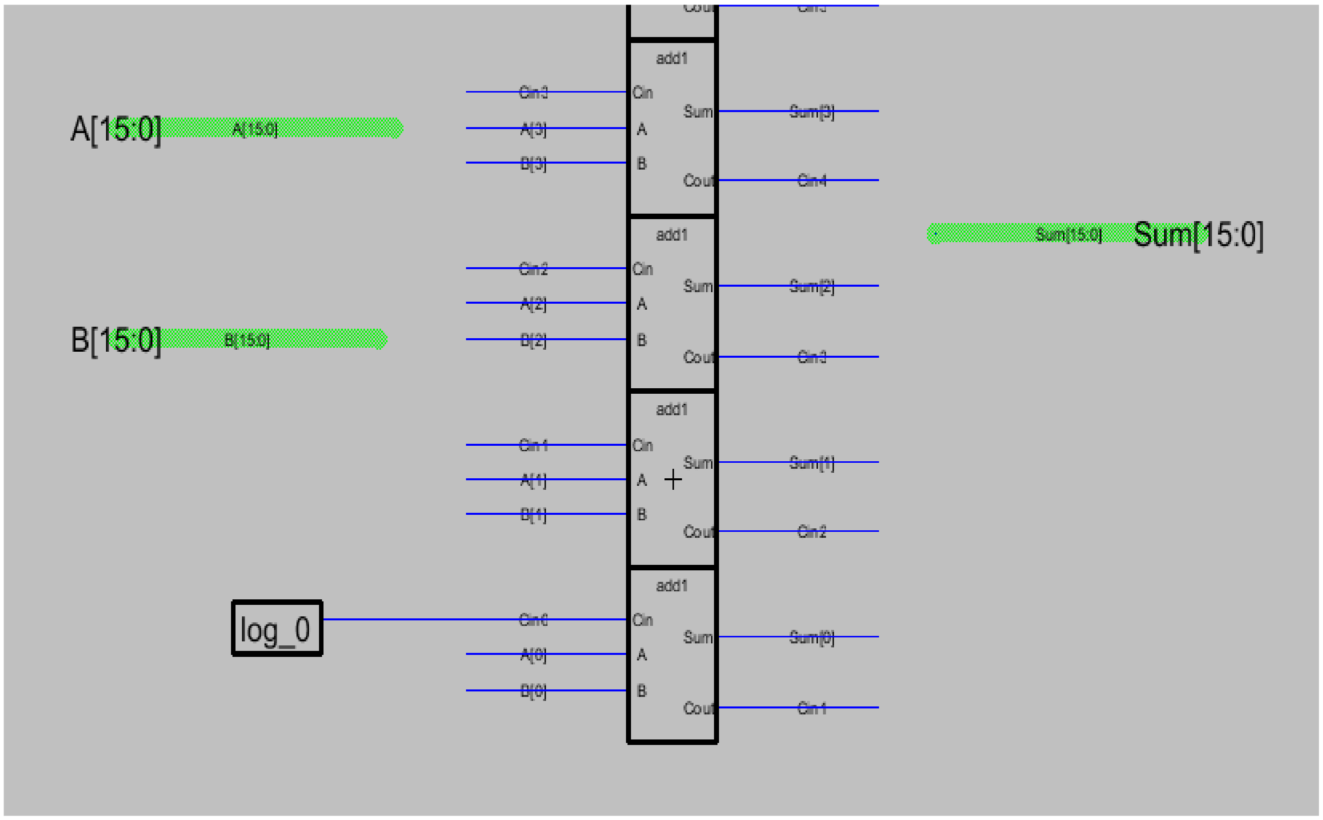


Here's ADD1, a 1-bit full adder. I did it the lazy way by using a 2X1 mux. Note that I had to name all the wires to get the connectivity working correctly.

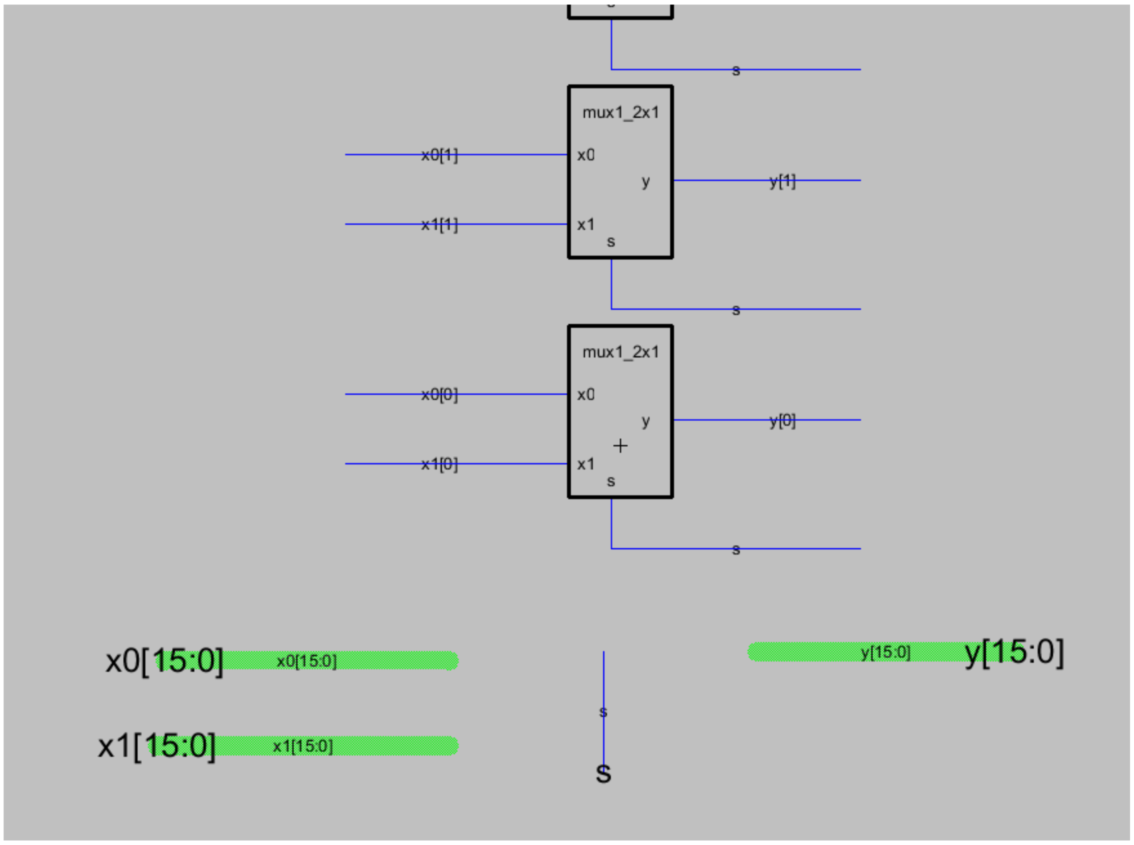




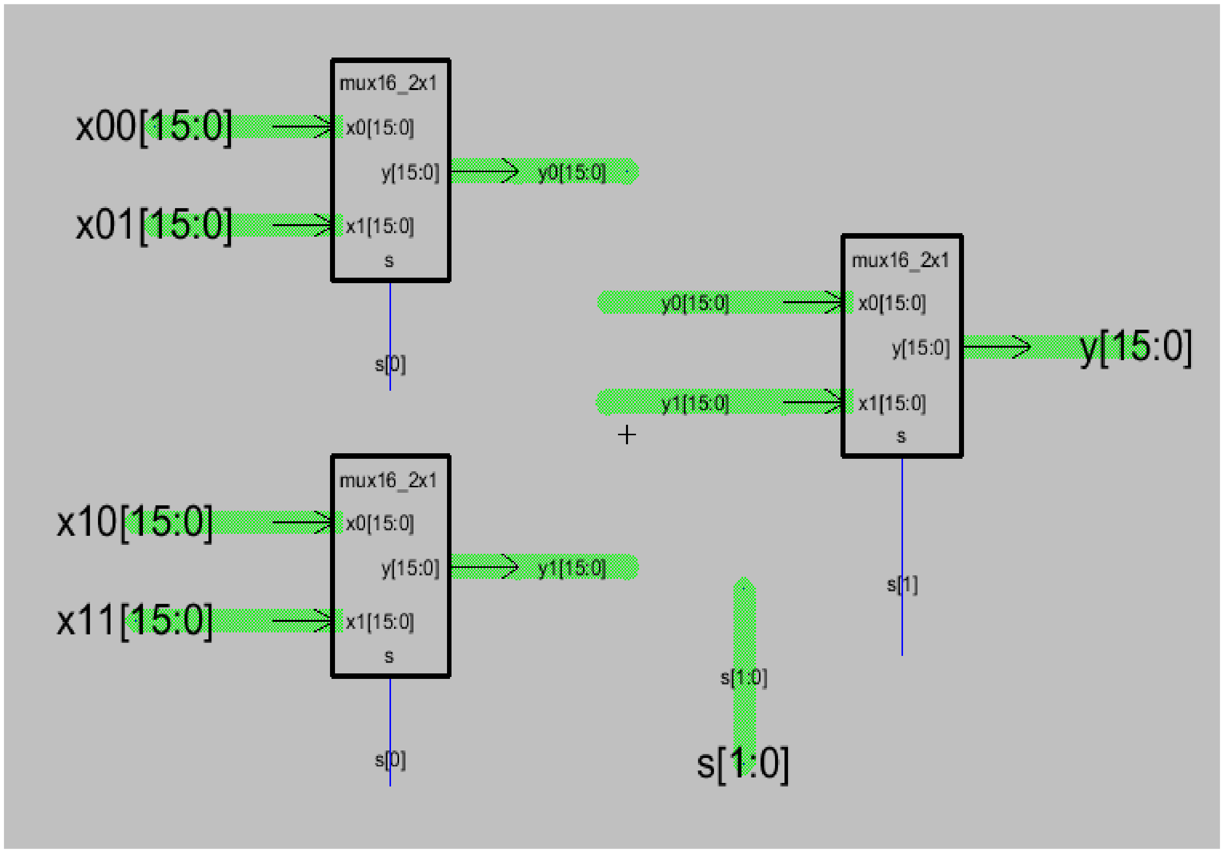
Here's a portion of ADD16, a 16-bit full adder. The LC3 doesn't have a subtract; so, the Cin is set to 0. Also, the carry out, C16, is ignored as LC3 doesn't have an ALU output flag for that. Again, the carries had to be hand edited to chain them together. That's why they don't have bus names.



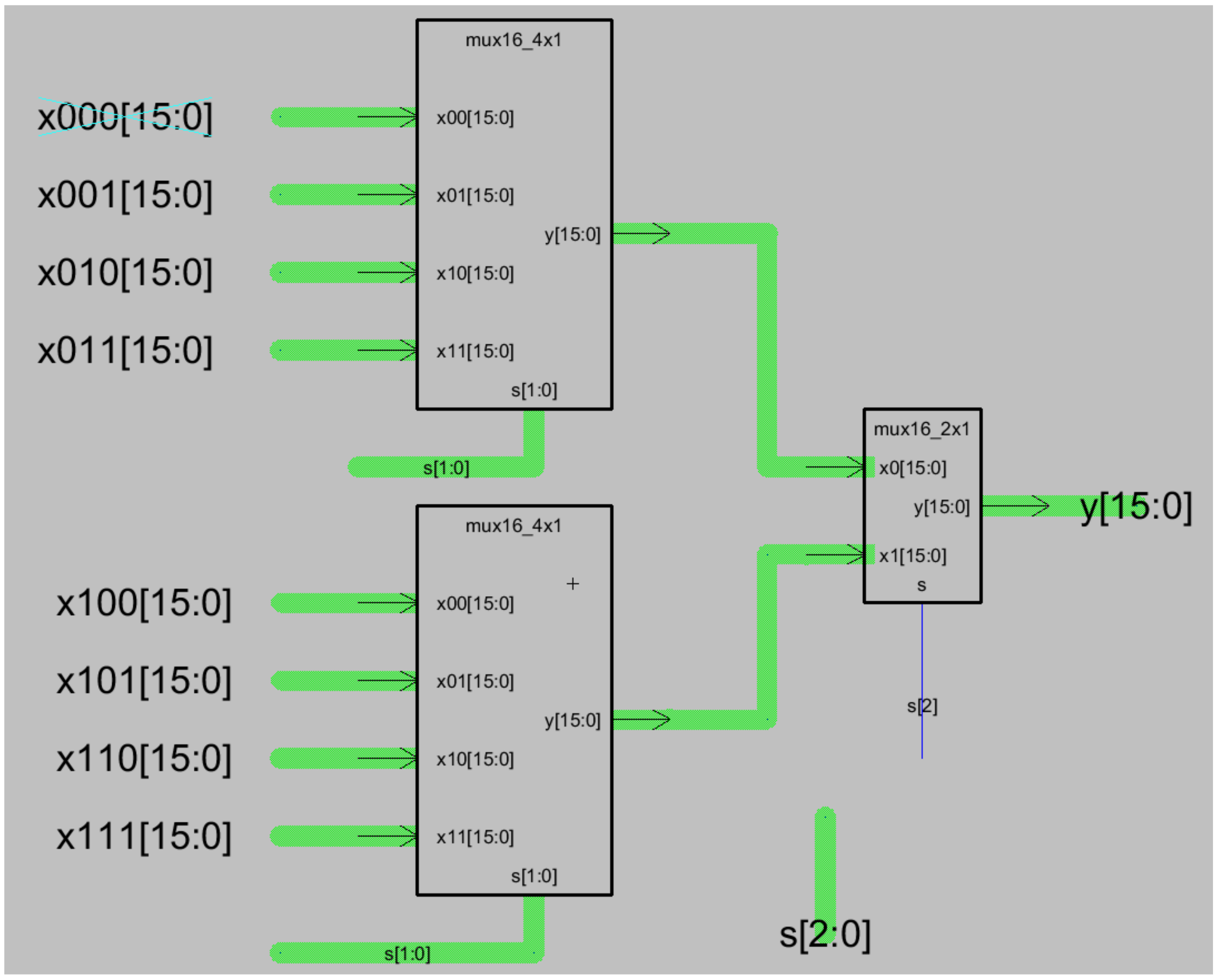
Here's a portion of mux16_2X1. It uses mux1_2X1 as a base part. Note the selects had to be manually renamed to "s", after making the array.



Below is mux16_4X1. It uses three mux16_ 2X1s.



Here's the mux16_8X1. It uses one mux16_2X1 and two mux16_4X1s.


mux16_4x1
x000[15:0]
x001[15:0]
x010[15:0]
x011[15:0]
x00[15:0]
x01[15:0]
x10[15:0]
x11[15:0]
y[15:0]
s[1:0]
s[1:0]
mux16_4x1
x100[15:0]
x101[15:0]
x110[15:0]
x111[15:0]
x00[15:0]
x01[15:0]
x10[15:0]
x11[15:0]
y[15:0]
s[1:0]
s[1:0]
mux16_2x1
x0[15:0]
y[15:0]
x1[15:0]
s
s[2]
y[15:0]
s[2:0]