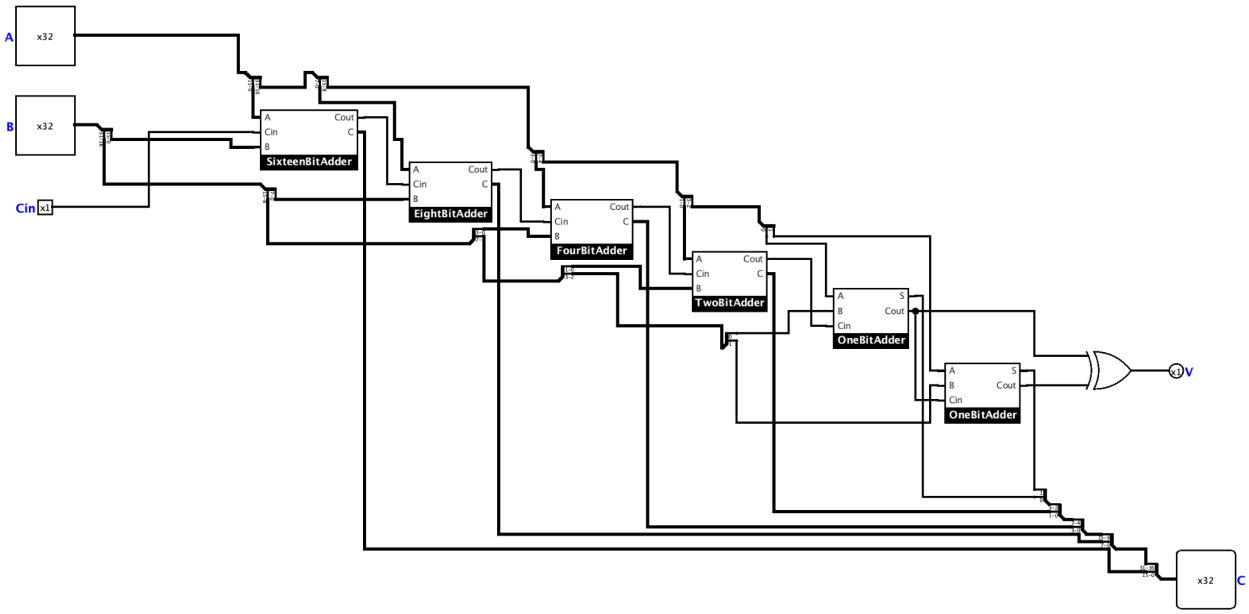
Michael Torku
CS3410
Project 1

DESIGN DOC

ADD32:
Is the circuit which I used to implement arithmetic addition of input A[32] and B[32]. It is composed of an abstraction of 1-bit adders into sub-circuits of 2-bit adders, 4-bit adders, 8-bit adders and a sixteen bit adder. I used an XOR gate to obtain the output V (which is representative of overflow) because when there's a difference between Cout and Cin of the MSB, it means there's overflow.



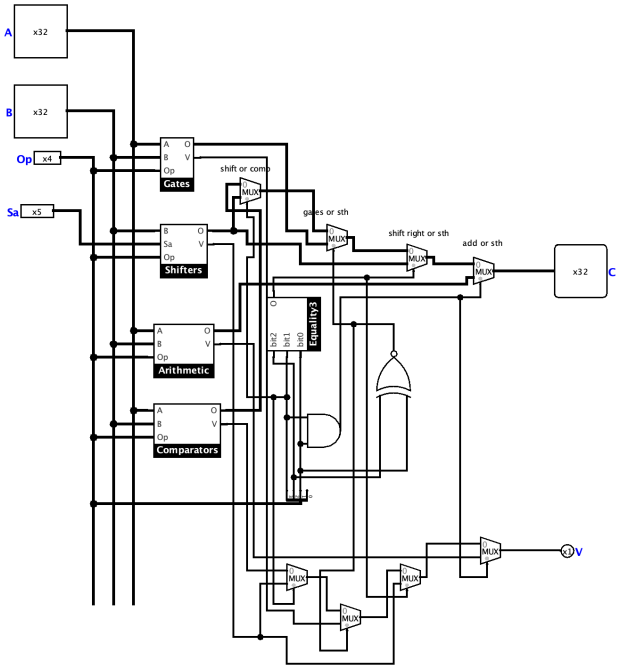
ALU: The way my ALU works (at a higher level) is that it recognizes the op code input and uses muxes to decide the requested output.

The first selection criterion is the first and third input: if they are both 1s then the op code requires arithmetic (addition or subtraction) hence the mux named (add or sth - sth represents any operations other than add). The AND gate which connects to said mux checks the validity of this criterion.

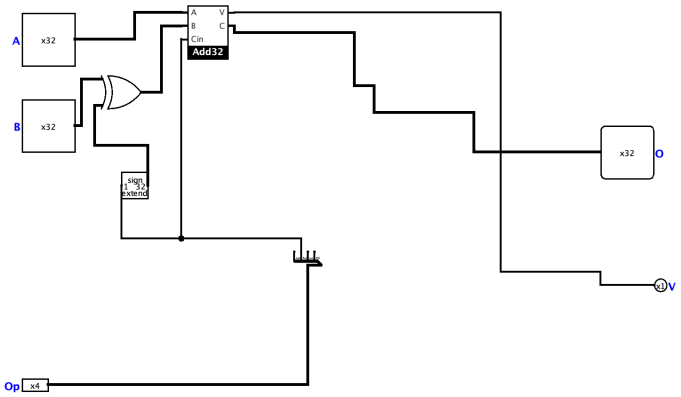
The next criterion for determining the operation to be outputted (assuming it's not addition or subtraction) is that when there's equality between the first three MSBs of the op code, then the output is either a right shift logical or a right shift arithmetic operation. Hence the mux named right shift or sth. The select bit for this mux connects to the equality 3 sub-circuit. The overarching idea of the equality3 sub-circuit is that it outputs 1 if all three inputs are the same, else 0. Hence equality3 is connected to the first three MSBs of the op code to assess the validity of criterion 2.

If criterion 2 falls through as well, then the operation must be a left logical shift or a comparison operation or a logical operation. Criterion 3 speaks to the fact that of the three outstanding possibilities, we have the situation of a 0 and a 1 between the two middle bits of the op code only if the operation requires output of a logical gate. I use an XOR gate to check the validity of this criterion on bit1 and bit2 of the op code.

Following this criterion, the aim of criterion 4 is to distinguish between the op code of a comparator or a left shift by checking if the MSB of the op code is a 0 or a 1. Which is a necessary/valid condition to distinguish between a left shift logical op code or a comparator op code.
So I more or less developed a query to explore the desired output (C[32]) of the provided op code.
Determining output V[1] uses the exact same querying algorithm as C[32]. The muxes towards the bottom of the circuit work similarly to the muxes higher up in the circuit and are used to determine the V which should be outputted by the circuit by selecting the V which was outputted by the component whose C[32] is being outputted.

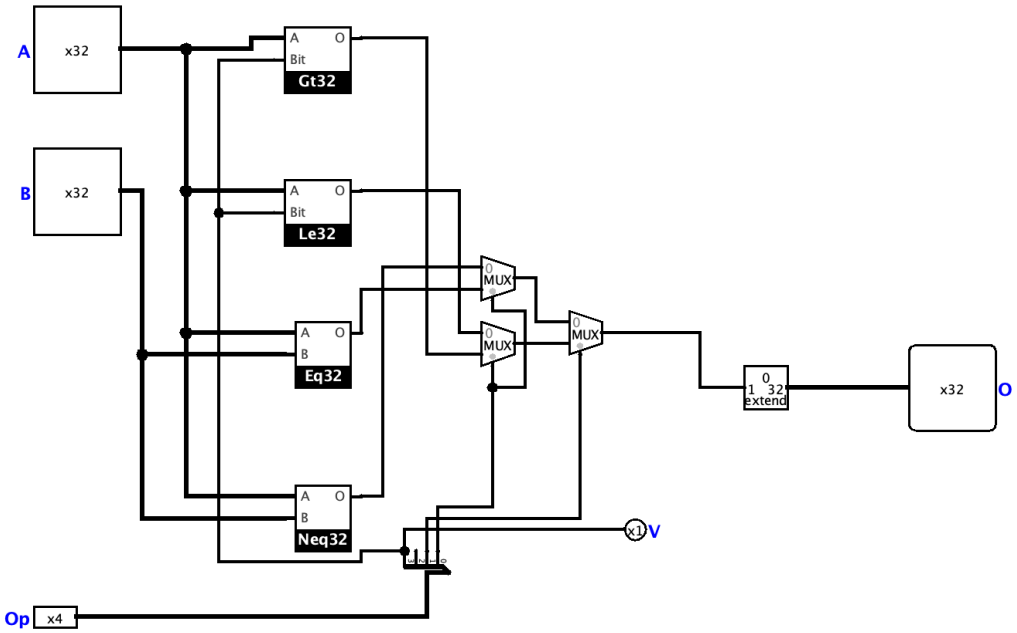


Arithmetic: This circuit uses Add32 to calculate the A+B or A-B given inputs A and B. If the Op code contains a 1 on bit2, then we invert B and provide a Cin of 1 : This gets the two's complement value of B so we perform the correct calculation by adding A and B'. Else if the op code contains 0 on bit2, then we add B as is to A and send the output to O. We sign-extend bit2 so it matches the input width of B[32] so we can have a consistent/valid XOR gate.

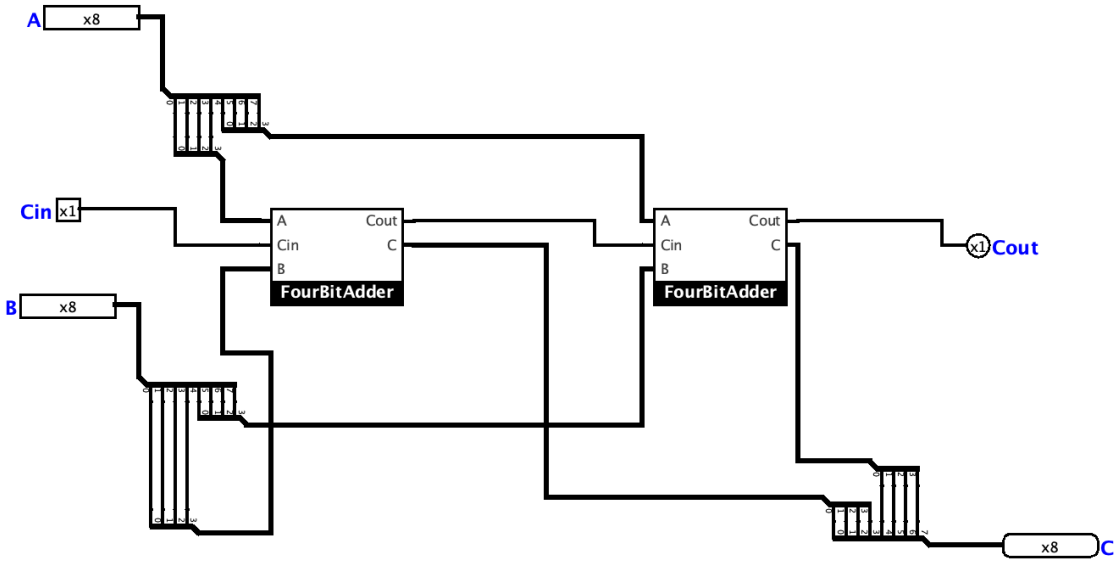


Comparators: I use this circuit to determine the logic operation to be outputted and provide that as 0. For all comparators, bit3 has value 0 so I use that as the output for V. I then use 3 muxes to distinguish between which comparator corresponds to the provided Op code according to the following 3 criterions (which I use as the signal bits for my muxes):

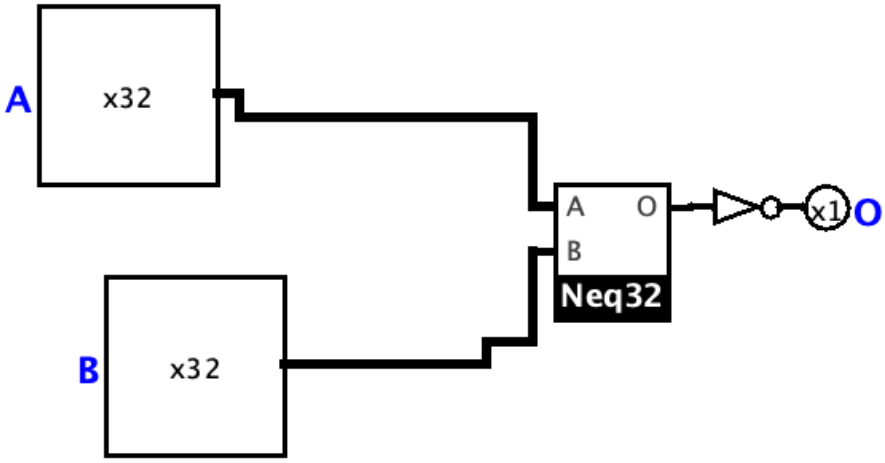
1. If there is a 0 on bit1 then the Op code matches either a Equal to or a NotEqual to gate. (else it matches to a Greater than or a Less than gate)
2. If there is a 0 on bit 1 then a 1 on bit 0 then the Op code is a request to a NotEqual to gate
3. If there is a 0 on bit 1 then a 0 on bit 0 then the Op code is a request to an Equal to gate
4. there is a 1 on bit 1 then a 1 on bit 0 then the Op code is a requests a Greater than gate
5. there is a 1 on bit 1 then a 0 on bit 0 then the Op code is a requests a Less than or equal to gate8-bit Adder:This is a sub circuit of adders (1-bit adders) abstracted for simplicity, towards my construction of Add32



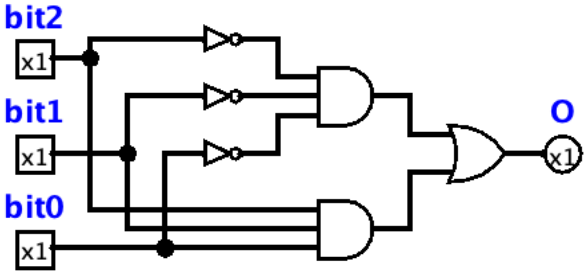
8-Bit Adder: This is a sub circuit of adders (1-bit adders) abstracted for simplicity, towards my construction of Add32



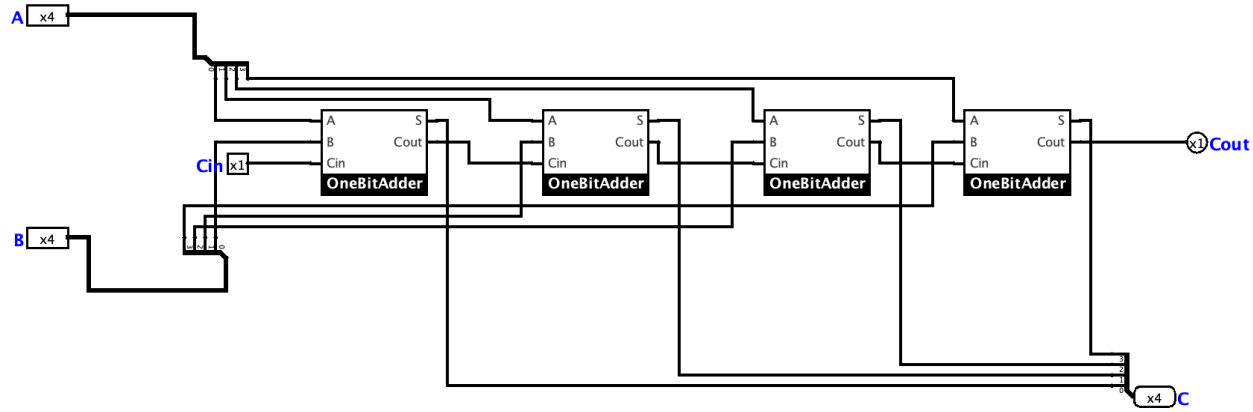
Eq32: This sub-circuit asserts equivalence between A and B by getting the output of if A and B are not equal, and inverting it.



Equality3: I use this to check equality between 3 inputs, by outputting 1 if they are all 1s i.e. A AND B AND C or 1 if NOT A AND NOT B AND NOT C.

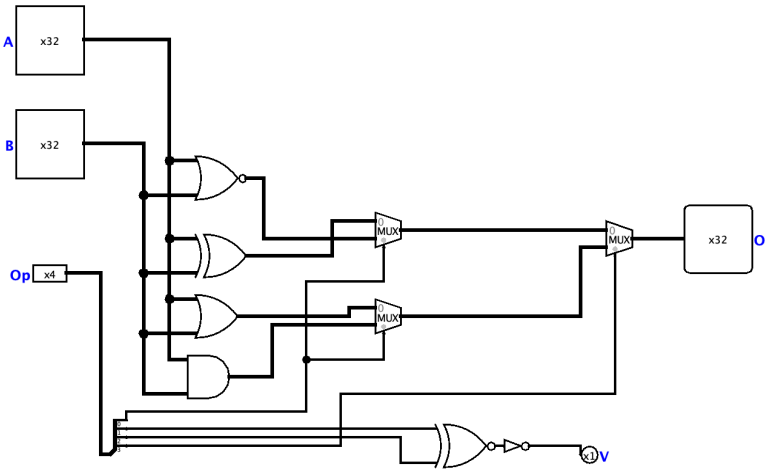


4-bit Adder: This is a sub circuit of adders (1-bit adders) abstracted for simplicity, towards my construction of Add32

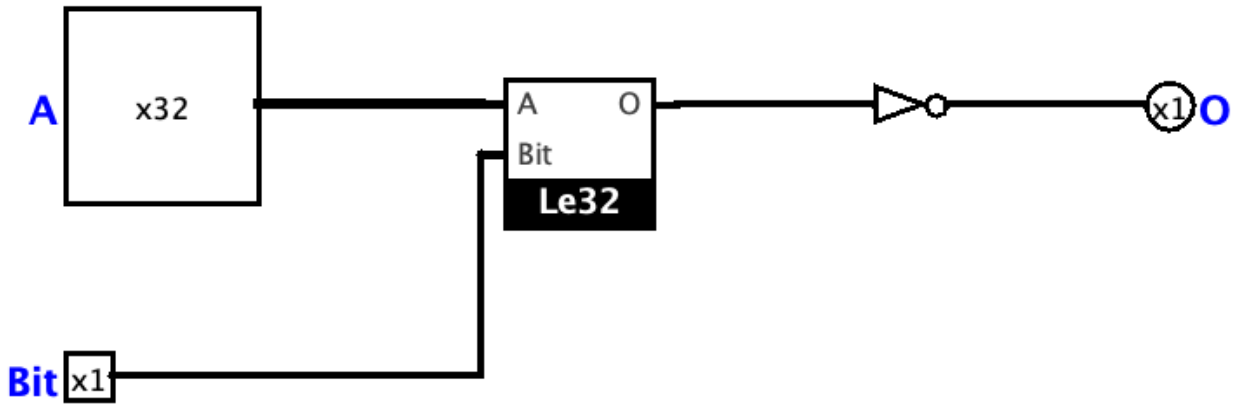


Gates: These are simply 32-input gates I use for the logical calculation of AND, OR, XOR, and NOR operations. I use NOT (bit1 XNOR bit2) as my output for V to ensure that V is 0 because the Op code for AND, OR , XOR or NOR all have both middle bits being the same, so by checking their equivalence (with XNOR) and negating it with not, I make sure V is always 0.
I also use 3 muxes to determine the required gate by using selector bits which leverage the difference between the opcodes of the 4 logical operations.

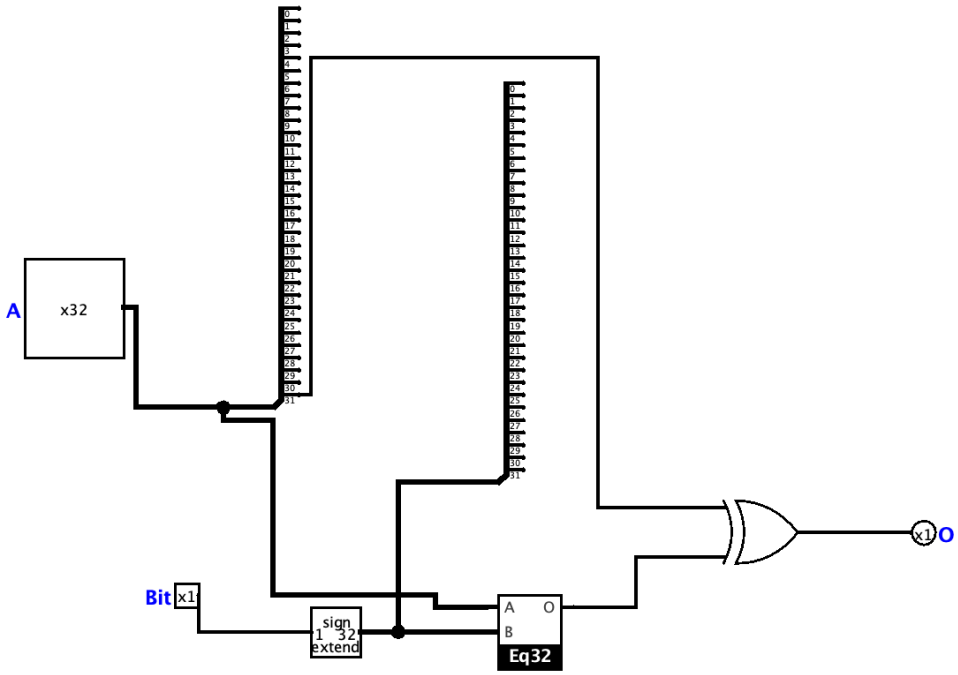
1. If there is a 0 on bit3 then the Op code matches either an XOR or a NOR gate. (else it matches a Greater than or a Less than gate)
2. If there is a 0 on bit 3 then a 1 on bit 0 then the Op code is a request to a NOR gate
3. If there is a 0 on bit 3 then a 0 on bit 0 then the Op code is a request to an XOR gate
4. there is a 1 on bit 3 then a 1 on bit 0 then the Op code is a requests a AND gate
5. there is a 1 on bit 3 then a 0 on bit 0 then the Op code is a requests an OR gate



Gt32: This circuit negates the output of LE32 to determine if input A is greater than 0

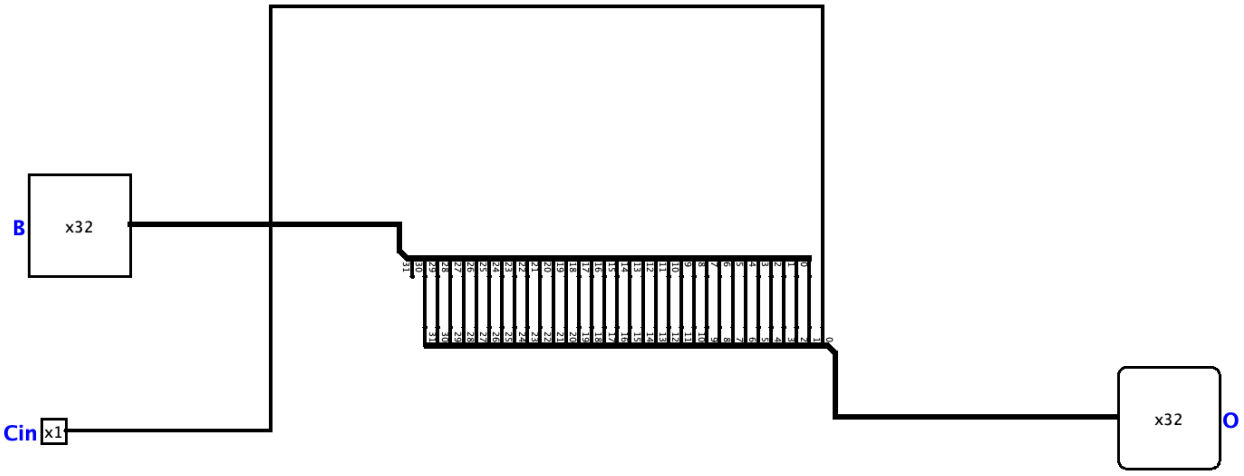


LE32: Checks if input A is less than or equal to 0 by checking if it's MSB is one (which means the value is negative) implying that A is less than 0, or assessing equality with the input Bit which must always be 0 because it is set to a bit in the Op code of comparators which is 0 across all comparators (bit3).

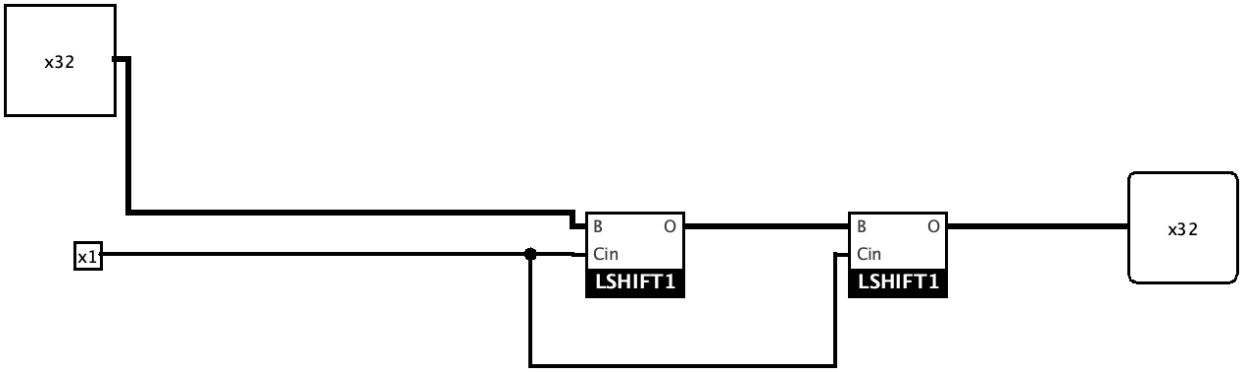


LeftShift1: I used this splitter to left shift a 32-bit input by one bit by connecting it to a splitter which takes the first 31 bits of the input and replaces the last input with the Cin input value.

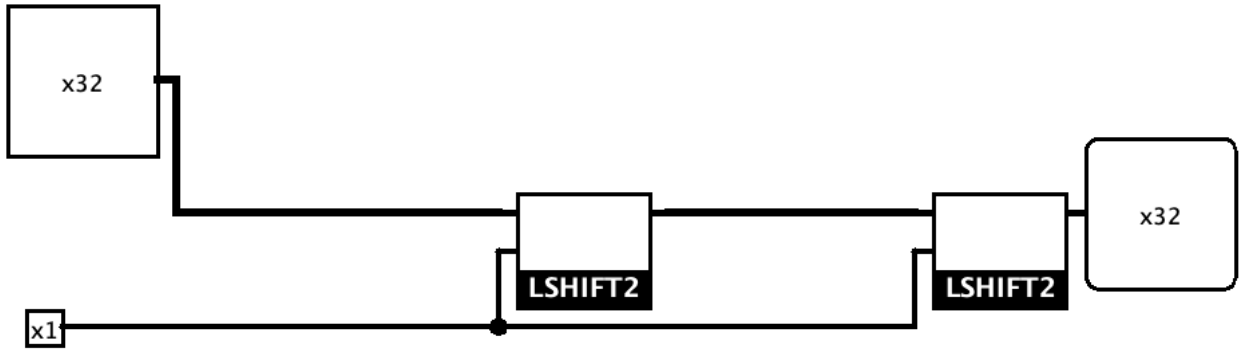
Then outputs the new 32-bit value to O.



LShift2: This is a sub-circuit which combines sub-circuits of leftshift1 towards my implementation of leftshift32

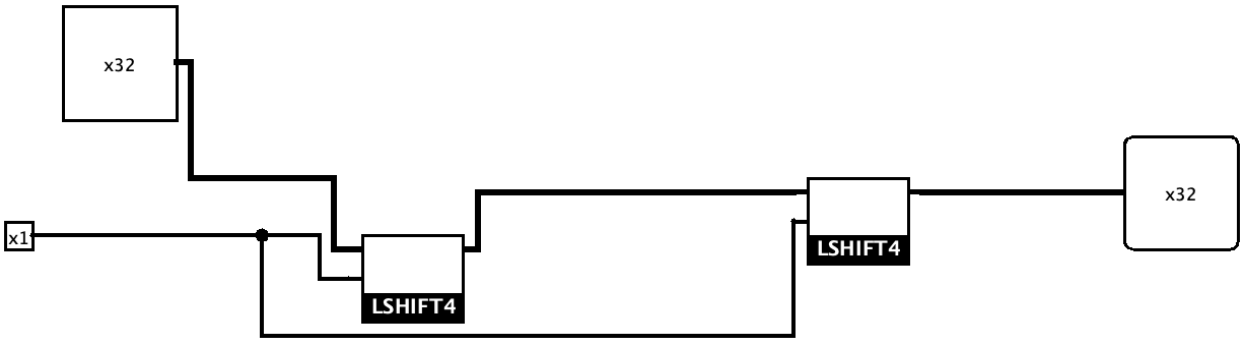


LShift4: This is a sub-circuit which combines sub-circuits of leftshift1 towards my implementation of leftshift32

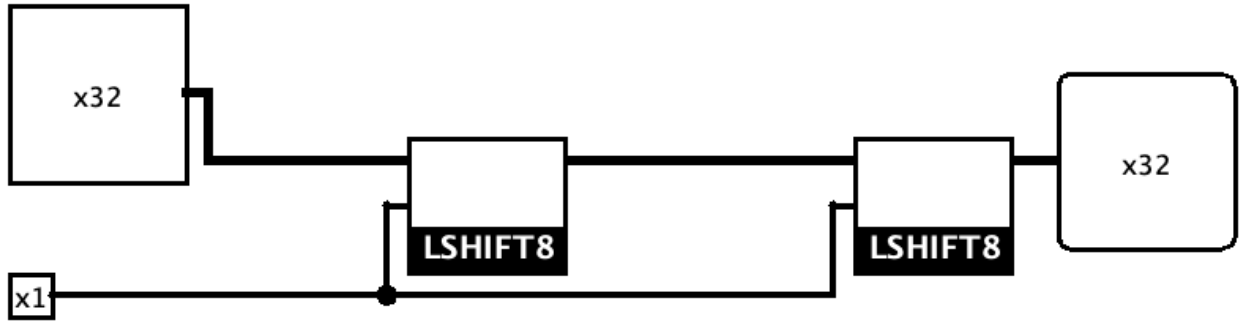


LShift8: This is a sub-circuit which combines sub-circuits of leftshift1 towards my implementation

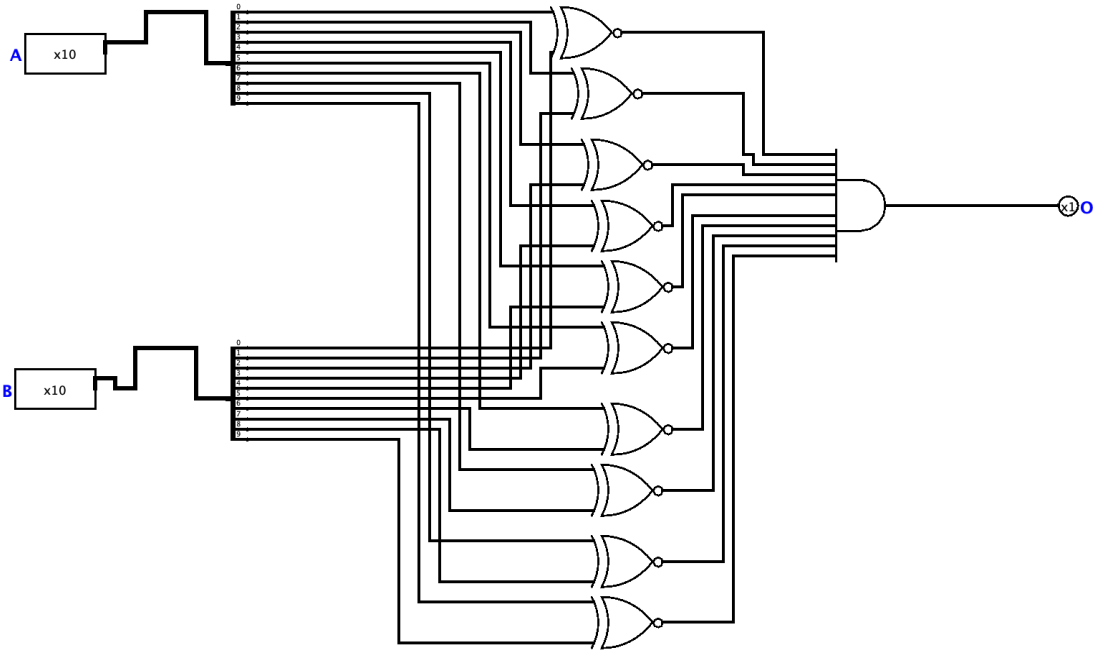
of leftshift32



LShift16: This is a sub-circuit which combines sub-circuits of leftshift1 towards my implementation of
leftshift32

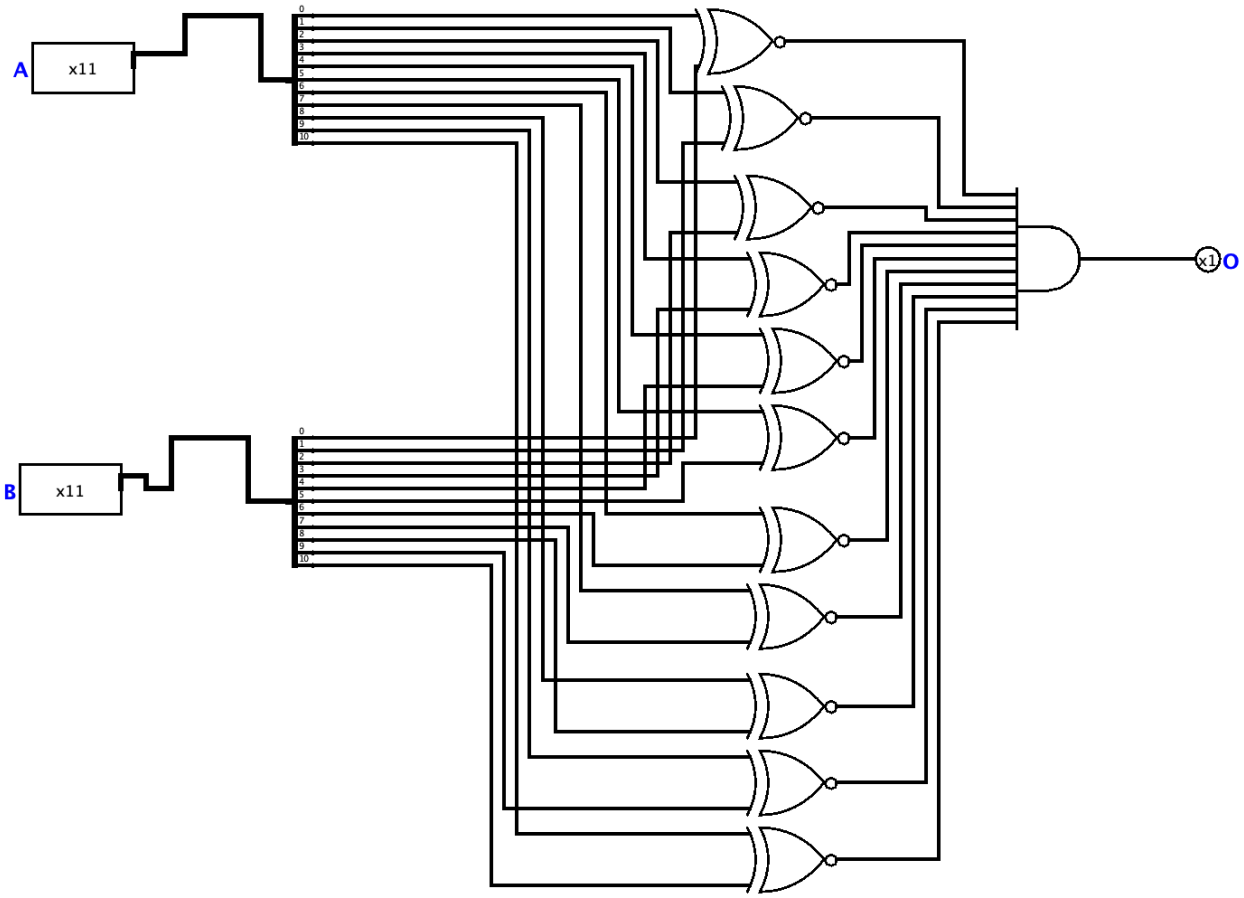


Neq10: I use the XNOR gate to check equivalence between each bit in A and each bit in B, so this circuit outputs 1 if all bits in A are the same as all bits in B and outputs 0 if there's a difference between any corresponding bits of A and B. (Unintuitively- because it's a sub-circuit

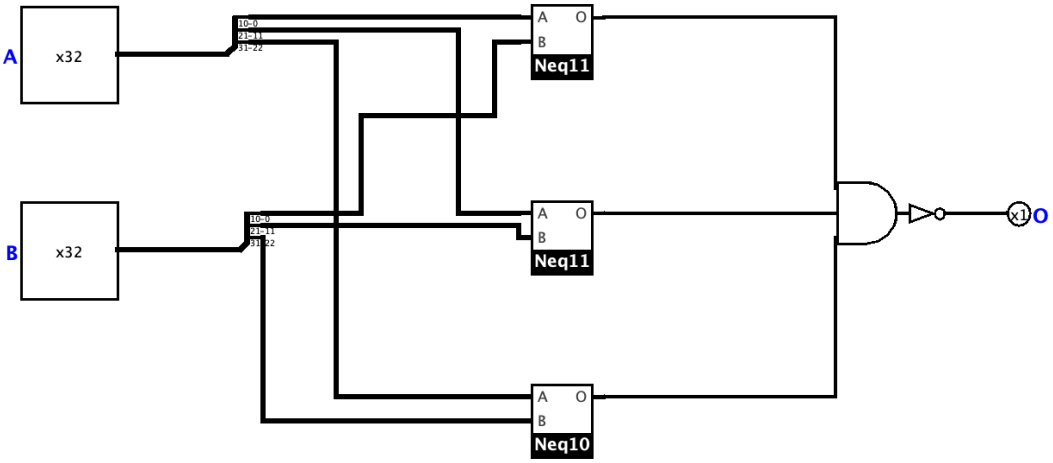


of Neq32)

Neq11: I use the XNOR gate to check equivalence between each bit in A and each bit in B, so this circuit outputs 1 if all bits in A are the same as all bits in B and outputs 0 if there's a difference between any corresponding bits of A and B. (Unintuitively- because it's another sub-circuit of Neq32)

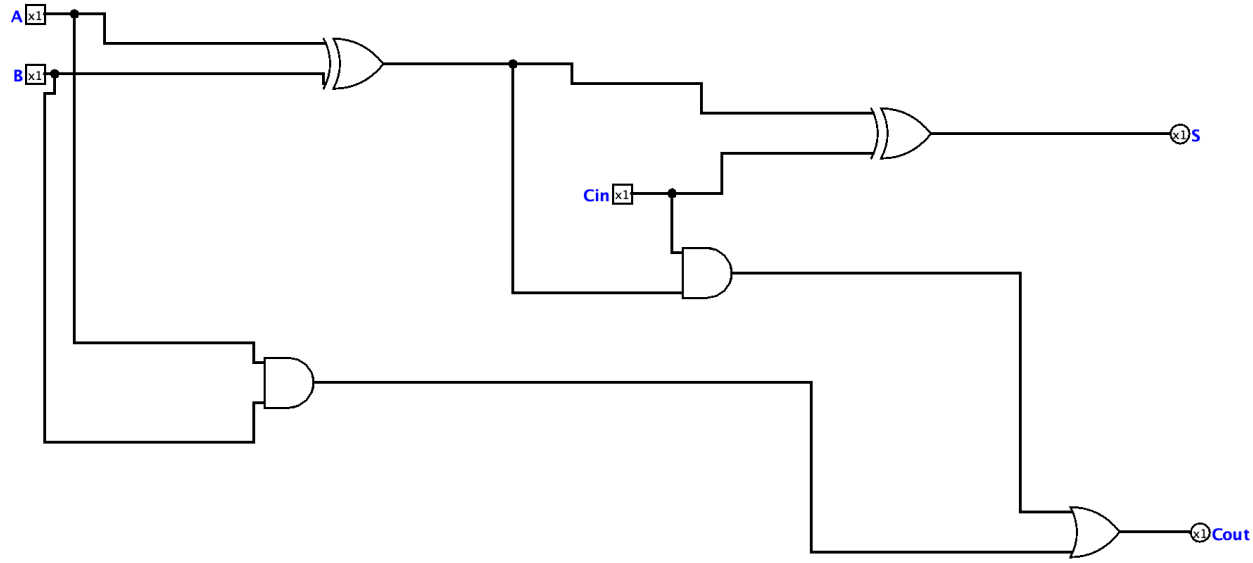


Neq32: This sub-circuits outputs 1 if A !=B and outputs 0 if A == B. Because Neq11 and Neq10 output 1 if A and B are the same, I form Neq32 by placing their outputs in an AND gate to check that all bits in A and B are equivalent and then the output is inverted so that this circuit outputs 1 if A !=B and outputs 0 if A == B.



1-Bit Adder: I used this to perform the addition of two bits A and B. A one-bit full adder was designed to support the addition of two 1-bit inputs with a carry bit. This circuit is necessary for the construction of more advanced full adders. Cin is the carry-in bit, S is the output bit, while Cout is the carry-out bit.
It is the optimal circuit for the following truth table:

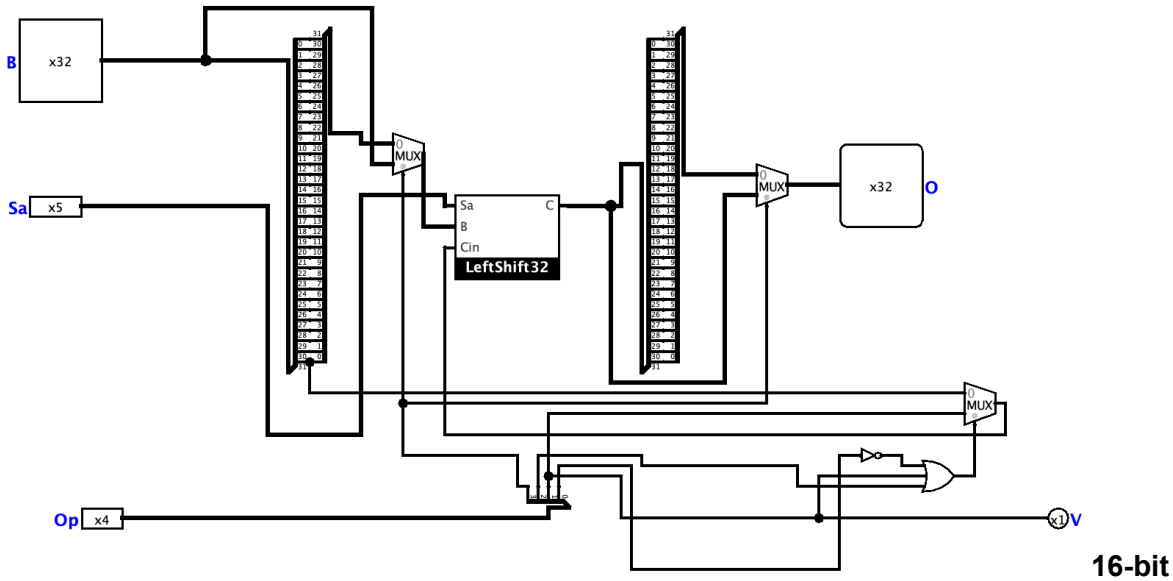
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



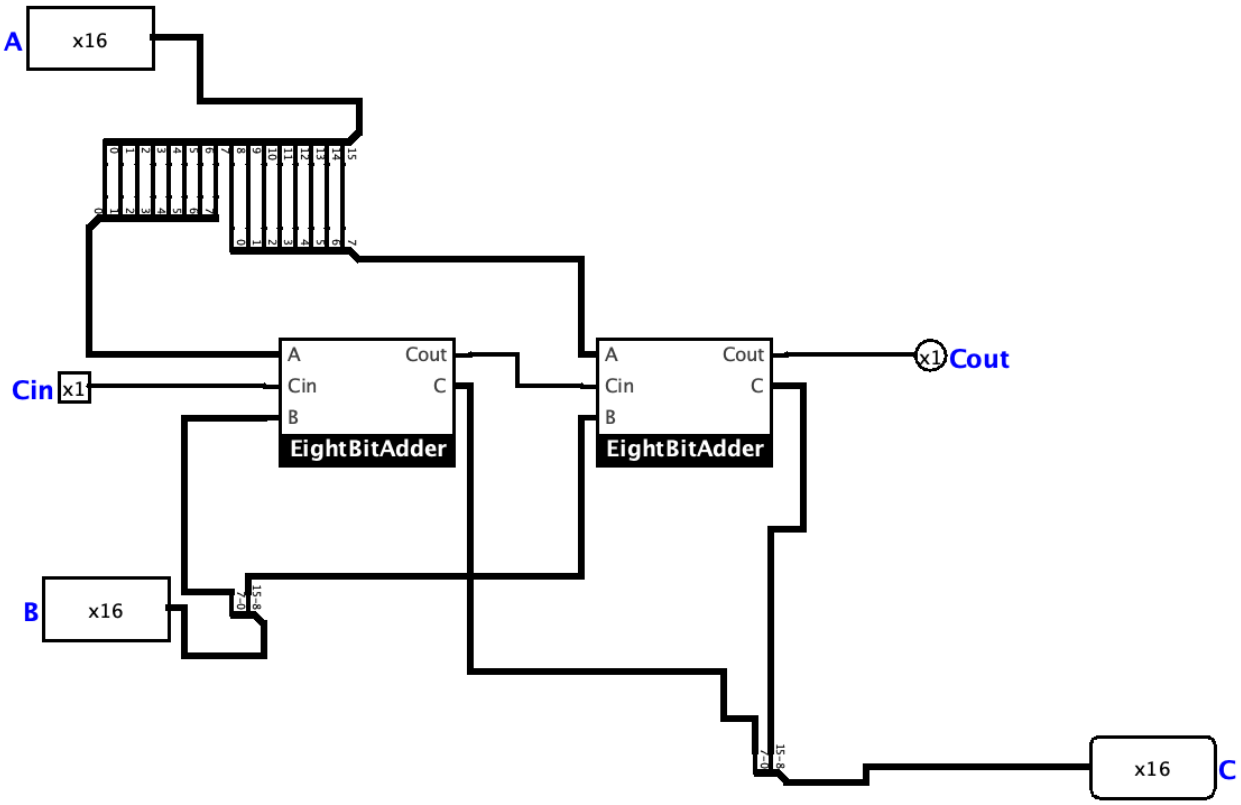
Shifters: I use this sub-circuit to structure my shifting operation relative to the provided Op code. If the Op code. This is the truth table for the component of the circuit taking the 3 MSB's of the Op code (towards the bottom of the circuit). It returns 0 when the Op code indicates a request for an arithmetic right shift (0001). This is used as a signal for the mux, to decide Cin: because Cin can either be the MSB or 0. I also make use of two muxes to determine whether I need to flip B before shifting in order to execute a right shift using my left shift32 and then flip back to obtain the correctly ordered right-shifted value. As a select bit for my two muxes, I use the MSB of the Op code because it distinguishes between a left shift and a right shift: if the MSB

is 1 then the Op code is a request for a left shift, and if it's 0, the Op code is a request for a right shift.

bit0	bit1	bit2	O
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Adder: This is a sub circuit of adders (1-bit adders) abstracted for simplicity, towards my construction of Add32



2-bit Adder: This is a sub circuit of adders (1-bit adders) abstracted for simplicity, towards my construction of Add32