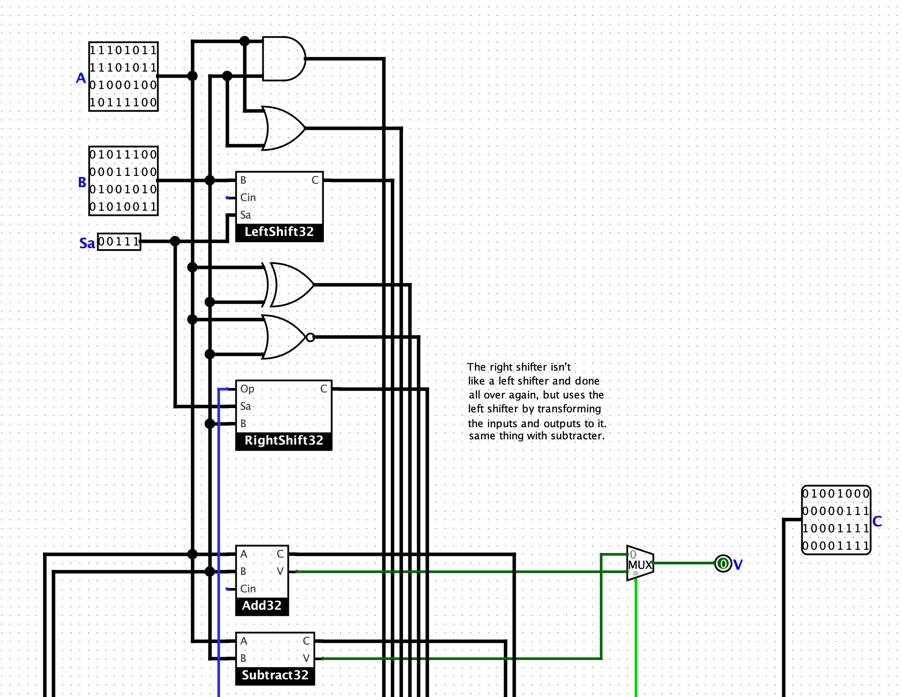
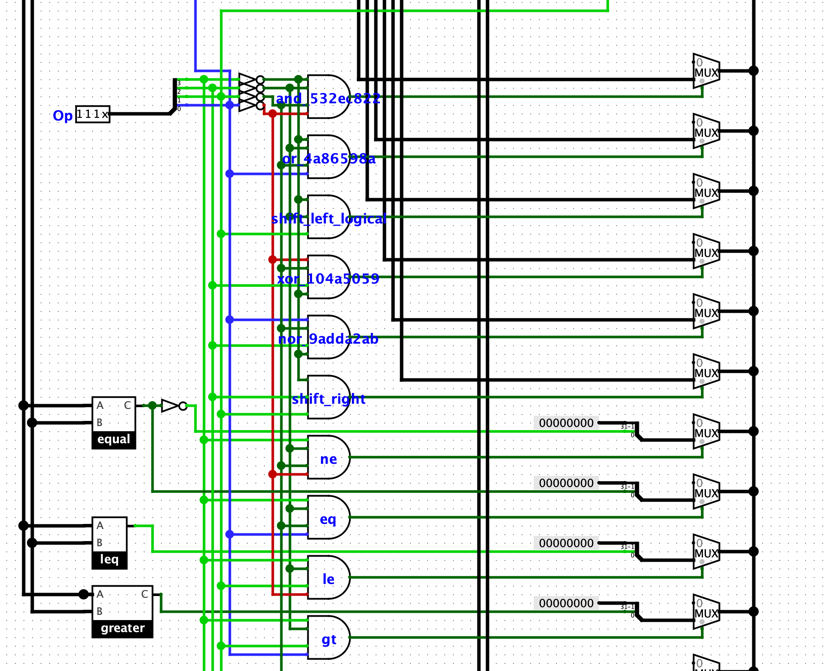
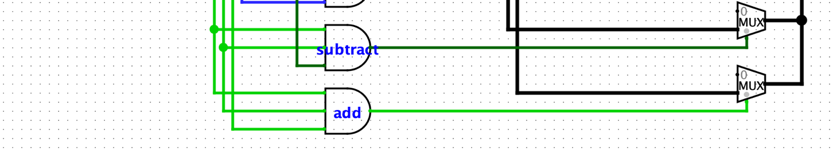
# ALU32





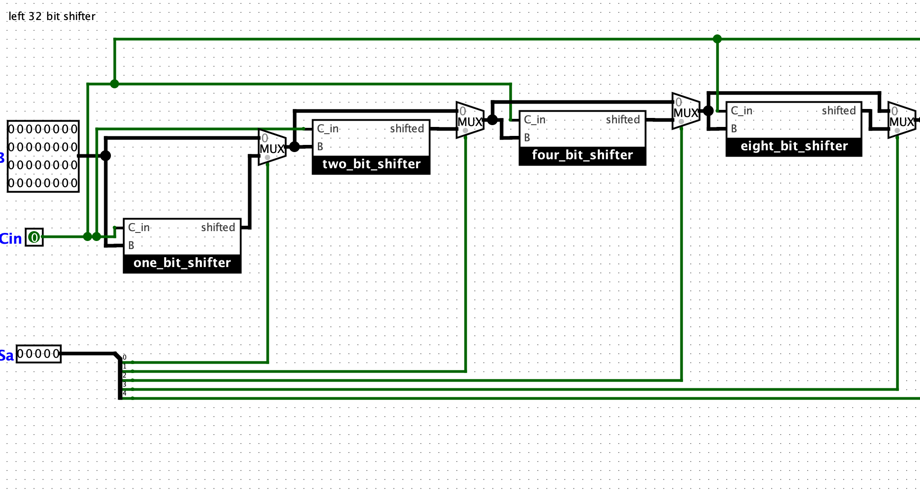
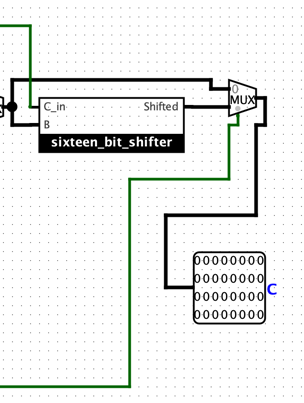


I implemented the Op codes using multi-input (3- or 4- input) AND gates and transferred the corresponding operation result to the output through a multiplexer without the input.

I did not make a right shifter like how I made a left shifter, but followed the hint about transforming inputs and outputs to the right shifter.

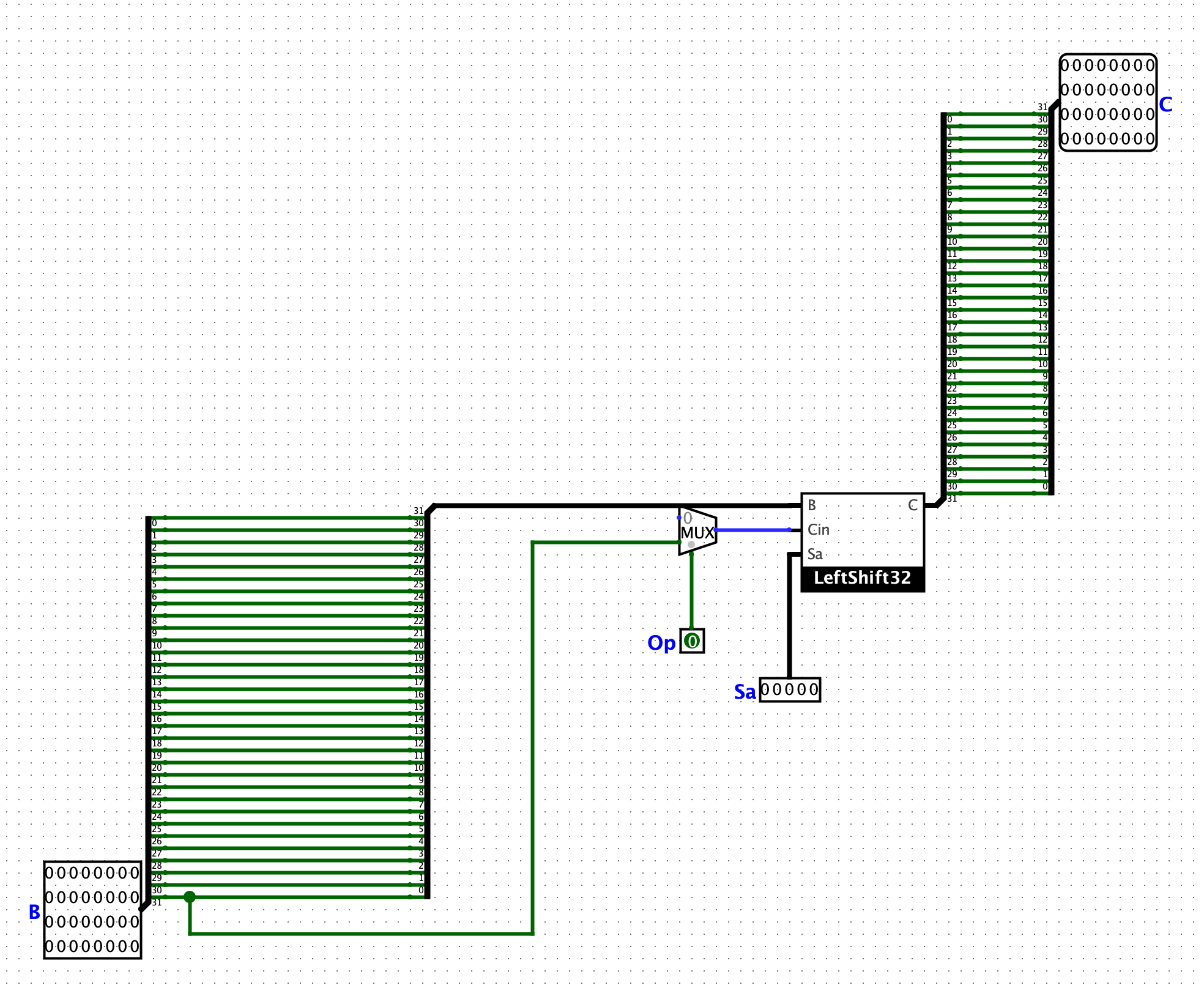
Similarly, with the subtractor, I just transformed B to be its two’s complement and implemented overflow following rules about signed binary number addition/subtraction overflow.

# LeftShift32



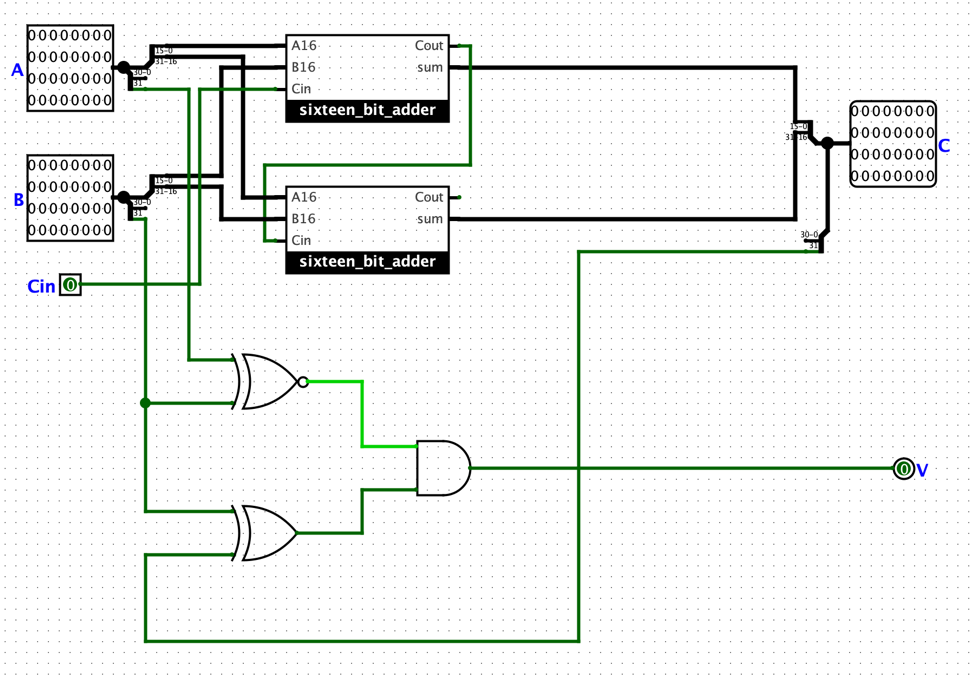
I used a one-bit shifter, a two-bit shifter, a four-bit shifter, an eight-bit shifter, and a sixteen-bit shifter to construct my thirty-two-bit shifter.

## RightShift32



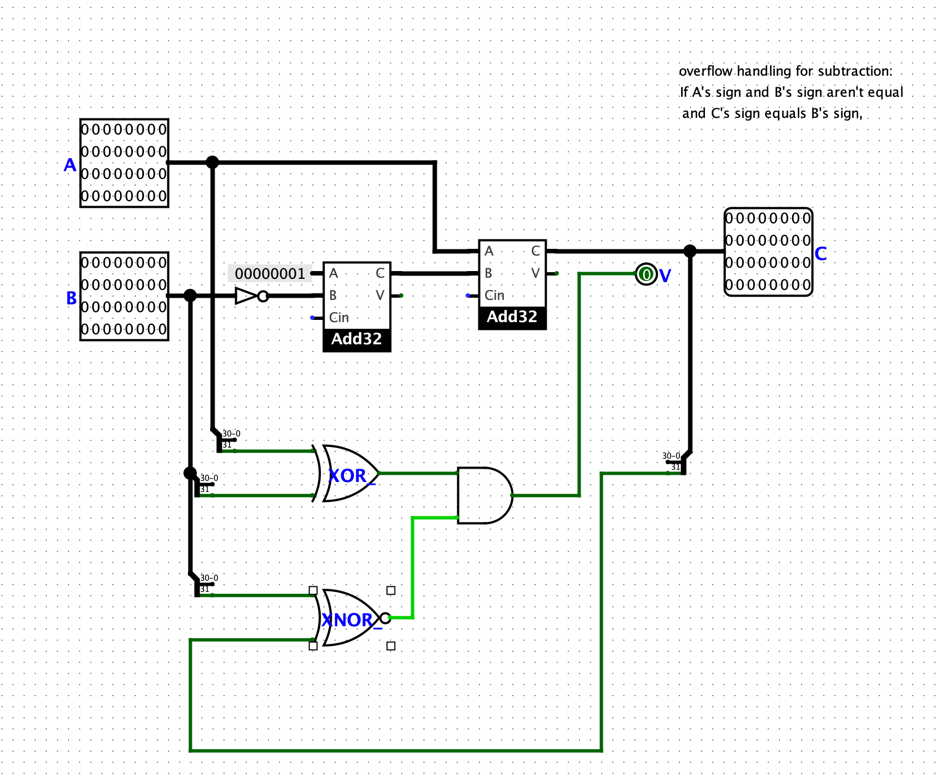
My thirty-two-bit right shifter is implemented by reversing (e.g. ) the input then shifting it left using LeftShift32 then reversing it back. specifies whether it’s arithmetic or logical shift.

# Add32



I used two sixteen-bit adders in my thirty-two-bit adder. Each sixteen-bit adder is comprised of 4 four-bit adders, which are then comprised of 4 one-bit adders each. I handled overflow in the 32-bit adder by making it an unsigned adder first, then following the rule of if and are of the same sign and is of the opposite sign, then overflow is .

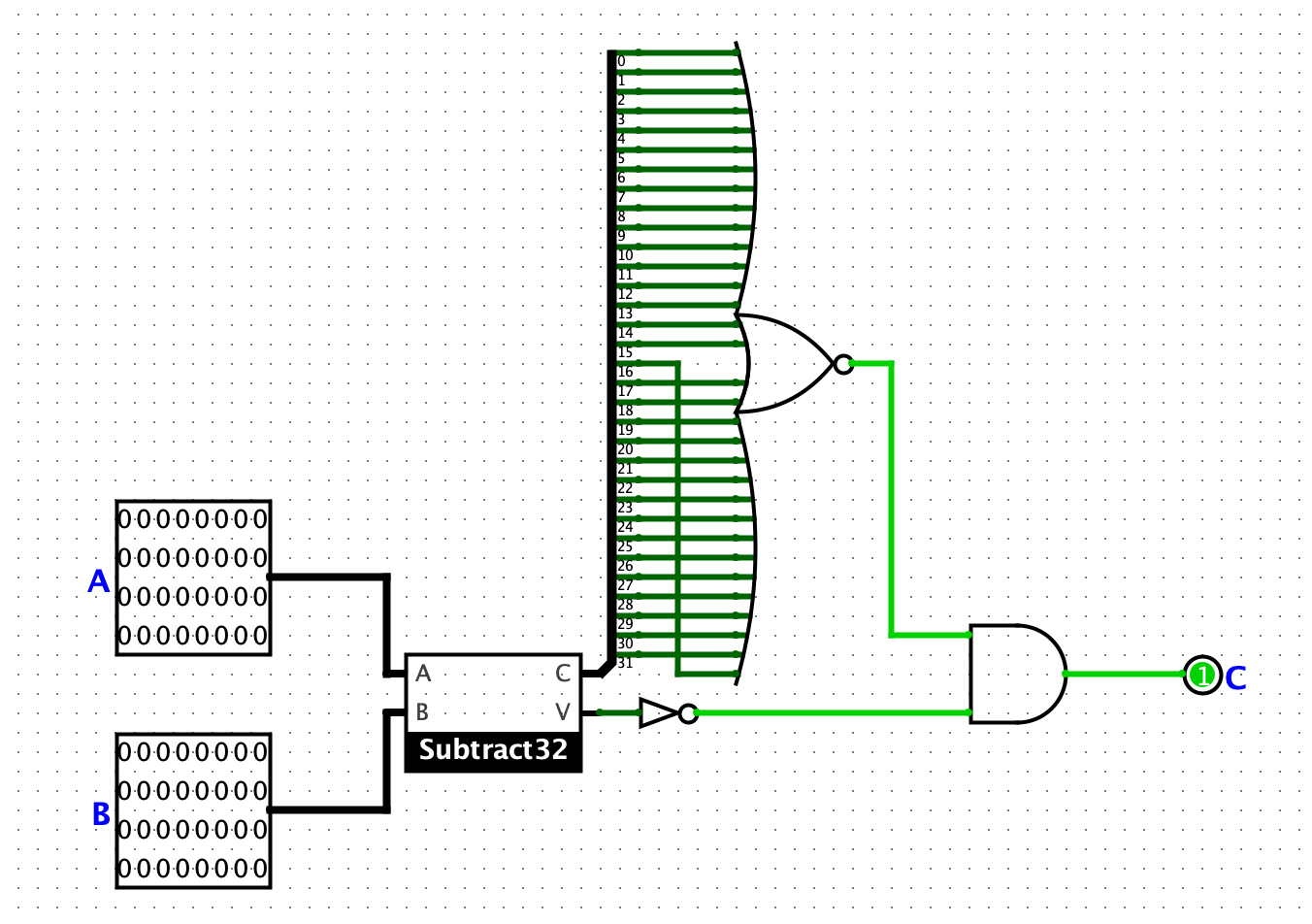
## Subtract32



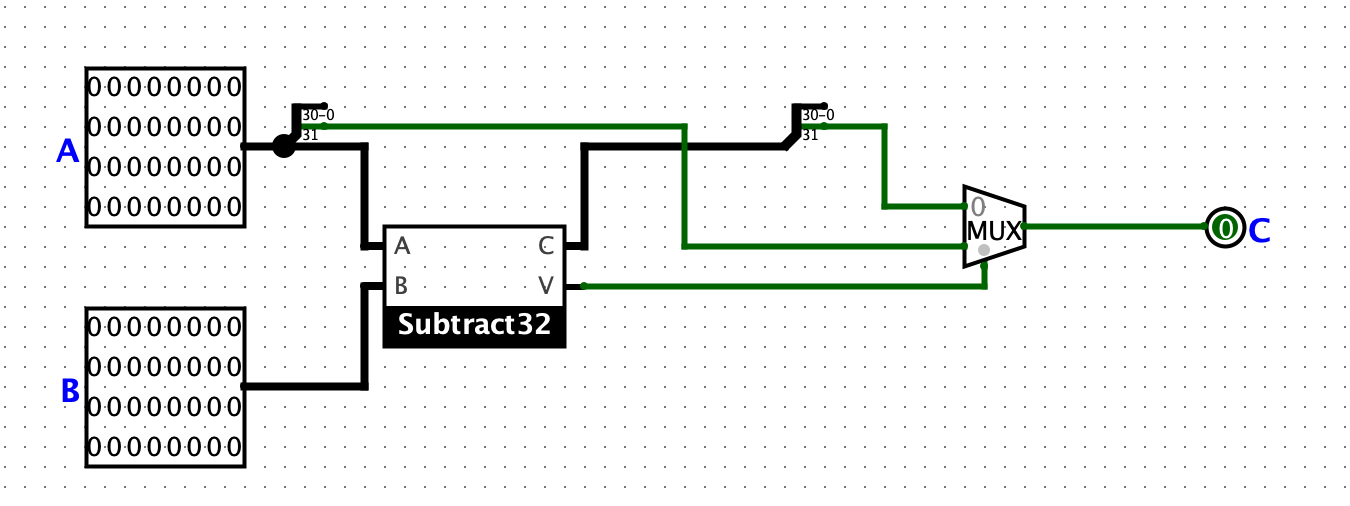
. So I turned into its two’s complement through NOT-ing it (inverting each bit) then adding through my Add32, then added to Add32. I handled subtraction overflow by implementing the rule of if and are of different signs and has the same sign as , then there’s overflow, so I have to set to .

# Comparators

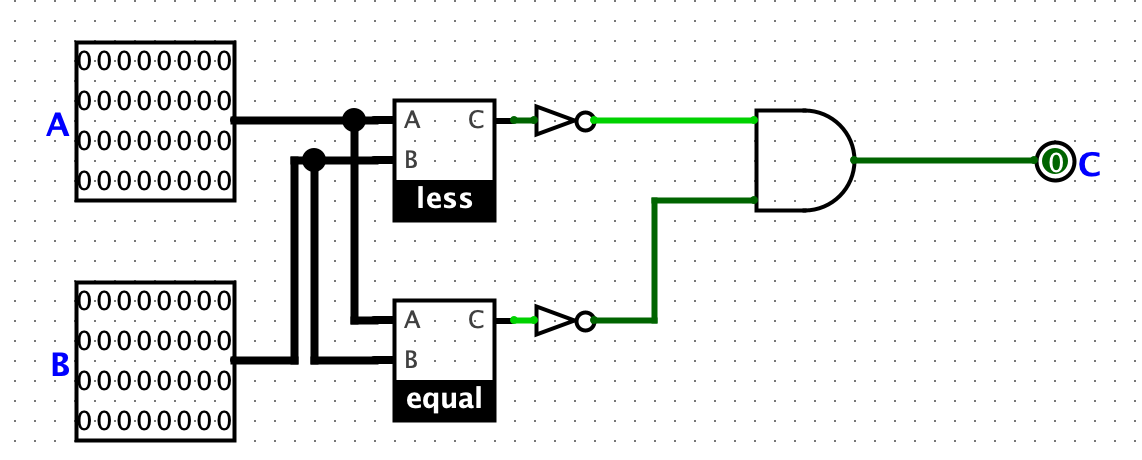
## equal

When , then .

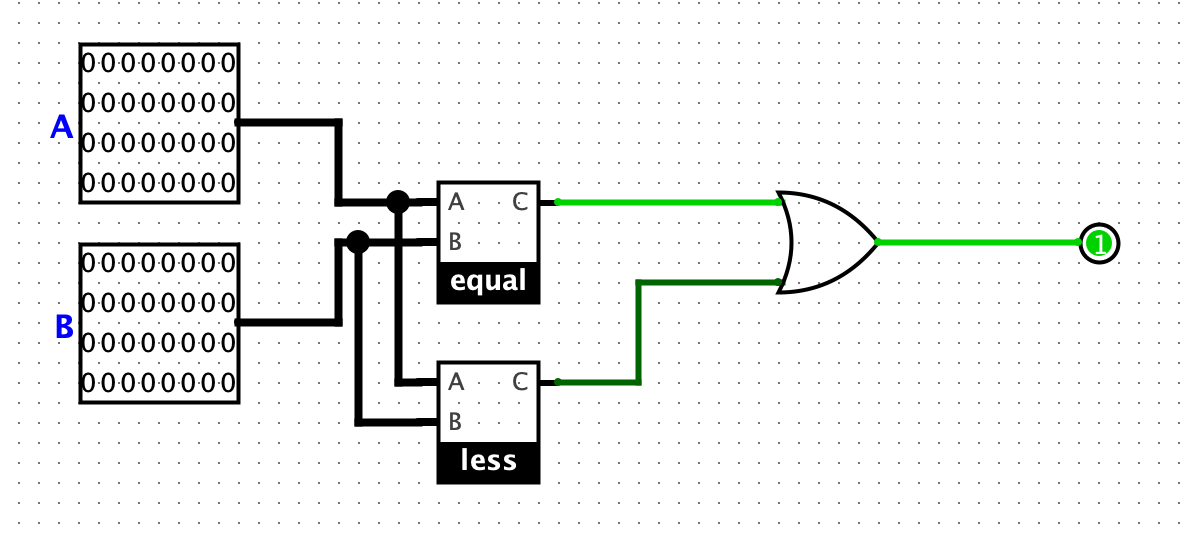
## less

When , then . When , then .

## greater

 if and only if .

## leq

 if and only if .