Proof of Study 5

Qianlang Chen

U1172983

Problem 1

For this problem, I chose to build the prime-number circuit: *"A circuit that takes a 4-bit unsigned binary number as input, and produces an output that is true only if the input represents a prime number."* Since the input number is only 4-bit long, to approach this, I figured that the easiest way is to use brute-force — manually select numbers that are prime without doing any arithmetic. An unsigned 4-bit number is ranged from 0 to 15, and only 6 of which are prime numbers: 2, 3, 5, 7, 11, and 13. This would imply that the output would only be *true* when the input was equal to any of the prime numbers above, leading to the following truth table, where the inputs A to D are the bits of the input number, with D being the least significant bit:

A	B	C	D	OUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

The above truth table gives all the possibilities for the combination of the 4 bits to produce an output of *true*. However, by noticing some patterns in the inputs and outputs, we can reduce the conditions:

- When the inputs B, C, and D are equal to 011 or 101, the output will be *true* no matter what the input A is;
- When the inputs A, C, and D are equal to 011, *true* no matter what B is;
- When the inputs A, B, and C are equal to 001, *true* no matter what D is.

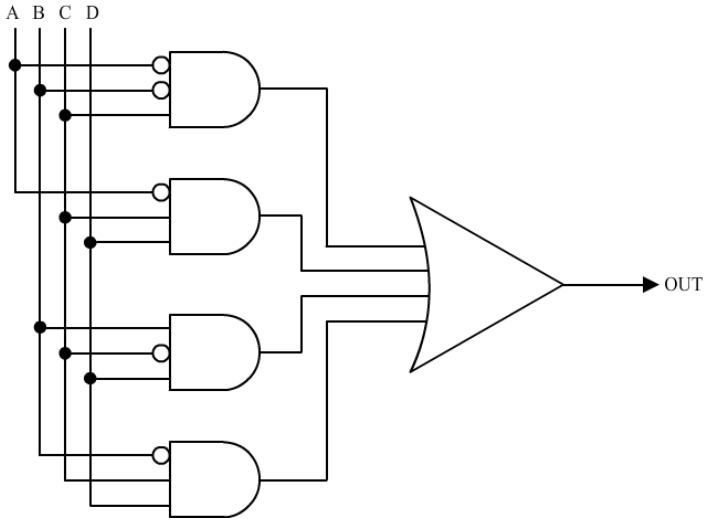
By taking advantage of these features, the conditions of the output being *true* can be reduced:

A	B	C	D	OUT
×	0	1	1	1
×	1	0	1	1
0	×	1	1	1
0	0	1	×	1

Now, we can construct our formula for the output (OUT) according to the table above:

$$\mathbf{OUT = \bar{B}CD + B\bar{C}D + \bar{A}CD + \bar{A}\bar{B}C}$$

Which can then be implemented by the following circuits:



Finally, to demonstrate how Demorgan's Law could be applied to the above equation:

$$\text{OUT} = \overline{B}CD + B\overline{C}D + \overline{A}CD + \overline{A}\overline{B}C$$

$$\textbf{!(OUT)} = (B + \overline{C} + \overline{D}) \times (\overline{B} + C + \overline{D}) \times (A + \overline{C} + \overline{D}) \times (A + B + \overline{C})$$

$$= (BC + B\overline{D} + \overline{B}\overline{C} + \overline{C}\overline{D} + \overline{B}\overline{D} + C\overline{D} + \overline{D})$$

$$\times (A + AB + A\overline{C} + A\overline{C} + B\overline{C} + \overline{C} + A\overline{D} + B\overline{D} + \overline{C}\overline{D})$$

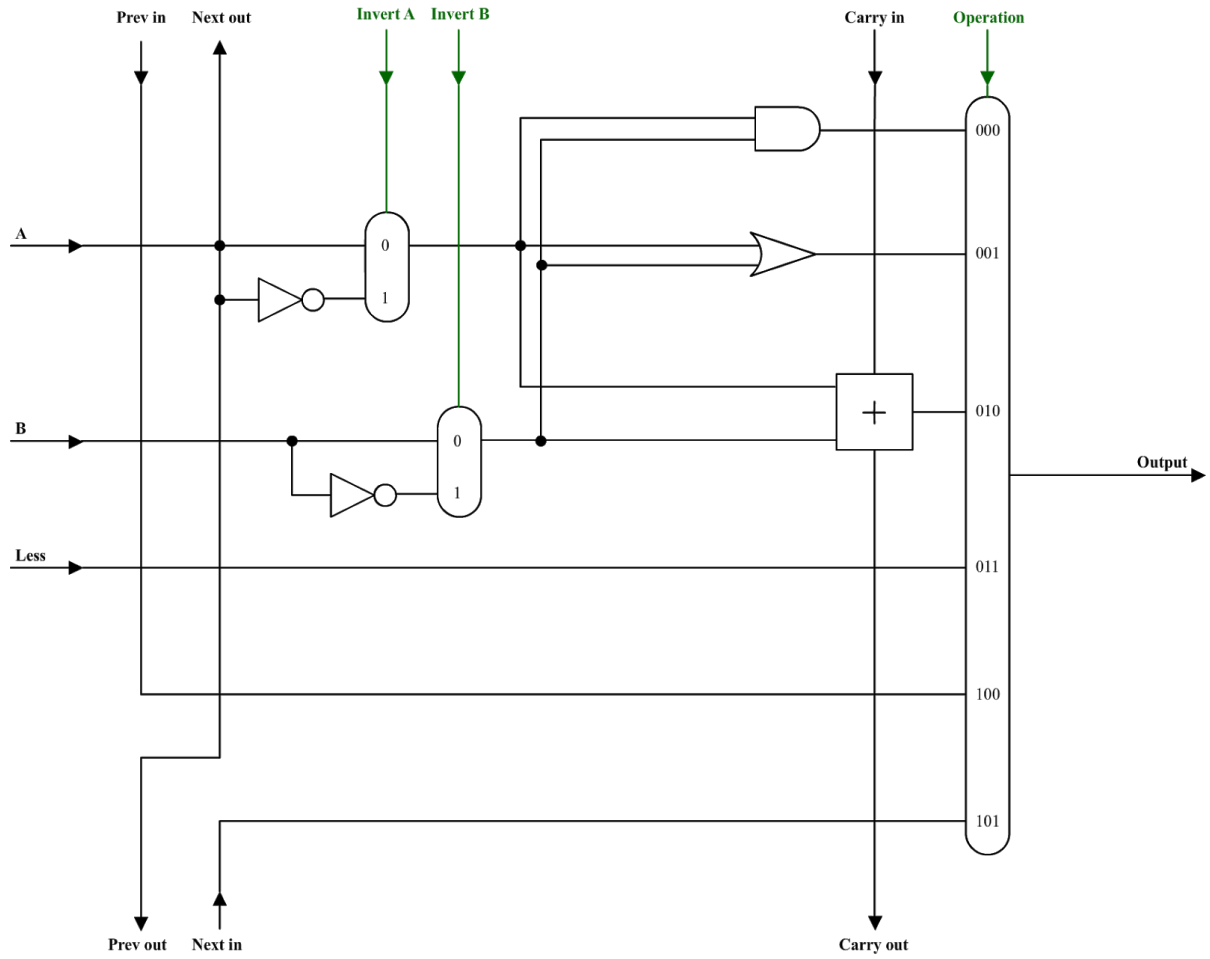
$$= (BC + \overline{B}\overline{C} + \overline{D}) \times (A + B\overline{C} + B\overline{D} + \overline{C})$$

$$= ABC + BC\overline{D} + A\overline{B}\overline{C} + \overline{B}\overline{C} + A\overline{D} + B\overline{C}\overline{D} + B\overline{D} + \overline{C}\overline{D}$$

$$\mathbf{= ABC + A\overline{B}\overline{C} + A\overline{D} + \overline{B}\overline{C} + B\overline{D} + \overline{C}\overline{D}}$$

Problem 2

The shifting operations directly move the bits in a number to the left or right for one position. When a shift-left operation takes place, each bit in the number (except for the first, the least significant bit) gets the value of the previous bit, while the first bit gets a value of 0. A shift-right operation does a similar thing but in the other direction. After understanding these details about the shifting operations, the strategy for implementing them becomes very clear. During a shift-left operation, each bit passes its value to the next bit, and that value goes straight to the MUX at the end. Similar events happen during a shift-right operation, where the bit values are passed to the previous bits instead. (Except for the first and last bits, which will be treated differently with the complete 32-bit circuits designed on the next page.) With these in mind, we can easily build an ALU with shifting operations like so:



Notice that it is just an ordinary ALU (with the operations of and, or, add, and set-on-less-than) with two more wires in the front, and they simply pass the value to/get the value from the bits around them. **To perform a shifting operation, simply set both inverse-controls to 0, initial carry to 0, and the operation code to 100 (shift-left) or 101 (shift-right).** This one-bit ALU can then be cloned and reused to achieve operations for 32-bit values, with some small modifications on the first and last ALUs: (Notice how the first and last bits are treated differently; the first bit always gets a 0 when shifting left, and the last bit always copies and preserves its own value when shifting right.)