Final 4-bit ALU Design for 4-bit CPU

Nahin Ul Sadad
Lecturer
CSE, RUET

ALU in CPU

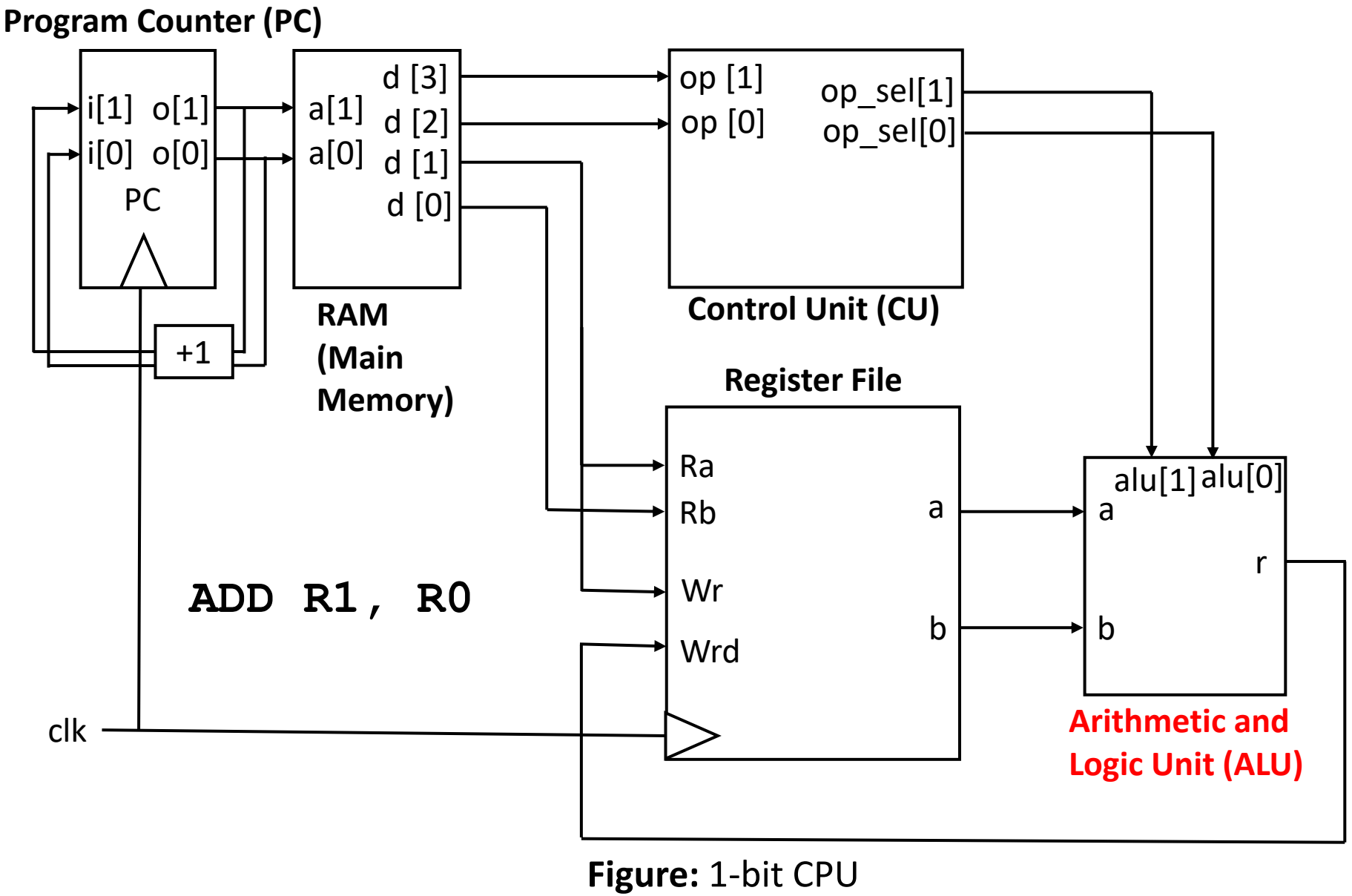


Figure: 1-bit CPU

1. Program Counter will have address of next instruction to be executed in current clock cycle.
2. Address in PC will be sent to RAM to retrieve instruction.
3. Instruction will be decoded by control unit and will select registers and/or immediate values.
4. Data within registers and/or immediate values will be sent to **Arithmetic and Logic Unit (ALU)** to perform operations.
5. The **ALU** will perform operation and result will be sent to the register to be written.
6. Finally, PC will be incremented to point to the next instruction in next clock cycle.

ALU in CPU

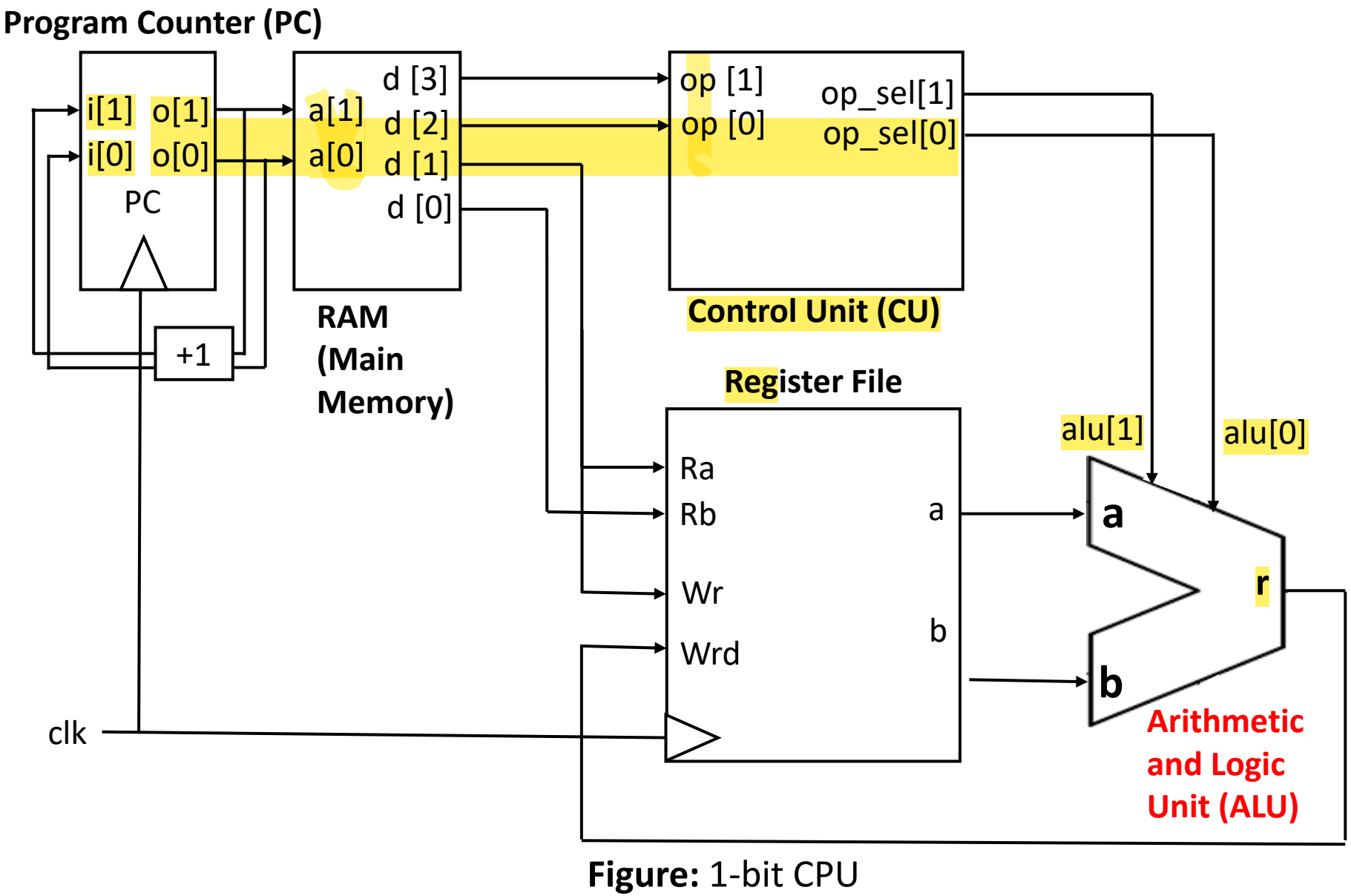


Figure: 1-bit CPU

1. Program Counter will have address of next instruction to be executed in current clock cycle.
2. Address in PC will be sent to RAM to retrieve instruction.
3. Instruction will be decoded by control unit and will select registers and/or immediate values.
4. Data within registers and/or immediate values will be sent to **Arithmetic and Logic Unit (ALU)** to perform operations.
5. The **ALU** will perform operation and result will be sent to the register to be written.
6. Finally, PC will be incremented to point to the next instruction in next clock cycle.

ALU Basics

ALU

Arithmetic Logic Unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. It is a fundamental building block of Central Processing Unit (CPU) of computers.

The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed. The ALU's output is the result of the performed operation.

In many designs, the ALU also has status/FLAG inputs or outputs, or both, which convey information about a previous operation or the current operation, respectively, between the ALU and external status/FLAG registers.

Functions of ALU

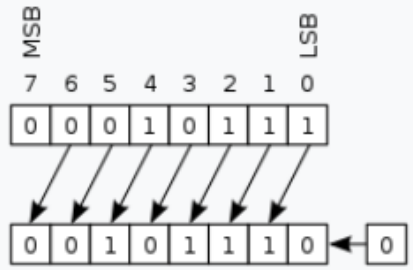
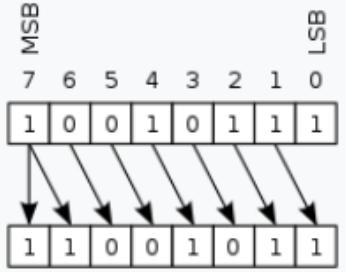
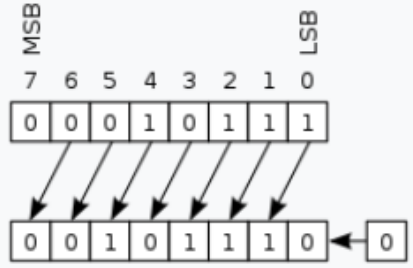
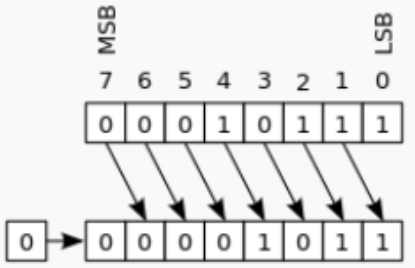
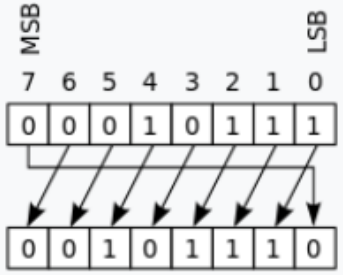
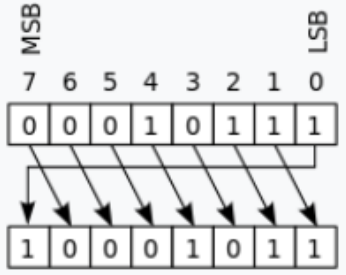
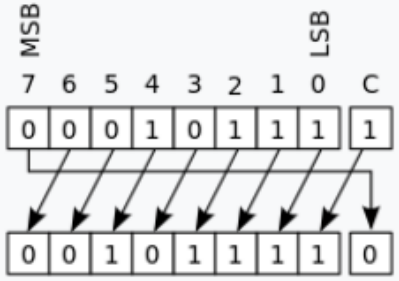
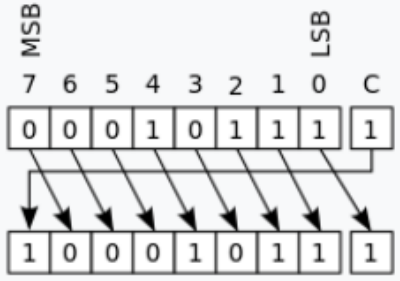
a. **Arithmetic operations:**
   1. **Add:** **`A`** and **`B`** are summed and the sum appears at **`Y`** and **`carry-out`**.
   2. **Add with carry:** **`A`**, **`B`** and **`carry-in`** are summed and the sum appears at **`Y`** and **`carry-out`**.
   3. **Subtract:** **`B`** is subtracted from **`A`** (or vice versa) and the difference appears at **`Y`** and **`carry-out`**. For this function, carry-out is effectively a "borrow" indicator. This operation may also be used to compare the magnitudes of **`A`** and **`B`**; in such cases the **`Y`** output may be ignored by the processor, which is only interested in the status bits (particularly zero and negative) that result from the operation.
   4. **Subtract with borrow:** **`B`** is subtracted from **`A`** (or vice versa) with borrow (**`carry-in`**) and the difference appears at **`Y`** and **`carry-out`** (borrow out).
   5. **Two's complement (negate):** **`A`** (or **`B`**) is subtracted from zero and the difference appears at **`Y`**.
   6. **Increment:** **`A`** (or **`B`**) is increased by one and the resulting value appears at **`Y`**.
   7. **Decrement:** **`A`** (or **`B`**) is decreased by one and the resulting value appears at **`Y`**.

Functions of ALU

b. **Bitwise logical operations:**
   1. **AND:** the bitwise AND of **`A`** and **`B`** appears at **`Y`**.
   2. **OR:** the bitwise OR of **`A`** and **`B`** appears at **`Y`**.
   3. **Exclusive-OR:** the bitwise XOR of **`A`** and **`B`** appears at **`Y`**.
   4. **Ones' complement:** all bits of **`A`** (or **`B`**) are inverted and appear at **`Y`**.

c. **Bit shift operations:** ALU shift operations cause operand **`A`** (or **`B`**) to shift left or right (depending on the opcode) and the shifted operand appears at **`Y`**.
   1. **Arithmetic shift:** The operand is treated as a two's complement integer, meaning that the most significant bit is a "sign" bit and is preserved.
   2. **Logical shift:** A logic zero is shifted into the operand. This is used to shift unsigned integers.
   3. **Rotate:** The operand is treated as a circular buffer of bits so its least and most significant bits are effectively adjacent.
   4. **Rotate through carry:** The carry bit and operand are collectively treated as a circular buffer of bits.

Functions of ALU

Type	Left	Right
Arithmetic shift		
Logical shift		
Rotate		
Rotate through carry		

FPU

A floating-point unit (FPU, colloquially a math coprocessor) is a part of a computer system specially designed to carry out operations on floating-point numbers. Typical operations are addition, subtraction, multiplication, division, and square root.

In general-purpose computer architectures, one or more FPUs may be integrated as execution units within the Central Processing Unit (CPU). However, many embedded processors/microcontrollers do not have hardware support for floating-point operations.

When a CPU is executing a program that calls for a floating-point operation, there are three ways to carry it out:

1. A floating-point unit emulator (A floating-point library/Software).
2. Add-on FPU (Coprocessor/Not inside Processor).
3. Integrated FPU (Inside Processor).

ALU

ALU chip is shown below:

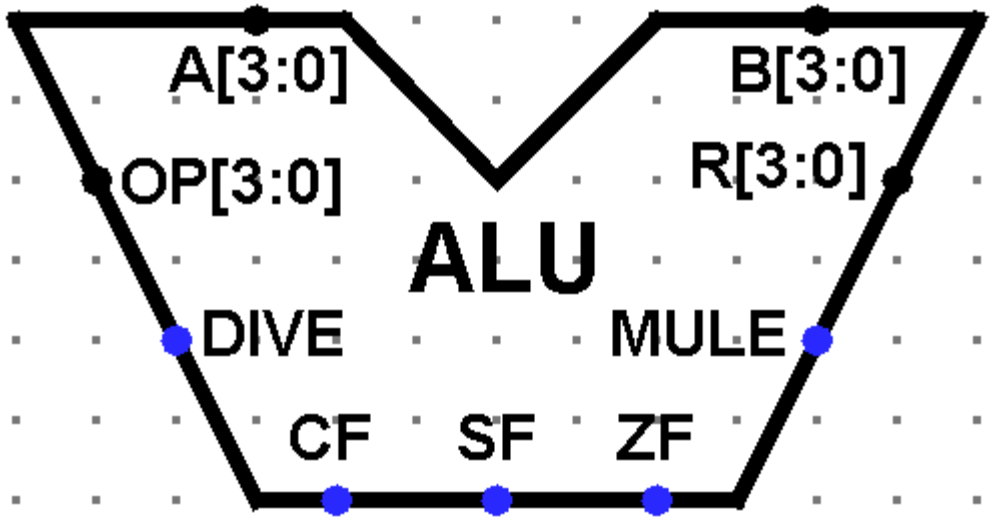


Figure: 4-bit ALU where it performs operations on A and B based on OP value.

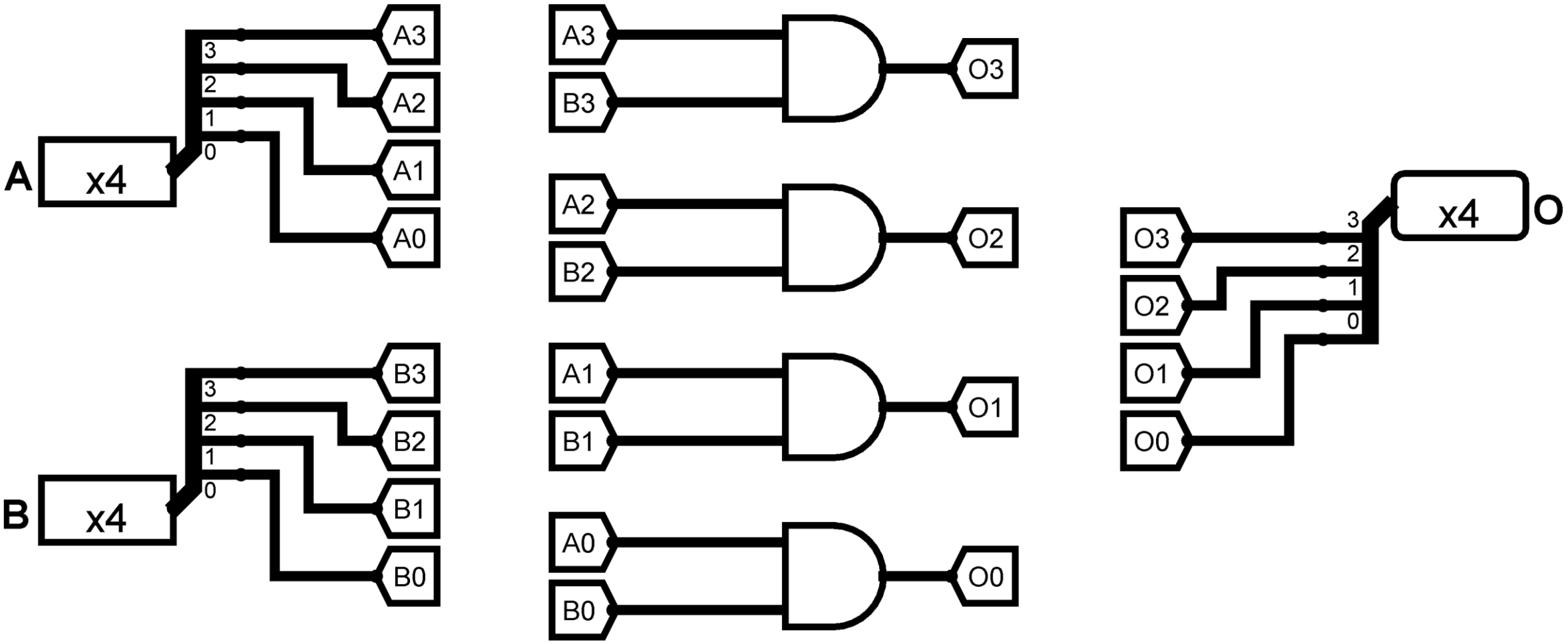
Here,
A = Data1 of ALU
B = Data2 of ALU
OP = Opcode of ALU
R = Result of ALU

CF = Carry Flag
SF = Sign Flag
ZF = Zero Flag

DIVE = DIV Exception
MULE = MUL Exception

Logic Circuits
AND Gate
OR Gate
XOR Gate
NOT Gate

4-bit AND gate



4-bit AND Gate Simulation

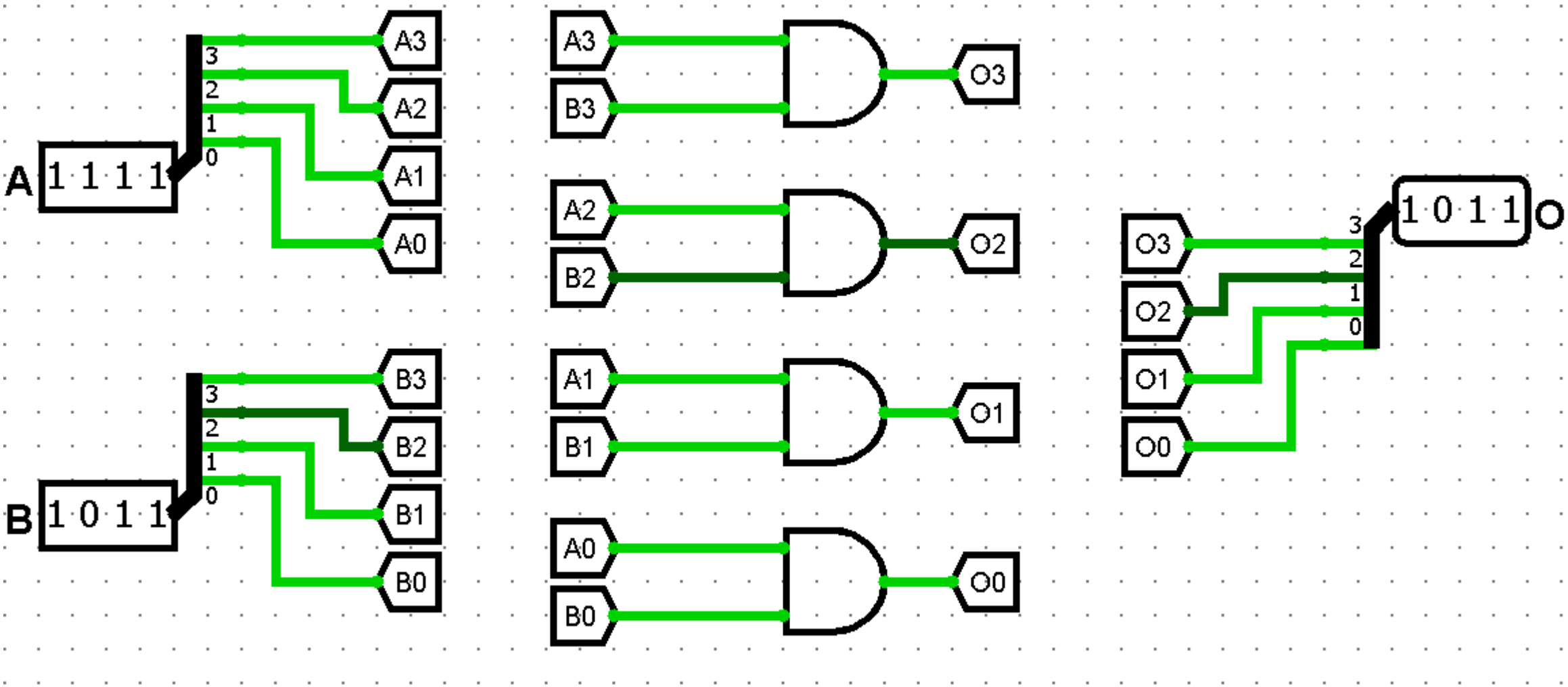
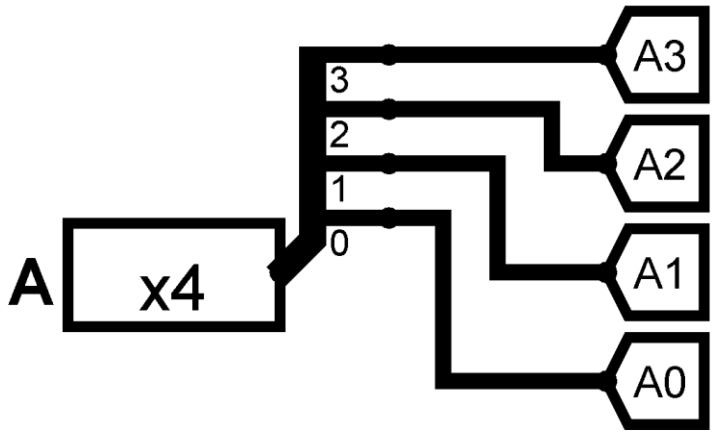
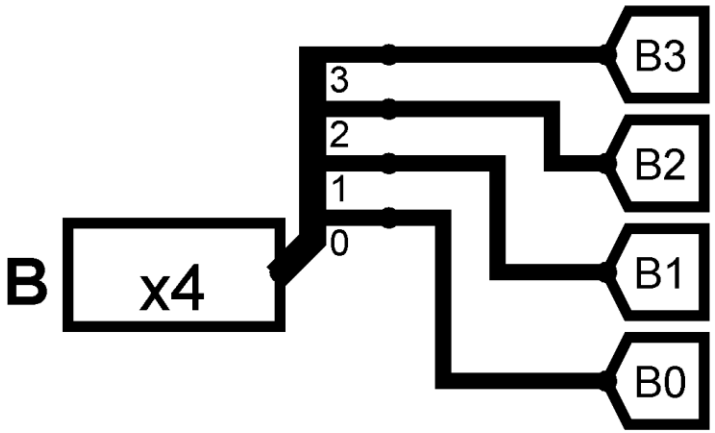
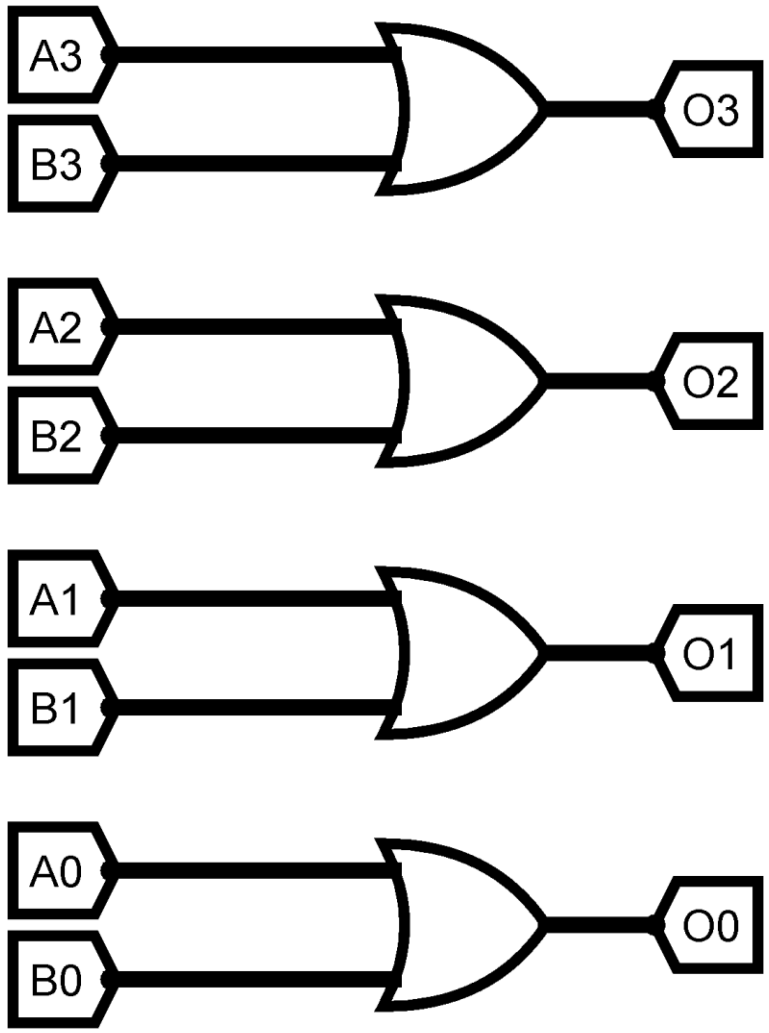


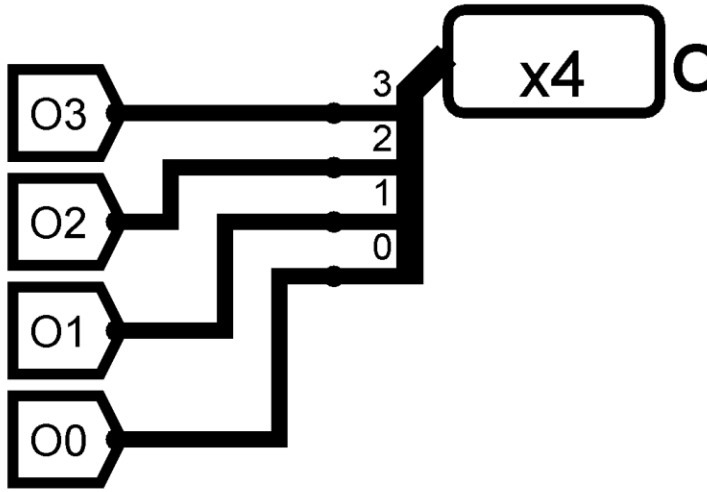
Figure: 4-bit AND Gate Simulation for input A = 1111 and B = 1011.
Output is O = 1011

4-bit OR gate


A
x4
3
2
1
0
A3
A2
A1
A0


B
x4
3
2
1
0
B3
B2
B1
B0


A3
B3
O3
A2
B2
O2
A1
B1
O1
A0
B0
O0


O3
O2
O1
O0
3
2
1
0
x4
O

4-bit OR Gate Simulation

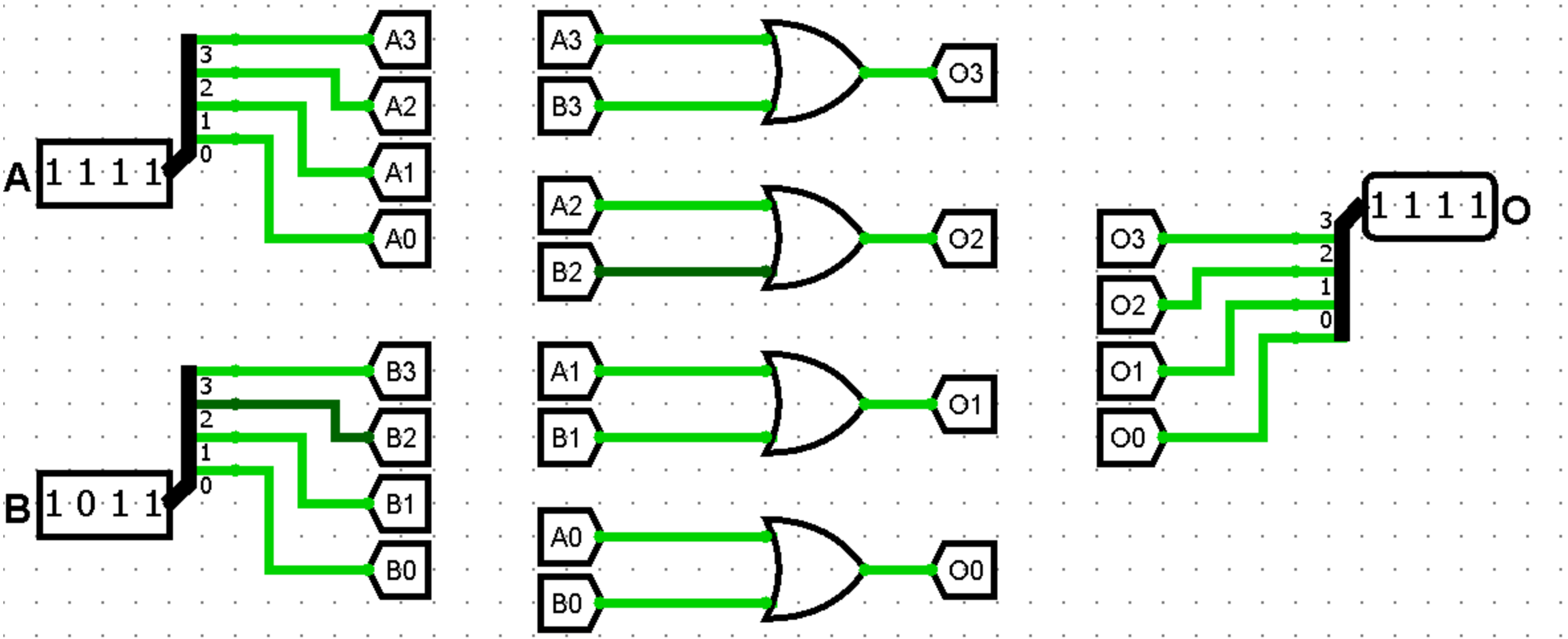
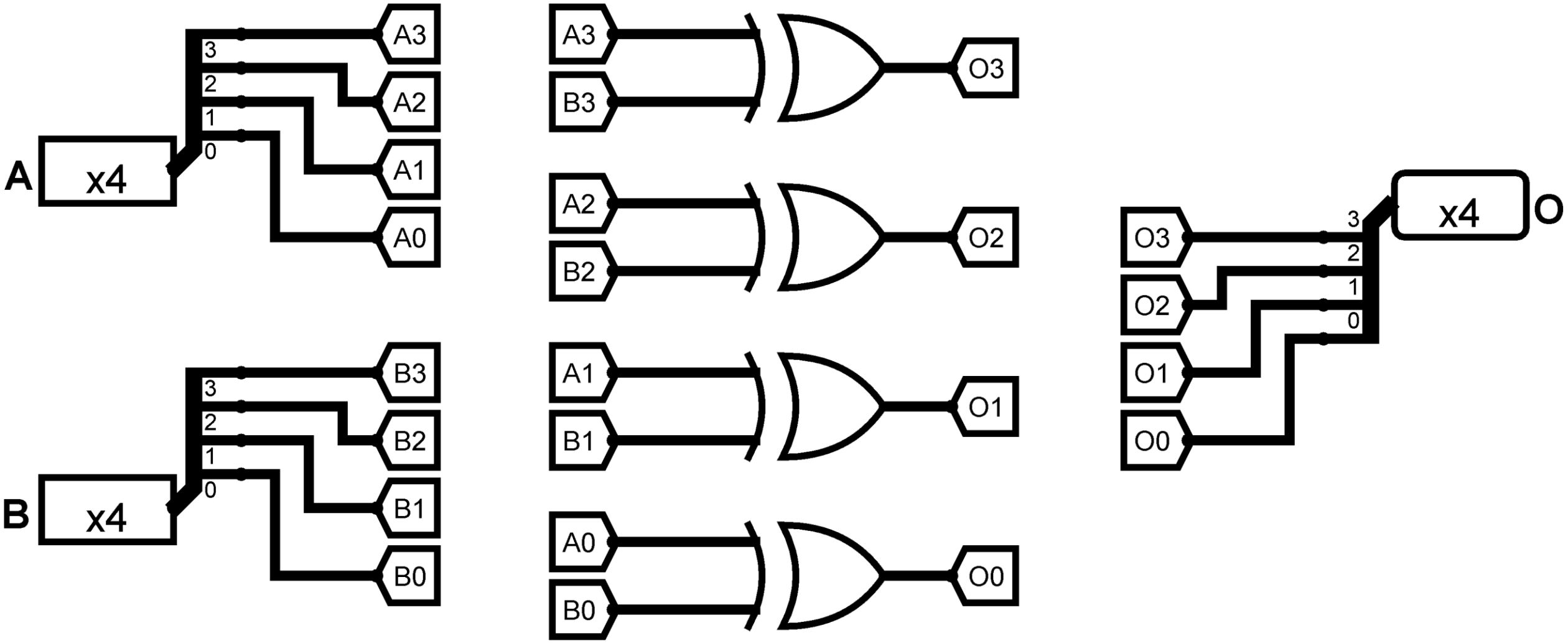


Figure: 4-bit OR Gate Simulation for input A = 1111 and B = 1011.
Output is O = 1111

4-bit XOR gate



4-bit XOR Gate Simulation

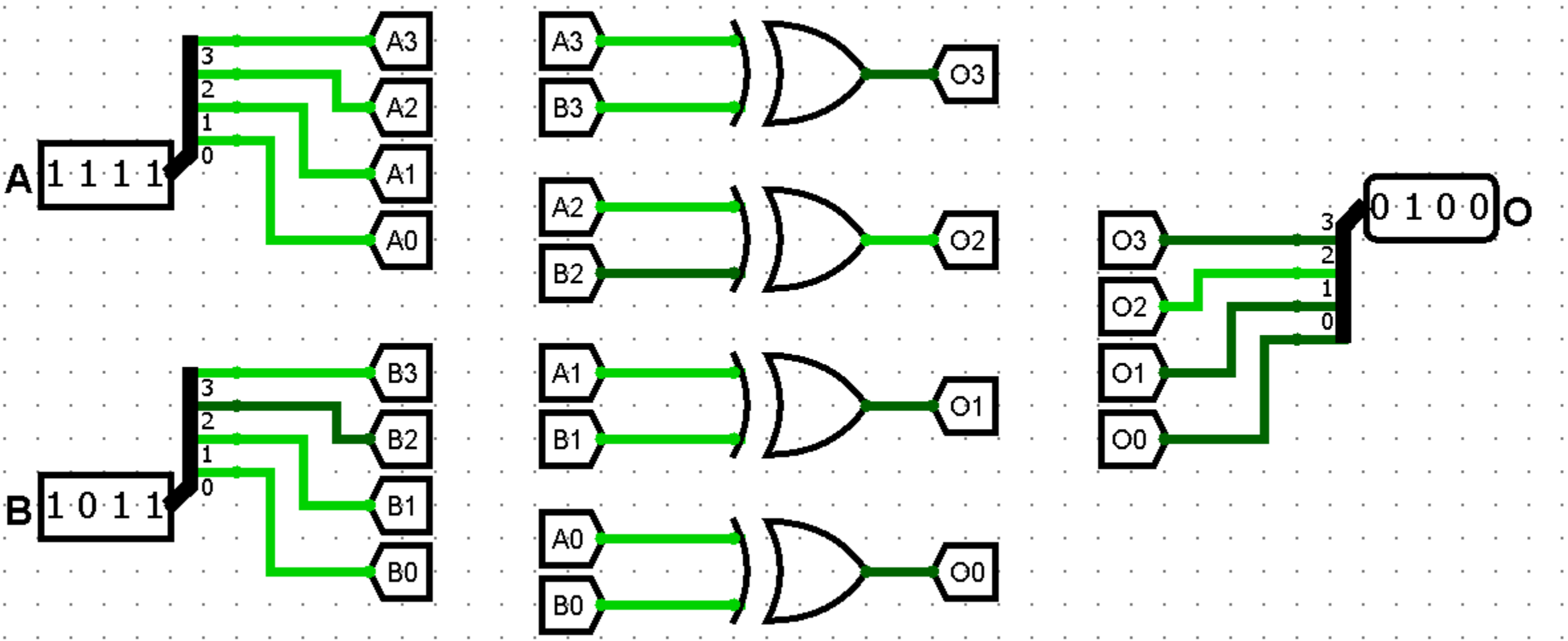
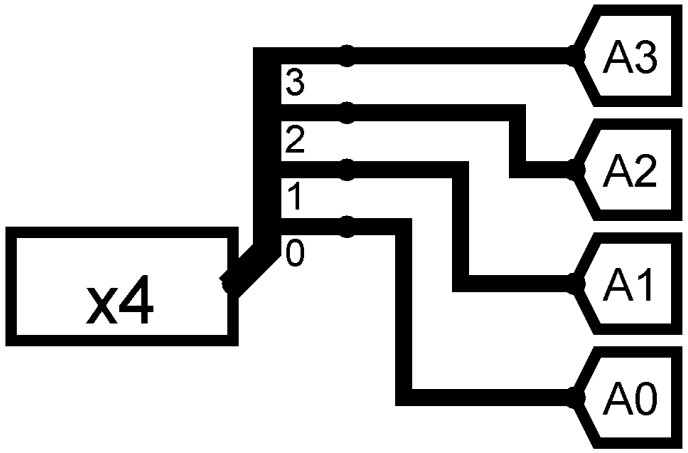
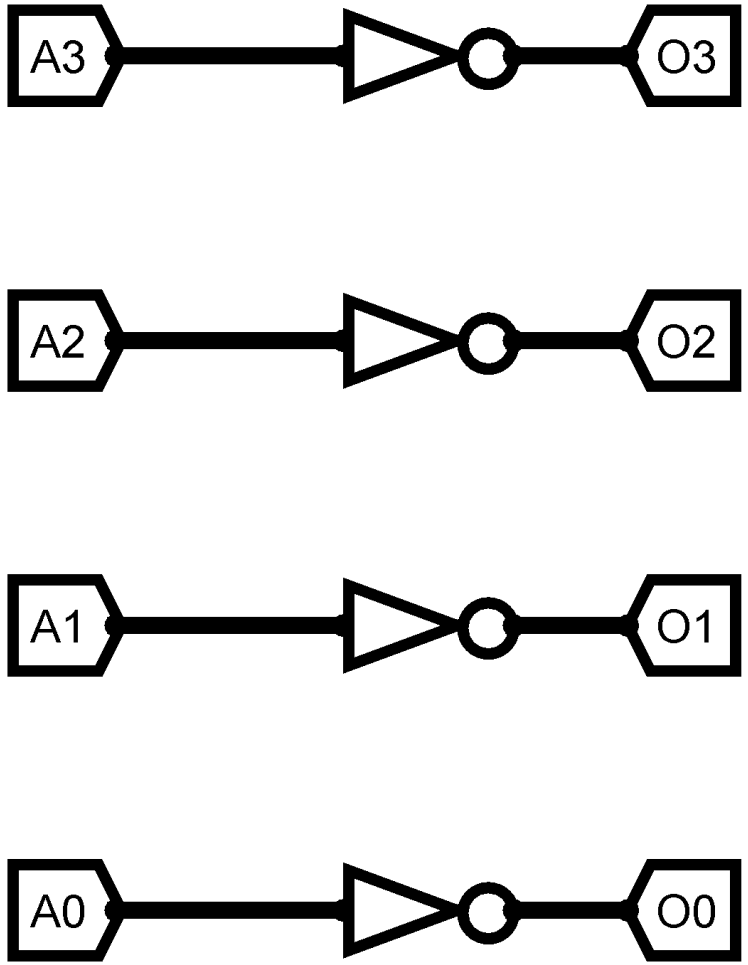
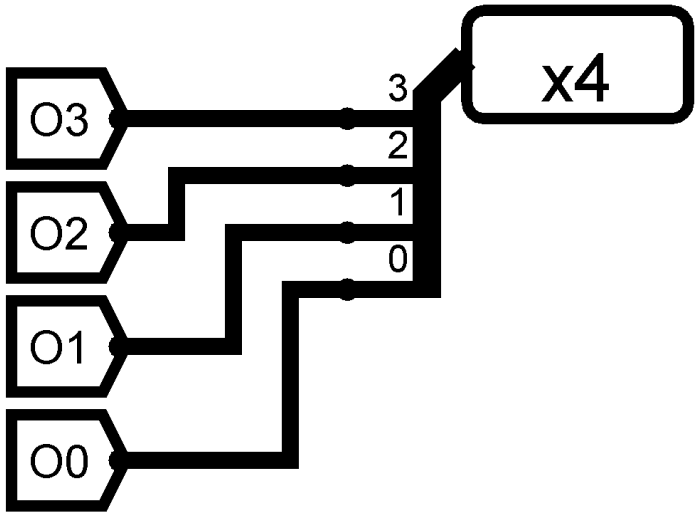


Figure: 4-bit XOR Gate Simulation for input A = 1111 and B = 1011.
Output is O = 0100

4-bit NOT Gate







4-bit NOT Gate Simulation

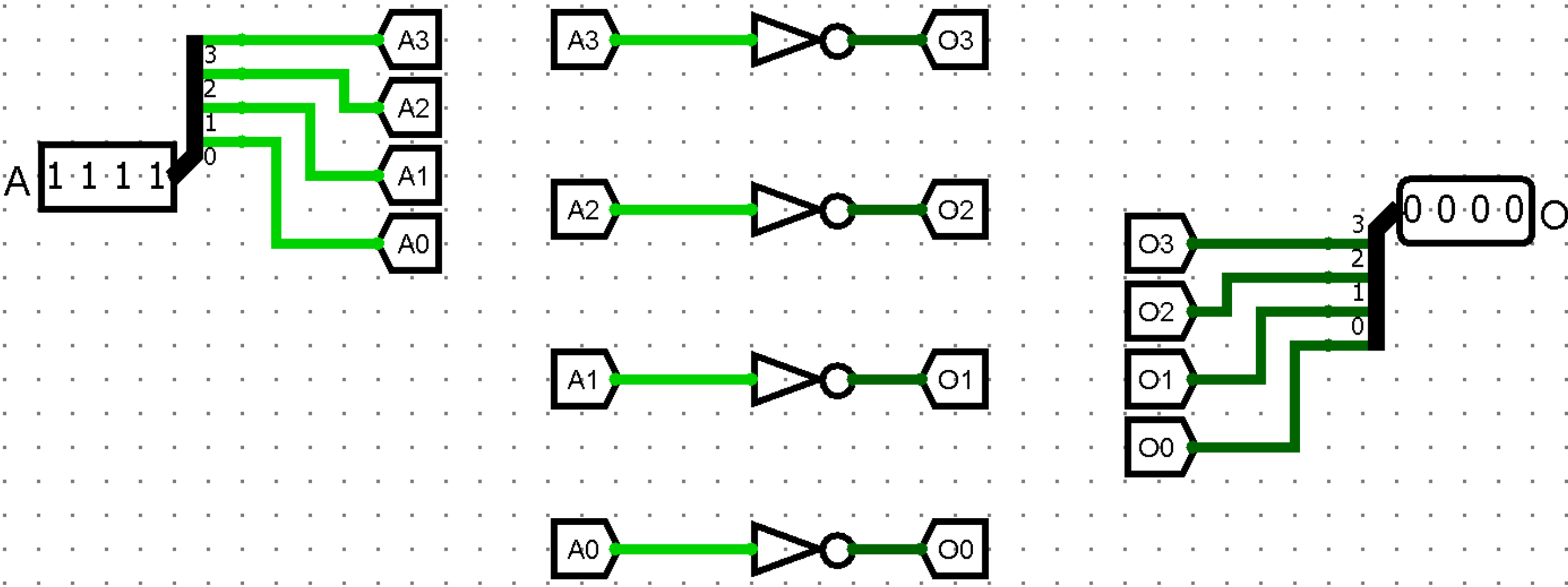
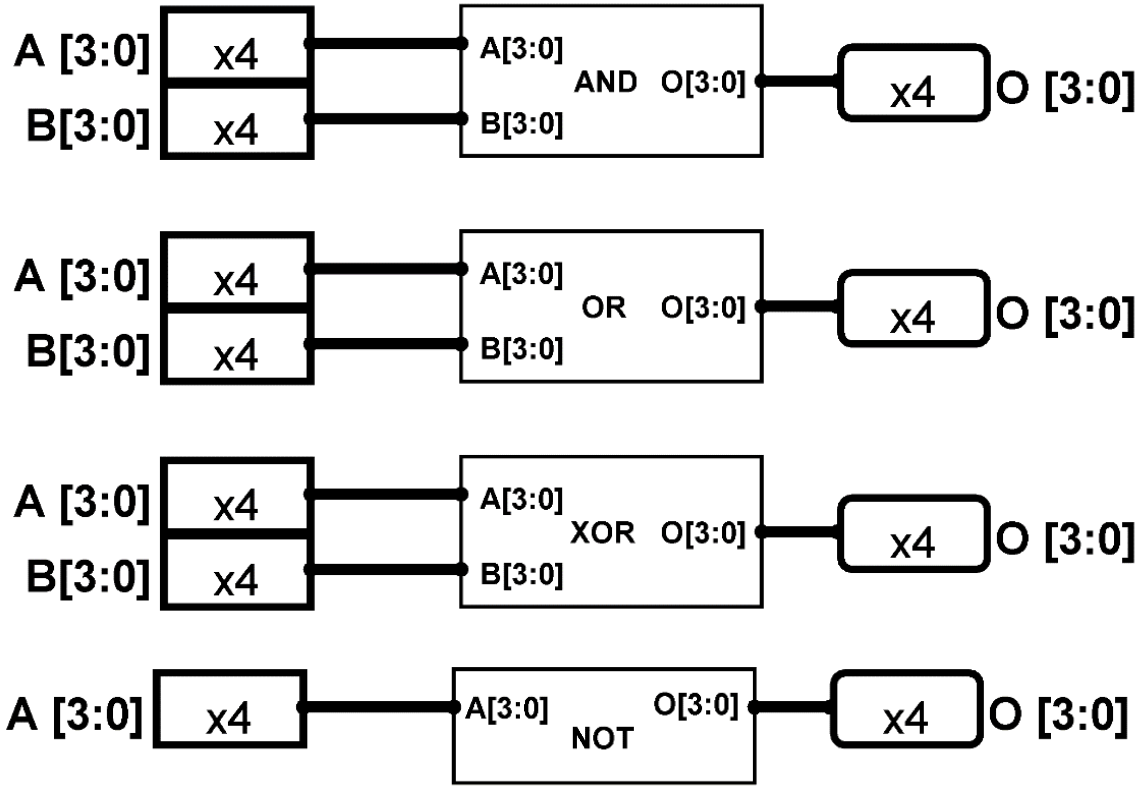


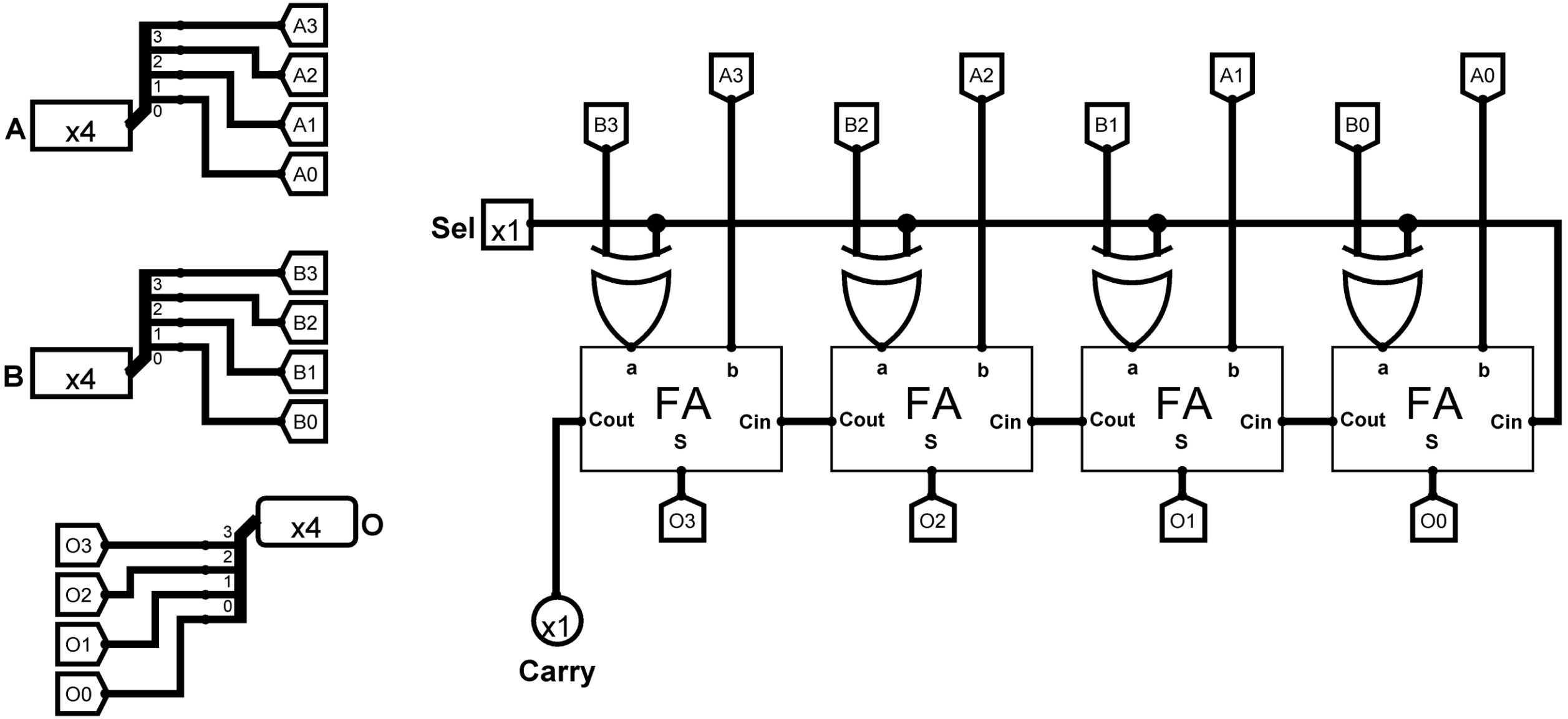
Figure: 4-bit NOT Gate Simulation for input A = 1111. Output is O = 0000

Logic Circuits

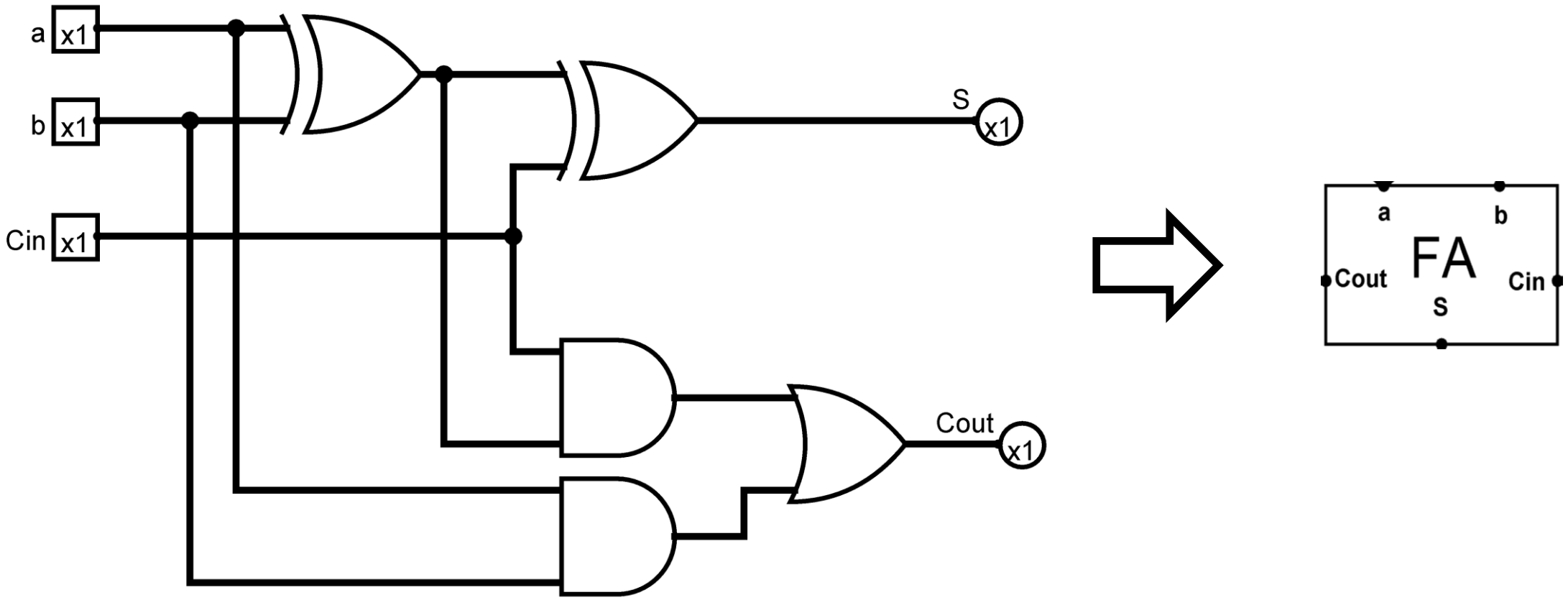


Arithmetic Circuits
Adder
Subtractor
Multiplier
Divider

4-bit Adder/Subtractor



4-bit Adder/Subtractor



4-bit Adder/Subtractor Simulation

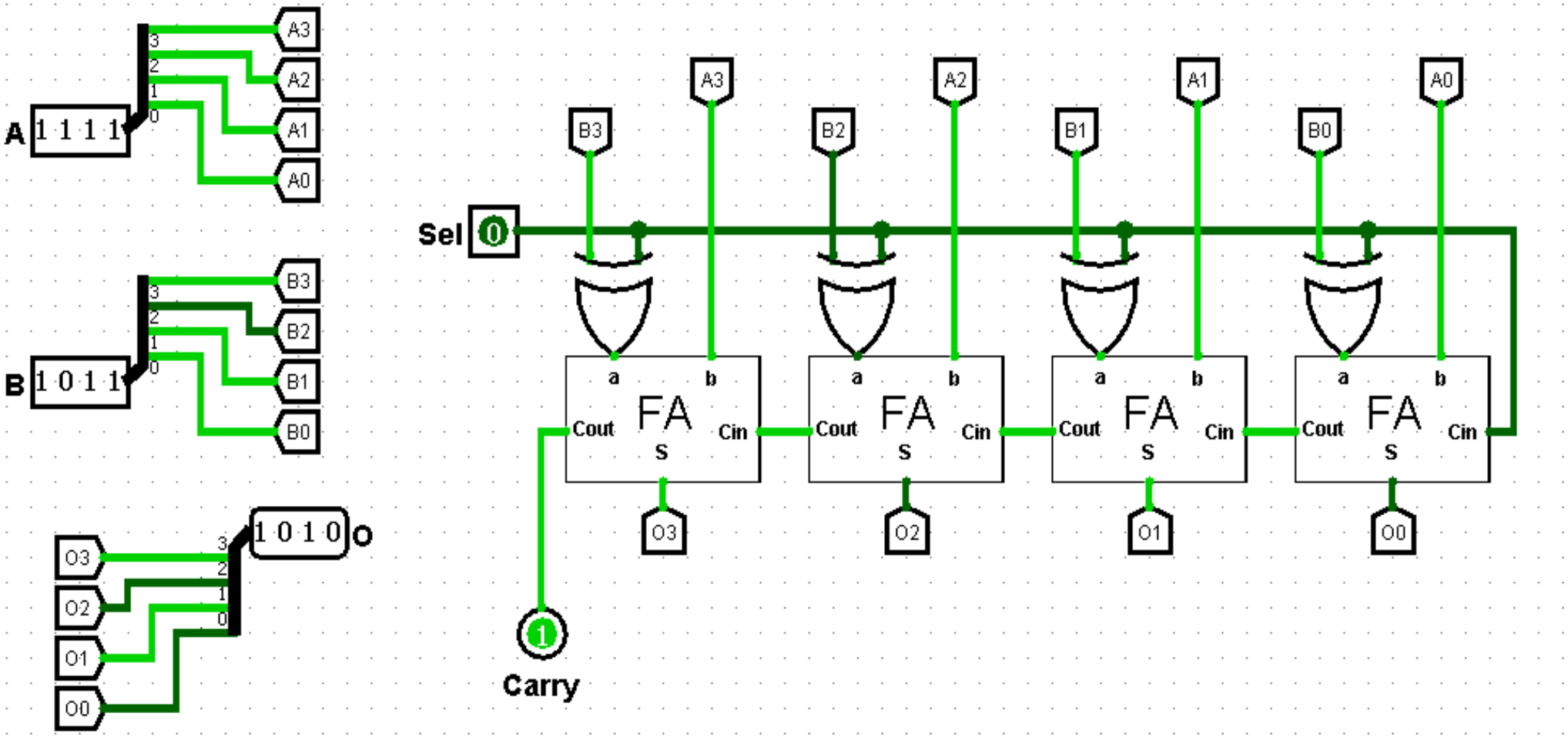


Figure: 4-bit Adder/Subtractor Simulation for input A = 1111, B = 1011, Sel = 0 (ADD). Output O = 1010 and Carry = 1

4-bit Adder/Subtractor Simulation

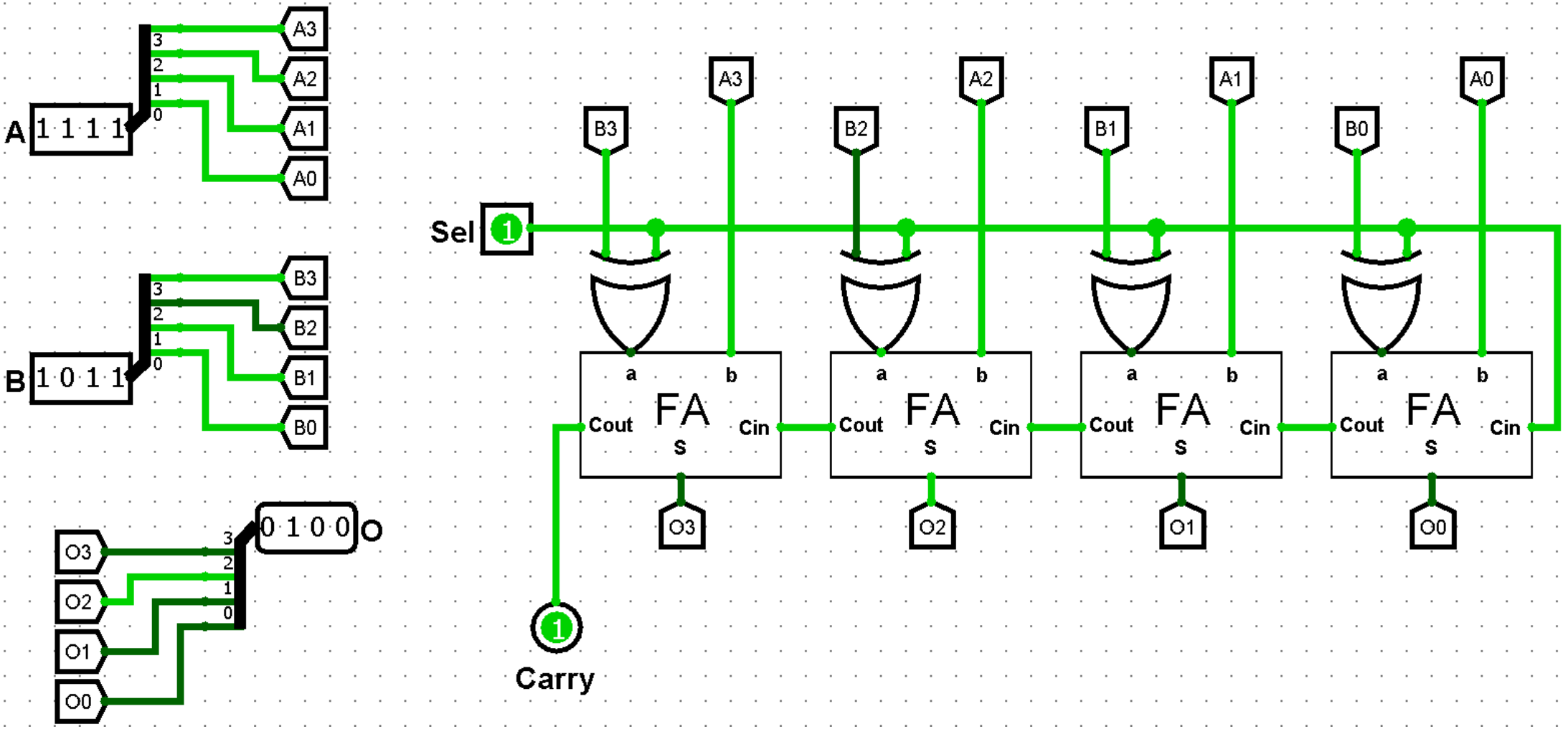
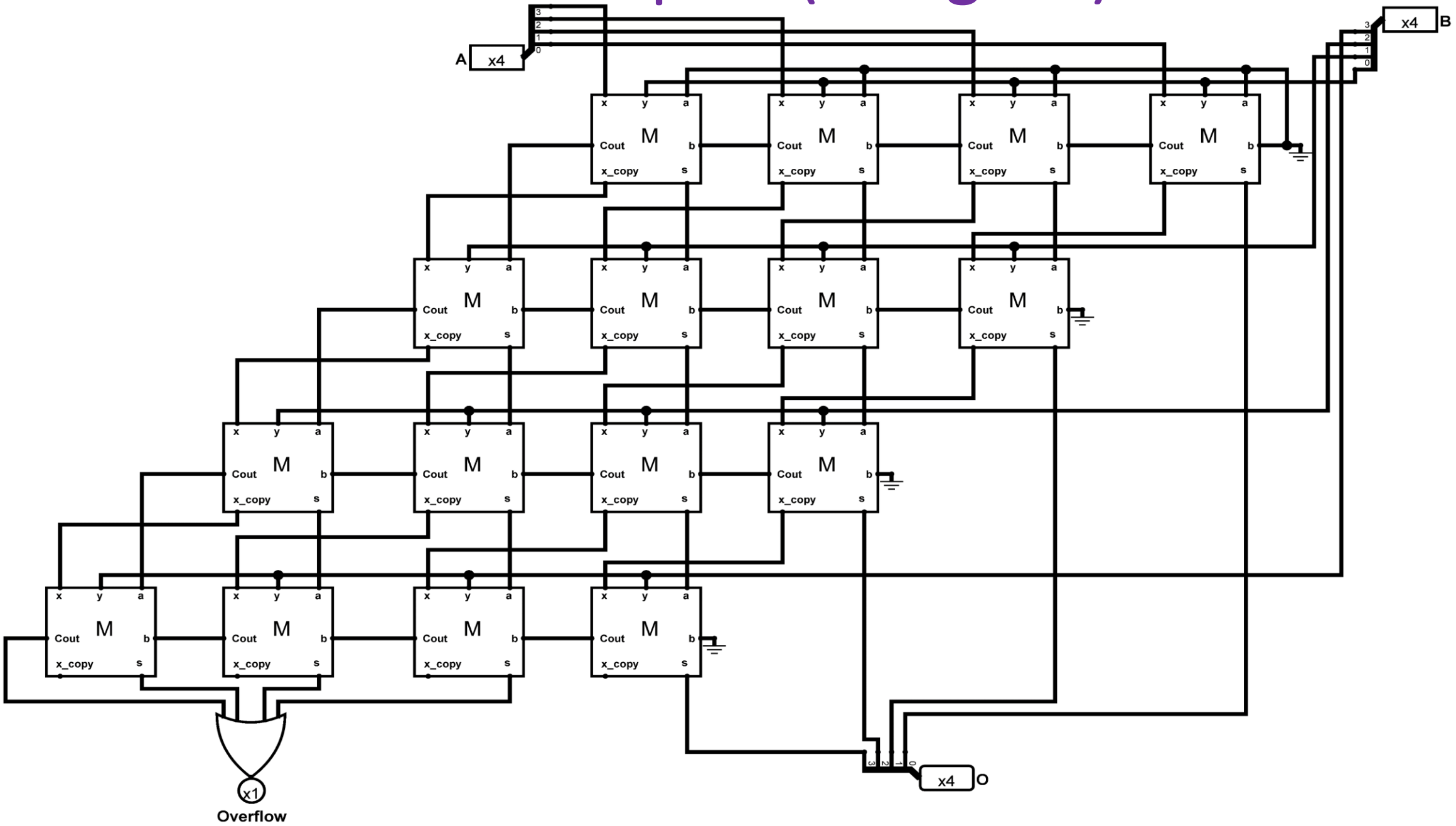


Figure: 4-bit Adder/Subtractor Simulation for input A = 1111, B = 1011, Sel = 0 (SUB). Output O = 0100 and Carry = 1

4-bit Multiplier (Unsigned)



4-bit Multiplier (Unsigned) Simulation

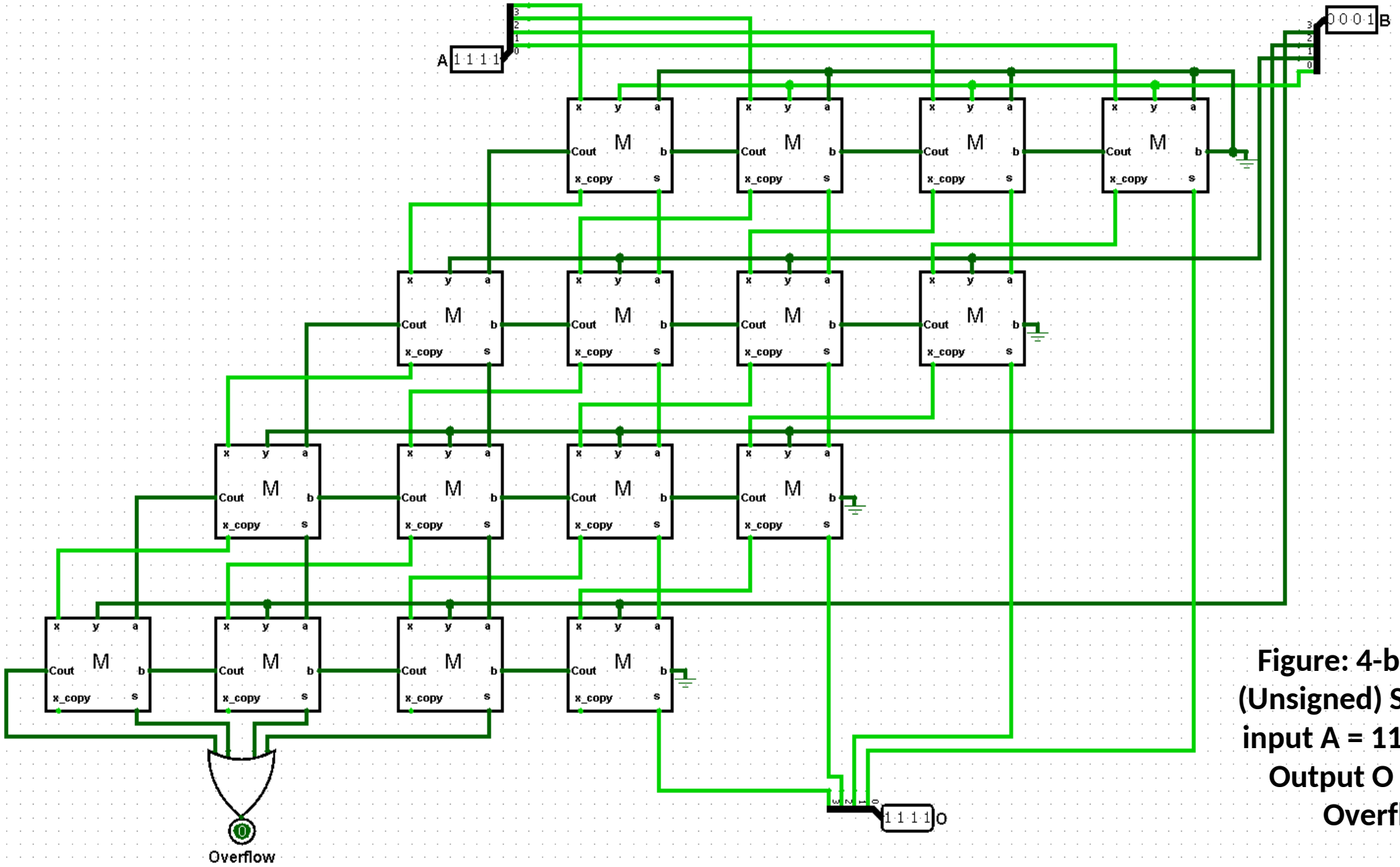
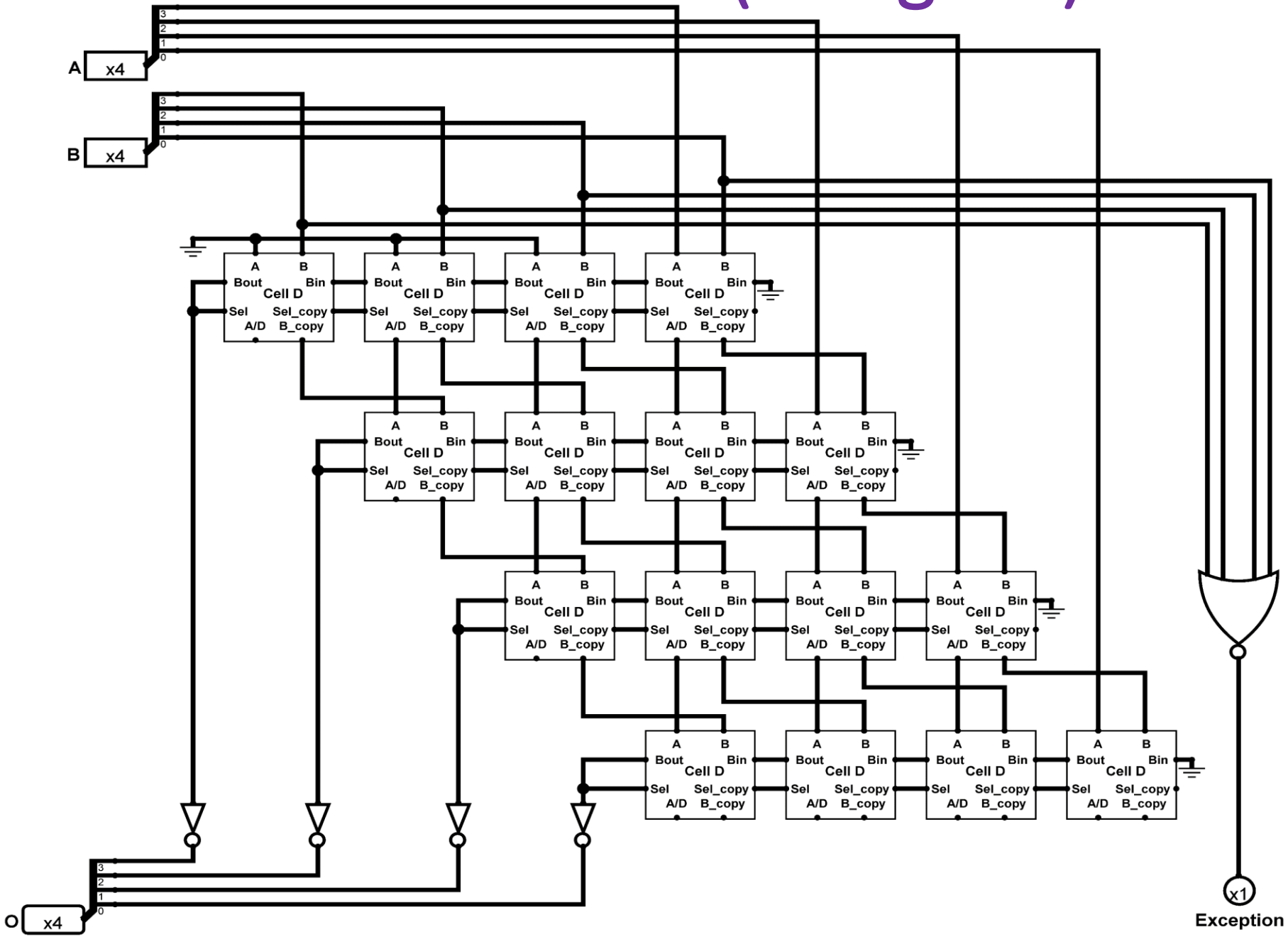


Figure: 4-bit Multiplier (Unsigned) Simulation for input A = 1111, B = 0001. Output O = 1111 and Overflow = 0

4-bit Divider (Unsigned)



4-bit Divider (Unsigned) Simulation

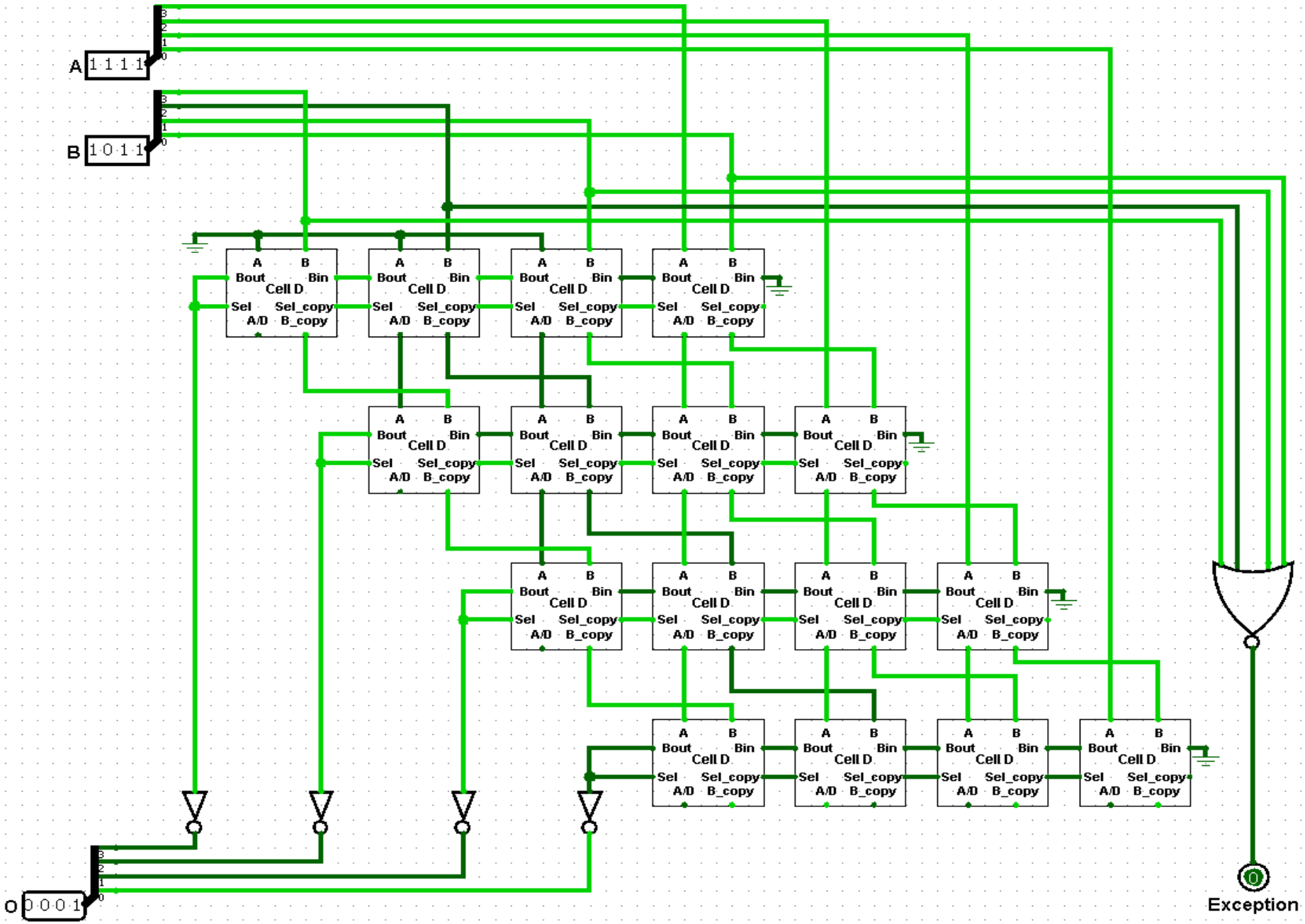
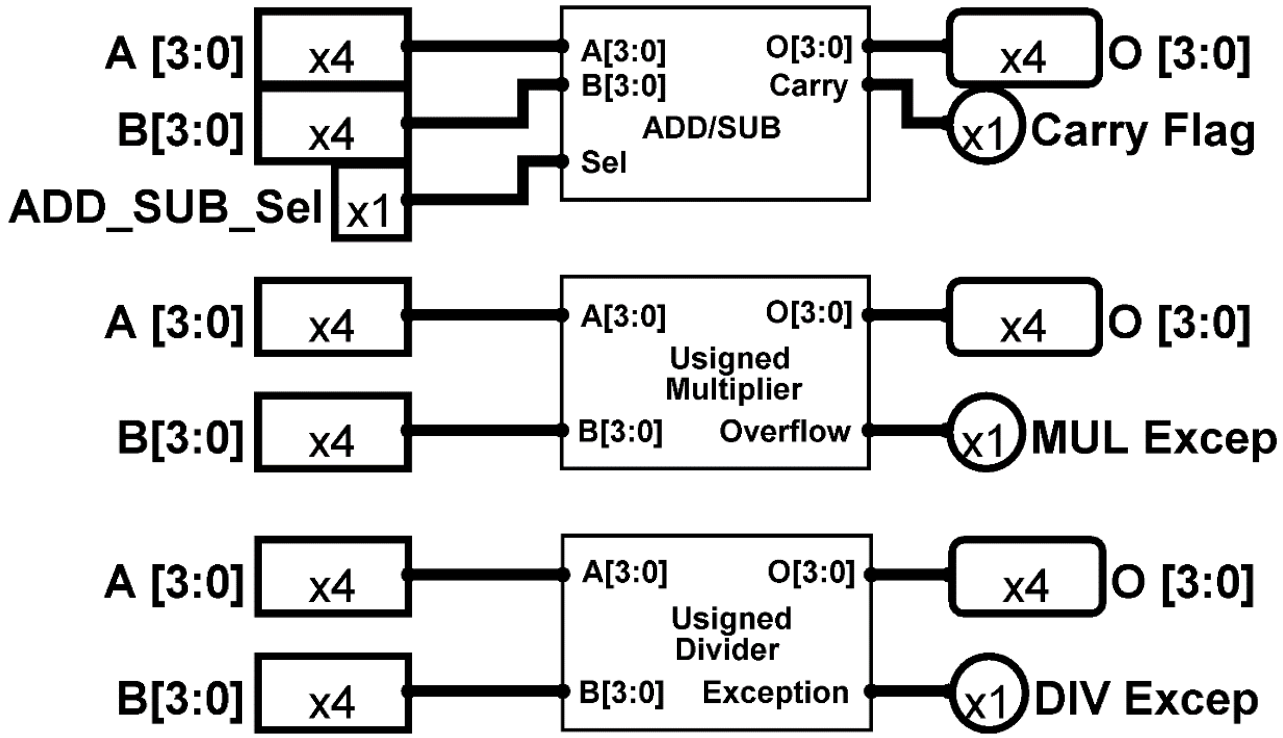


Figure: 4-bit Divider (Unsigned) Simulation for input A = 1111, B = 1011. Output O = 0001 and Exception = 0

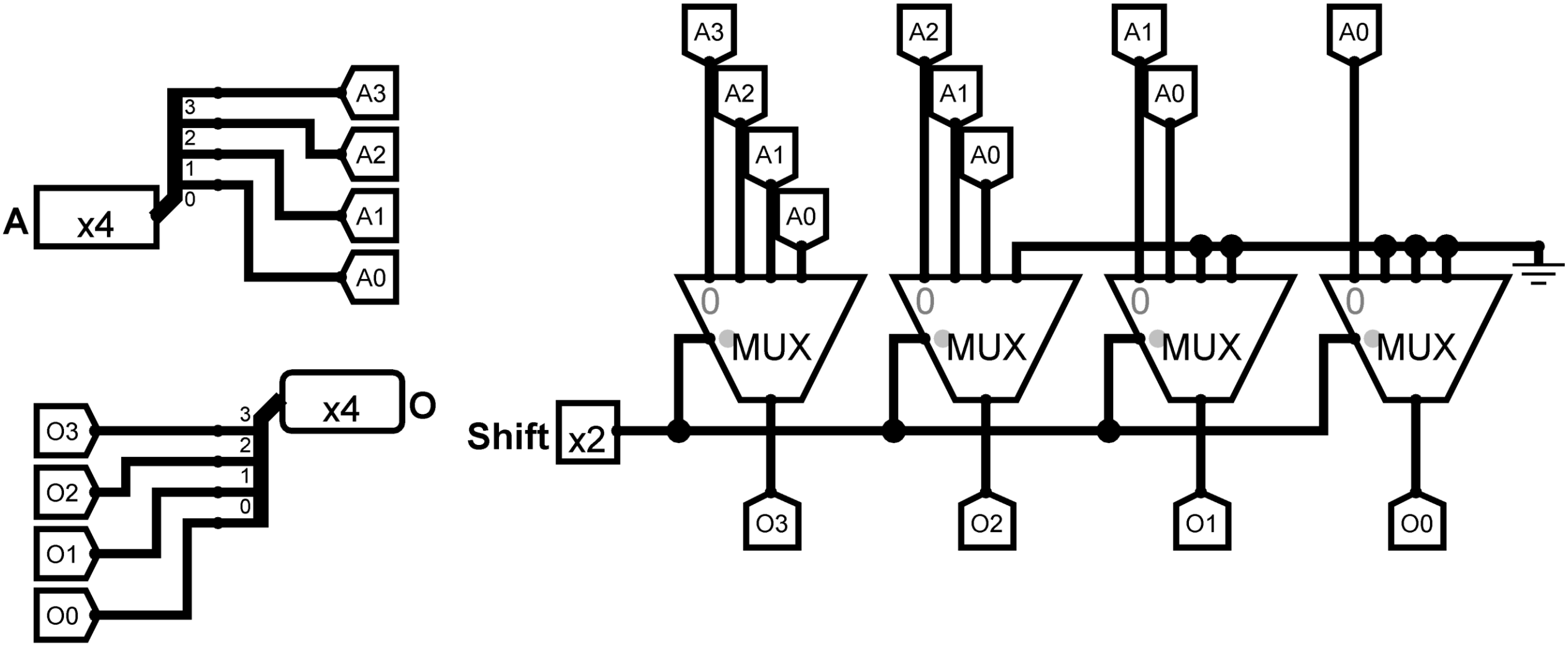
Arithmetic Circuits Block

If $\mathbf{ADD\_SUB\_Sel = 0,\ O = ADD}$
If $\mathbf{ADD\_SUB\_Sel = 1,\ O = SUB}$

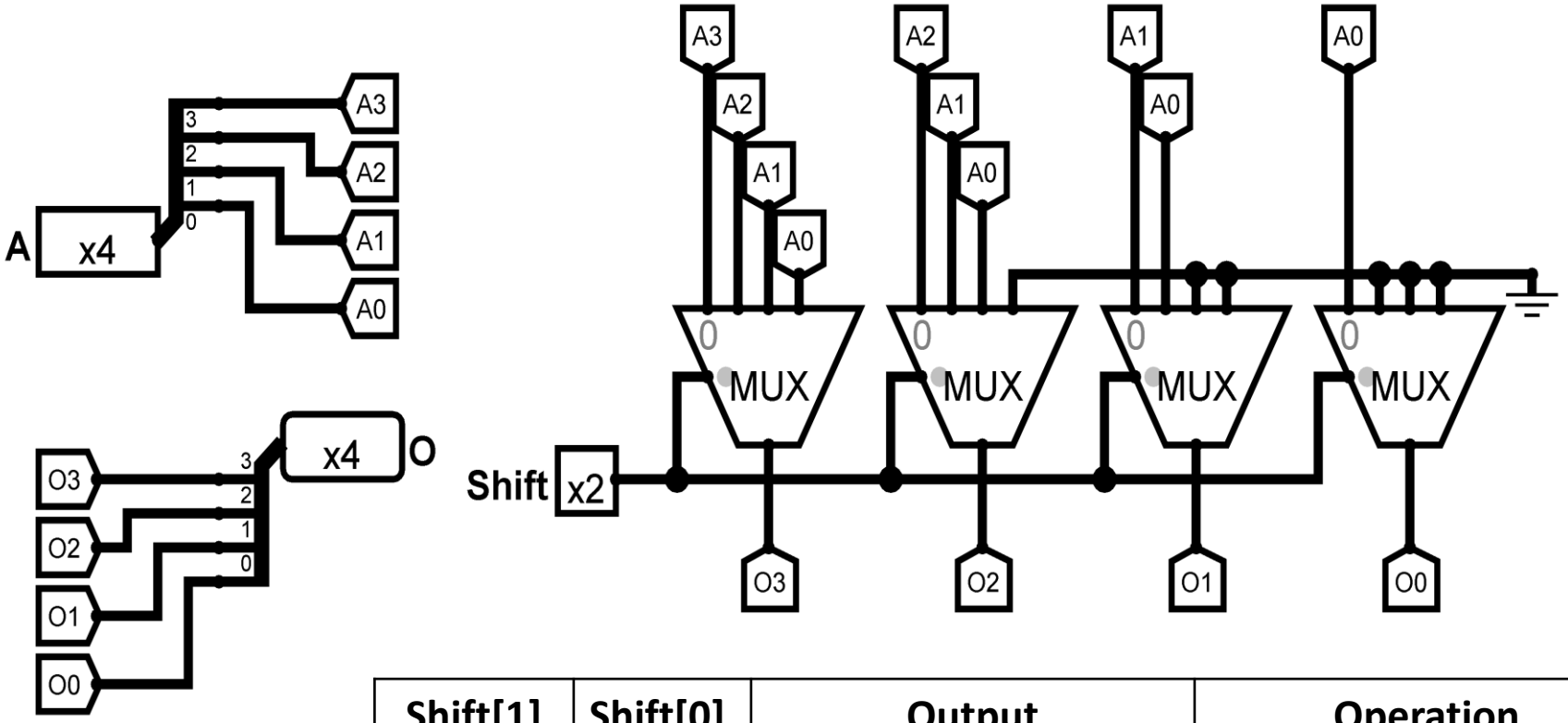


Shifter and Rotate Circuits
Shifter
Rotate (Barrel Shifter)

4-bit Left Shifter



4-bit Left Shifter



Shift[1]	Shift[0]	Output	Operation
0	0	**A[3]A[2]A[1]A[0]**	No Shift
0	1	**A[2]A[1]A[0]0**	1 bit Left Shift
1	0	**A[1]A[0]00**	2 bit Left Shift
1	1	**A[0]000**	3 bit Left Shift

4-bit Left Shifter Simulation

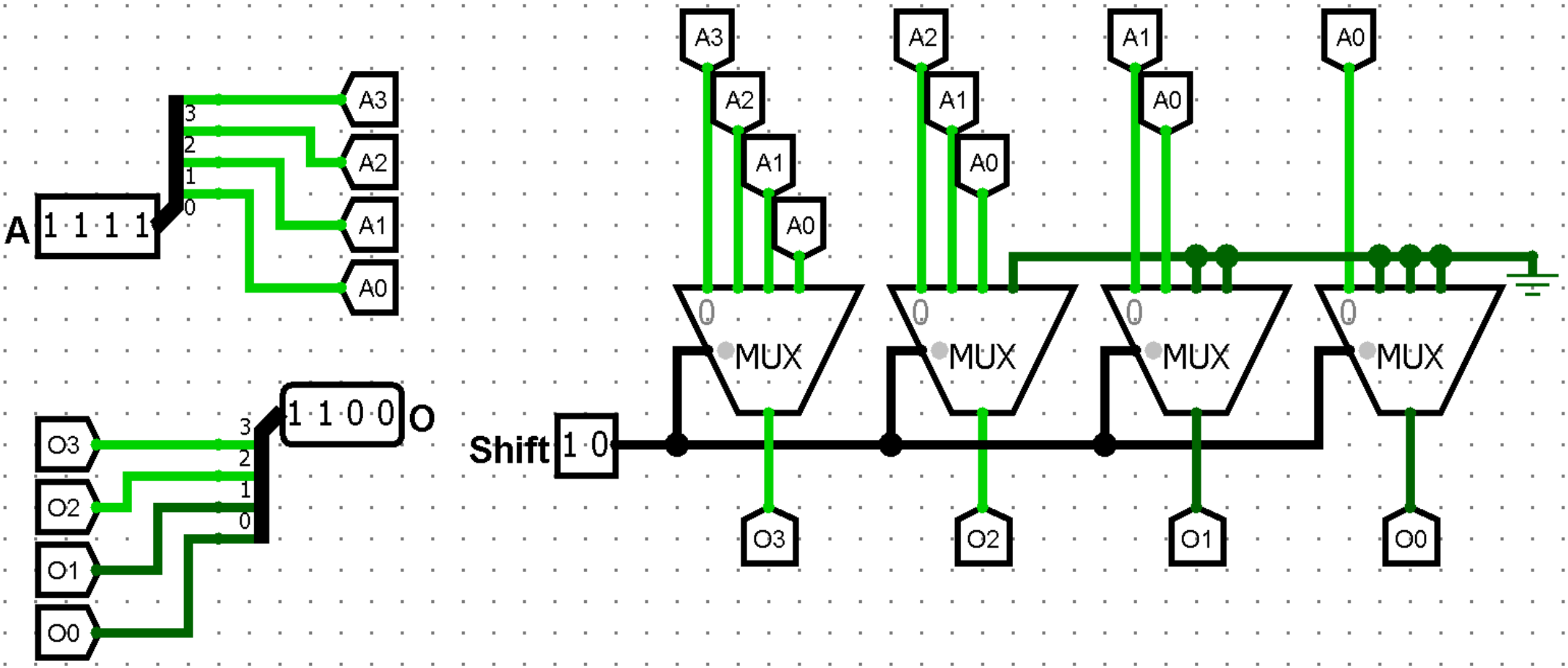
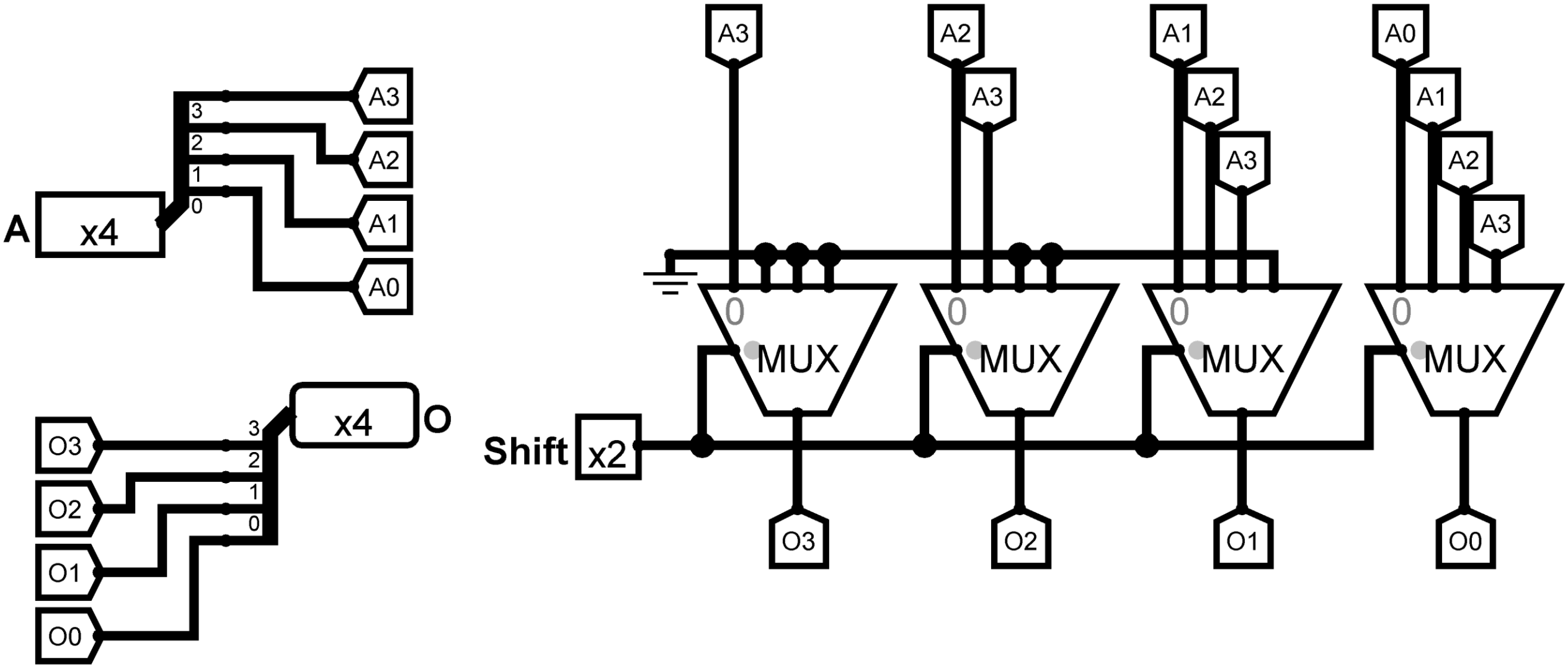
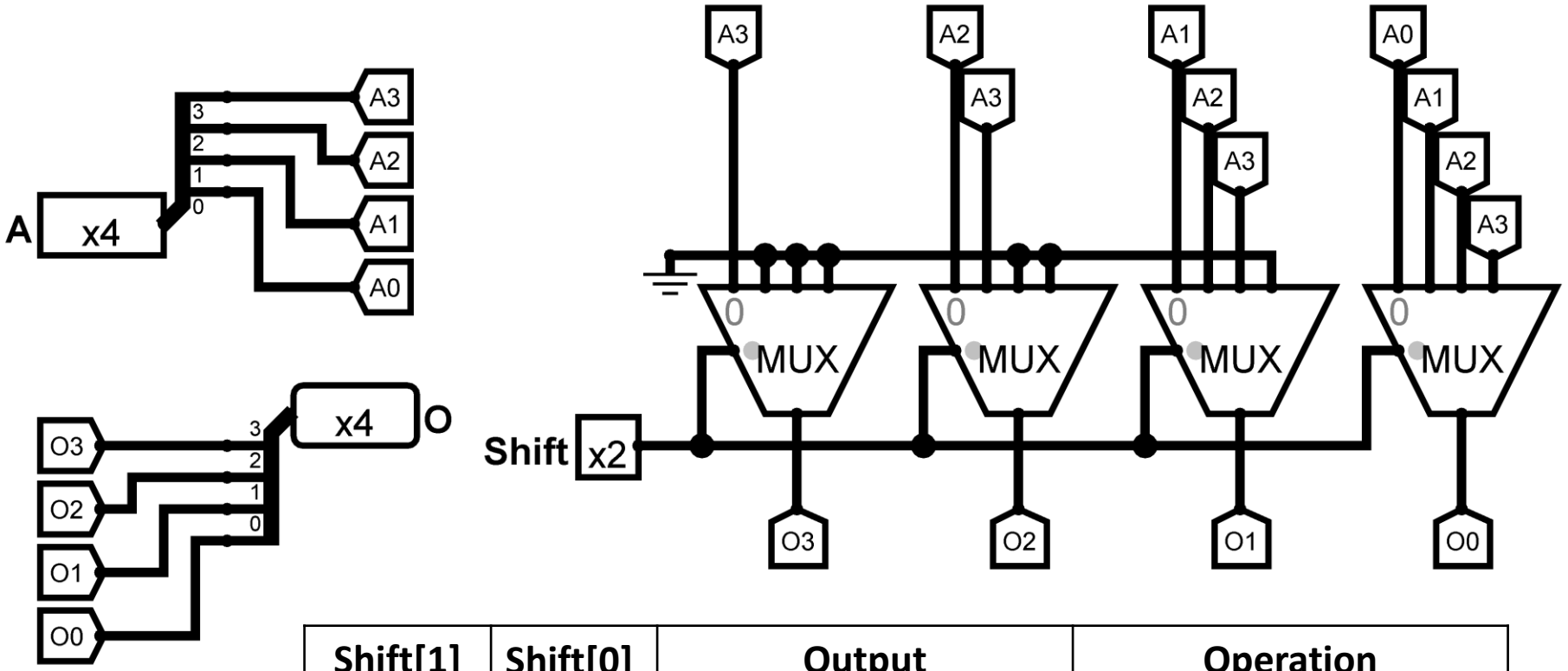


Figure: 4-bit Left Shifter Simulation for input A = 1111, Shift = 10 (2-bit Left Shift).
Output O = 1100

4-bit Right Shifter



4-bit Right Shifter



Shift[1]	Shift[0]	Output	Operation
0	0	**A[3]A[2]A[1]A[0]**	No Shift
0	1	**0A[3]A[2]A[1]**	1 bit Right Shift
1	0	**00A[3]A[2]**	2 bit Right Shift
1	1	**000A[3]**	3 bit Right Shift

4-bit Right Shifter Simulation

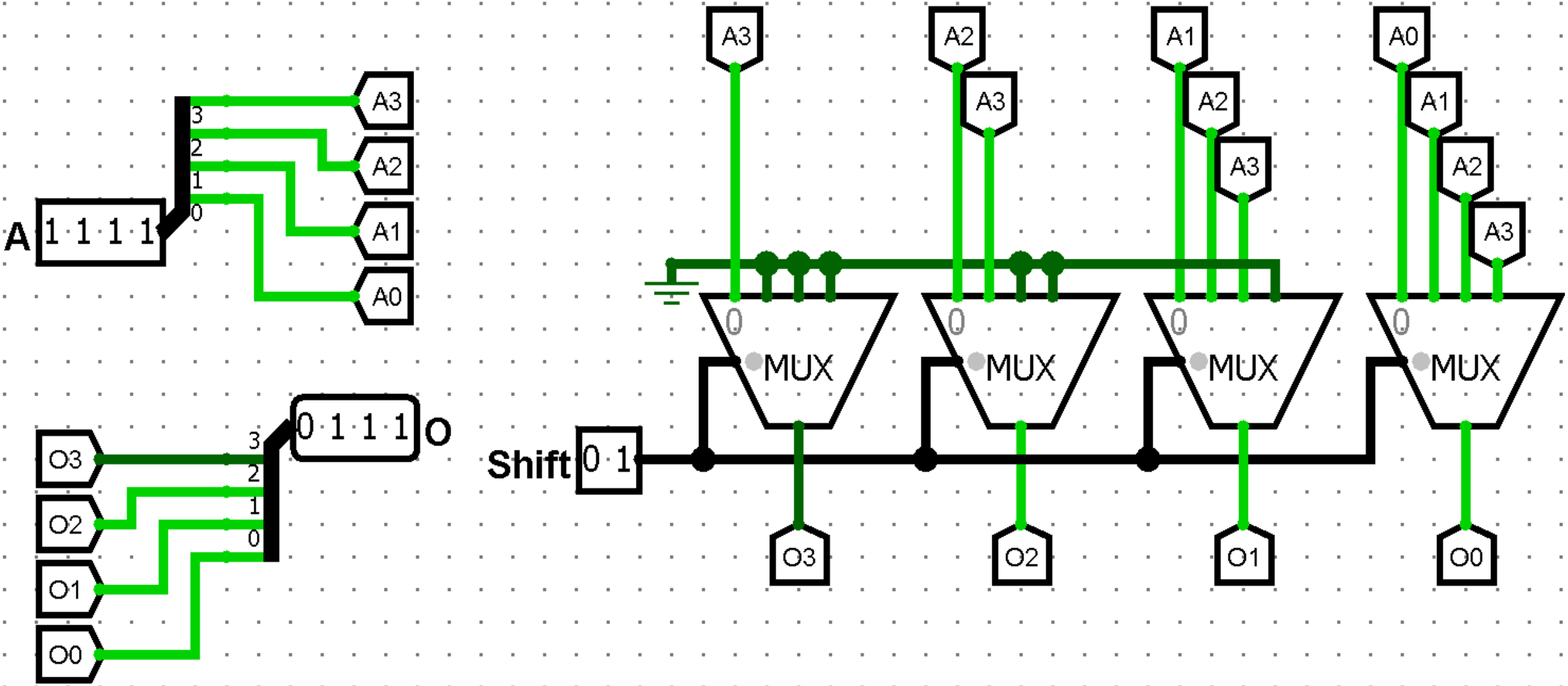
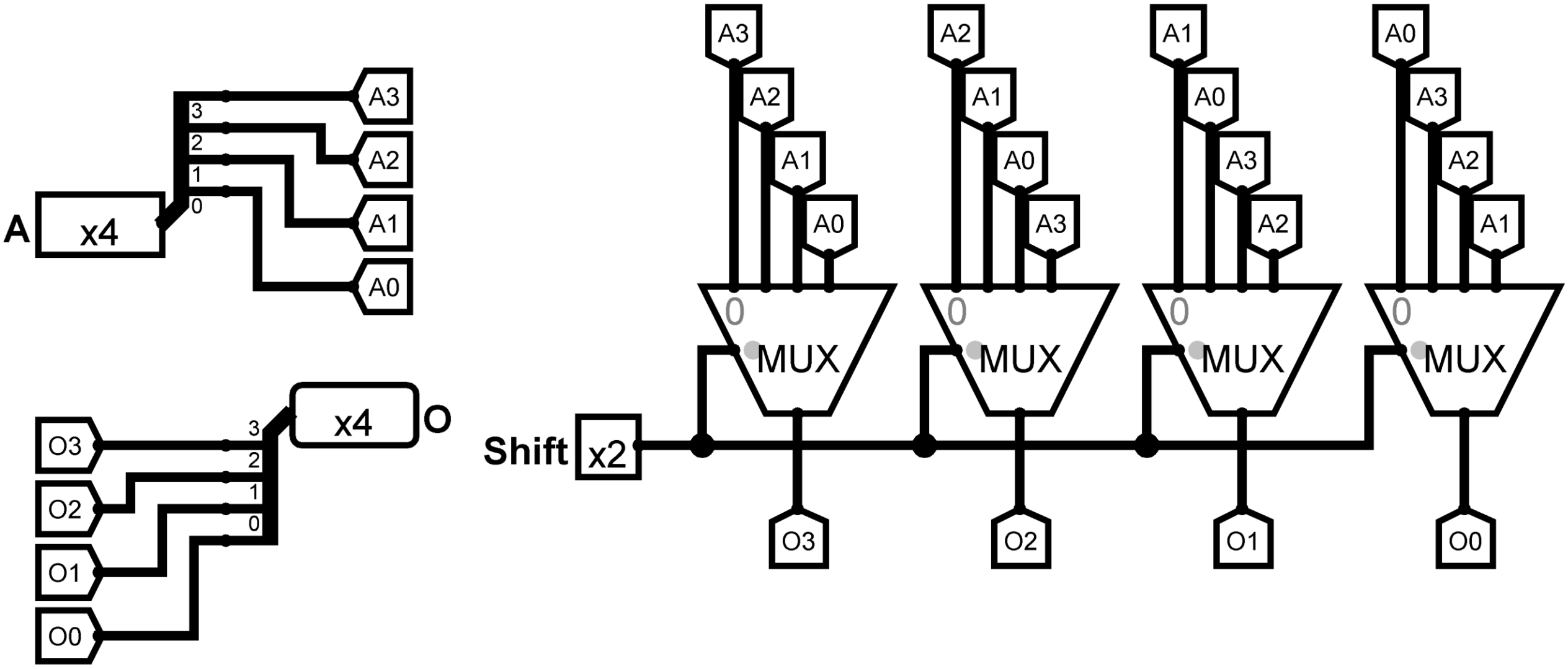
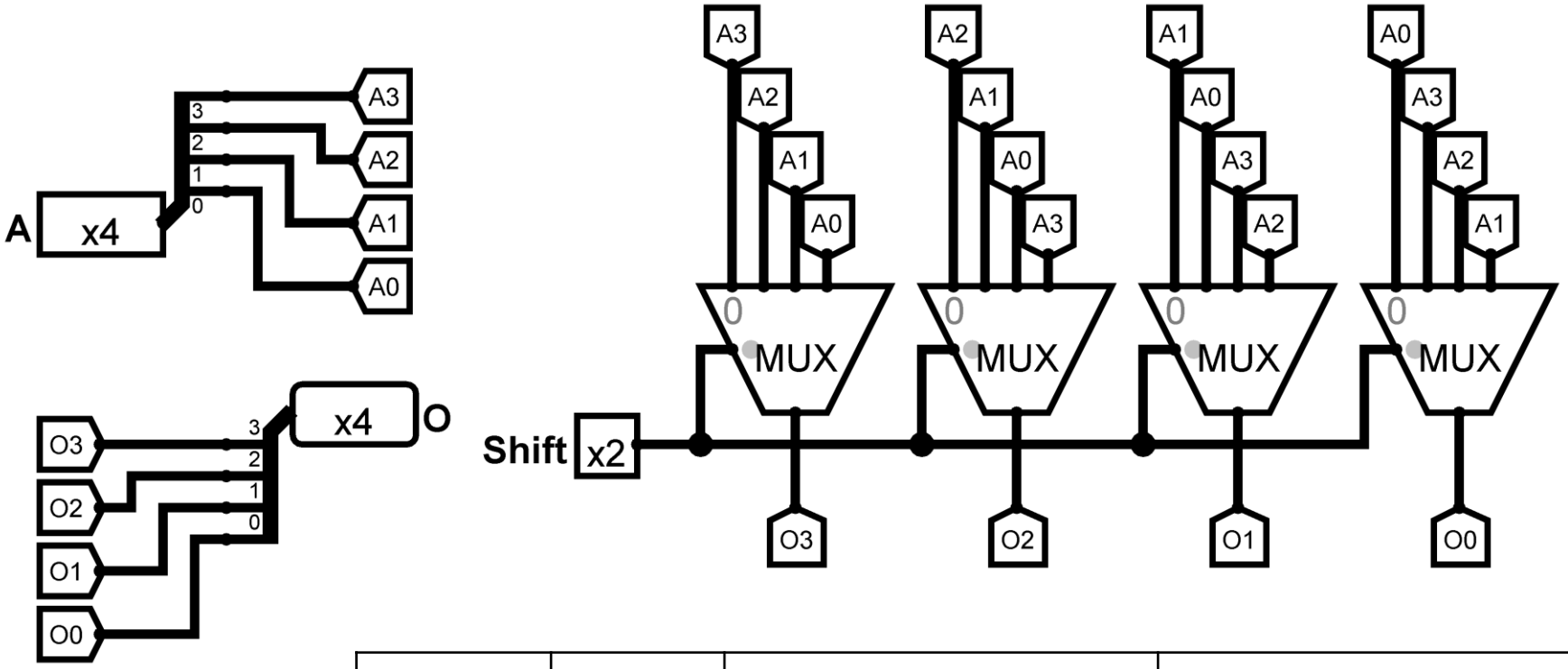


Figure: 4-bit Right Shifter Simulation for input A = 1111, Shift = 01 (1-bit Right Shift).
Output O = 0111

4-bit Left Rotate



4-bit Left Rotate



Shift[1]	Shift[0]	Output	Operation
0	0	**A[3]A[2]A[1]A[0]**	No Rotate
0	1	**A[2]A[1]A[0]A[3]**	1 bit Left Rotate
1	0	**A[1]A[0]A[3]A[2]**	2 bit Left Rotate
1	1	**A[0]A[3]A[2]A[1]**	3 bit Left Rotate

4-bit Left Rotate Simulation

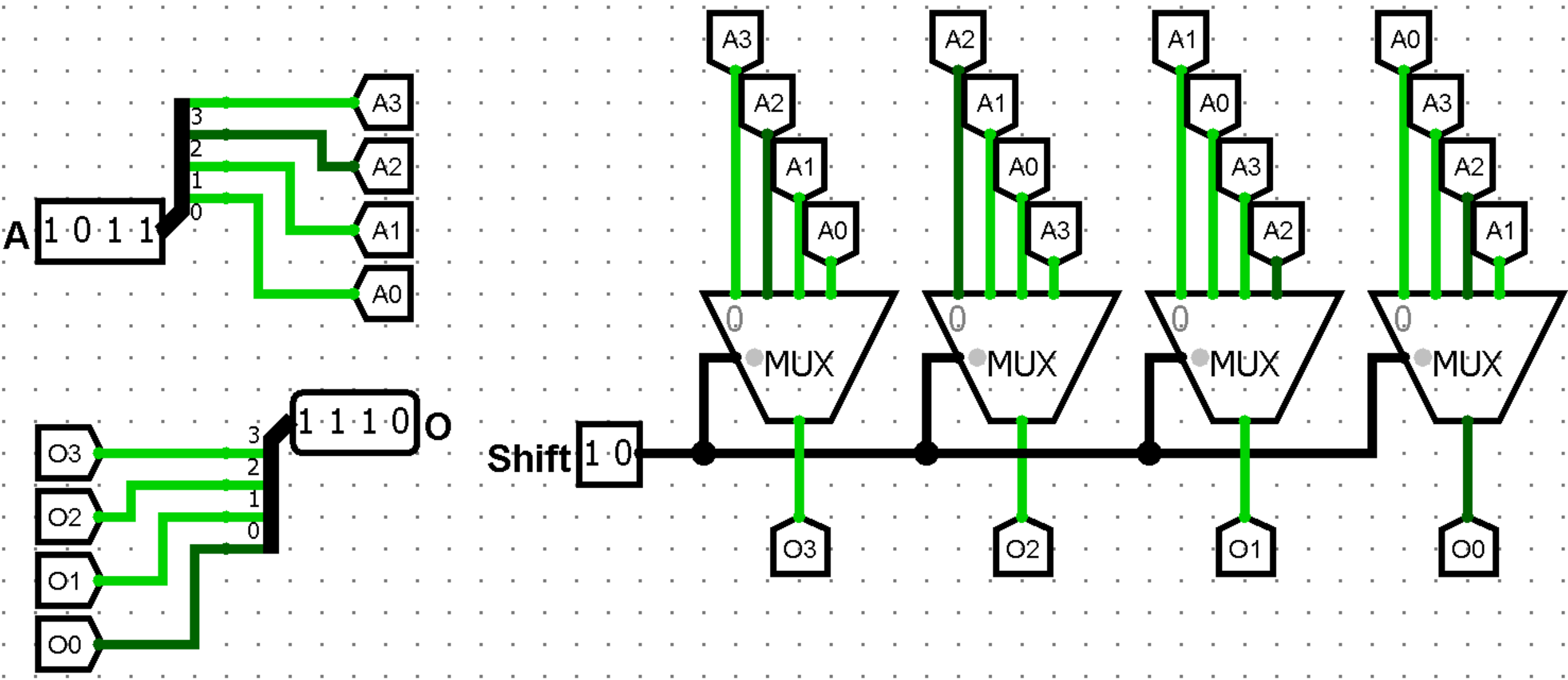
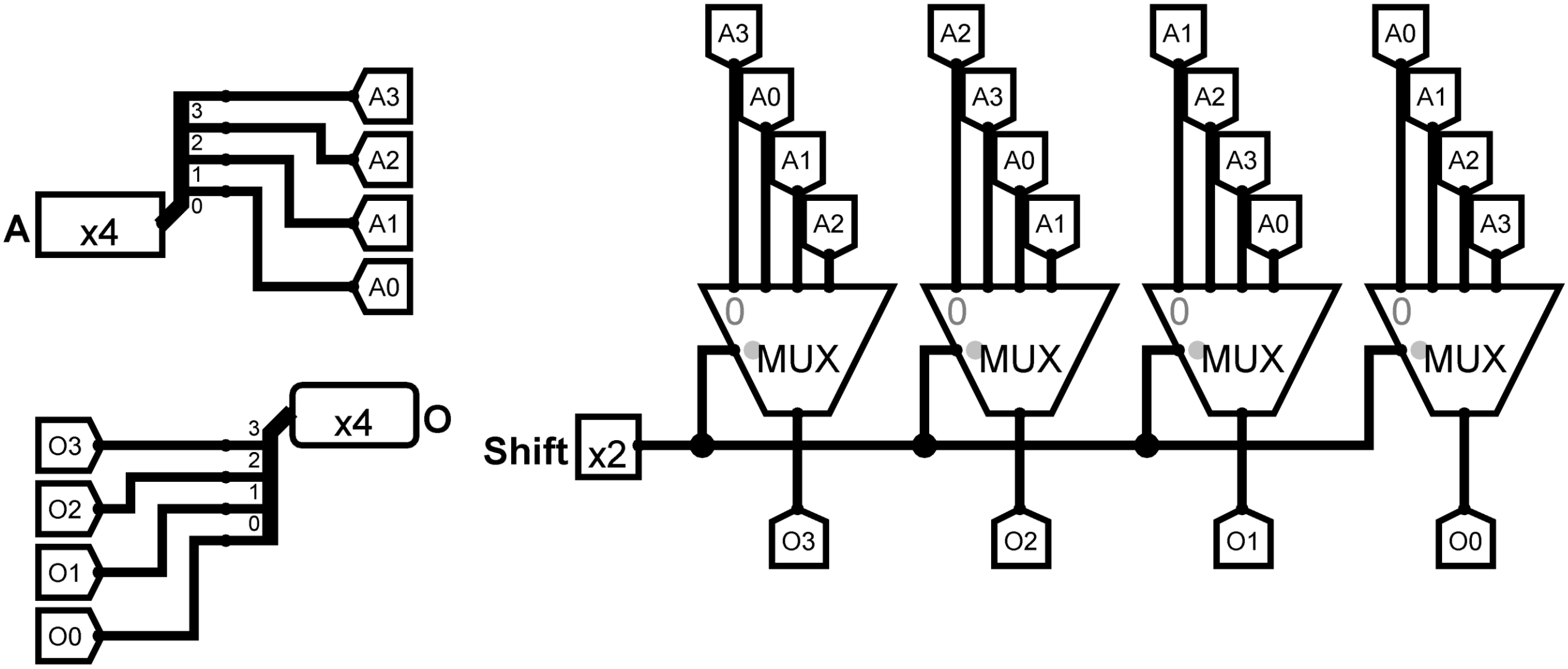
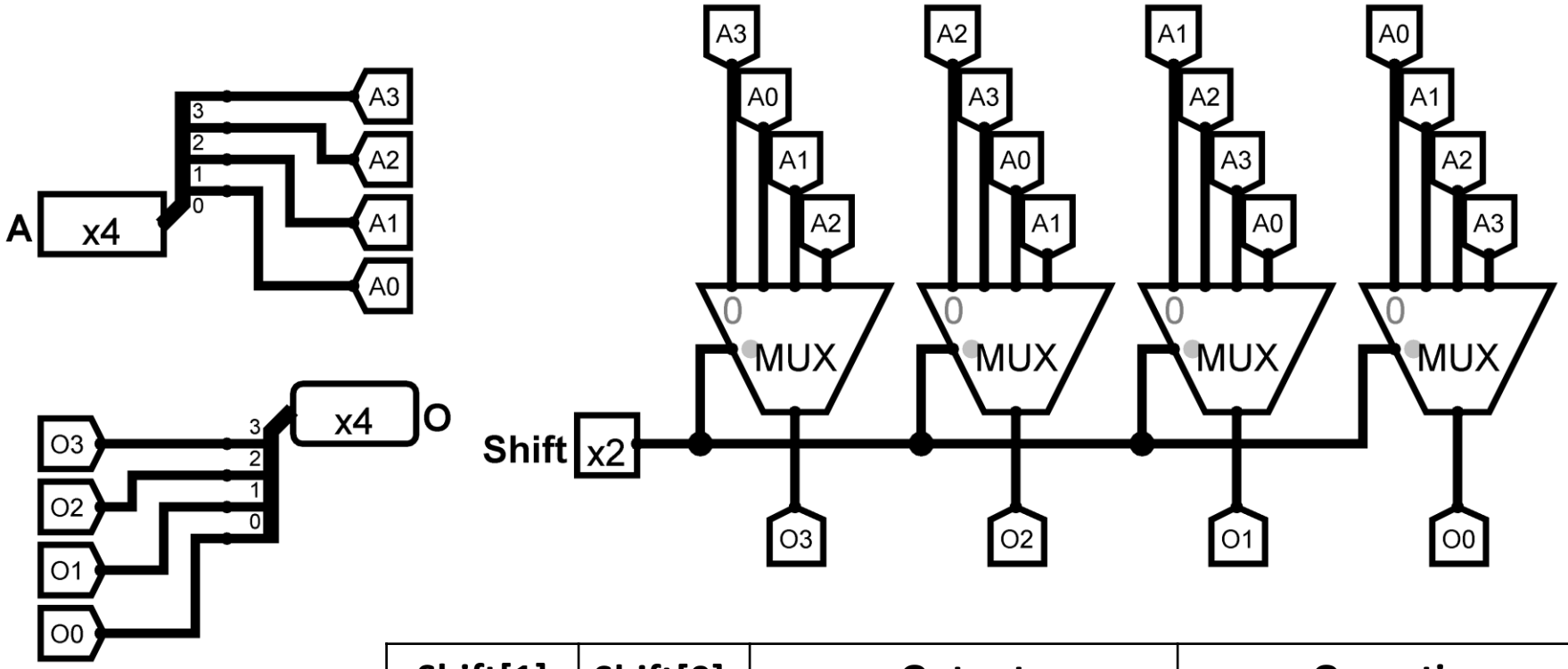


Figure: 4-bit Left Rotate Simulation for input A = 1111, Shift = 10 (1-bit Left Rotate). Output O = 1110

4-bit Right Rotate



4-bit Right Rotate



Shift[1]	Shift[0]	Output	Operation
0	0	**A[3]A[2]A[1]A[0]**	No Rotate
0	1	**A[0]A[3]A[2]A[1]**	1 bit Right Rotate
1	0	**A[1]A[0]A[3]A[2]**	2 bit Right Rotate
1	1	**A[2]A[1]A[0]A[3]**	3 bit Right Rotate

4-bit Right Rotate Simulation

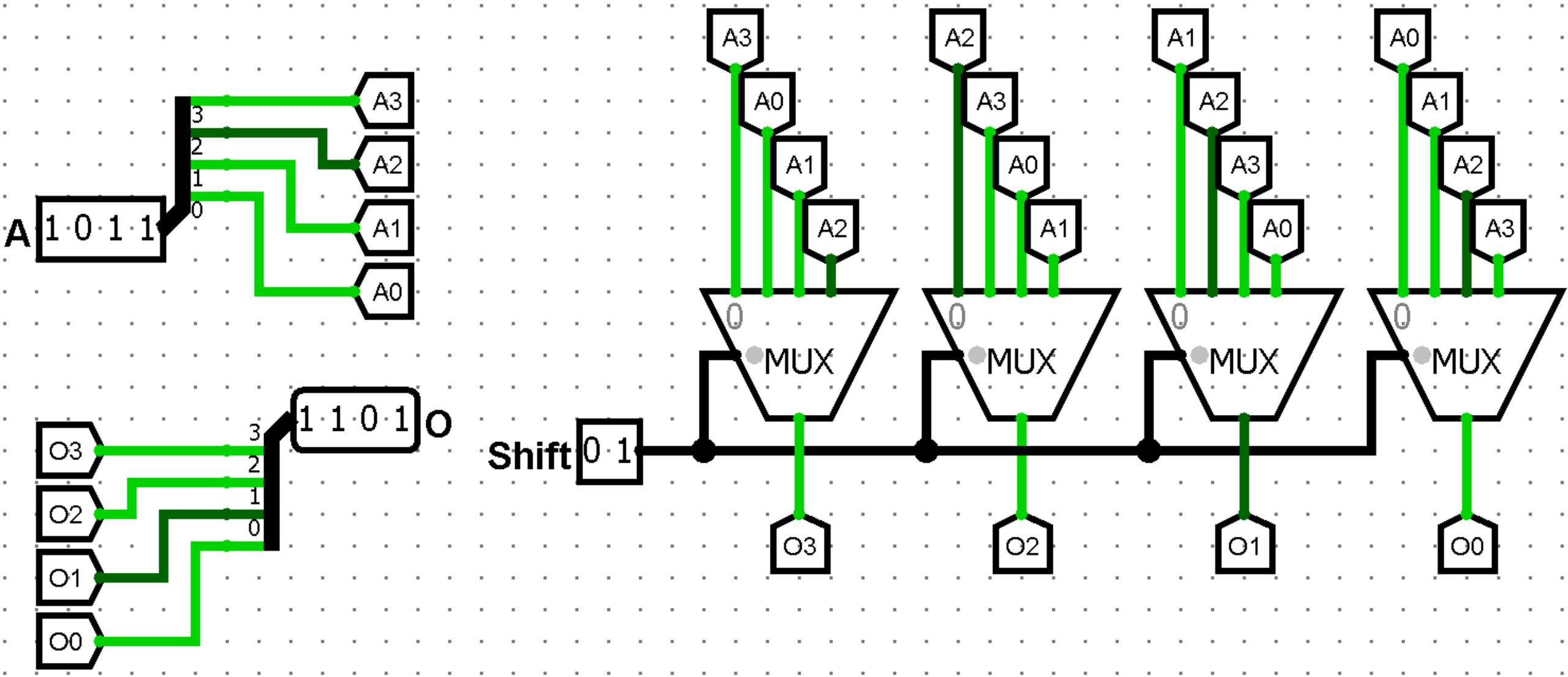
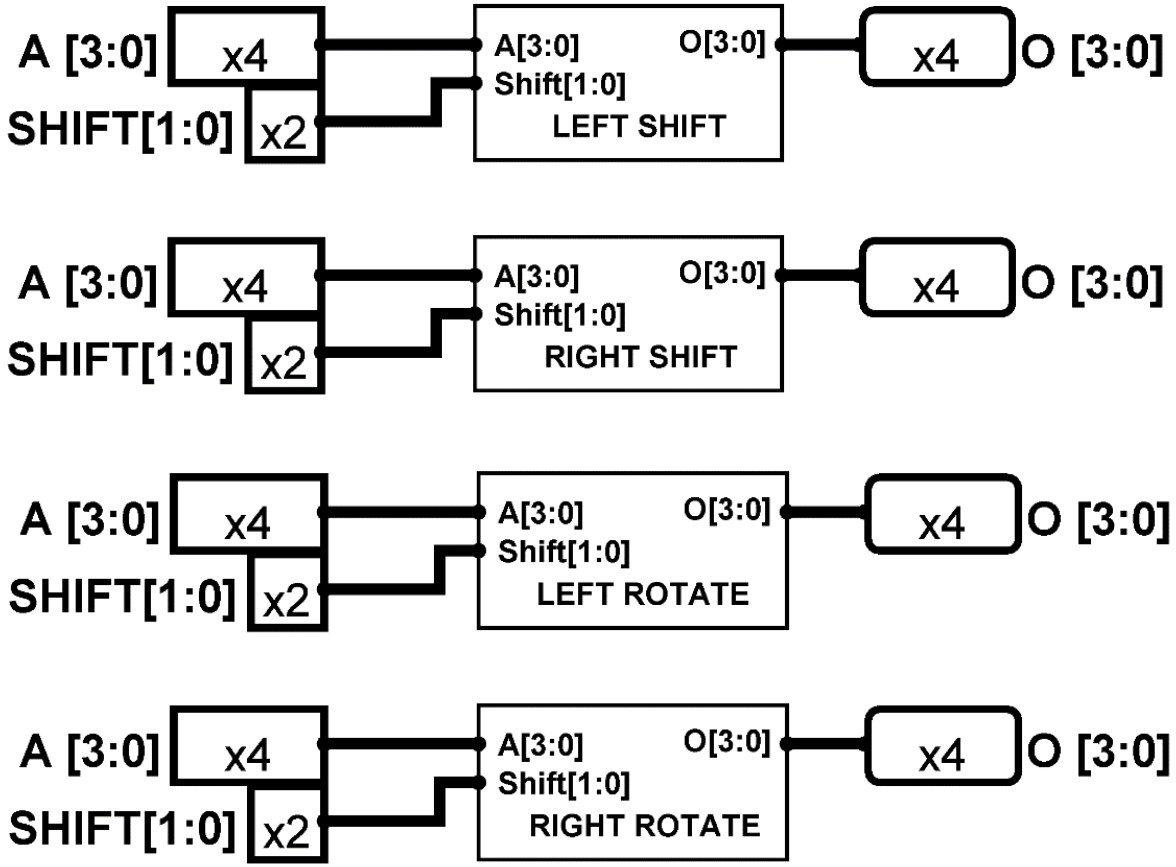
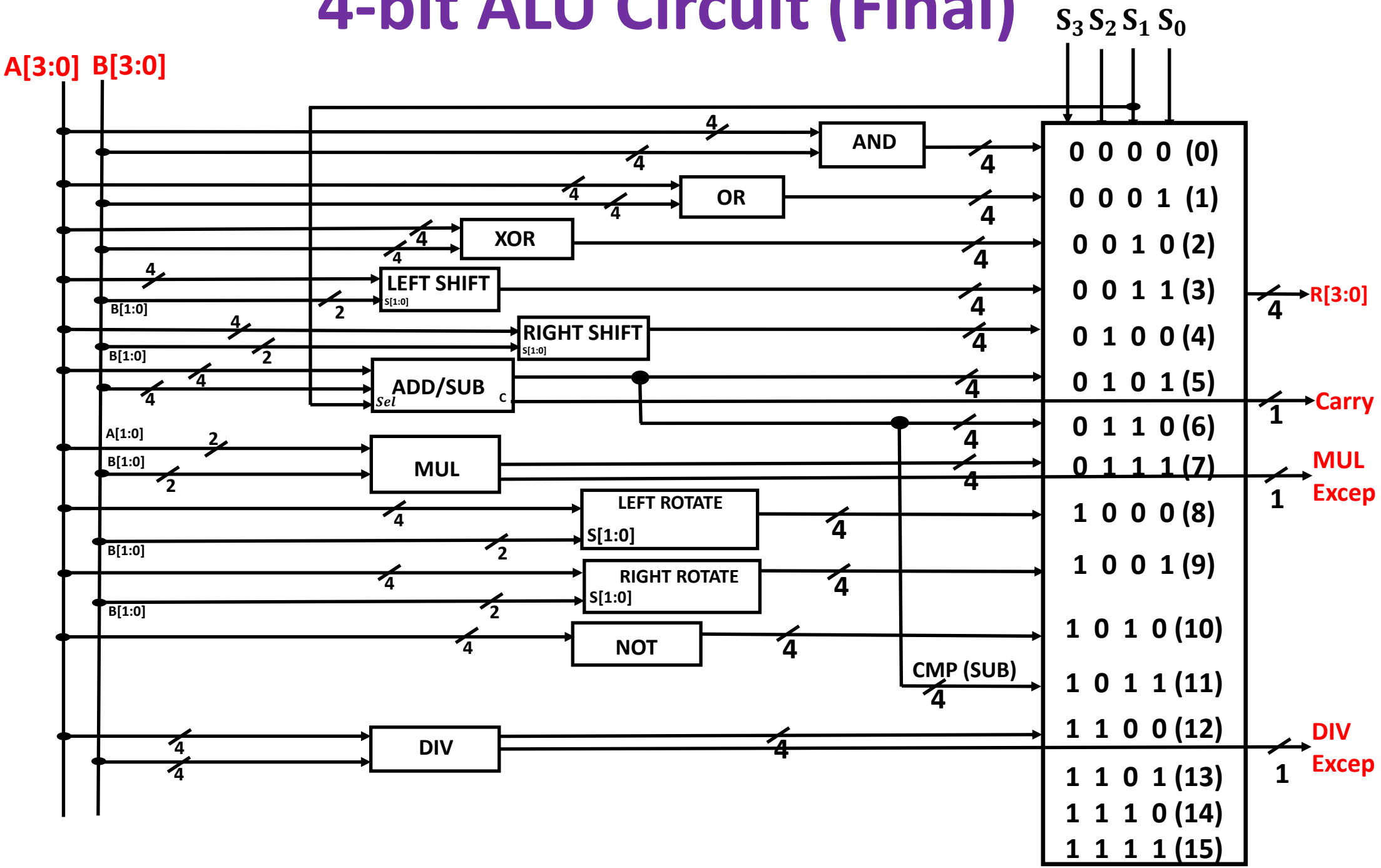


Figure: 4-bit Right Rotate Simulation for input A = 1111, Shift = 10 (1-bit Right Rotate). Output O = 1110

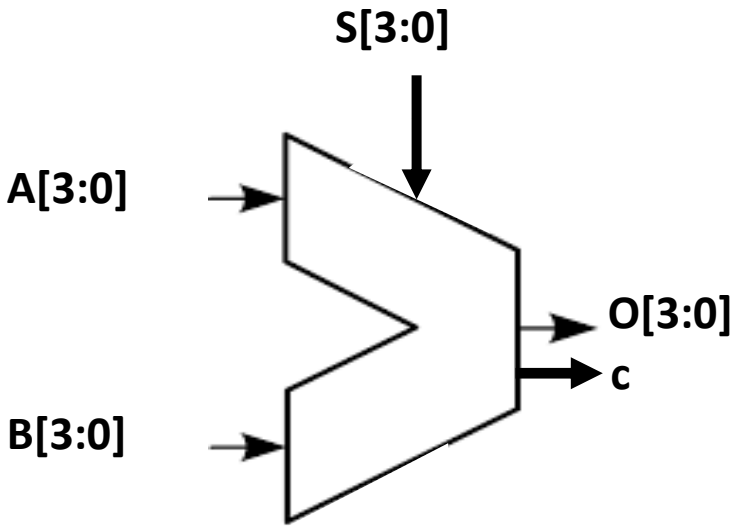
Shift and Rotate Circuits Block



4-bit ALU Circuit (Final)



4-bit ALU Circuit

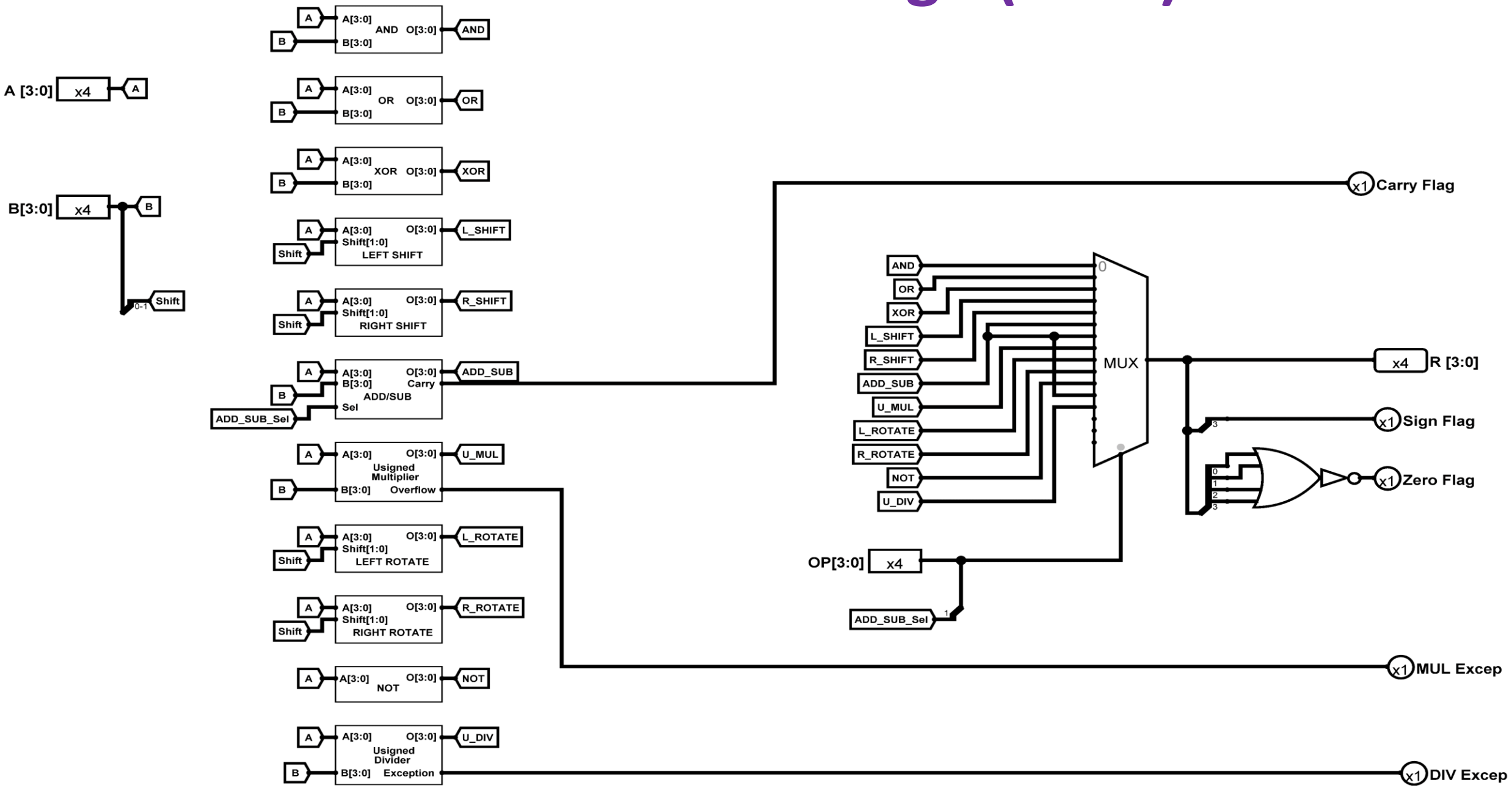


Operation	Selection lines			
	S_3	S_2	S_1	S_0
AND	0	0	0	0
OR	0	0	0	1
XOR	0	0	1	0
NOT	1	0	1	0
ADD	0	1	0	1
SUB	0	1	1	0
MUL	0	1	1	1
DIV	1	1	0	0
SHL	0	0	1	1
SHR	0	1	0	0
ROL	1	0	0	0
ROR	1	0	0	1
CMP	1	0	1	1

Here,
A[3:0] is data to be shifted or rotated.
And B[1:0] is number of shift/rotate (Max 3).

Here, CMP is same as SUB but it doesn't update register value.

4-bit ALU Design (Final)



4-bit ALU Simulation

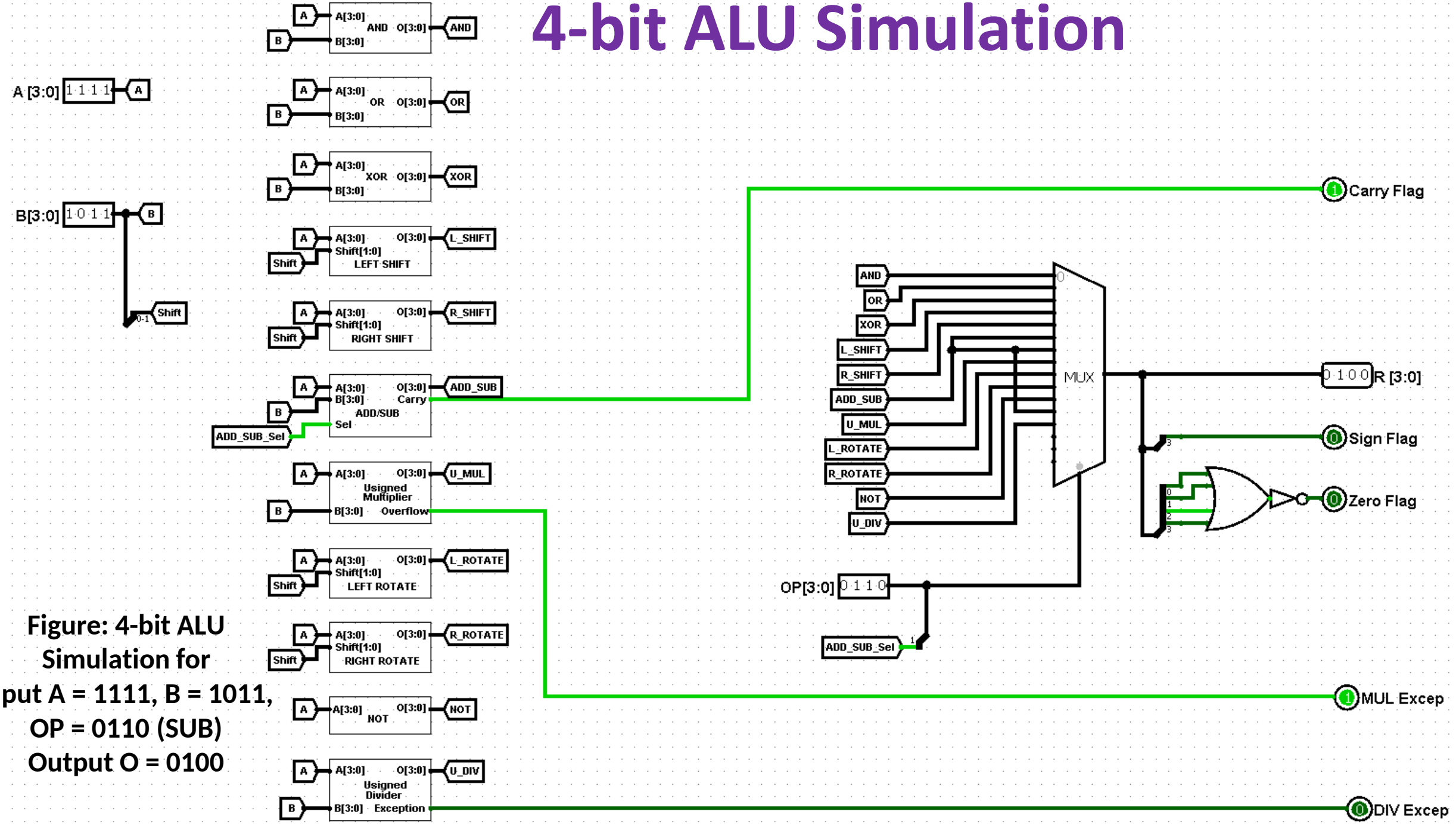


Figure: 4-bit ALU Simulation for input A = 1111, B = 1011, OP = 0110 (SUB) Output O = 0100

Example: ALU

Question: Design a 4-bit ALU that supports RIGHT ROTATE and DIV (Unsigned) operations.

Answer:

4-bit RIGHT ROTATE Circuit:

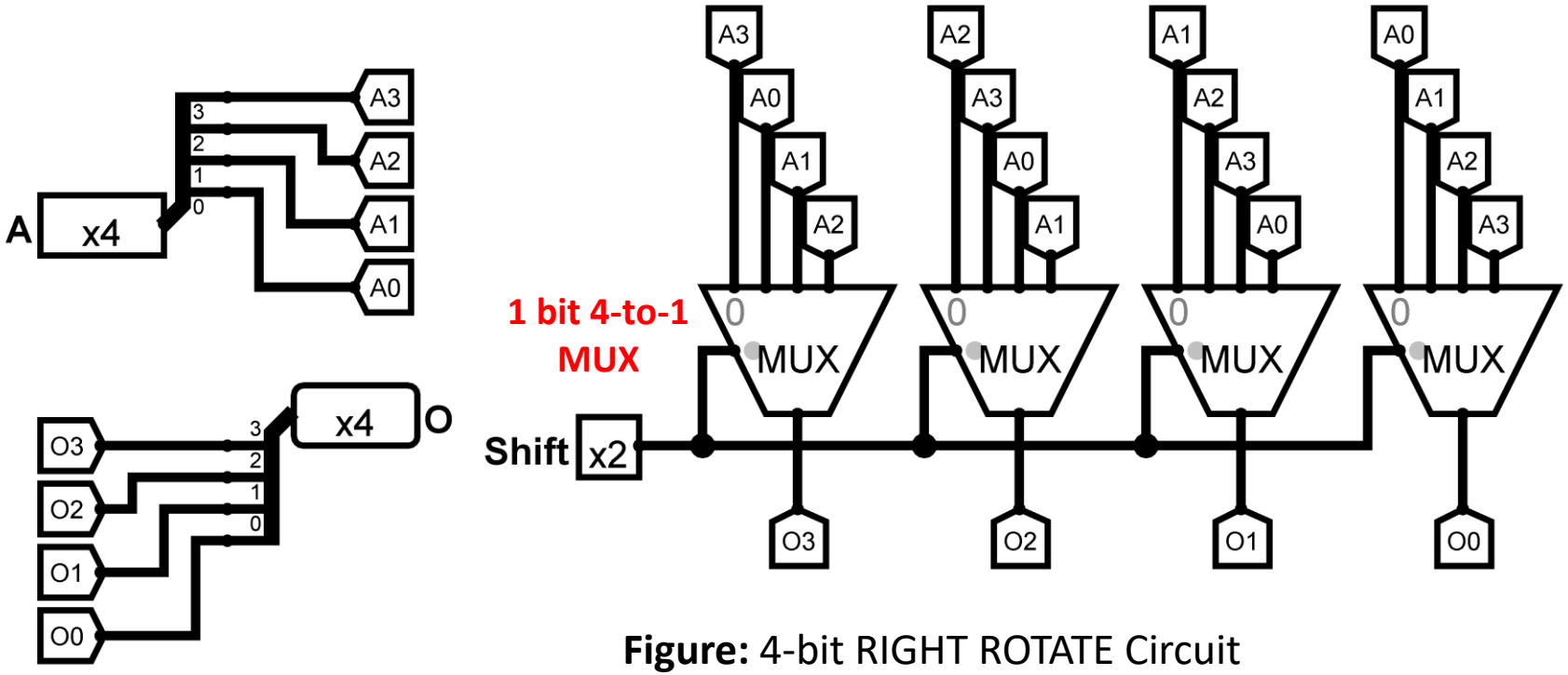


Figure: 4-bit RIGHT ROTATE Circuit

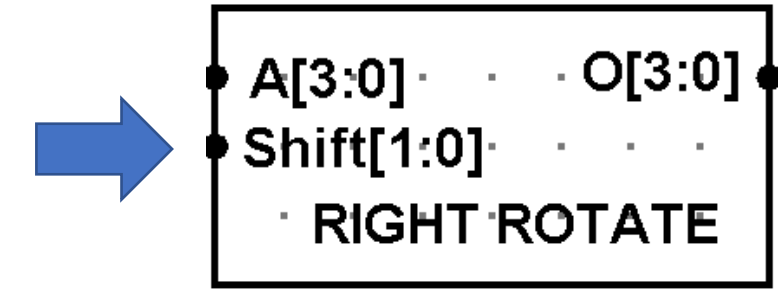


Figure: 4-bit RIGHT ROTATE Chip

Example: ALU

4-bit DIV (Unsigned) Circuit:

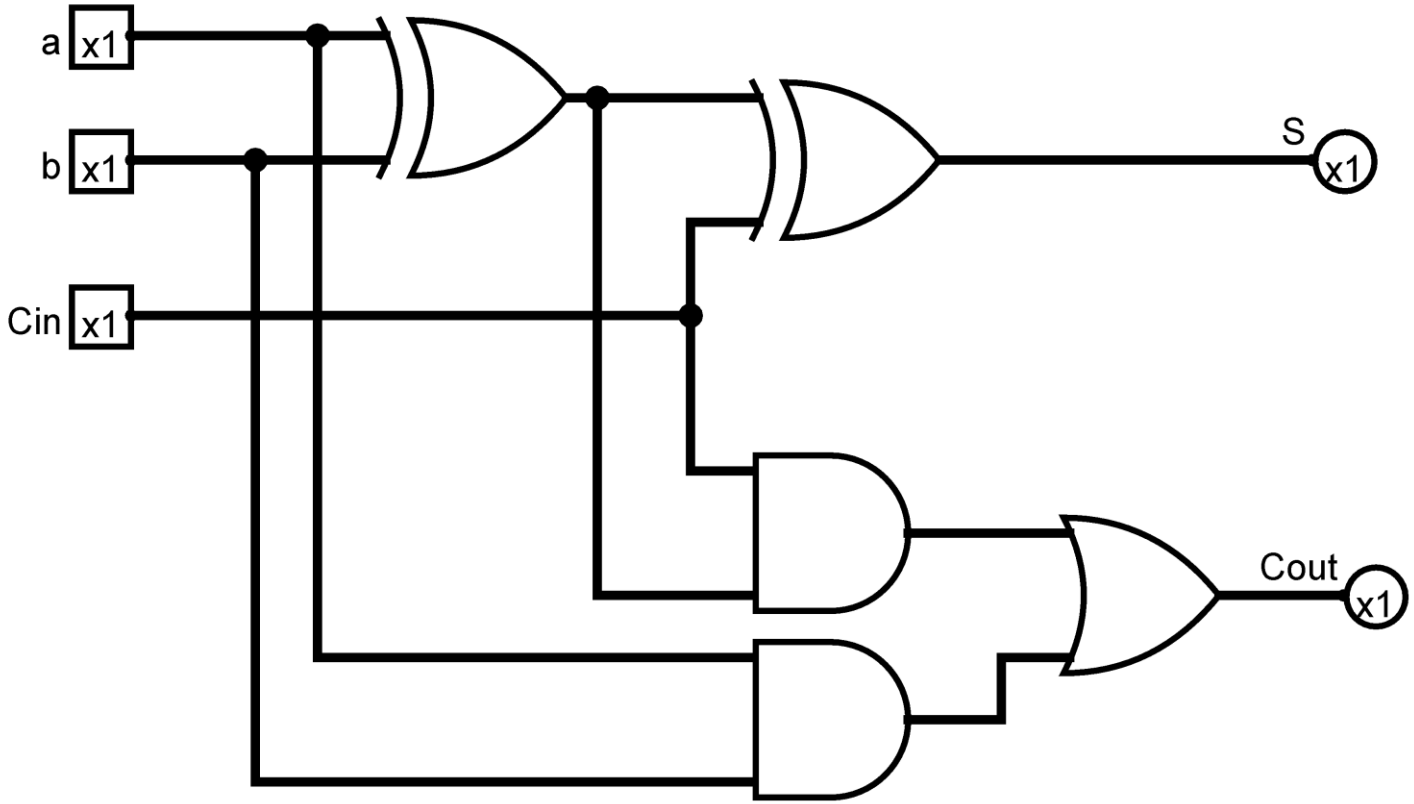


Figure: 1-bit Full Adder Circuit

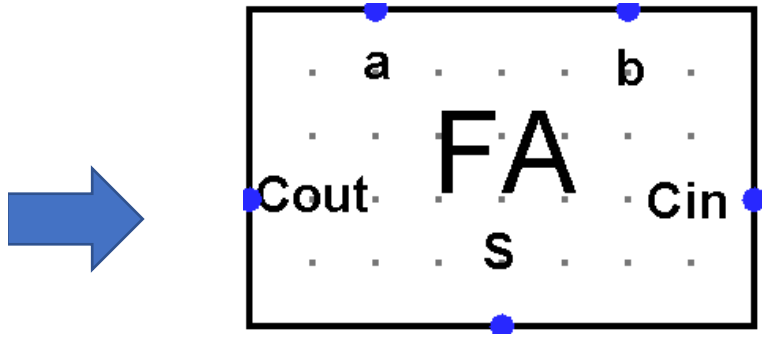


Figure: 1-bit Full Adder Chip

Example: ALU

4-bit DIV (Unsigned) Circuit:

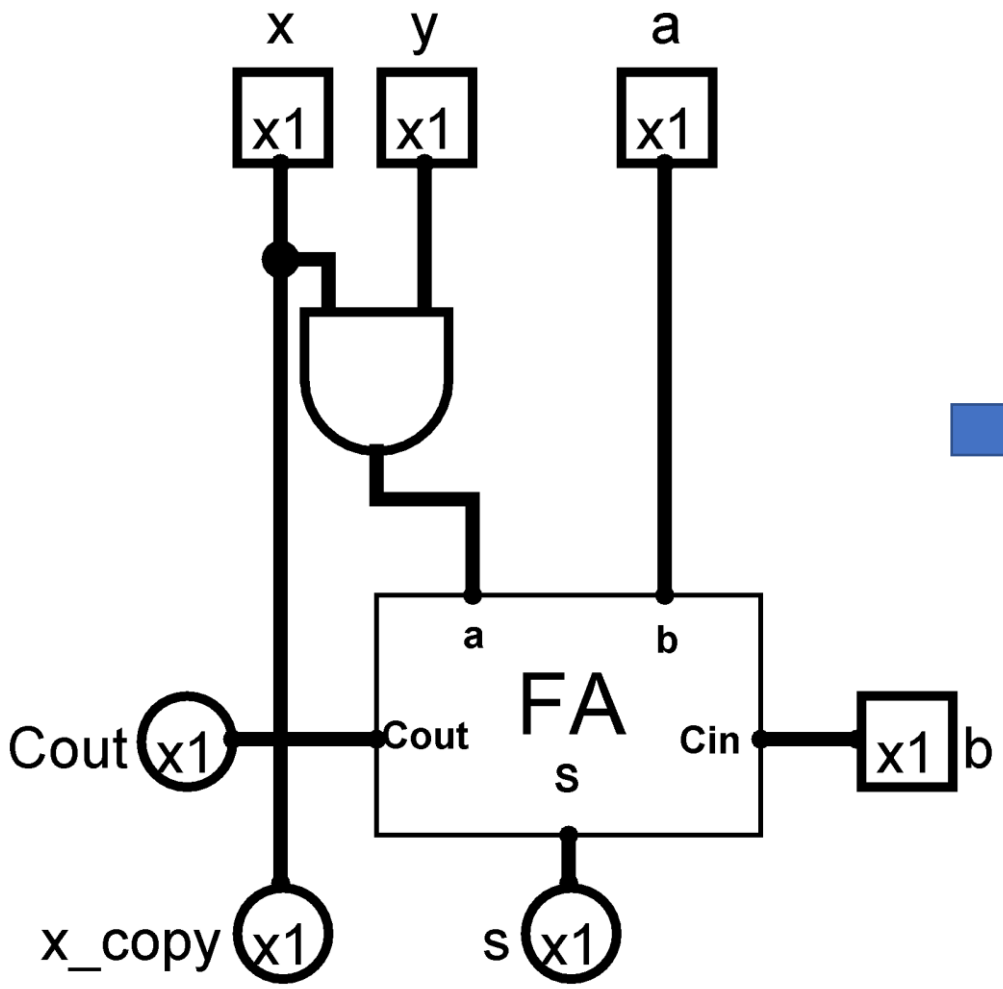


Figure: Cell M (Divider Block)

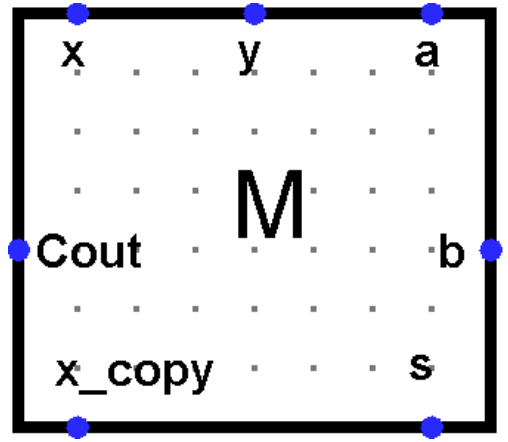


Figure: Cell M (Divider Block) Chip

Example: ALU

4-bit DIV (Unsigned) Circuit:

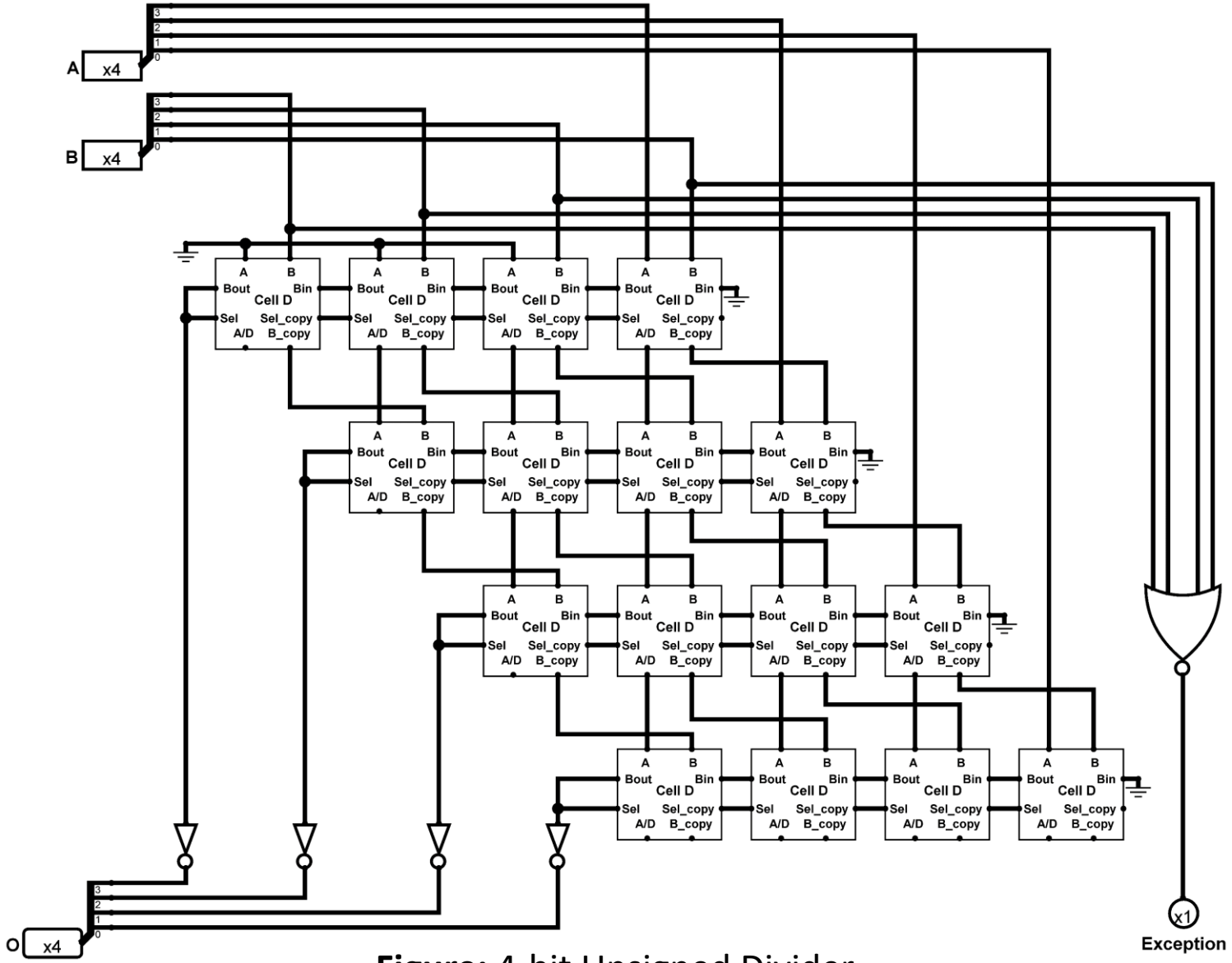


Figure: 4-bit Unsigned Divider

Example: ALU

4-bit DIV (Unsigned) Circuit:

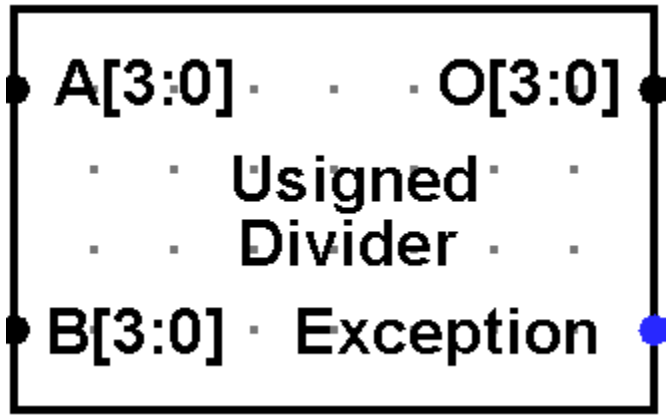


Figure: 4-bit Unsigned Divider Chip

Example: ALU

4-bit ALU Circuit:

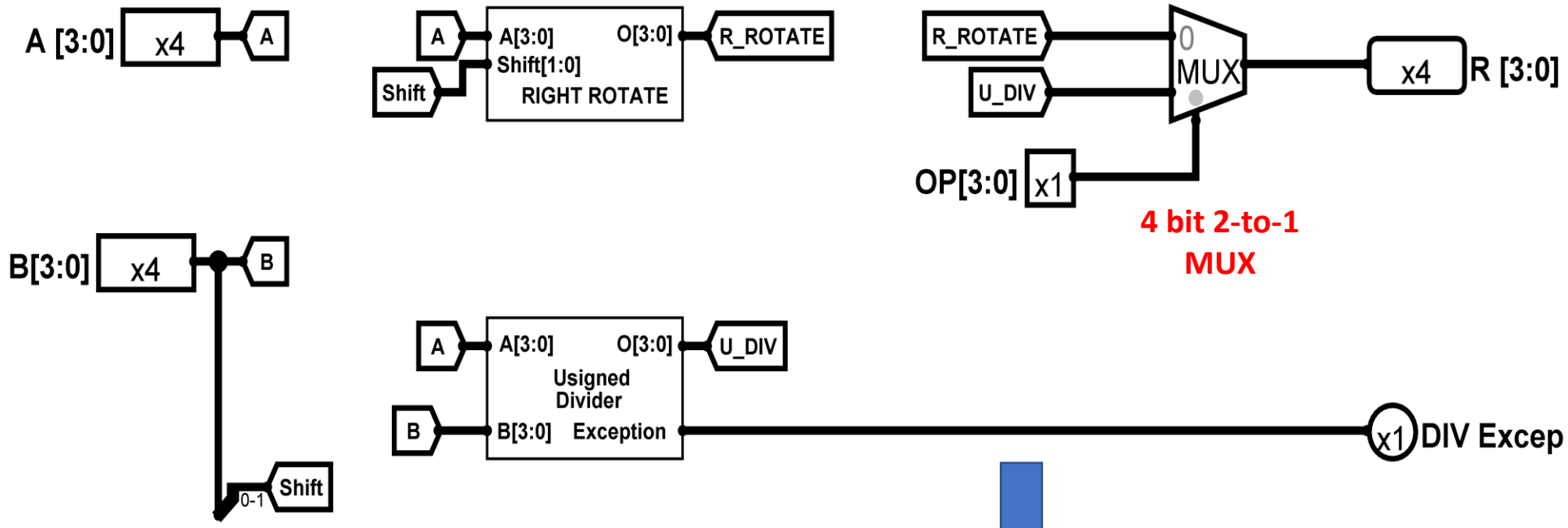


Figure: 4-bit ALU Circuit that supports RIGHT ROTATE and DIV

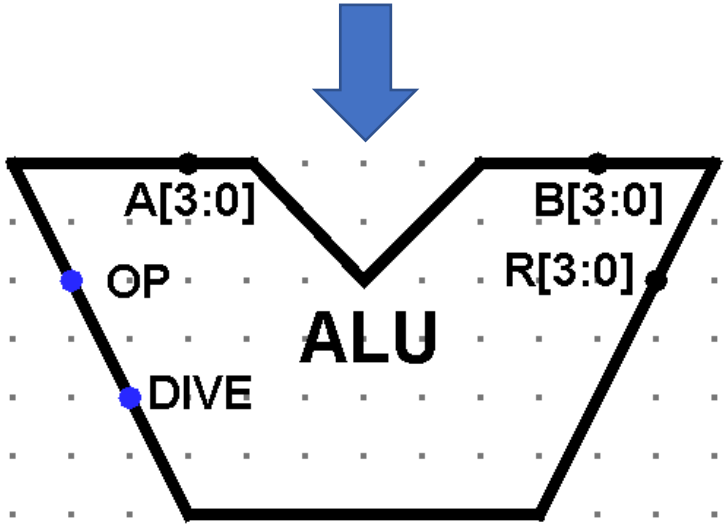


Figure: 4-bit ALU Circuit Chip

Exercises

1. Draw/Design/Implement an 1-bit/2-bit/3-bit/4-bit/5-bit ALU that supports following operations:

i. ADD /
ii. SUB /
iii. MUL /
iv. DIV /
v. LEFT SHIFT /
vi. RIGHT SHIFT /
vii. LEFT ROTATE /
viii. RIGHT ROTATE /
ix. AND /
x. OR /
xi. XOR /
xii. NOT /
xiii. CMP

Calculating FLAG Values in ALU

Question: How can we implement conditional branch instructions like JE, JNE, JG, JL, JLE etc.?

Answer:
We can implement conditional instructions by using FLAG values. Flag values are stored in FLAG Register.

FLAG Register

FLAG register always save state of previous instruction. Its flags will be on/off depending on result of previous instruction.

We learned about 8 flags in 8086 processor. They are:

Sign Flag
Zero Flag
Auxiliary Cary Flag
Parity Flag
Carry Flag
Overflow Flag
Directional Flag
Interrupt Flag
Trap Flag

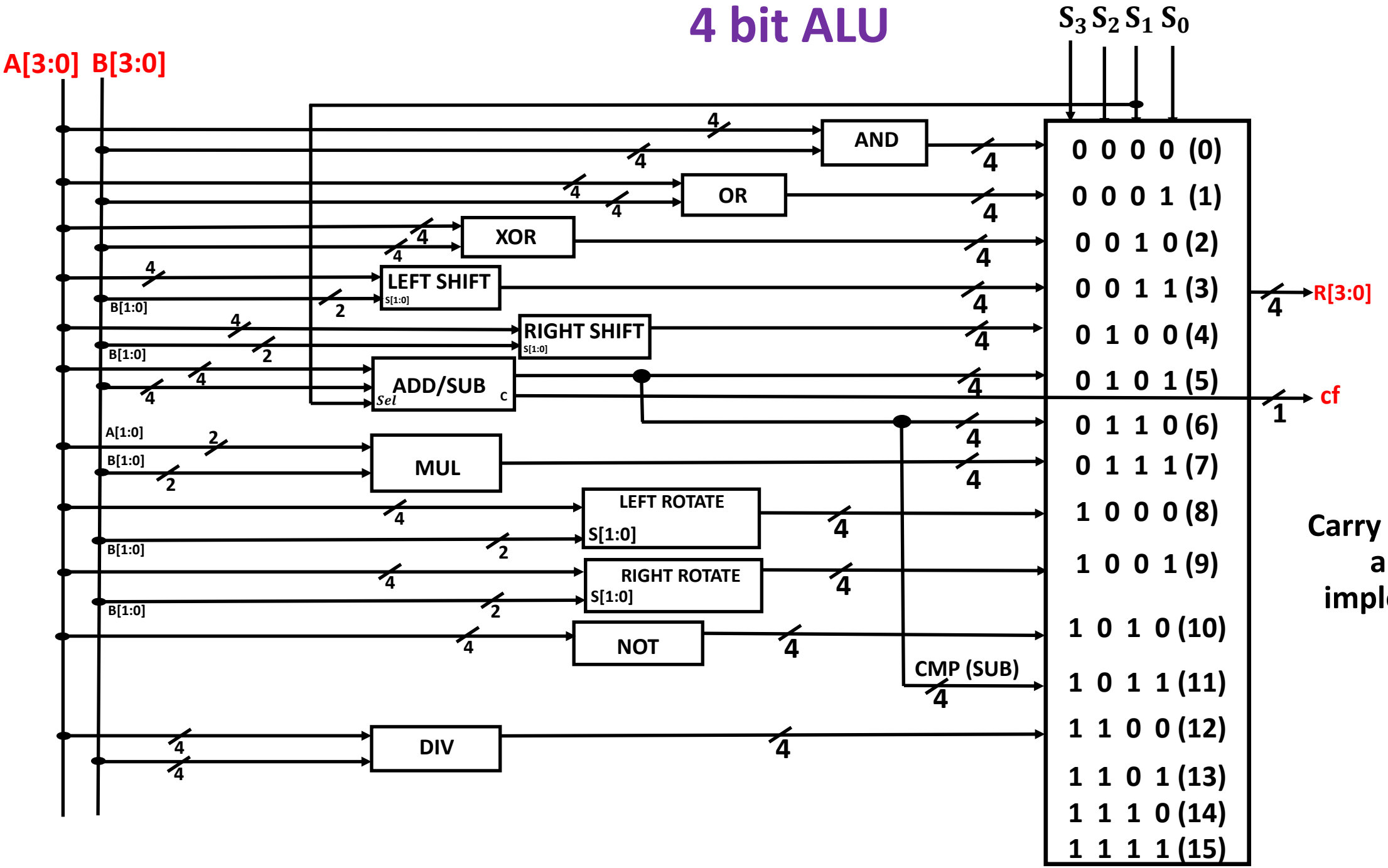
FLAG Register

We will implement 3 flags in our CPU. They are:

1. **Carry Flag (CF):** It will be ON/1 when result of ADD/SUB have carry.
2. **Sign Flag (SF):** It will determine whether result is positive or negative.
3. **Zero Flag (ZF):** It will determine whether result is 0.

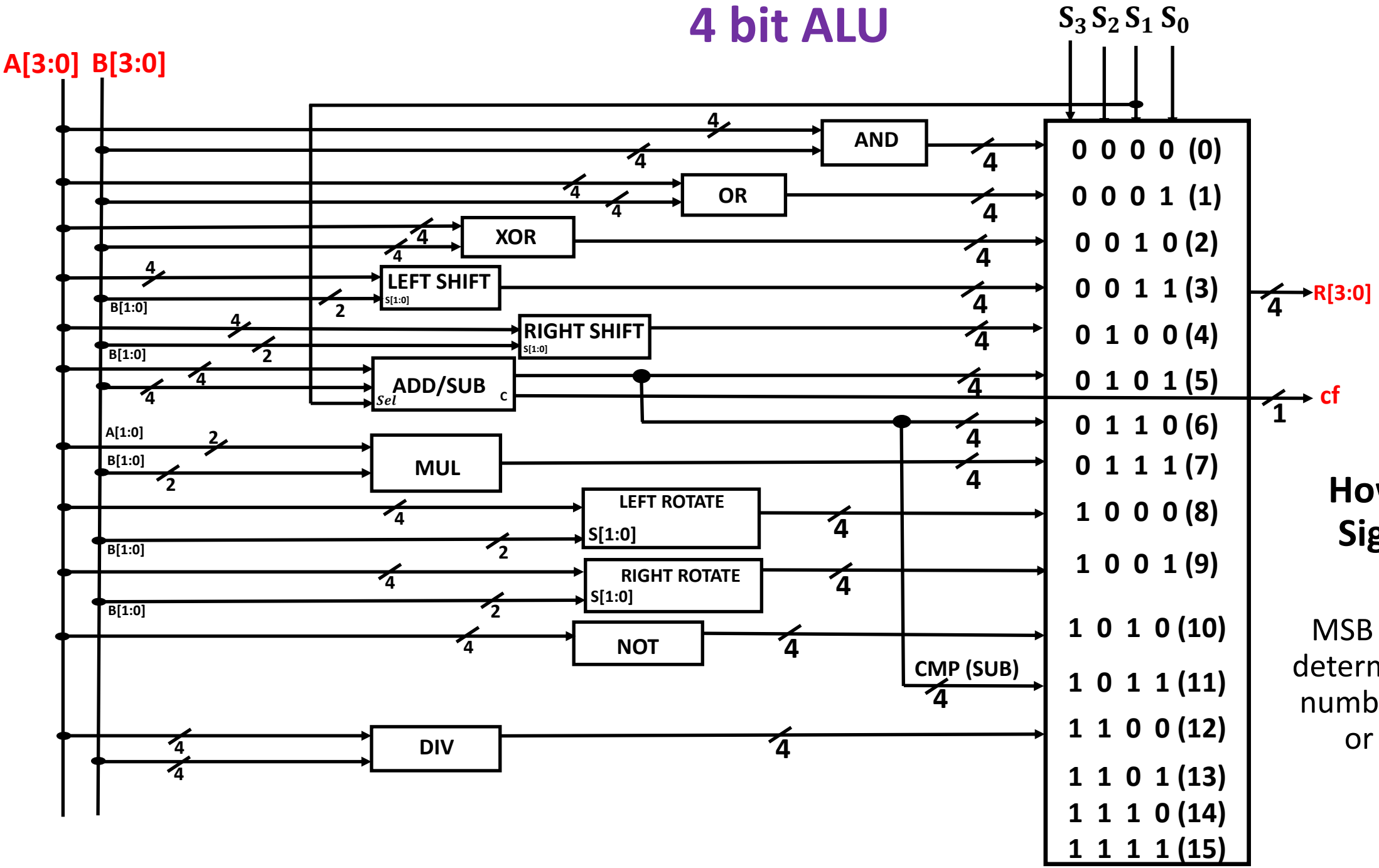
Flag value depends on Result of ALU.

4 bit ALU



Carry Flag (CF) is already implemented.

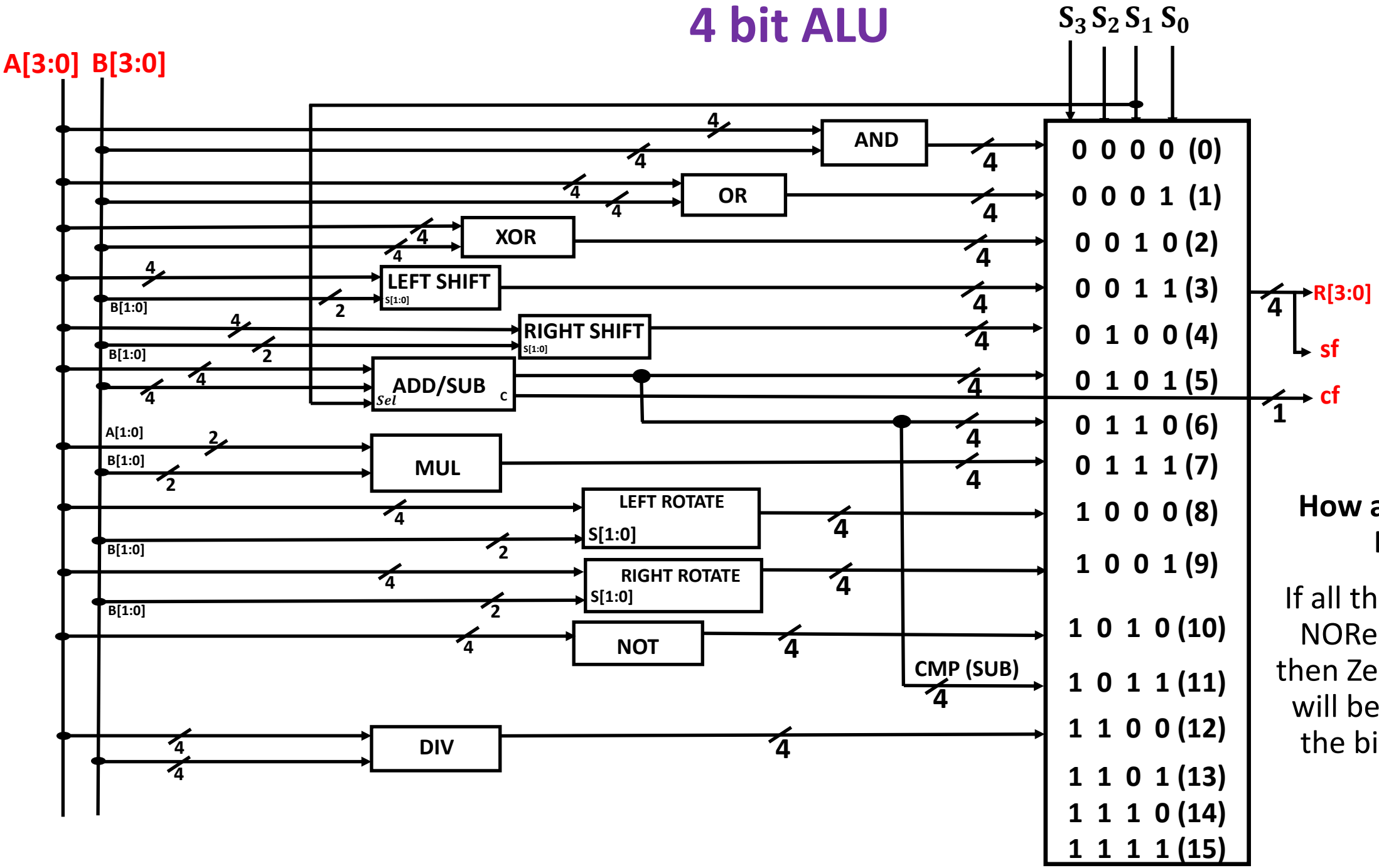
4 bit ALU



How about Sign Flag?

MSB bit of result determine whether number is positive or negative.

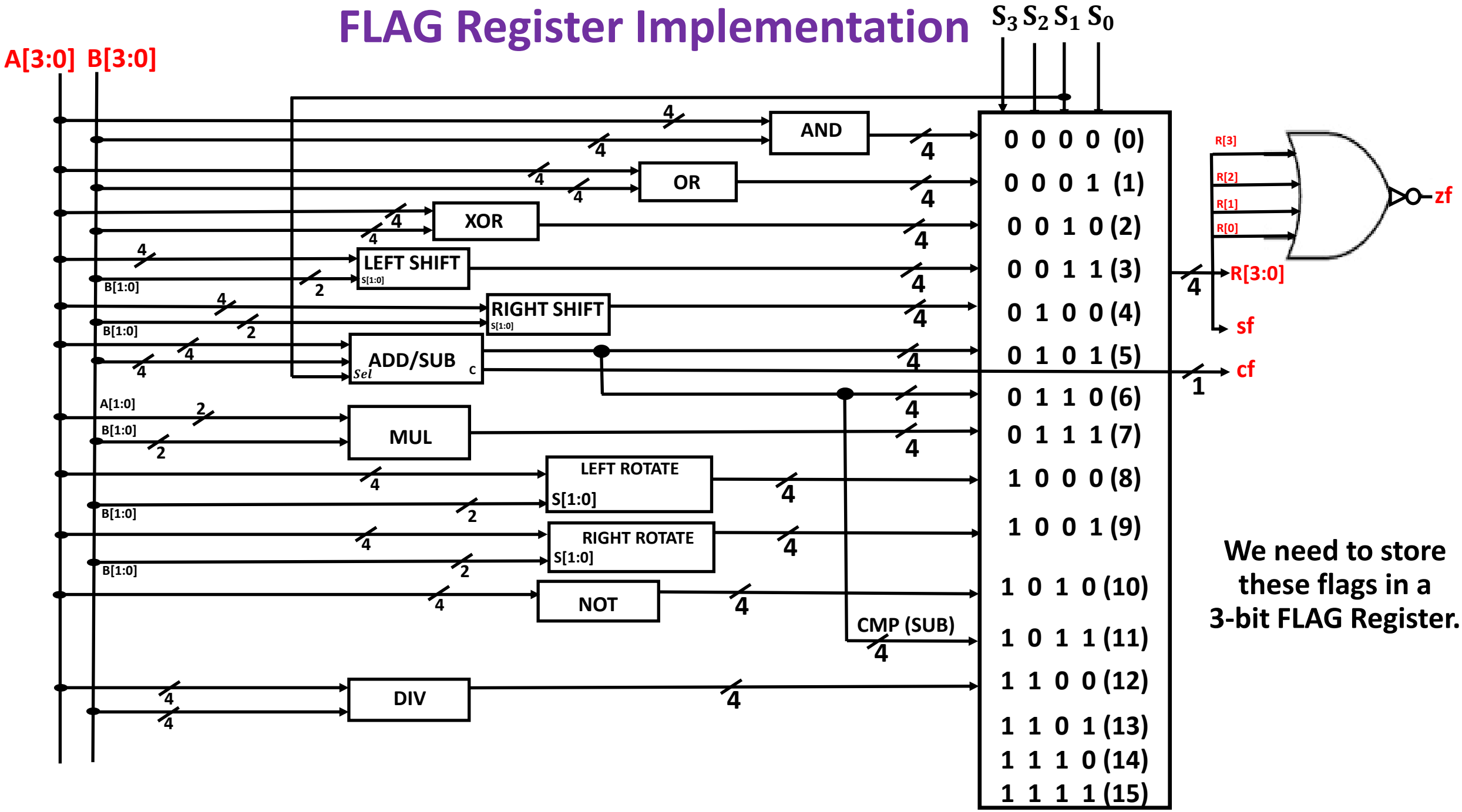
4 bit ALU



How about Zero Flag?

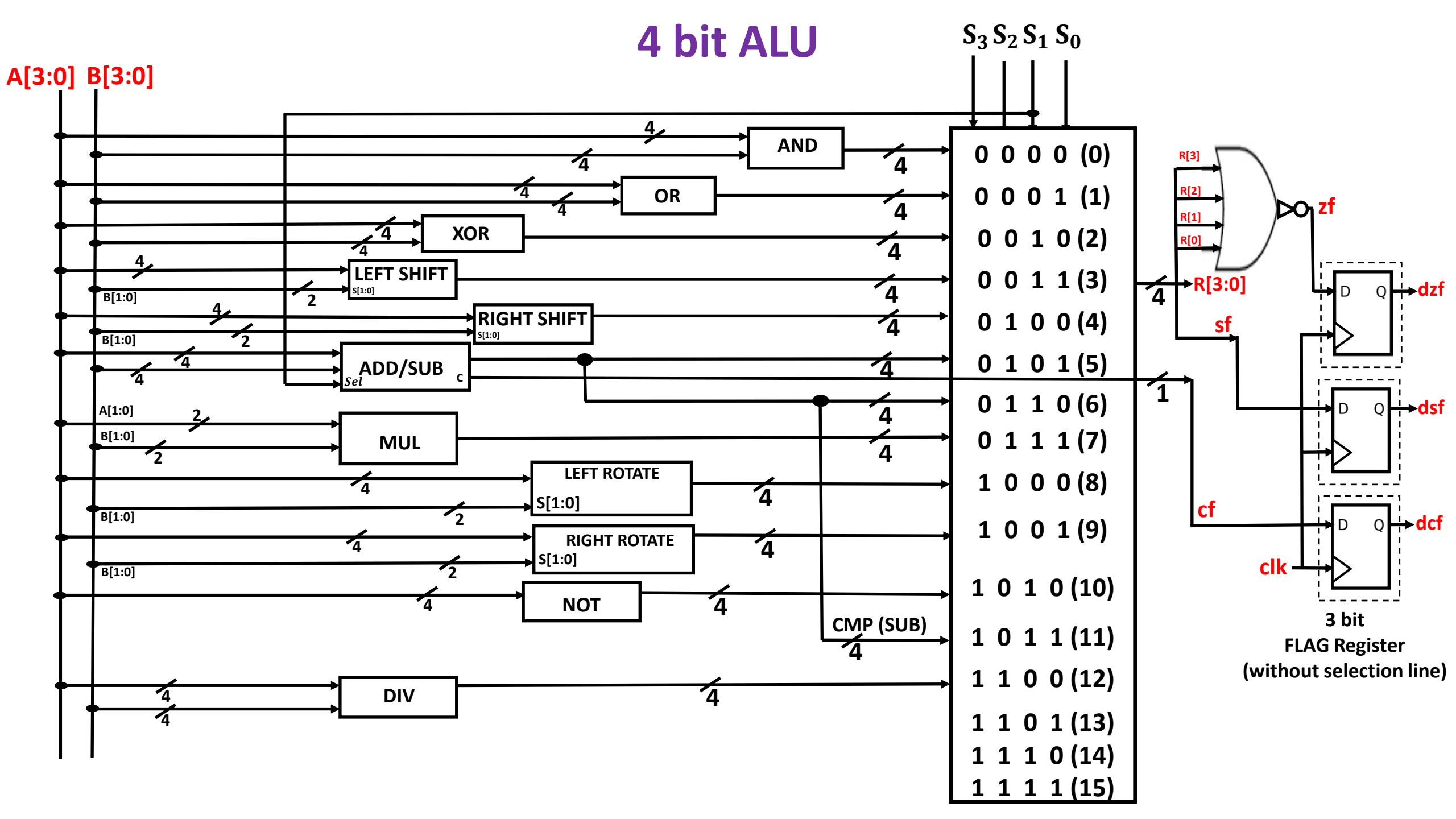
If all the bits of R is NORed together, then Zero Flag value will be 1 only if all the bits of R is 0.

FLAG Register Implementation



We need to store these flags in a 3-bit FLAG Register.

4 bit ALU


A[3:0] B[3:0]
S_3 S_2 S_1 S_0
AND
OR
XOR
LEFT SHIFT
S[1:0]
RIGHT SHIFT
S[1:0]
ADD/SUB
Sel
C
MUL
LEFT ROTATE
S[1:0]
RIGHT ROTATE
S[1:0]
NOT
DIV
CMP (SUB)
B[1:0]
A[1:0]
0 0 0 0 (0)
0 0 0 1 (1)
0 0 1 0 (2)
0 0 1 1 (3)
0 1 0 0 (4)
0 1 0 1 (5)
0 1 1 0 (6)
0 1 1 1 (7)
1 0 0 0 (8)
1 0 0 1 (9)
1 0 1 0 (10)
1 0 1 1 (11)
1 1 0 0 (12)
1 1 0 1 (13)
1 1 1 0 (14)
1 1 1 1 (15)
R[3]
R[2]
R[1]
R[0]
R[3:0]
zf
sf
cf
D
Q
dzf
dsf
dcf
clk
3 bit
FLAG Register
(without selection line)

Thank You ☺