8-BIT ALU DESIGN USING lOGISIM

CS21B002
ANDALURI S P V M ADITYA

USAGE OF THE DESIGNED ALU:

First of all, let's look at the design table of 8-bit ALU:

ALU DESIGN :

S0	S1	S2	T	S	CIN	OPERATION
0	0	0	0	0	0	ADD
0	0	0	0	0	1	ADD WITH CARRY
0	0	0	0	1	0	SUB WITH BORROW
0	0	0	0	1	1	SUB
0	0	0	1	0	0	TRANSFER A
0	0	0	1	0	1	INCREMENT A
0	0	0	1	1	0	DECREMENT A
0	0	0	1	1	1	TRANSFER A
0	0	1	X	X	X	MULTIPLIER
0	1	0	X	X	X	BITWISE AND
0	1	1	X	X	X	BITWISE NAND
1	0	0	X	X	X	BITWISE OR
1	0	1	X	X	X	BITWISE NOR
1	1	0	X	X	X	BITWISE XOR
1	1	1	X	X	X	BITWISE XNOR

*** Before reading this section Refer below picture of ALU in page 2**

The core of ALU shown in the below picture has two 8 bit inputs ,**A** ,**B** ,and one 8 bit output, **Result** .

There are some other inputs as well :

S2,**S1**,**S0** are the select bits for choosing the operation we are interested in doing with ALU.

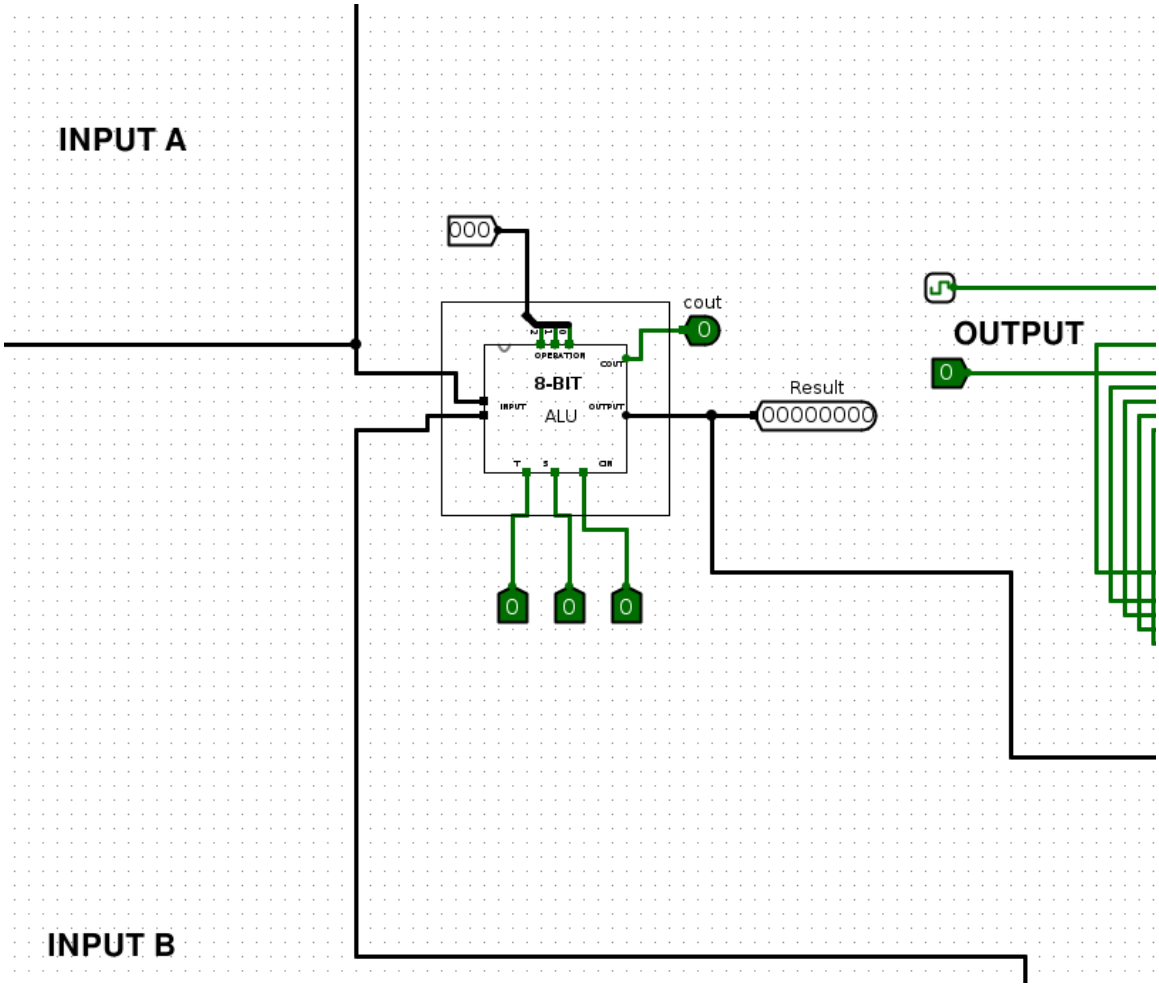
T is another input , I used this to get the ease of transferring and incrementing operations.

S is another input , I used to do subtract and decrement operations.

CIN is another input ,to switch (ADD operation to ADD with carry) or (SUB (subtraction) with borrow to subtraction).

Basically , we will first set the other inputs to the respective binary digits for the required operation . And then we enter input A of 8 bit using the DIP switches and enter input B using DIP switches. Then we will do clock simulation for the input to be loaded in the flip flops and to get desired in the output flip flops and then are also displayed using the 7 segment displays. All the given inputs and outputs are shown in the 7 segment displays.

logisim design of ALU :



Let's visit the **outer functionalities** then we understand the **inside design** of ALU

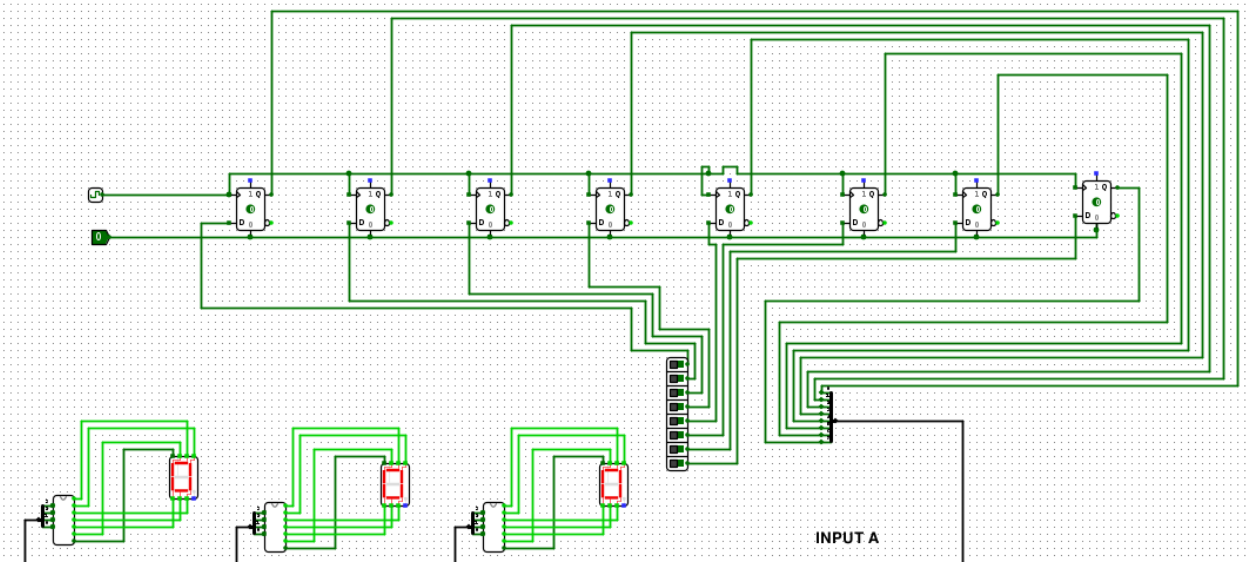
OUTER FUNCTIONS :

- **SETTING OPERATION:**

  We can select the s0,s1,s2,T,S,CIN inputs according to the truth table given For the interested operation.

- **Taking INPUTS** :

  As already mentioned inputs A and B are 8 bit and are should be given using the dip switches (picture attached below). Now let's start simulating the clock (use ctrl+k) and stop the clocks (again use ctrl + k), so that the inputs are loaded in the d flip flops. I have used parallel data transfer for this.



  In the set of DIP switches. The Lowest bit is the MSB. Uppermost bit is LSB.The bit below clock is the reset.The loaded input shown in the 3 seven segment displays.

The full details of conversion of 8 bit binary to 7 segment is explained at the end of the designed document.

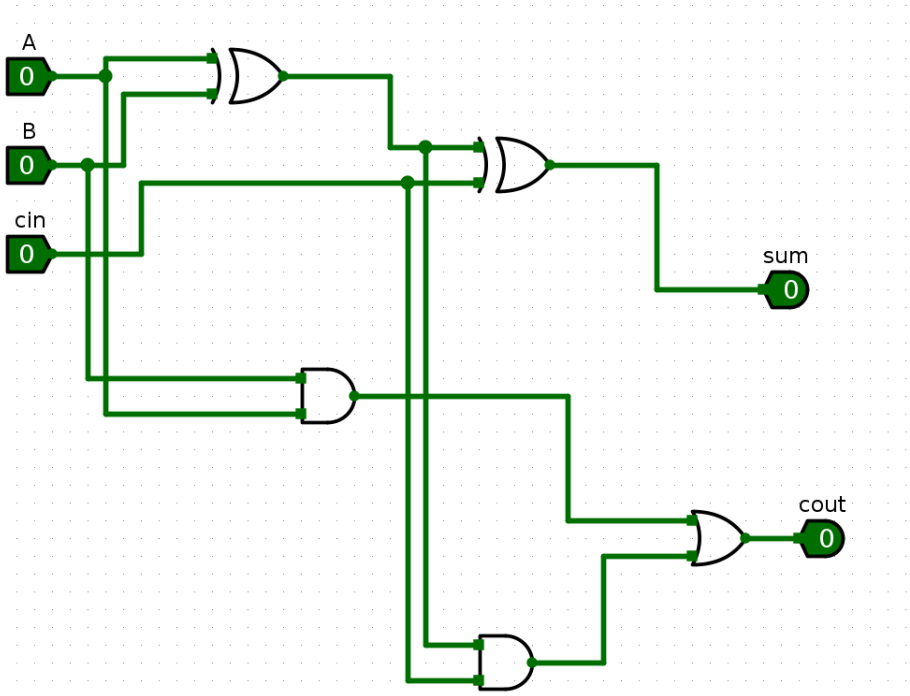
- **Getting Outputs**:

  Once the clocking simulation is done, immediately the output is shown in the output seven segments.

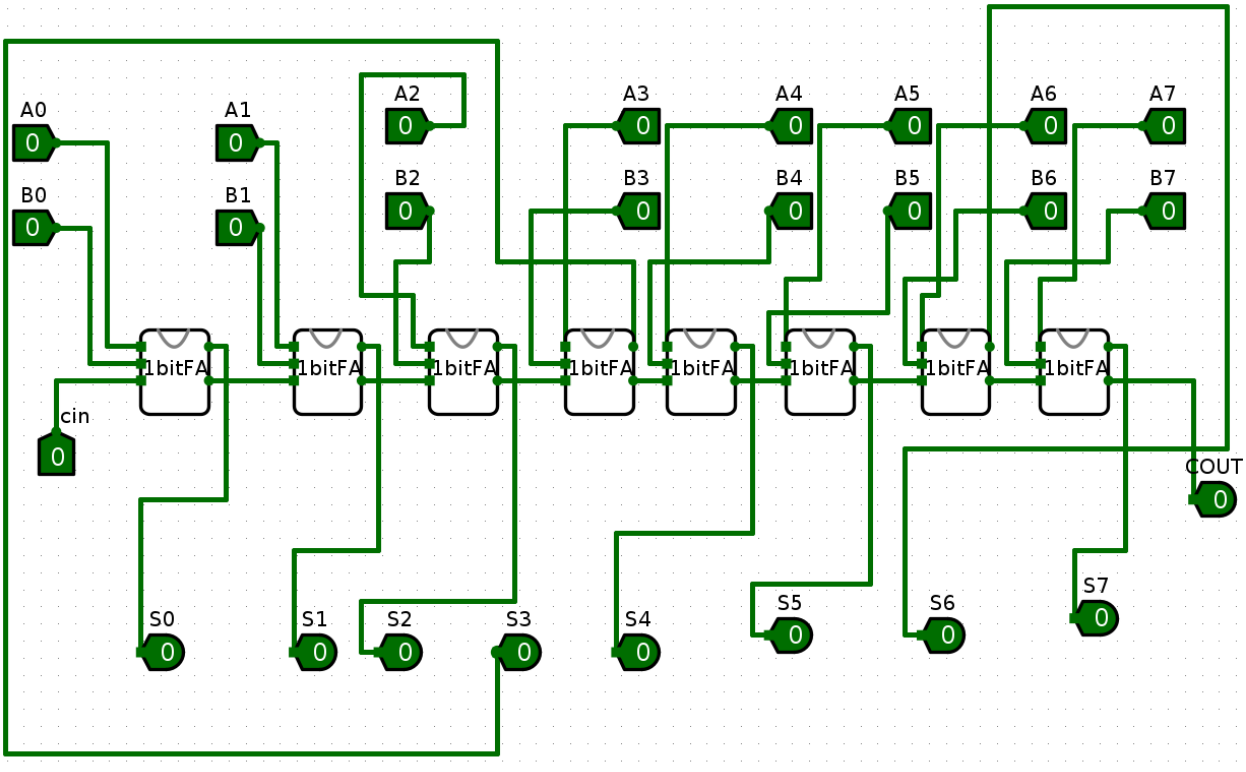
INNER DESIGN OF ALU :

Starting with **one bit full adder** :

A	B	cin	sum	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

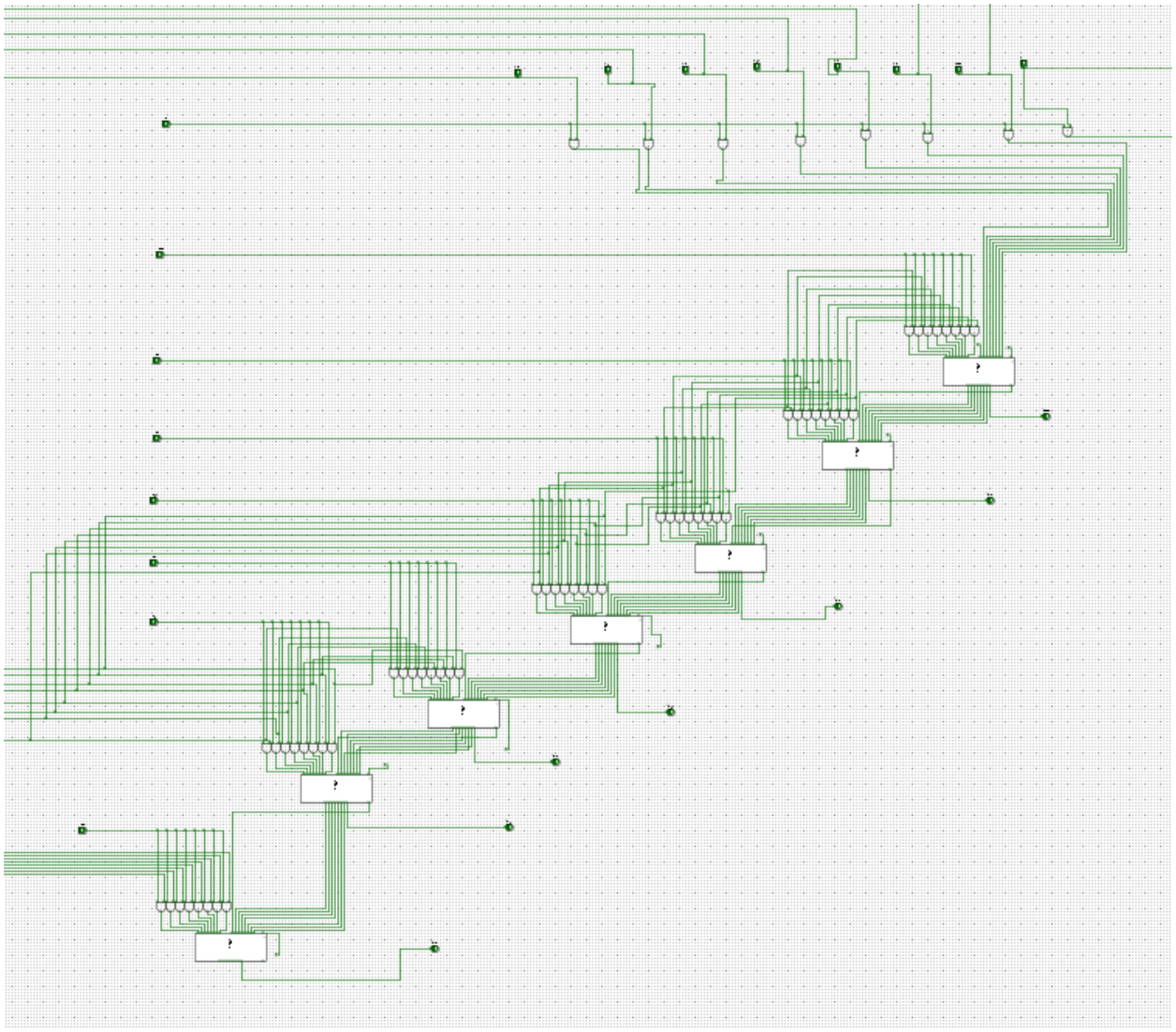


8 BIT Full adder:



Can be created just by connecting the cout of one 1bitFA to cin of consecutive 1bitFA.

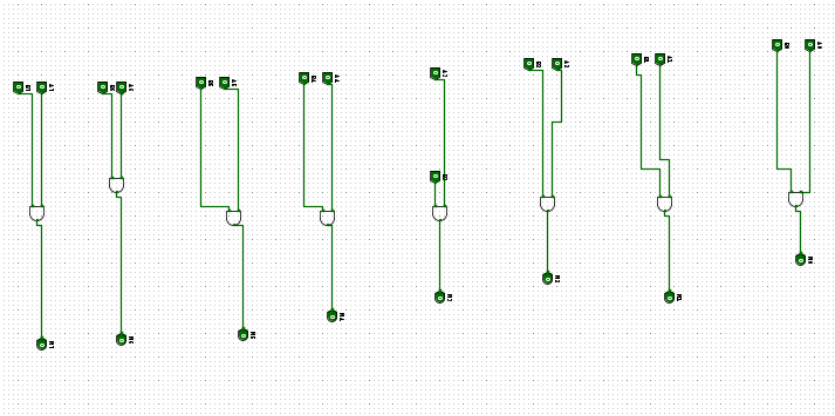
8 BIT MULTIPLIER:



This is just the analogous to 4 *3 multiplier explained in the course.
Contains 7 eight bit full adders for sum of partial products.

8 bitwise AND,OR,XOR,XNOR,NOR:
These are made simply by taking the respective bits and putting them in the respective gates.

One of these is shown below :

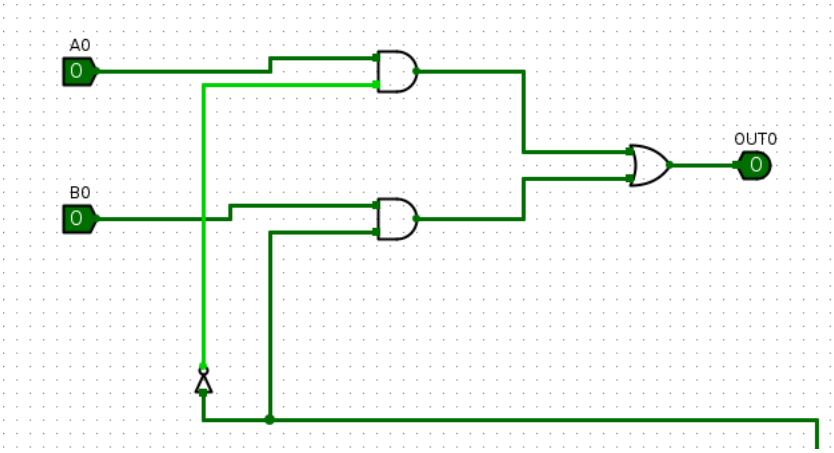


8 to 1 mux with 2 to one mux:

This is designed according to this truth table:

Select	Inputs		Output
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	1

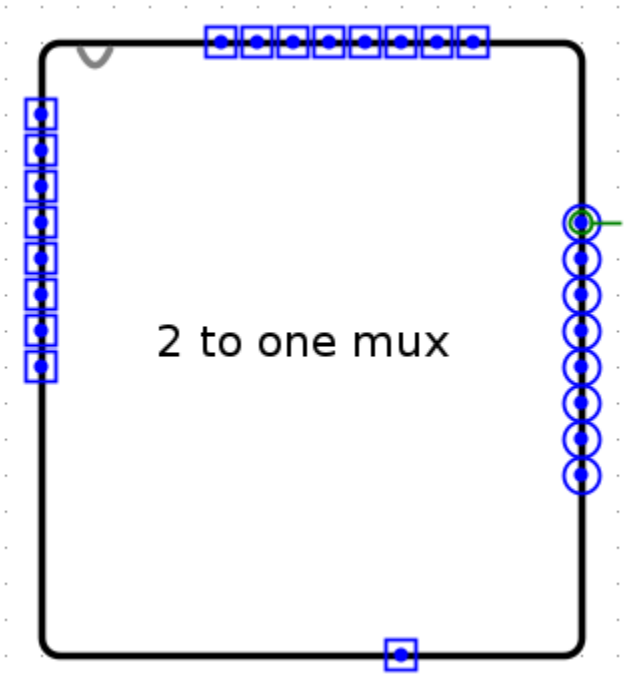
431 × 191

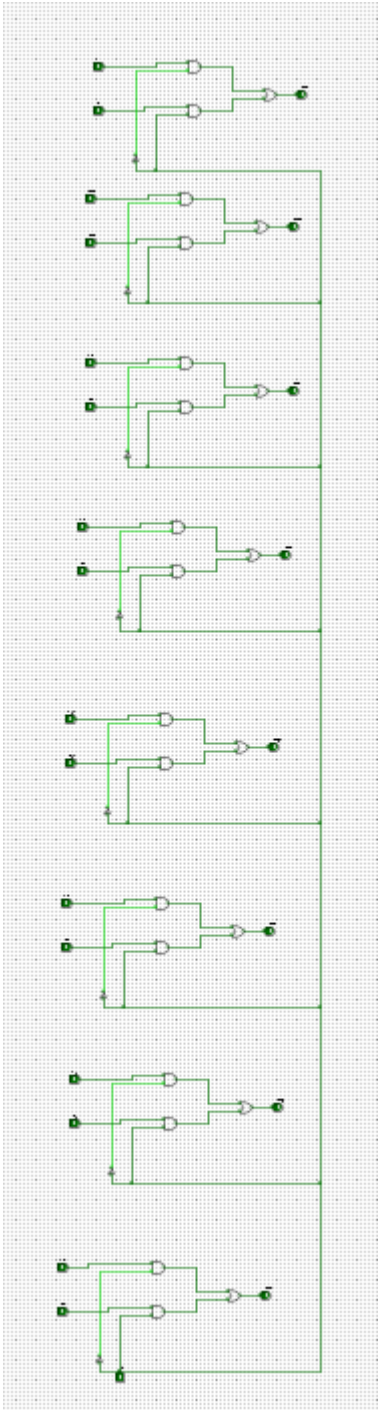


This is for the LSB. All the bits are similarly designed and joined together.

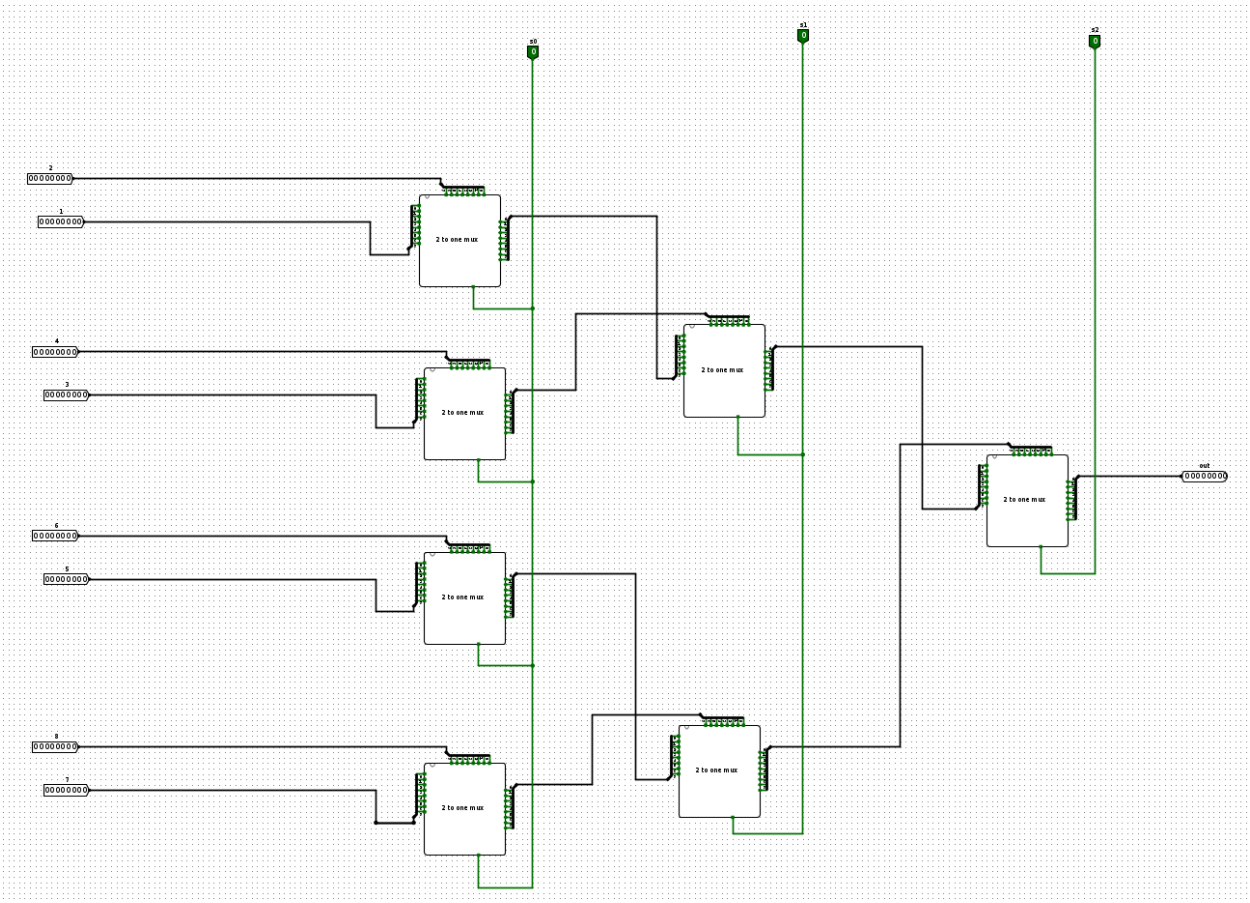
8 to one mux is implemented by joining 7 two to one multiplexers.
Shown in below pictures.

WE WANT THIS 8 TO ONE MUX TO SELECT THE OPERATION AMONG 8 OPERATIONS WE HAVE.





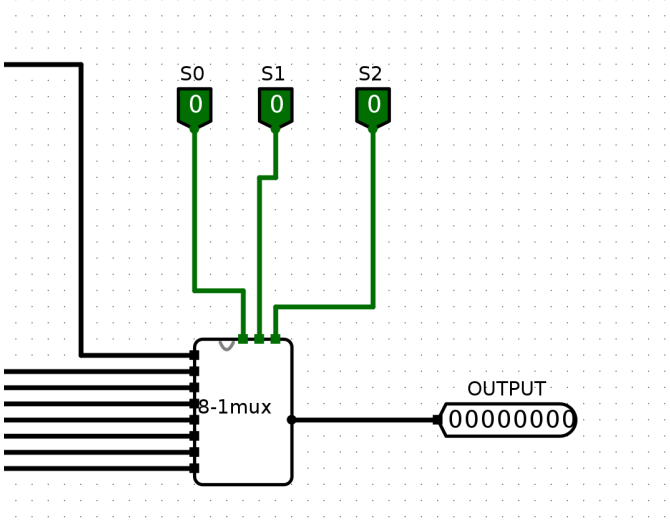
2 to one mux of 8 bits.



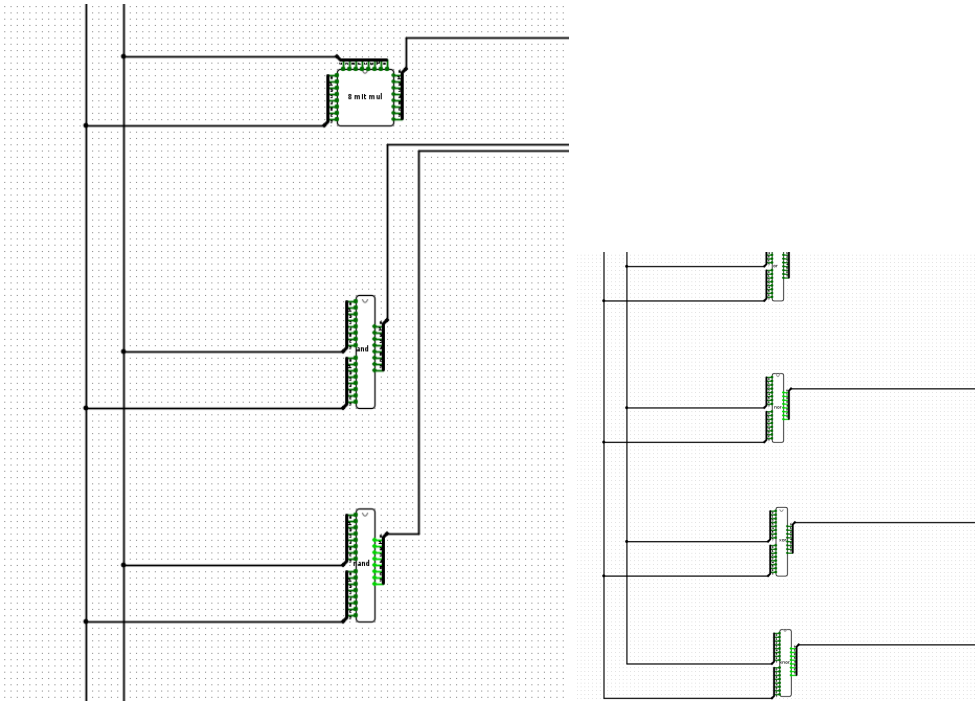
8 to one mux with 2 to one mux

8 bit ALU:

Truth table for this is already shown in page1

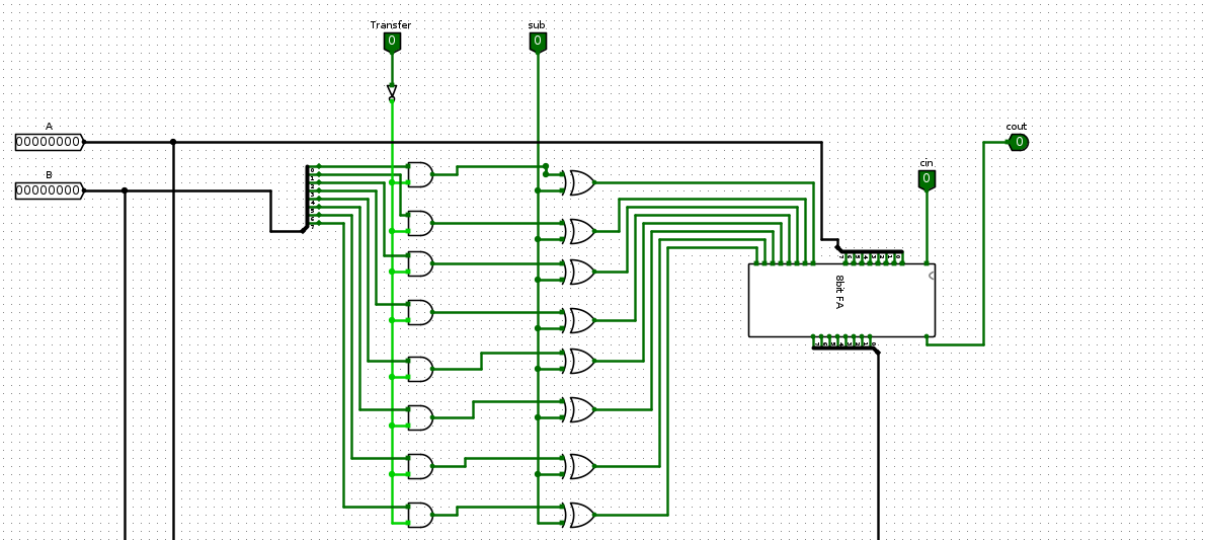


In the above picture you can see the 8 wires coming from 8 different operations.
The output from each operation is controlled by s0,s1,s2 select bits.
OPERATIONS FROM 2 TO 8 :



The outputs of all these seven operations are simply obtained by connecting the inputs to the respective multiplier (s2s1s0-001), bitwise and (s2s1s0-010),bitwise or(s2s1s0-011) ,bitwise nand (s2s1s0-100), bitwise nor(s2s1s0-101) ,bitwise xor(s2s1s0-110), bitwise xnor(s2s1s0-111).

OPERATION 1:



This output can be selected by giving 000 as select bits.
Clearly from the picture, if Transfer=0 and SUB=0, then B is as it is given to 8 bit full adder ,so addition operation without carry,since cin=0,if cin=1, then this leads to addition with carry.
if Transfer=0 and SUB=1, then xor gates will complement the input B and is given to 8 bit full adder with cin=1 to give subtracted answer ,again if cin=0,subtraction with borrow.
If Transfer=1 and SUB=0, all the B bits are AND with zeros, so that 00000000 is given to full adder so A is obtained as result if cin=0,again if cin=1,00000001 is added to A to increment A.

If Transfer=1 and SUB=1, all the B bits are AND with zeros and is complemented, so that 11111111 is given to full adder so A-1 is obtained as result if cin=0,again if cin=1,00000000 is added to A to Transfer A.

Finally conversion of 8 bit binary to 7 segment display in its decimal value.

DOUBLE DABBLE ALGORITHM:

The double dabble algorithm is used to convert binary numbers into binary-coded decimal (BCD) notation.
Suppose the original number to be converted is stored in a register that is *n* bits wide. Reserve a scratch space wide enough to hold both the original number and its BCD representation; $n + 4\times ceil(n/3)$ bits will be enough. It takes a maximum of 4 bits in binary to store each decimal digit.

Then partition the scratch space into BCD digits (on the left) and the original register (on the right). For example, if the original number to be converted is eight bits wide, the scratch space would be partitioned as follows:

100s	Tens	Ones	Original
0010	0100	0011	11110011

Let's do sample problem of converting 243 (i.e, 11110011)

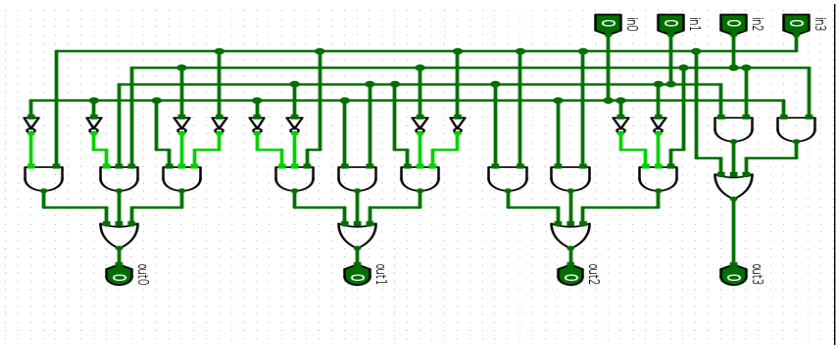
0000 0000 0000	11110011	Initialization
0000 0000 0001	11100110	Shift
0000 0000 0011	11001100	Shift
0000 0000 0111	10011000	Shift

0000 0000 1010 10011000 Add 3 to ONES, since it was 7
0000 0001 0101 00110000 Shift
0000 0001 1000 00110000 Add 3 to ONES, since it was 5
0000 0011 0000 01100000 Shift
0000 0110 0000 11000000 Shift
0000 1001 0000 11000000 Add 3 to TENS, since it was 6
0001 0010 0001 10000000 Shift
0010 0100 0011 00000000 Shift
2 4 3
BCD

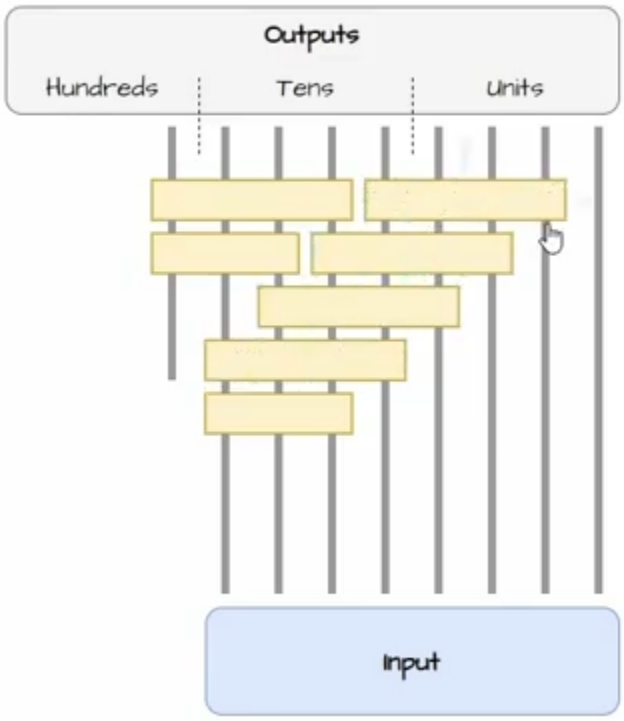
Implementation :

We will first develop a circuit which follows the step :
If number greater than or equals to 5 then add 3:

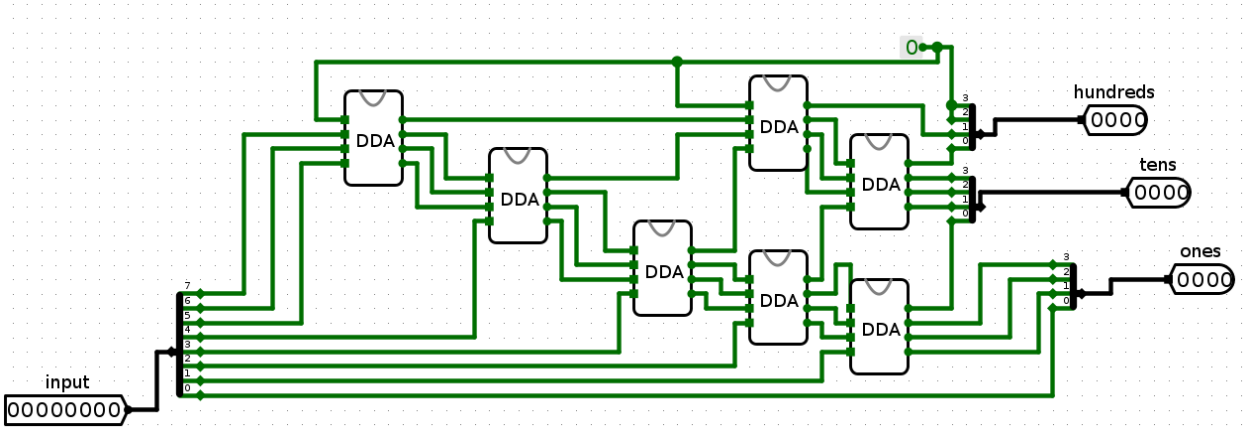
in3	in2	in1	in0	out3	out2	out1	out0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	1	1	1	1
1	1	0	1	1	1	0	0
1	1	1	0	1	1	0	1
1	1	1	1	1	1	1	0



Now assume this circuit as a yellow box.
Then the final 8 bit binary to bcd converter is as follows:



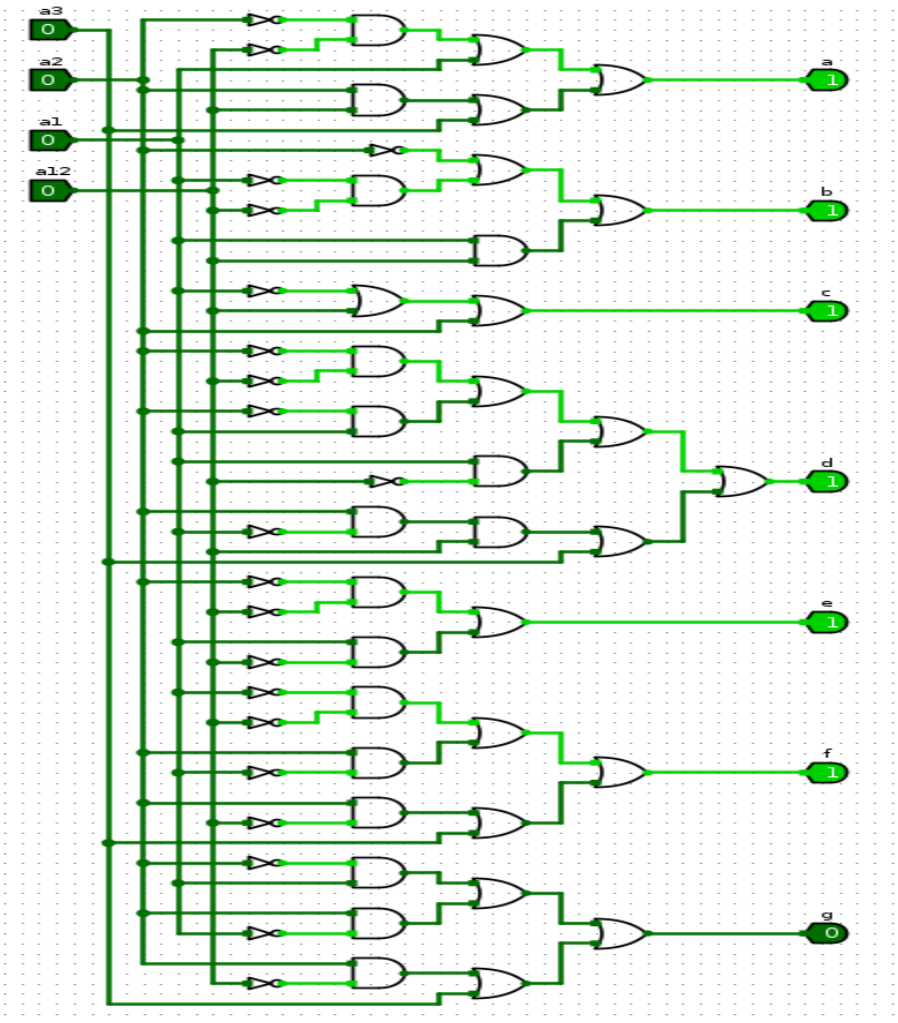
So ,



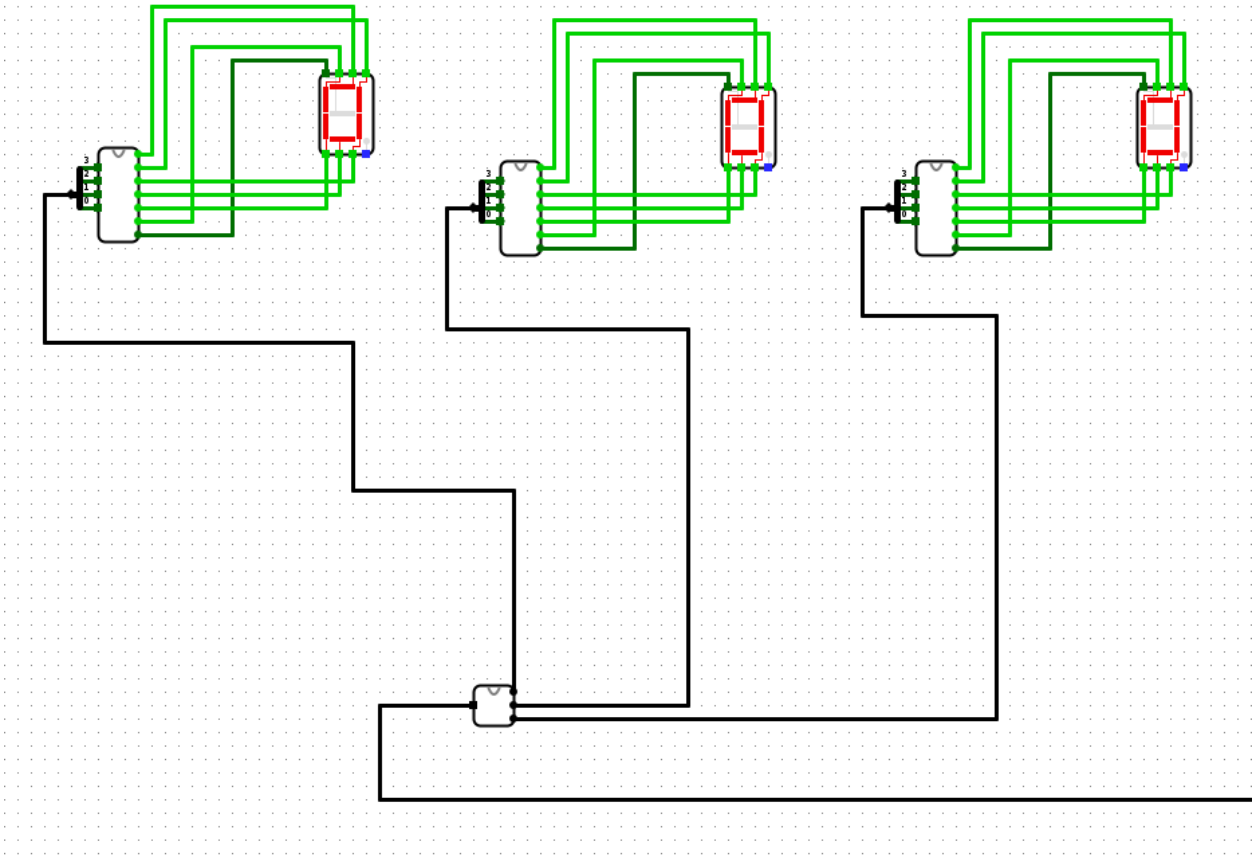
This is implemented according to the above diagram.

Now our task is to display 4 bit bcds to 7 segments.

a3	a2	a1	a12	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	0	1	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1	1
1	1	0	1	1	0	1	1	0	1	1
1	1	1	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1



These 7 outputs are connected to a 7 segment display.
The binary to 7 segment finally looks like this:



Rectangle elements are bcd to 7 segment converters.
Small square element the end is 8 bit binary to bcd converter.

Therefore the implementation of ALU is completed.

I Thank Dr.Noor sir for giving me this wonderful project.