EC302 VLSI Design Lab Project

Simple 4-bit ALU in Magic VLSI

BY KRUTI DEEPAN PANDA (191EC126),

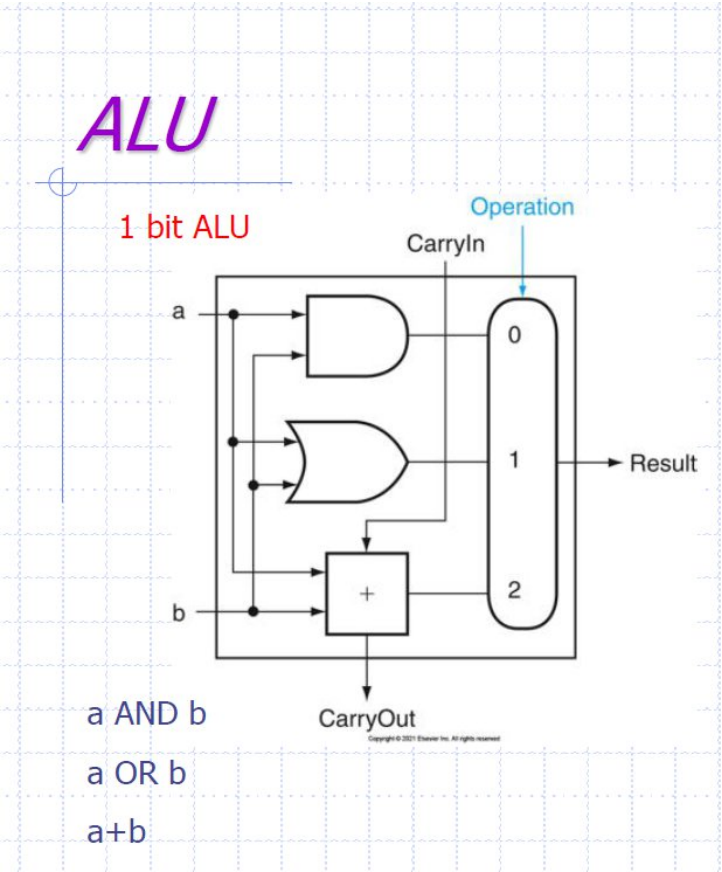
RAHUL MAGESH (191EC145),

ANIRUDH T (191EC106)

Objective:

To design a simple ALU that can perform basic operations such as addition, bitwise and, and bitwise or; along with necessary verification of the ALU.

Block circuit diagram:



This is the block diagram of the 1 bit ALU. We have made use of 4 such units to make our 4 bit ALU.

1 bit ALU corresponding to bit 3
1 bit ALU corresponding to bit 2
1 bit ALU corresponding to bit 1
1 bit ALU corresponding to bit 0

Description:

We have a made a general purpose 1 bit ALU that can be easily stacked on top of each other to create a n-bit ALU.

Each 1 bit ALU can perform 3 operations based on the opcode given to it -

Opcode : 00 = OR operation

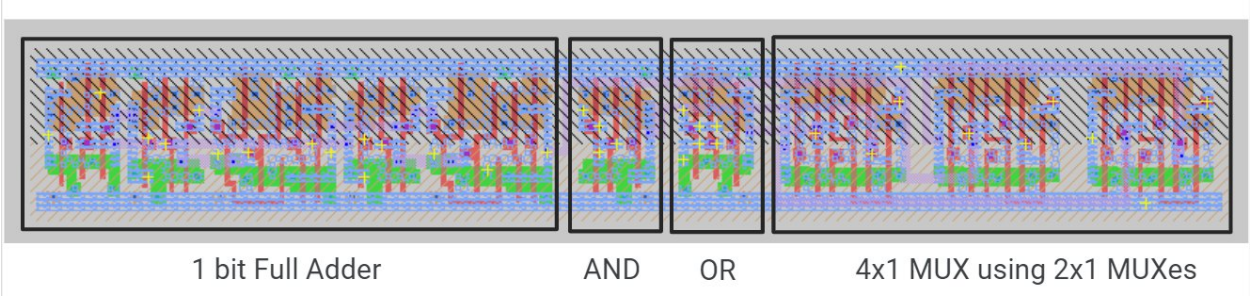
01 = AND operation

10 = ADD operation

The implemented ADD is an add with carry operation, which means we can give the first bit of the ALU a carry value that could be derived from the output of an external circuit.

In this case we have stacked 4 such ALUs to make a 4-bit ALU.

Implementation:

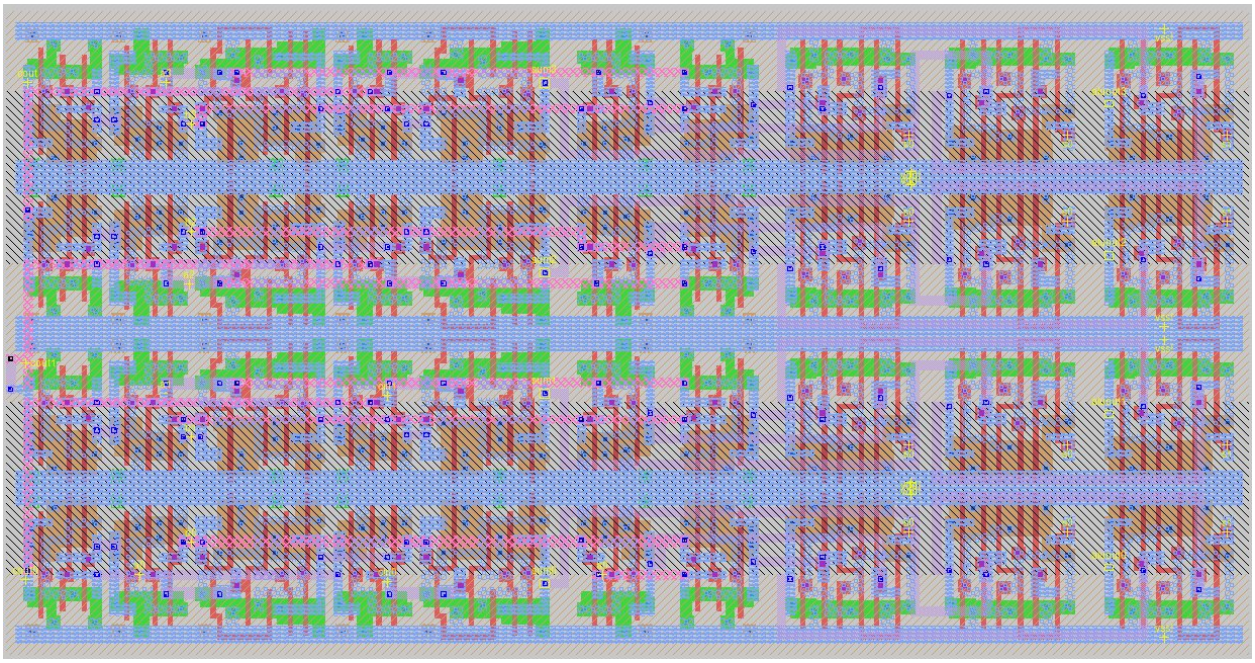


The vcslib library of standard cells from pharosc technology was used to make this ALU. The 1 bit ALU is made up of standard cells of 2x1 muxes used to make a 4x1 mux for our selection of output. The standard cells for OR and AND gates was also used.

A standard cell for full adder was made and used for the ALU.

All intracell connections were made using metal 1. Whereas intercell connections used metal2 and metal3, along with the use of via1 for metal1 to metal2 connection; and via2 for metal2 to metal3 connection.

Layout:



Test strategy:

The verification is started by running the shell script ***verify.sh*** which then runs the necessary commands for further testing.

In-order to test our ALU we have implemented a python script that randomly generates instructions for our ALU to run. It asks the user the number of instructions to generate.

After that it runs another python script to verify our ALU's output.

To start simulation type the following in the CLI -

`bash verify.sh`

verify.sh:

```
#!/bin/bash

python3 cmdGenerate.py
echo "CMD files generated"

irsim scmos100.prm ALU.sim -cmd/test.cmd
echo "IRSIM execution finished"

python3 logVerify.py
echo "Verification complete"
echo "Ending procedure"
```

cmdGenerate.py:

```
import random

tests = int(input('Enter how many tests you want to perform : '))

high = 'vdd!'
low = 'vss!'

node_opA = ['a3', 'a2', 'a1', 'a0']
node_opB = ['b3', 'b2', 'b1', 'b0']
node_carry = 'cout'
node_out = ['aluout3', 'aluout2', 'aluout1', 'aluout0']
node_op = ['s1', 's0']
node_in = node_op + node_opA + node_opB

vectors = { 'A': ['A', node_opA],
            'B': ['B', node_opB],
            'Opcode': ['op', node_op],
            'Carry out': ['C_out', node_carry],
            'Out': ['out', node_out],
            'In' : ['in', node_in]
            }

# filename_cmd = cmd_path + 'test.txt'
filename_cmd = 'cmd/test.cmd'
filename_log = 'log/test.txt'

file = open(filename_cmd, 'w')

#IRSIM codes
#-------------------------------------------------------
#setting up the file
file.write(f'logfile {filename_log}\n')
file.write('stepsize 50\n')
file.write(f'h {high} \n')
file.write(f'l {low} \n')
file.write(f'l cin \n')

#generating vectors
for vector_name in vectors:
    vector = vectors[vector_name]
    file.write(f'vector {vector[0]} ')
    for node in vector[1]:
        if vector_name == 'Carry out':
            file.write(f'{vector[1]} ')
            break
        file.write(f'{node} ')
    file.write(f'\n')

#watching vectors
```

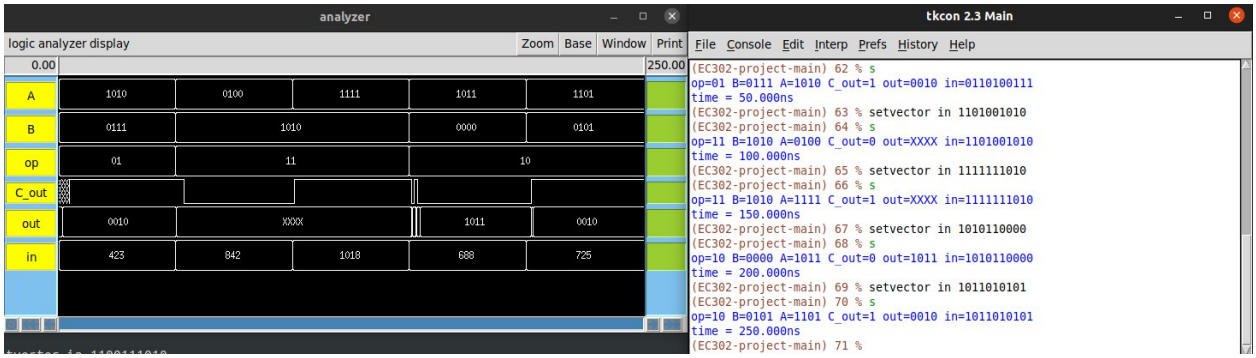
```
file.write(f'w in out C_out a b op \n')

#setting up analyzer
file.write('analyzer ')
for vector_name in vectors:
    vector = vectors[vector_name]
    file.write(f'{vector[0]} ')
file.write('\n')

#running simulations
for i in range(tests):
    instruction = random.randint(0, 2**10)
    file.write(f'setvector in {instruction:010b}\n')
    file.write('s\n')
file.write('logfile\n')
file.write('exit')
file.close
```

logVerify.py:

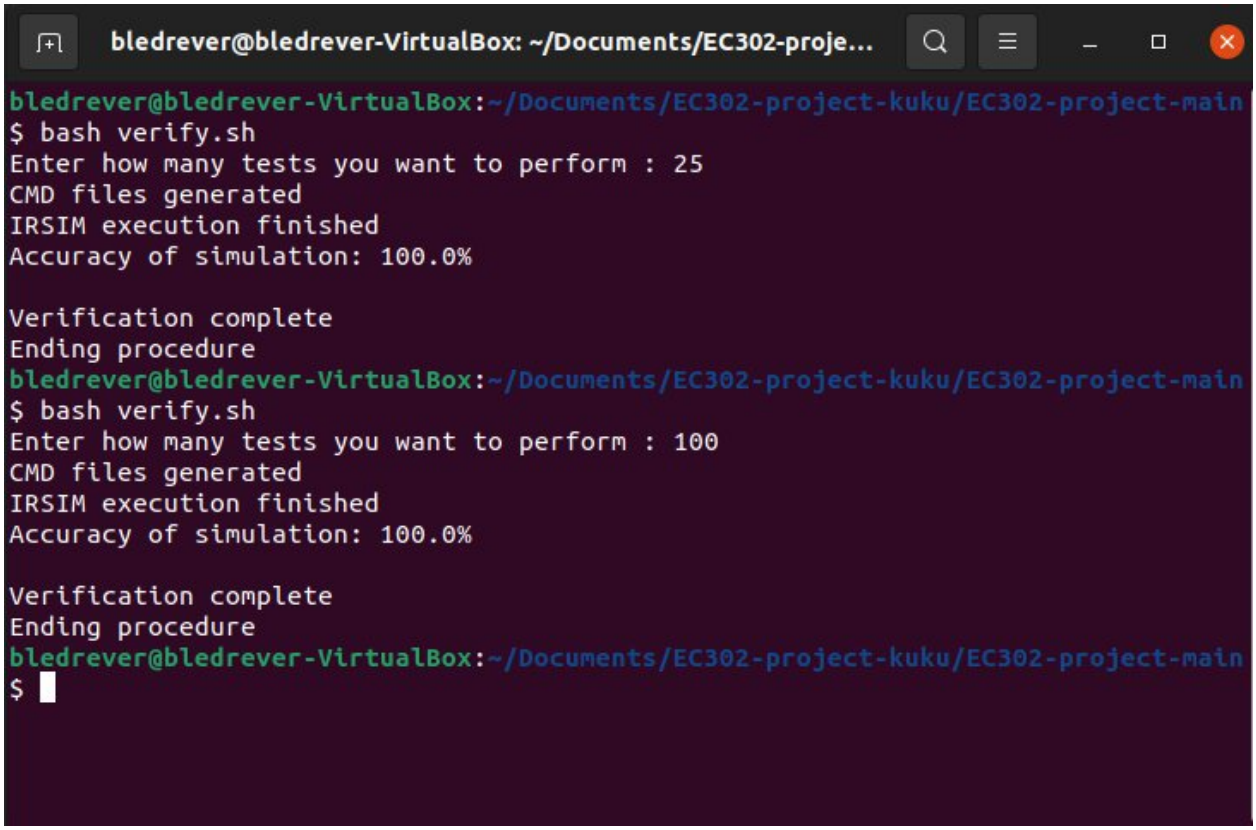
```
def checkAND(opA, opB, aluout):
    expected = opA & opB
    if aluout==expected:
        return True
    return False

def checkADD(opA, opB, aluout, cout):
    expected = opA + opB
    # cout for overflow
    if aluout + 16*cout == expected:
        return True
    return False

def checkOR(opA, opB, aluout):
    expected = opA | opB
    if aluout==expected:
        return True
    return False

correct=0
testcases=0

file = open('log/test.txt', "r")
lines = file.readlines()

#reading the log file
for i in range(0, len(lines), 2):
    log = lines[i].split()

    # read data from file
```

```
        op = (int(log[1][3:], 2))
        B = (int(log[2][2:], 2))
        A = (int(log[3][2:], 2))
        cout = (int(log[4][6], 2))

        out = 0
        if op != 3:
            out = (int(log[5][4:], 2))
        else:
            out = -100

        if op==0:
            flag=checkOR(A, B, out)
        elif op==1:
            flag=checkAND(A, B, out)
        elif op==2:
            flag=checkADD(A, B, out, cout)
        else:
            if log[5][4:] == 'XXXX':
                flag = True
            else:
                flag = False

        if flag:
            correct+=1
        else:
            print("The following inputs are causing unexpected behaviour-")
            print(f"\tA: {A}, B: {B}, op:{op}")
        testcases+=1

    accuracy = (correct*100.0)/testcases
    print(f"Accuracy of simulation: {accuracy}%\n")
    file.close()
```

Results:



Note that for cases where the opcode is 11 the output is don't care as no operation has been defined.



It takes an area of 169061 sq. microns.

Conclusion:

The designed ALU is 100% accurate. The designed 4 bit ALU is working as expected.