PROJECT REPORT

EC302 - VLSI DESIGN LAB

COMPILED BY
AKASH A, NAVRATAN AND ANAND KUMAR SINGH
ROLL NUMBERS : 191EC102, 191EC133, 191EC104

PROFESSOR INCHARGE : DR. RAMESH KINI M

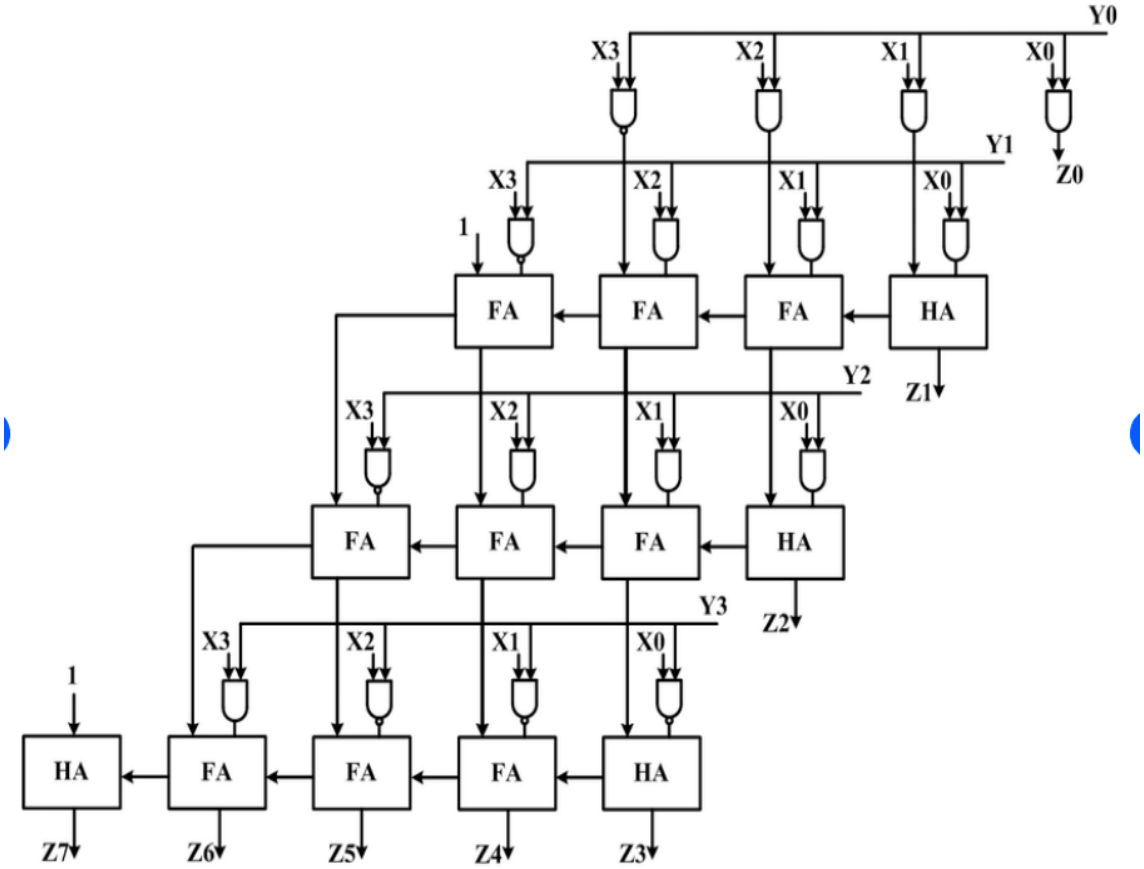
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL

Four Bit Signed Two's Complement Multiplier

OBJECTIVES

The project aims to perform the design, layout and characterization of a 4 bit multiplier in two's complement form. The multiplier takes in two 4-bit numbers (could be signed or unsigned) in two's complement form and the result is the product of the two numbers which is an 8-bit two's complement number.

CIRCUIT DIAGRAM



DESCRIPTION OF DESIGN

There are 3 main methods for multiplying binary numbers in 2's complement representation: shift-and-add method, Booth/Wallace method and array method. In this project, the array method of multiplication has been implemented which is the simplest and has the advantage of being easily pipelined for higher performance.

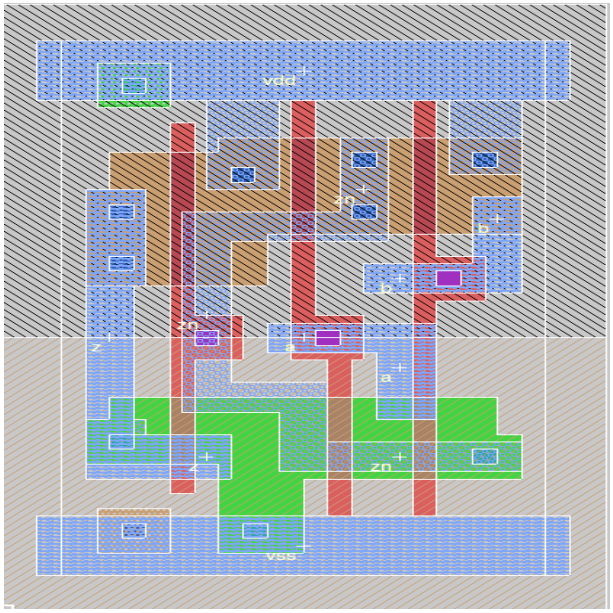
The formula for multiplication of binary arguments expressed as sums of products of bit values and weights of positions in the 2's complement representation can be transformed, through appropriate term grouping and introduction of bit complements, to the form, which is a basis for constructing an array multiplier composed of elementary adders, shown in the figure above. In the array of elementary adders so many rows appear how many bits are in the number that are multiplied. On the inputs of the initial layer of elementary adders and to carry inputs at lower levels, appropriate logical products are supplied of multiplicand and multiplier bits (simple and complemented). The results of elementary additions (sum and carry bits) are supplied on inputs of adders in lower layers or they constitute already result bits, depending on the position of a given adder in the array.

IMPLEMENTATION DETAILS

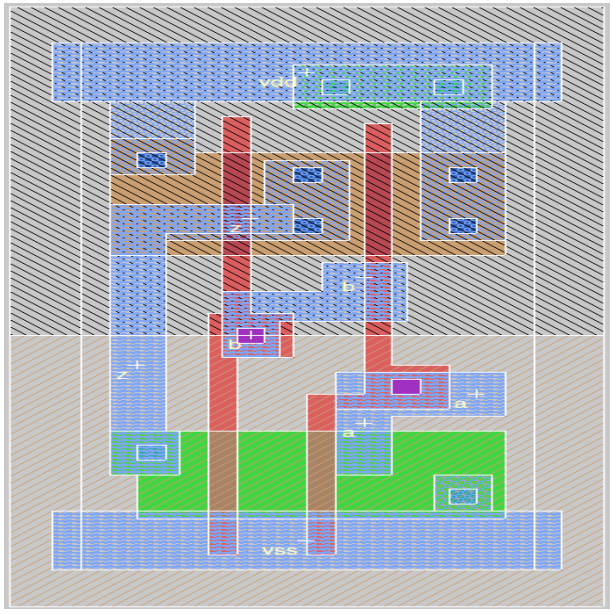
The RTL was designed using standard cells precreated using the technology file *pharosc.tech*. For the layout, over the cell routing technique was used as far as possible with some feedthroughs included for the sake of providing space for wiring. This helped in save space and create the circuit in a smaller area. For intra-cell routing, metal layer M1 was used while metal layers M2 and M3 were used for inter-cell routing. Top metal i.e. metal 3 in this case was used for the V_{DD} and V_{SS} rails to avoid the ground bounce problem since top metals have lesser resistance.

Cells Used

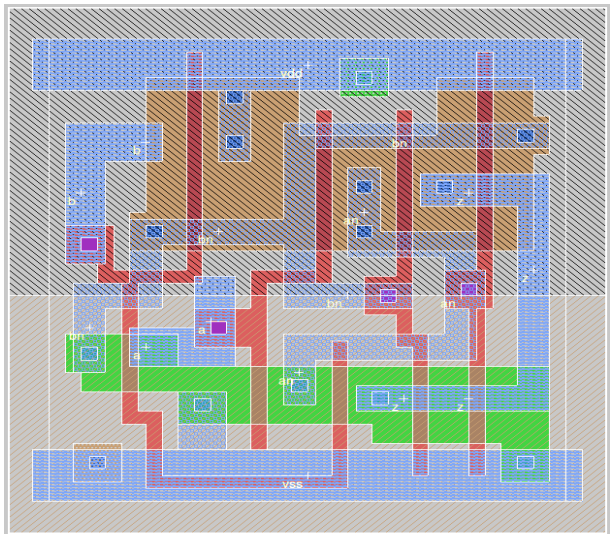
a) Two input AND Gate to generate partial products and to generate carry for half adder



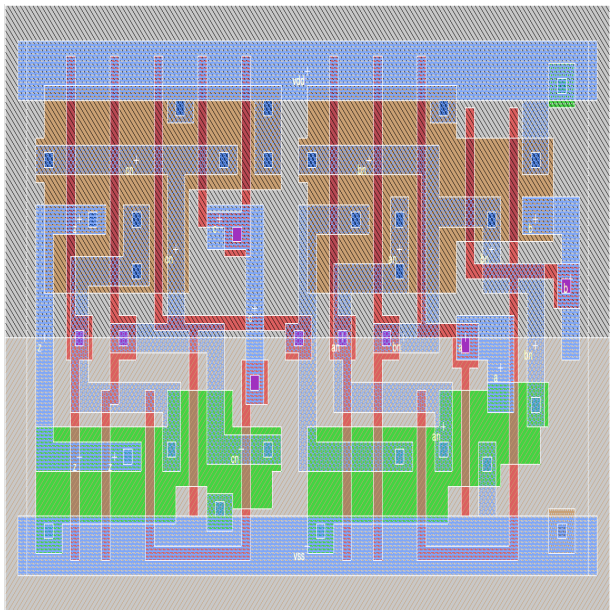
b) Two input NAND gate for signed partial products



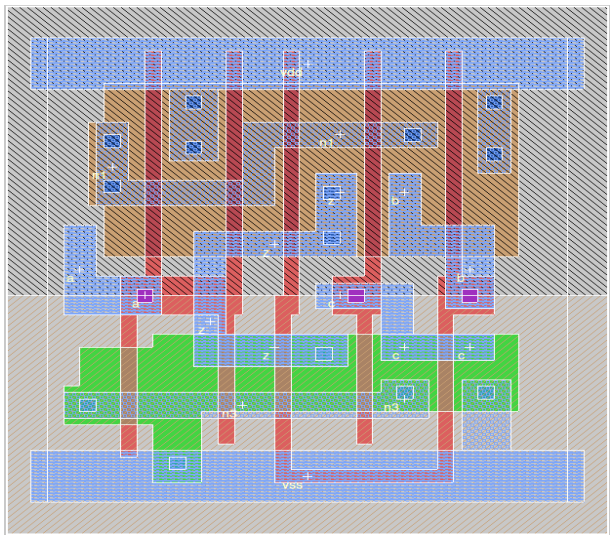
c) Two input XOR gate for half adder



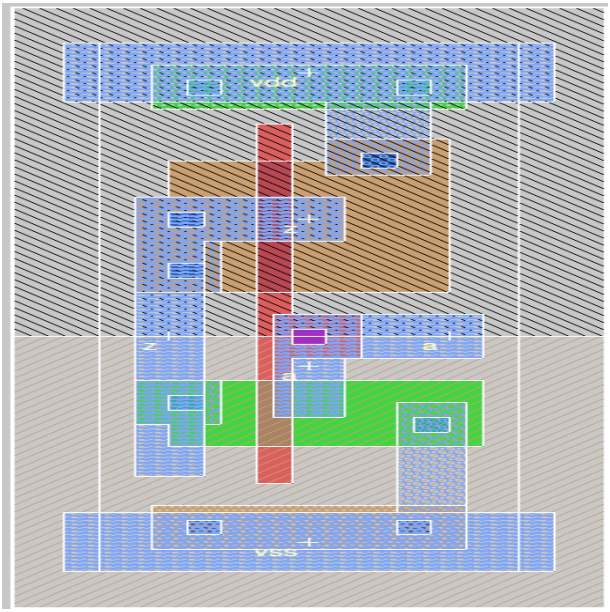
d) Three input XOR gate for full adder



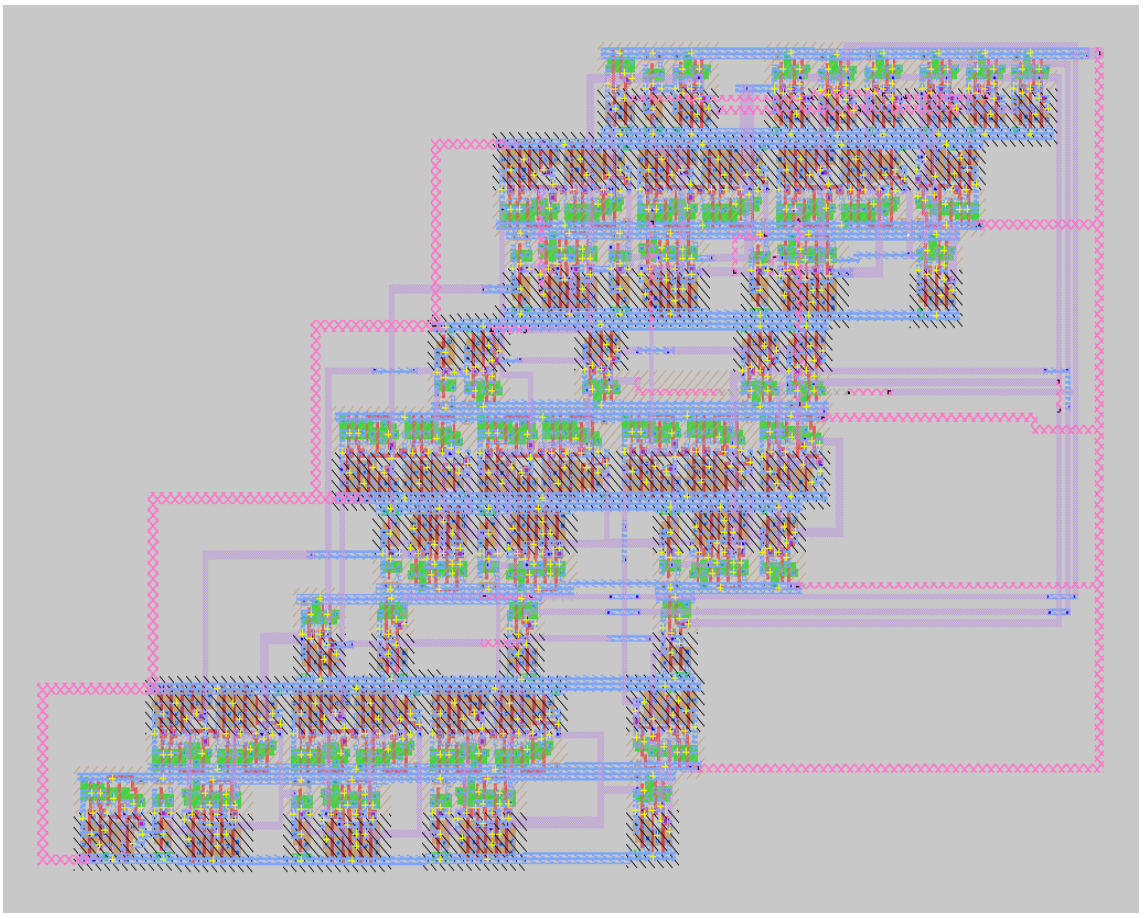
e) Three input carry generator to generate full adder carry



f) Inverter to invert output of carry generator



TOP LEVEL LAYOUT



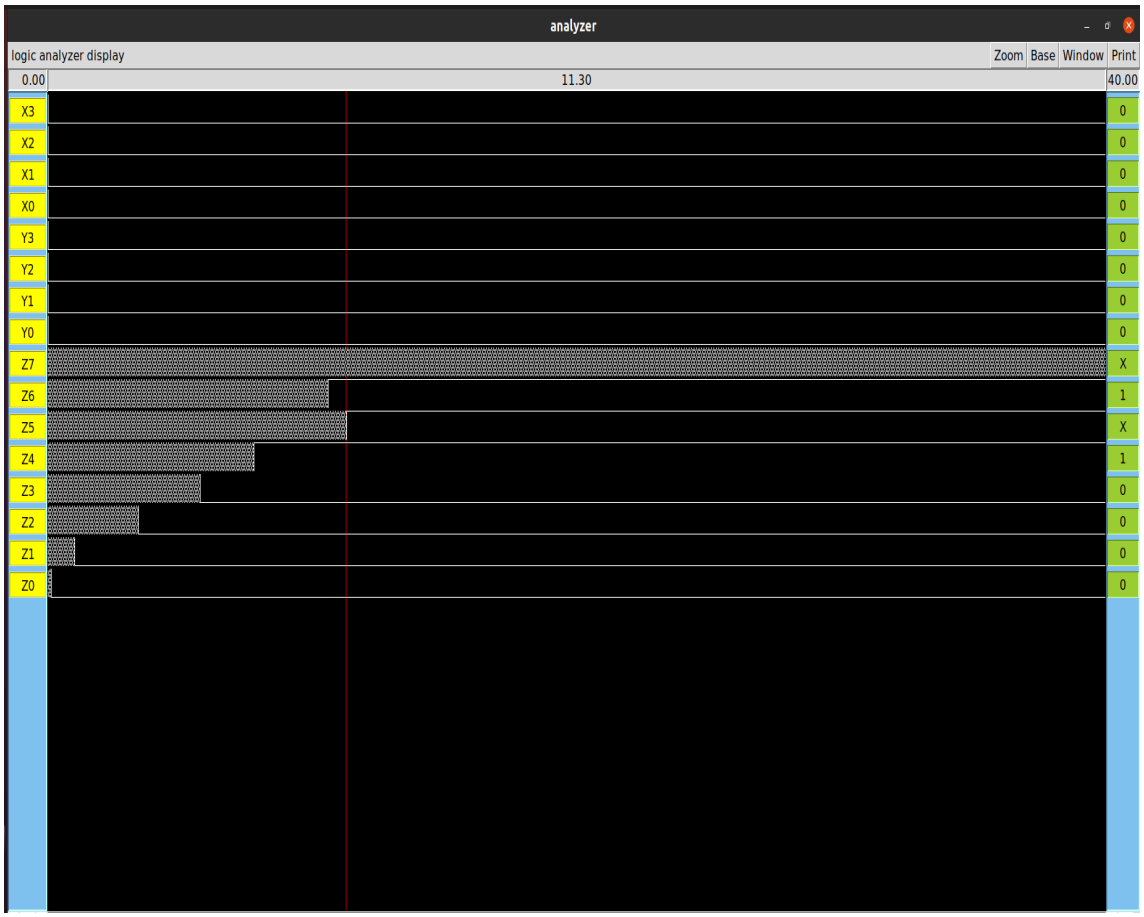
Total Number of Cells = 52
Area = (186+829)*(151+496) = 656,705 sq. microns

TEST STRATEGY

IRSIM, a circuit level simulation software was used to test the created .sim file for accuracy. In order to verify if the created circuit is correct, it was essential to test it for all possible binary numbers from 0000 to 1111 i.e. all possible four-digit binary numbers. The following python script can be used to generate all possible 4-digit binary numbers which henceforth needed to be permuted with itself to obtain the actual simulation test vextors for the two four-digit binary numbers.

```
def printTheArray(arr, n):

    for i in range(0, n):
        print(arr[i], end = " ")

    print()

# Function to generate all binary strings
def generateAllBinaryStrings(n, arr, i):

    if i == n:
        printTheArray(arr, n)
        return

    # First assign "0" at ith position
    # and try for all other permutations
    # for remaining positions
    arr[i] = 0
    generateAllBinaryStrings(n, arr, i + 1)

    # And then assign "1" at ith position
    # and try for all other permutations
    # for remaining positions
    arr[i] = 1
    generateAllBinaryStrings(n, arr, i + 1)

# Driver Code
if __name__ == "__main__":

    n = 4
    arr = [None] * n

    # Print all binary strings
    generateAllBinaryStrings(n, arr, 0)
```

WAVEFORM



CONCLUSION

We observe that the results obtained are not exactly as we might have expected. There are some errors which are obtained. This can be attributed to bad layout practices since odd metal layers were run vertically and horizontally and same was true for even metal layers. Another reason could have been the lack of a modular approach in the design. These are two things that could have been improved on and could have given the correct result if followed.