**Week 12**

**1.Write a program to implement Digital Logic Gates – AND, OR, NOT, EX-OR**

**Program(and):**

def AND(a,b):

if a==1 and b==1:

return True

else:

return False

print(AND(1,1),"=AND(1,1)")

print(AND(0,0),"=AND(0,0)")

print(AND(1,0),"=AND(1,0)")

print(AND(0,1),"=AND(0,1)")

**output:**

True =AND(1,1)

False =AND(0,0)

False =AND(1,0)

False =AND(0,1)

**Program(or):**

def OR(a,b):

if a==1 or b==1:

return True

else:

return False

print(OR(1,1),"=OR(1,1)")

print(OR(0,0),"=OR(0,0)")

print(OR(1,0),"=OR(1,0)")

print(OR(0,1),"=OR(0,1)")

**output:**

True =OR(1,1)

False =OR(0,0)

True =OR(1,0)

True =OR(0,1)

**Program(not):**

def NOT(a):

if a==1 :

return False

else:

return True

print(NOT(1),"=NOT(1)")

print(NOT(0),"=NOT(0)")

**output:**  
False =NOT(1)

True =NOT(0)

**Program(EX-OR):**

def XOR\_gate(a, b):

if a != b:

return 1

else:

return 0

print("XOR Gate:", XOR\_gate(5,5))

**output:**

XOR Gate: 0

1. **Write a program to implement Half Adder, Full Adder, and Parallel Adder.**

**Program(Half Adder):**def XOR(a,b):

if a!=b:

return 1

else:

return 0

def AND(a,b):

if a==b:

return 1

else:

return 0

def half\_adder(a,b):

sum=XOR(a,b)

carry=AND(a,b)

return sum,carry

sum,carry=half\_adder(0,0)

print(sum,carry)

sum,carry=half\_adder(0,1)

print(sum,carry)

sum,carry=half\_adder(1,0)

print(sum,carry)

sum,carry=half\_adder(1,1)

print(sum,carry)

**output:**

0 1

1 0

1 0

0 1

**Program(full adder):**  
def half\_adder(a,b):

sum=a^b

carry = a and b

return carry,sum

def full\_adder(carry\_in,a,b):

carry1,sum1=half\_adder(carry\_in,a)

carry2,sum=half\_adder(sum1,b)

carry=carry1 or carry2

return carry,sum

carry,sum=full\_adder(0,0,0)

print(sum,carry)

carry,sum=full\_adder(0,0,1)

print(sum,carry)

carry,sum=full\_adder(0,1,0)

print(sum,carry)

carry,sum=full\_adder(0,1,1)

print(sum,carry)

carry,sum=full\_adder(1,0,0)

print(sum,carry)

carry,sum=full\_adder(1,0,1)

print(sum,carry)

carry,sum=full\_adder(1,1,0)

print(sum,carry)

carry,sum=full\_adder(1,1,1)

print(sum,carry)

output:

0 0

1 0

1 0

0 1

1 0

0 1

0 1

1 1

**Prallel Adder:**

#

# To use these functions, you can run python and then import like -

# from binary\_adder import \*

#

# These methods carry out binary addition via 'digital logic'

# This is really what happens at the logic circuit level.

# So, this is a pretty abstract use of programming to illustrate

# what happens on silicon using code many, many, levels above that!

#

# A binary half adder -- performing addition only using logic operators,

# A half adder simply adds two bits and outputs a sum and carry

#

def half\_adder(a, b):

# ^ is logical xor in python

sum = a ^ b

carry = a and b

return carry,sum

# A binary full adder

# The full adder can add 3 bits (can handle an incoming carry)

# Also returns a sum and carry

#

def full\_adder(carry\_in, a, b):

carry1,sum1 = half\_adder(carry\_in,a)

carry2,sum = half\_adder(sum1,b)

carry = carry1 or carry2

return carry,sum

# This method virtually chains together binary full adders in order

# to add binary numbers of arbitrary size.

#

# a and b are expected to be strings representing binary integers.

#

#

def binary\_adder(a,b):

an = len(a)

bn = len(b)

# Convert strings to list of bits -- very functional syntax here

al = list(int(x,2) for x in list(a))

bl = list(int(x,2) for x in list(b))

# Pad smaller list with 0's

dif = an - bn

# more digits in a than b

if dif > 0:

for i in range(dif):

bl.insert(0,0)

else:

for i in range(abs(dif)):

al.insert(0,0)

print(al)

print(bl)

result = []

carry = 0

# Iterate through list right to left, calling full\_adder each time and

# inserting the sum each time

for i in range(len(al)-1,-1,-1):

carry,sum = full\_adder(carry,al[i],bl[i])

result.insert(0,sum)

print (result)

result.insert(0,carry)

return ''.join(str(x) for x in result)

def test\_binary\_adder(a,b):

result = binary\_adder(a,b)

print(result)

if (int(a,2) + int(b,2)) == int(result,2):

print("Woo hoo! It works")

else:

print("FAIL!!")

print(str(int(a,2)) + " + " + str(int(b,2)) + " = " + str(int(result,2)))

test\_binary\_adder('11111','11111')

**Output:**

[1, 1, 1, 1, 1]

[1, 1, 1, 1, 1]

[0]

[1, 0]

[1, 1, 0]

[1, 1, 1, 0]

[1, 1, 1, 1, 0]

111110

Woo hoo! It works

31 + 31 = 62