Ques-1

‘a’ and ‘b’ are the two 4-bit unsigned numbers given as input to the ALU. ‘result’ is the 8-bit unsigned number which is output after performing the operation. ‘sel’ is the 3-bit mode selection input. And depending on the ‘sel’ value , we had to perform different operations using structural VHDL code only.

So in each of the parts, I created a input pin which will check the condition of sel pin.
If input is 1, then the particular part will return the result and if input pin is 0, the particular part will send sequence of only 0’s as the output. In this way , I took the result of 7 subparts and to calculate the final result , I performed bitwise OR operation which will in turn perform the operation of selecting the subpart based on sel.

For example:
a = 0001 and b = 0001
If sel = 010 then outputs of the subparts will be:
00000000,00000001,00000000,00000000,00000000,00000000,00000000
And performing bitwise OR will give the final result same as the result of the 2nd subpart.

sel = 000 or 001 - Adder_Subtractor (not sel2 and not sel1)
sel = 010 - Multiplier (not sel2 and sel1 and not sel0)
sel = 011 - Comparator (not sel2 and sel1 and sel0)
sel = 100 - Bitwise NAND (sel2 and not sel1 and not sel0)
sel = 101 - Bitwise NOR (sel2 and not sel1 and sel0)
sel = 110 - Bitwise XOR (sel2 and sel1 and not sel0)
sel = 111 - Bitwise XNOR (sel2 and sel1 and sel0)

Subpart A - Adder and Subtractor:

It is a 4-bit unsigned adder-subtractor which had to be designed using a carry-look-ahead adder circuit. During this operation, the lower 4 bits of 'result' should represent the sum (for adder) and difference (for subtractor) and the 5th bit is the carry. The remaining upper 3 bits of 'result' are don't care. When sel is 000 perform addition and when sel is 001, we have to perform subtraction.

In general, there are 2 ways to subtract a and b. One is the normal borrow method which cannot be used here because we have to use a 4 bit ripple carry adder for subtraction also. The 2nd method is to first take 2's complement of b that is inverse all the bits of b and add 1 to get 2's complement of b and then add a and the 2's complement of b which will finally result in a-b.

Method for finding 2's complement of b:

First we can inverse the bits of b by performing a XOR operation of each of the bits of b with cin=1. This is known as the 1's complement of b. Now to convert 1's complement of b to 2's complement of b , we need to add 1. So instead of adding 1 here , i will give the 4 bit adder a carry of 1 to satisfy the 2's complement of b.

So finally **a - b = a + (2's complement of b)**

= a + (1's complement of b + 1)

= a + (1's complement of b) + (carry = 1)

When cin = 0

Input will be 'a' , 'b' , 'cin=0'

When cin = 1

Input will be 'a' , '1's complement of b', 'cin=1'

Here as we have to use only the adder circuit, we will calculate the subtraction using 2's complement method which is

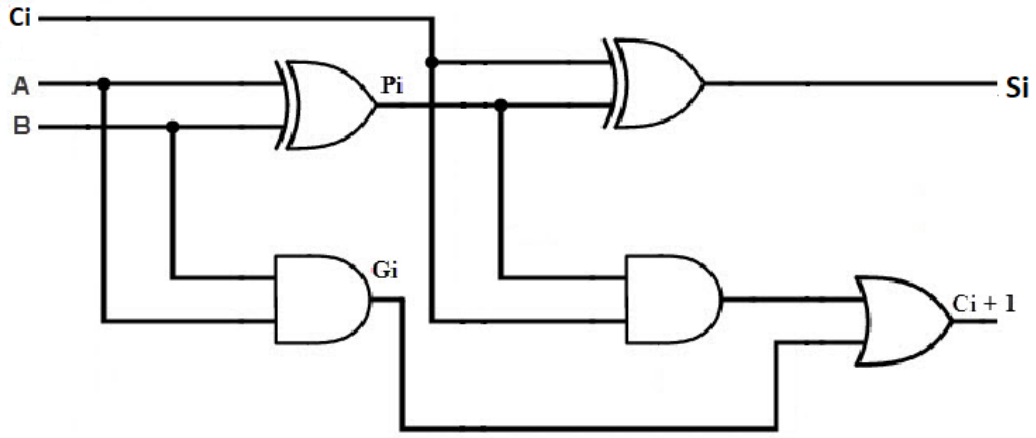
a - b = a + (2's complement of b)

Logic For Carry Look Ahead Adder:

In Stage i of a Full Adder, we are completely sure that there will be a carry generated when Ai = Bi = 1, independent of whether the carry-in from stage i-1 is 0 or 1.

Similarly there will be a carry propagated if the half sum is 1 and carry in Ci is 1.

These two signal conditions are called generate and propagate denoted by Gi and Pi respectively as shown in below circuit.



Pi = Ai XOR Bi

Gi = Ai AND Bi

Si = Pi XOR Ci

Ci+1 = Gi + Pi.Ci

In the ripple carry adder : Gi, Pi, Si are local to each cell of the adder and Ci is also local to each cell. But in the carry look ahead adder, in order to reduce the length of the carry chain, Ci is changed to a more global function spanning multiple cells.

Now we will flatten the equations for carry using Gi and Pi terms for less significant bits. We will begin at the cell 0 with the carry-in C0.

C1 = G0 + P0.C0

C2 = G1 + P1.C1
= G1 + P1.(G0 + P0.C0)
= G1 + P1.G0 + P1.P0.C0

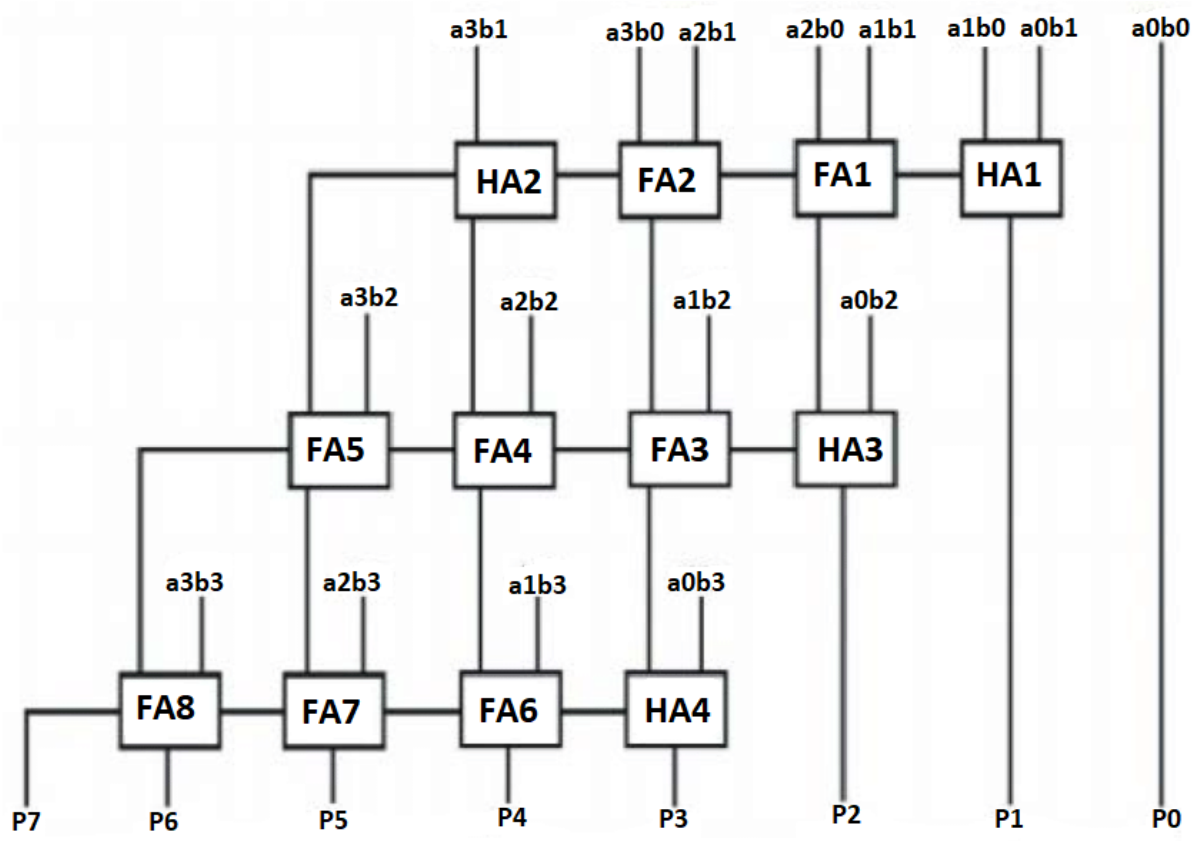
C3 = G2 + P2.C2
= G2 + P2.(G1 + P1.G0 + P1.P0.C0)
= G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0

C4 = G3 + P3.C3
= G3 + P3.(G2 + P2.G1 + P2.P1.G0 + P2.P1.P0.C0)
= G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.C0

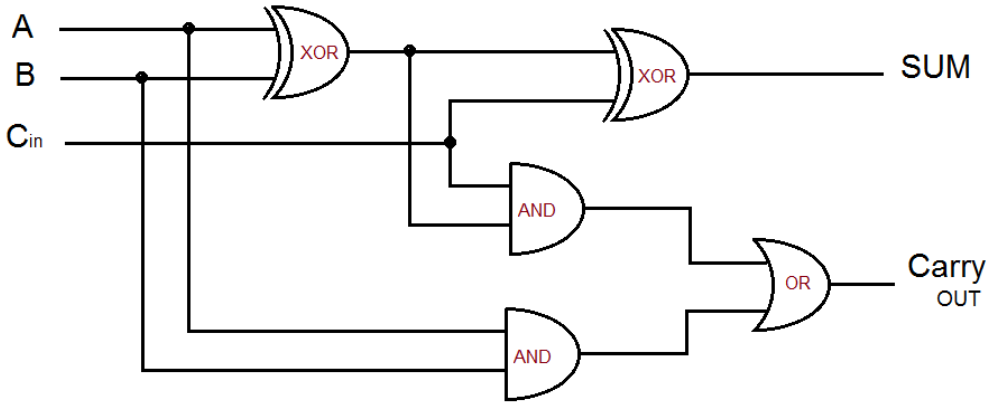
Subpart B - Multiplier:

				A3	A2	A1	A0
				B3	B2	B1	B0
				A3B0	A2B0	A1B0	A0B0
			A3B1	A2B1	A1B1	A0B1	
		A3B2	A2B2	A1B2	A0B2		
	A3B3	A2B3	A1B3	A0B3			
P7	P6	P5	P4	P3	P2	P1	P0

AiBi = Ai AND Bi



Array multiplier is just like the normal multiplication process that we use in our daily life.
Full Adder Diagram:



Subpart C - Comparator:

In this subpart, we have to design a 4 bit unsigned comparator using structural code. The comparator should be able to tell whether 'a' is less than or greater than or equal to 'b'. During comparator operation, only the lower 3 bits of 'result' will be used. The remaining upper 5 bits are don't care.

When a > b , output = xxxxx100
When a = b , output = xxxxx010
When a < b , output = xxxxx001

Let two inputs be a3 a2 a1 a0 and b3 b2 b1 b0
The basic logic of the comparator is that first we have to compare the most significant bit, If both are different the input containing 1 is larger but if both are same either 0,0 or 1,1 then we will have to compare the 2nd most significant bit and similarly we have to check all the bits, If all the 4 bits are equal then both the numbers are equal.

Here equality can be checked by using the xnor of the two bits.
The logic of the code is shown below:
s3 = a3 xnor b3
s2 = a2 xnor b2
s1 = a1 xnor b1
s0 = a0 xnor b0
Each will contain 1 if corresponding bits are equal and 0 otherwise.

Case-1 : When a = b

This case will be satisfied when all the 4 bits are equal of both the inputs.
And the output should be xxxxx010.
So result(1) = s3 AND s2 AND s1 AND s0
If all are 1 , then the two inputs are equal and the output will be as required

Case - 2: When a > b

This case will be satisfied during one of the 4 subcases mentioned below:
When a3 is 1 and b3 is 0 , code will be (a3 AND not b3)
When a3 and b3 are same i.e s3 is 1 and a2 is 1 and b2 is 0,
Code will be (s3 AND a2 AND not b2)

When a3 is same as b3 and a2 is same as b2 i.e s3=1 and s2=1 and a1 is 1 and b1 is 0
Code will be (s3 AND s2 AND a1 AND not b1)

When a3 = b3 and a2 = b2 and a1 = b1 i.e s3=s2=s1=1 and a0 is 1 and b1 is 0
Code will be (s3 AND s2 AND s1 AND a0 AND not b0)

When one of the above cases will be satisfied, a will be greater than b. So final logic will be or of these conditions. And as the output will be xxxxx100 , so result should be assigned to result(2).

result(2) = (a3 AND not b3) OR
(s3 AND a2 AND not b2) OR
(s3 AND s2 AND a1 AND not b1) OR
(s3 AND s2 AND s1 AND a0 AND not b0)

Case - 3: When a < b

This case will be satisfied during one of the 4 subcases mentioned below:
When a3 is 0 and b3 is 1 , code will be (b3 AND not a3)

When a3 and b3 are same i.e s3 is 1 and a2 is 0 and b2 is 1,
Code will be (s3 AND b2 AND not a2)

When a3 is same as b3 and a2 is same as b2 i.e s3=1 and s2=1 and a1 is 0 and b1 is 1
Code will be (s3 AND s2 AND b1 AND not a1)

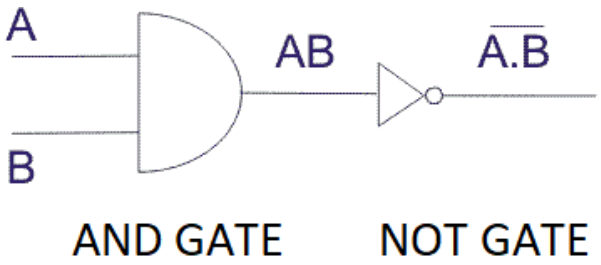
When a3 = b3 and a2 = b2 and a1 = b1 i.e s3=s2=s1=1 and a0 is 0 and b1 is 1
Code will be (s3 AND s2 AND s1 AND b0 AND not a0)

When one of the above cases will be satisfied, a will be greater than b. So final logic will be or of these conditions. And as the output will be xxxxx001 , so result should be assigned to result(0).

result(0) = (b3 AND not a3) OR
(s3 AND b2 AND not a2) OR
(s3 AND s2 AND b1 AND not a1) OR
(s3 AND s2 AND s1 AND b0 AND not a0)

Subpart D - Bitwise NAND :

In this subpart, we have to design a 4-bit bitwise NAND operation using a structural design.
NAND(a,b) = NOT(a AND b)

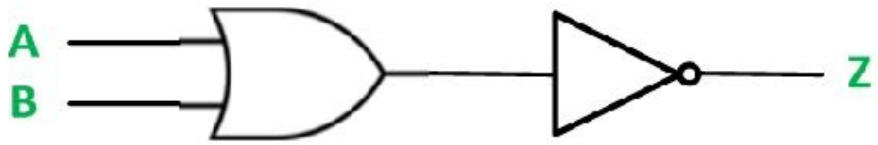


Input	Input	Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Let a = a3 a2 a1 a0 and b = b3 b2 b1 b0
a bitwiseNAND b can be defined as:
result(3) = a3 NAND b3
result(2) = a2 NAND b2
result(1) = a1 NAND b1
result(0) = a0 NAND b0

Subpart E - Bitwise NOR :

In this subpart, we have to design a 4-bit bitwise NOR operation using a structural design.
NOR(a,b) = NOT(a OR b)

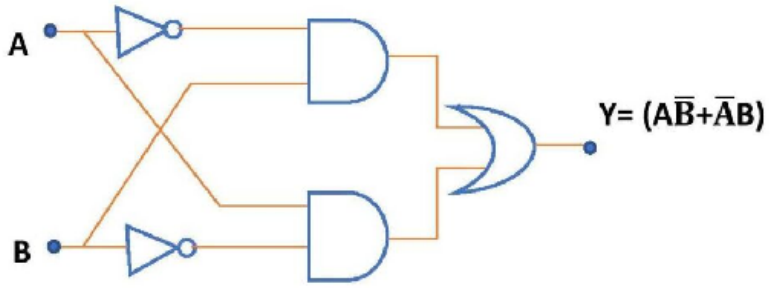


A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

Let a = a3 a2 a1 a0 and b = b3 b2 b1 b0
a bitwiseNOR b can be defined as:
result(3) = a3 NOR b3
result(2) = a2 NOR b2
result(1) = a1 NOR b1
result(0) = a0 NOR b0

Subpart F - Bitwise XOR :

In this subpart, we have to design a 4-bit bitwise XOR operation using a structural design.
XOR(a,b) = (a AND not b) OR (not a AND b)

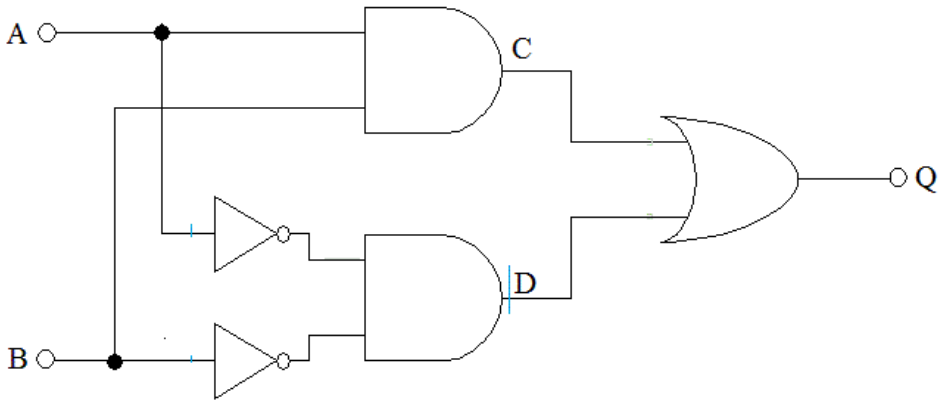


INPUTS		OUTPUTS
A	B	Y = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Let a = a3 a2 a1 a0 and b = b3 b2 b1 b0
a bitwiseXOR b can be defined as:
result(3) = a3 XOR b3
result(2) = a2 XOR b2
result(1) = a1 XOR b1
result(0) = a0 XOR b0

Subpart G - Bitwise XNOR :

In this subpart,we have to design a 4-bit bitwise XNOR operation using a structural design.
XNOR(a,b) = (a AND b) OR (not a AND not b)



Inputs		Outputs
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

Let a = a3 a2 a1 a0 and b = b3 b2 b1 b0
a bitwiseXNOR b can be defined as:
result(3) = a3 XNOR b3
result(2) = a2 XNOR b2
result(1) = a1 XNOR b1
result(0) = a0 XNOR b0