RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC203A
Course Name	Logic Design
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	03/2018
Course Leader/s	Mr. Narsimha Murthy

Declaration Sheet			
Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	03/2018
Course Code	CSC203A		
Course Title	Logic Design		
Course Date		to	
Course Leader	Mr. Narasimha Murthy		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Question No. 1

Solution to Question No. 1 Part A:

A 1.1 Introduction:

Namely there two types of adders, like serial adder and parallel adder.in a parallel adder, the length n of augend and the addend with the input carry of LSB are available in registers and given as inputs to the respective adders. This is a faster than serial adder. The speed is attenuated by time needed for the carry produced, propagates to stages upto MSB. Hence the name Ripple is acquired. The speed of a adder is reduced by the time. That much time should be given for the outputs to settle before the inputs vary. The output carry-in a certain stage of adder completely depends on the augend and addend of the last stage. Holding the argument until the final stag $(n-1)$ stage. In ripple carry adder there is much dependency of output. Hence there exists a delay. This needs the carries that could be generated or propagated through the stages from LSB to MSB. This requirement could be achieved through another adder called "Carry look ahead adder" or "Fast adder". The layout of ripple carry adder is simple, which allows for fast design time however, RCA is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder.

A 1.2 Comparison of the complexity of ripple carry adders and carry lookahead adders:

In ripple carry adders, for each adder block, the two bits that are to be added are available instantly. However, each adder block waits for the carry to arrive from its previous block. So, it is not possible to generate the sum and carry of any block until the input carry is known. The ith block waits for the $i-1^{th}$ block to produce its carry. So, there will be a considerable time delay which is carry propagation delay.

Carry lookahead adder is faster than ripple carry adder (also known as carry propagation adder) since it consists of carry lookahead circuit and all its inputs given by $G(x)$ and $P(x)$ generator functions are calculated simultaneously.

But cost of Carry Lookahead Adder should be more since cost in digital logic means how many gates we are using, what is the FAN-IN of those gates. So, in carry lookahead adder we have carry lookahead circuit that contains many gates compared to carry propagation. To be precise, no. of gates used in carry lookahead circuit = $O(n^2)$ which is much larger than no. of gates used in carry propagation adder which is $O(n)$. So cost of carry lookahead adder is obviously larger.

Time Complexity Analysis:

We could think of a carry look-ahead adder as made up of two "parts"

1. The part that computes the carry for each bit.

2. The part that adds the input bits and the carry for each bit position.

The $\log(n)$ complexity arises from the part that generates the carry, not the circuit that adds the bits.

Now, for the generation of the nth carry bit, we need to perform a AND between (n+1) inputs. The complexity of the adder comes down to how we perform the AND operation. If we have AND gates, each with a fan-in (number of inputs accepted) of k, then we can find the AND of all the bits in $\log_k(n+1)$ time. This is represented in asymptotic notation as $\Theta(\log n)$.

In the case of a ripple carry adder the carry bit is propagated from gate to gate, hence the time complexity would be $O(n)$.

The comparison Table gives us the information about delay in ripple carry adder and carry look ahead adder. The delay is more in case of ripple carry adder. While it is less in case of carry look ahead adder. The transistor count is also more in Carry look ahead adder, which makes it more complex in structure to design and implement. This is considered to be one of the major disadvantage of Carry look ahead adder.

Parameter	RCA (delay)	CLA (delay)
n bit	2n	5 or $3+2[\log n]$
8 bit	16	5 or 8
32 bit	64	5 or 13
64 bit	128	5 or 15
Transistor Count	1280	1514

A 1.3 Stance and justification:

I would disagree with the statement that "A ripple-carry adder is faster than a carry-lookahead adder" since after doing the time complexity analysis it can be seen that the propagation delay in a CLA is very less and it provides the fastest addition logic. From the above analysis it is evident that the delay time of 32-bit carry look ahead adder is less than that of 32-bit ripple carry adder. Trade-off between complexity and performance. Ripple carry adders are simpler but slower. Carry look ahead adders are faster but more complex. Ripple carry adder we need not to wait for the propagation of carries to get the sum. The disadvantage of CLA is that the carry logic block gets very complicated for more than four bits. For that reason CLA's are usually implemented as four bit modules and are used in hierarchical structure the realized adders that has multiples of four bits.

For very large numbers (hundreds or even thousands of bits) look ahead carry logic does not become any more complex, because more layers of super groups and supersupergroups can be added as necessary.

Question No. 2

Solution to Question No. 1 Part B:

B 1.1 Design of the circuit and analysis of its shortcomings:

A Combinational Circuit Block is to be designed that has at max 14 inputs and 3 outputs. This block should be able to work as an 8:3 Encoder and an 8:1 Multiplexer, one of the inputs is to be taken as a toggle that can switch between the encoder and the multiplexer.

The Designing is started by choosing the inputs and output lines for the encoder and the multiplexer. Both the Encoder and the Multiplexer have 8 Inputs each, hence they can share these 8 input lines, apart from that the Multiplexer needs 3 more inputs lines for the select lines, the Enable is another input, which can be used as the toggle between the two circuits.

The output lines are limited to 3, hence the encoder can use all the three output lines, and the last output line can be logical AND with the enable and the multiplexer output line, in this was when the toggle is 0 the Encoder functions and when Toggle is 1, the first two output lines should be don't care and the last output line should be that of the Multiplexer.

1. Designing the Encoder

The Encoder that is to be implemented would be an 8:3 Priority Encoder. In such an encoder we give priority to the MSB first, i.e. if I_7 is high then we don't care if other inputs are ON, the output is only for the I_7. Since Logisim is pretty easy to use with Boolean expression, we assign the inputs and the outputs and build the circuit.

Inputs: $I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7$

Outputs: Q_0, Q_1, Q_2

The expressions for the outputs being:

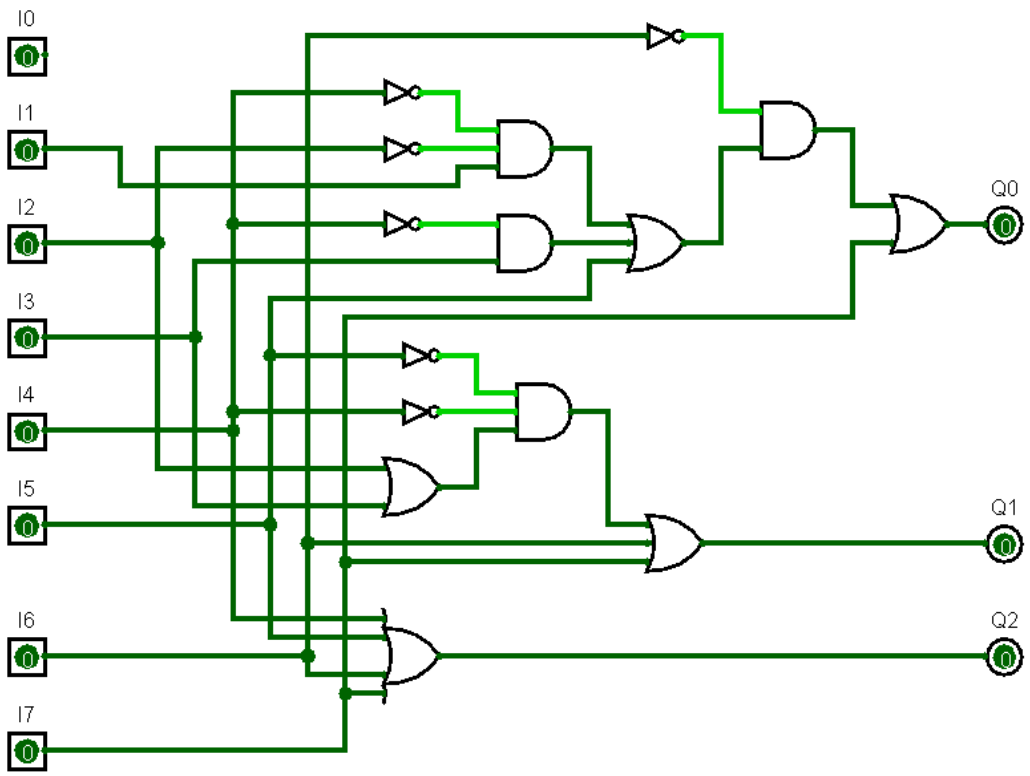
$$Q_0 = \sum(\overline{I_6}(\overline{I_4}\overline{I_2}I_1 + \overline{I_4}I_3 + I_5) + I_7)$$

$$Q_1 = \sum(\overline{I_5}\overline{I_4}(I_2 + I_3) + I_6 + I_7)$$

$$Q_2 = \sum(I_4 + I_5 + I_6 + I_7)$$

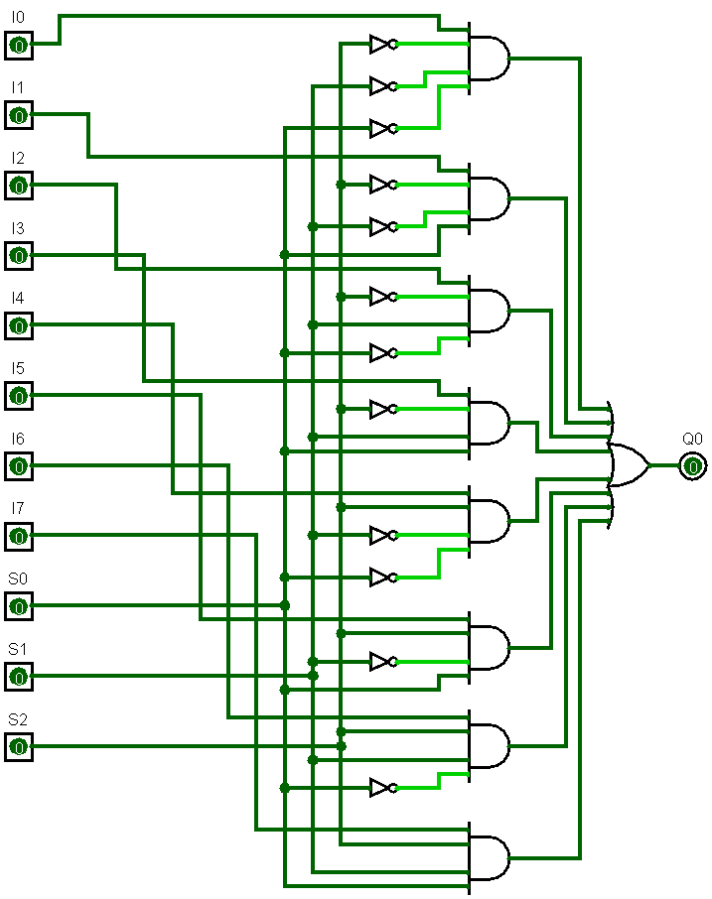
A Priority Encoder also has a Valid bit, but this cannot be accommodated since we are limited to only 3 outputs.

IC's such as $TTL\ 74LS148$ can be used but that won't give total control over the internals, nevertheless it will save time since we won't have to reimplement an encoder, if this was to be on an industrial scale that would be preferred.



2. Designing the Multiplexer

A Multiplexer is basically a selector circuit that selects an input from multiple inputs and passes it to the output, it is useful when only one of the inputs is to be passes to the output at a time.



The expression for the circuit is pretty simple again and writing just that gives us the circuit for an 8:1 Multiplexer.

$$\begin{aligned} Q_0 = I_0\overline{S_2}\,\overline{S_1}\,\overline{S_0} &+ I_1\overline{S_2}\,\overline{S_1}S_0 \\ &+ I_2\overline{S_2}S_1\overline{S_0} \\ &+ I_3\overline{S_2}S_1S_0 \\ &+ I_4S_2\overline{S_1}\,\overline{S_0} \\ &+ I_5S_2\overline{S_1}S_0 \\ &+ I_6S_2S_1\overline{S_0} \\ &+ I_7S_2S_1S_0 \end{aligned}$$

The Multiplexer also needs an Enable pin input, we can use the Enable bit to toggle between the Encoder and the Multiplexer.

A typical $IC\ 74151$ is an 8-to-1 multiplexer with eight inputs and two

outputs. The two outputs are active low and active high outputs. It has three select lines A, B and C and one active low enable input.

So, using the Enable as the Toggle the Output equations of our 83ENC81MUL becomes:

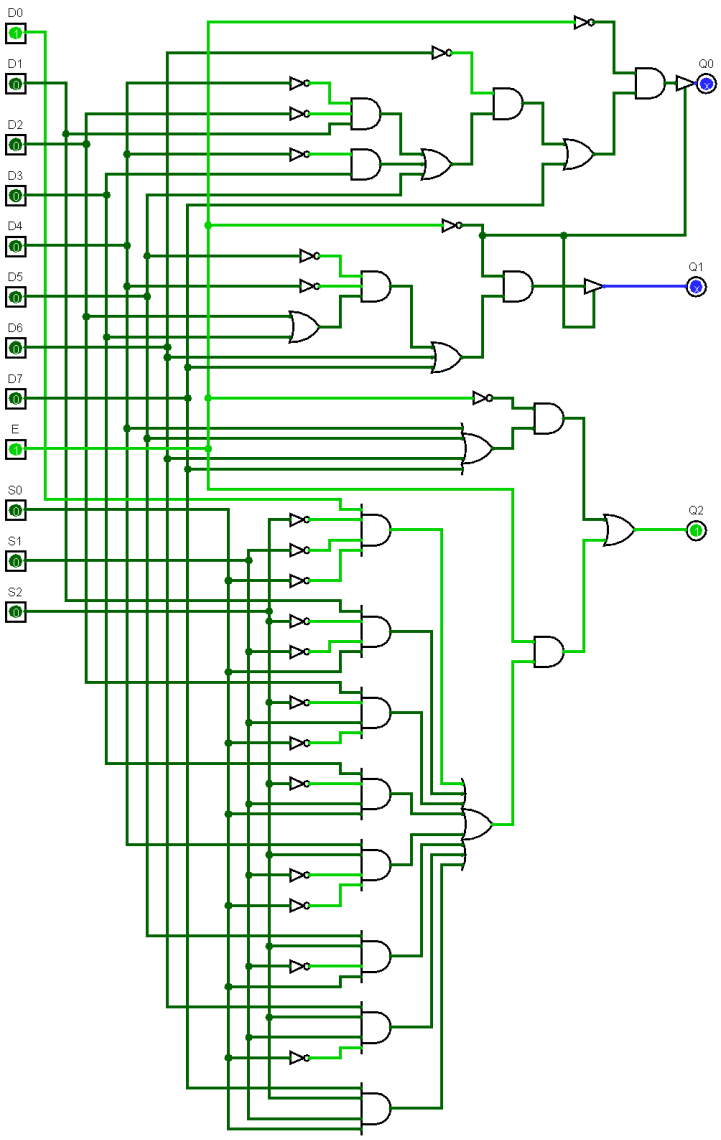
$$Q_0 = (\overline{I_6}(\overline{I_4}\,\overline{I_2}I_1 + \overline{I_4}I_3 + I_5) + I_7)$$

$$Q_1 = (\overline{I_5}\,\overline{I_4}(I_2 + I_3) + I_6 + I_7)$$

$$Q_2 = \overline{E}\,(I_4 + I_5 + I_6 + I_7) + E(I_0\overline{S_2}\,\overline{S_1}\,\overline{S_0} + I_1\overline{S_2}\,\overline{S_1}S_0 + I_2\overline{S_2}S_1\overline{S_0} + I_3\overline{S_2}S_1S_0 + I_4S_2\overline{S_1}\,\overline{S_0} + I_5S_2\overline{S_1}S_0 + I_6S_2S_1\overline{S_0} + I_7S_2S_1S_0)$$

Hence when the Enable is 0 the Encoder functions, and when the Enable is 1 the Multiplexer output is shown. Along with this a Tri-State Controlled Buffer is added, so when the Multiplexer is used, the other two outputs become don't care.

The final design turns out to be:



Shortcoming/Limitations:

1. Since we are limited to only 3 Outputs, when the Encoder is being used, there is no Valid Bit Output, i.e. there's no way to know if the output that is 000 is due to the input I_0 being HIGH, or the circuit being OFF, an encoder should be a function of input and output, this could lead to false output being passed.
2. The number of gates could be reduced by simplifying the expression, this will reduce the gate delay to get the output.

B 1.2 Truth table for encoder and multiplexer:

Truth Table for Priority Encoder:

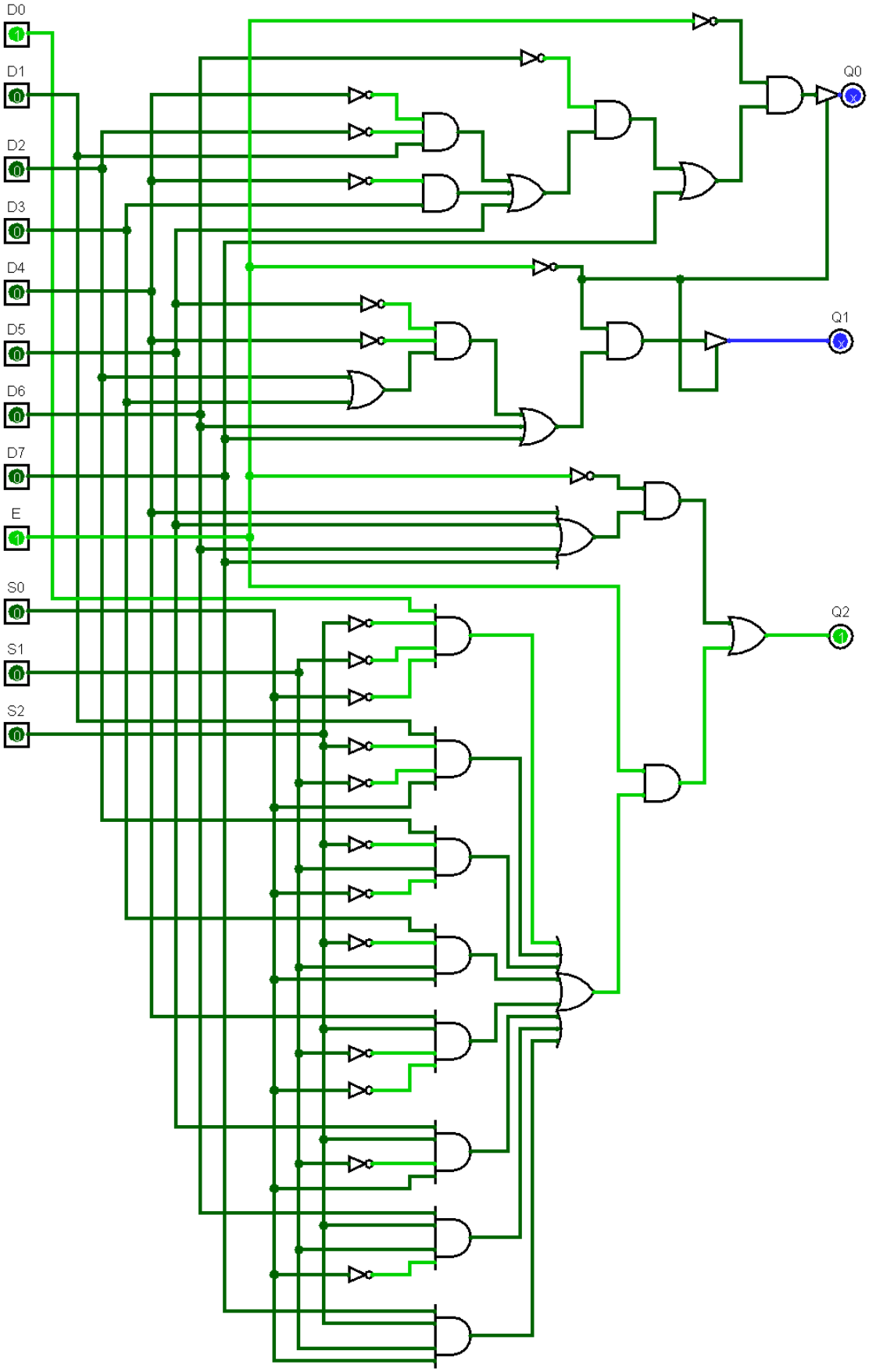
Digital Inputs									Binary Output		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	E	Q_2	Q_1	Q_0
1	0	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	0	1
X	X	1	0	0	0	0	0	0	0	1	0
X	X	X	1	0	0	0	0	0	0	1	1
X	X	X	X	1	0	0	0	0	1	0	0
X	X	X	X	X	1	0	0	0	1	0	1
X	X	X	X	X	X	1	0	0	1	1	0
X	X	X	X	X	X	X	1	0	1	1	1
X	X	X	X	X	X	X	X	1	X	X	X
0	0	0	0	0	0	0	0	0	0	0	0

Truth Table for Multiplexer:

Select Data Inputs			E	Output
S_0	S_1	S_2		Q_2
0	0	0	1	I_0
0	0	1	1	I_1
0	1	0	1	I_2
0	1	1	1	I_3
1	0	0	1	I_4
1	0	1	1	I_5
1	1	0	1	I_6
1	1	1	1	I_7
X	X	X	0	X

Q_0, Q_1 are X, X when the Enable is 1, since they are not required when the Multiplexer is functioning.

B 1.3 Implementation and simulation:

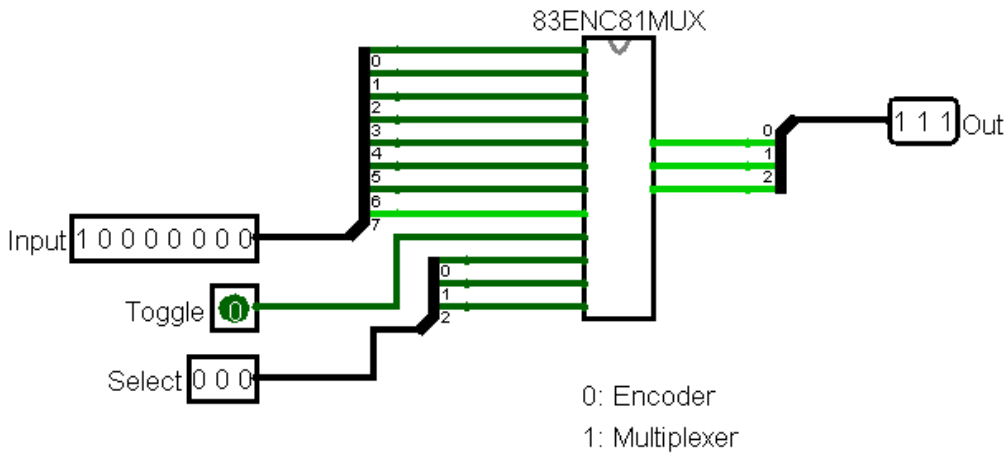


Simulation and Testing:

1. Encoder

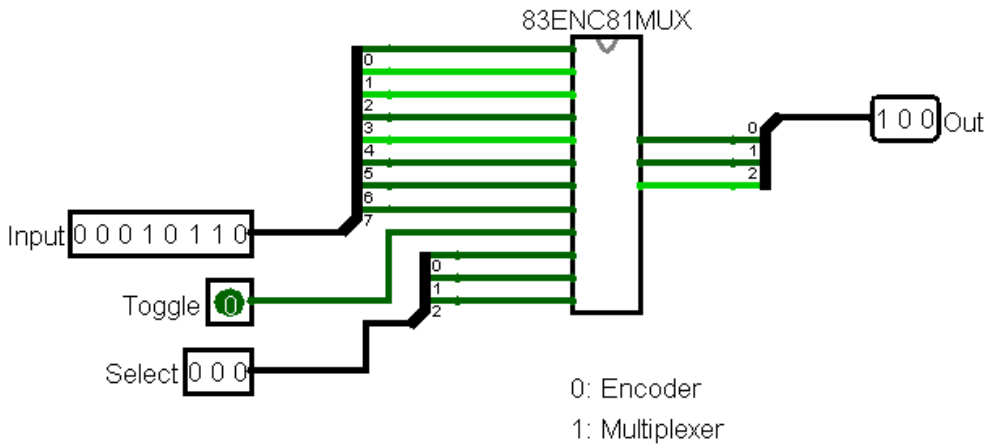
For the Encoder the Toggle is set to 0. The Select Inputs here are Don't Care since they don't affect the output.

With One Input HIGH at a time:



An Encoder here converts between an Octal to a Binary number, let's try to convert 7 in octal to binary, so I_7 is set to HIGH, the output obtained is $(1,1,1)$ which is the required output.

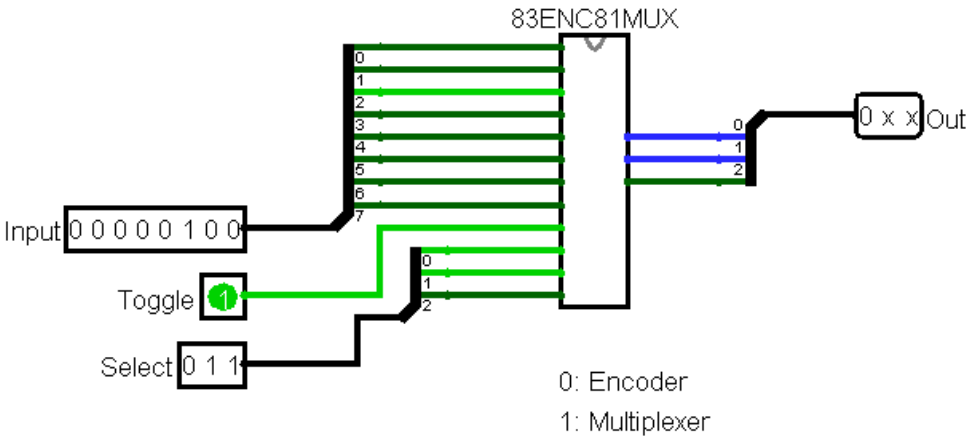
With Multiple Inputs HIGH at once:



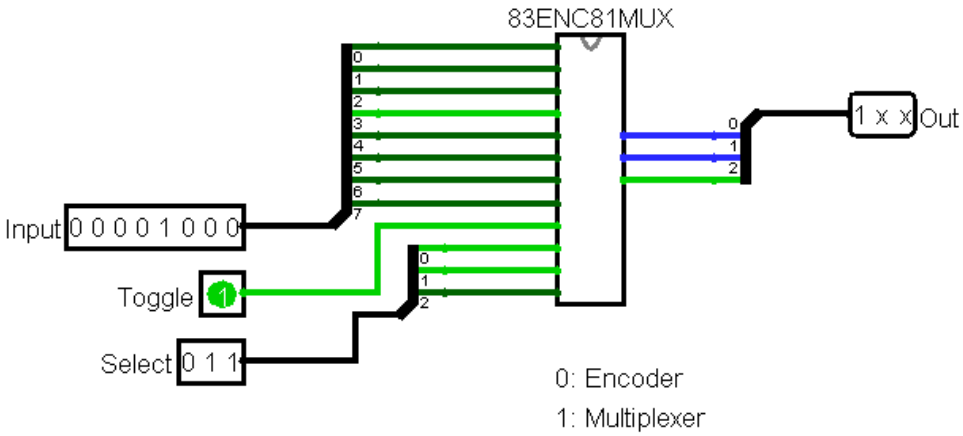
Here I_3, I_2, I_1 are set to HIGH, since it's a priority encoder the MSB is used and the rest are don't care, here I_3 is the MSB. So, the output obtained is $(1,0,0)$ which is the required output.

2. Multiplexer

For the Multiplexer the Toggle is set to 1. The Outputs MSB is used and the rest of the bits are set to don't care, as we used a tri state controlled buffer here.



The I_2 is set to HIGH and other input bits to LOW, and the Select Bits are set to $(0,1,1)$ i.e we select I_3 input, the output here is $(0, X, X)$.



We change the I_3 to HIGH, hence the Output here is $(1, X, X)$.

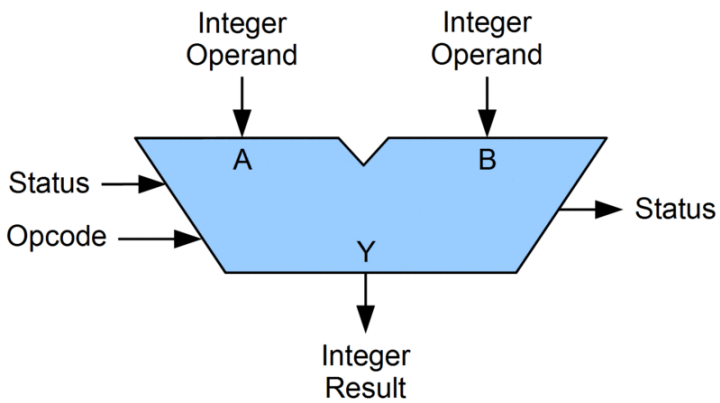
Question No. 3

Solution to Question No. 2 Part B:

B 2.1 Introduction

An ALU is to be designed that can perform Arithmetic and Logical Operations, which are BCD Addition, Comparator, and Complement. Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data, and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory.

The inputs are given as 8-bits and the output is also expected to be 8-bits.



As it can be seen that the ALU will have two operands, an OpCode which would specify which operation to perform and the Integer result, we would omit the Status part of the ALU since it's not required in this context. The Output for the BCD Adder will include the BCD Addition result along with the Carry-Out bit.

B 2.2 Documentation of the design and simulation processes:

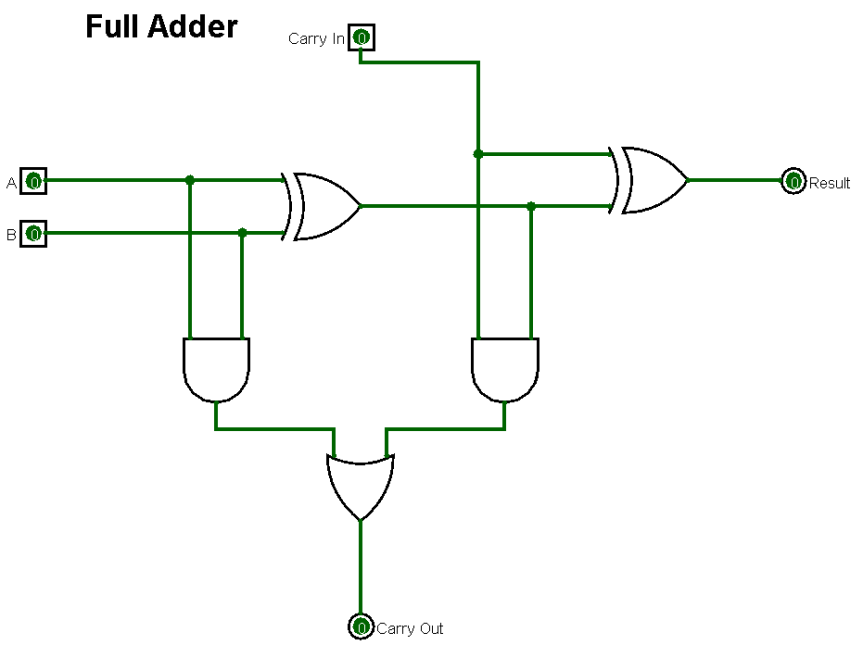
1. Full Adder

Performs Full Adder operation of two 1-bit numbers along with a 1 bit carry in, and outputs the result and carry out.

$$SUM = (A\ XOR\ B) XOR\ Cin = (A \oplus B) \oplus Cin$$

$$CARRY - OUT = A\ AND\ B\ OR\ Cin(A\ XOR\ B) = A.B + Cin(A \oplus B)$$

Which is then implemented using basic gates.

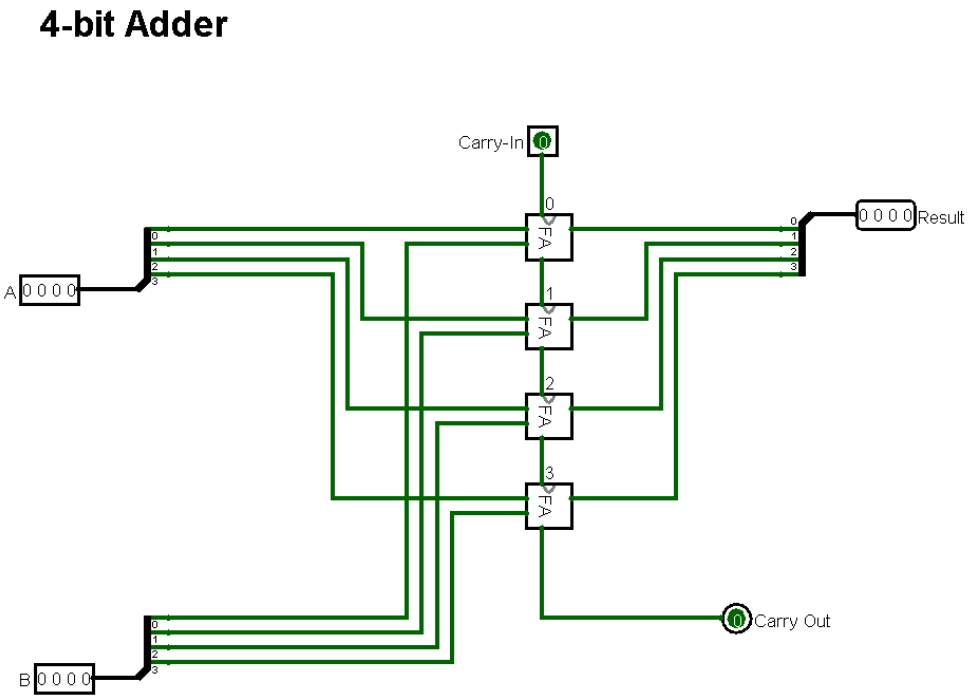


2. 4-Bit Adder

if we wanted to add together two n-bit numbers, then n number of 1-bit full adders need to be connected or “cascaded” together to produce what is known as a Ripple Carry Adder.

A “ripple carry adder” is simply “n“, 1-bit full adders cascaded together with each full adder representing a single weighted column in a long binary addition. It is called a ripple carry adder because the carry signals produce a “ripple” effect through the binary adder from right to left, (LSB to MSB).

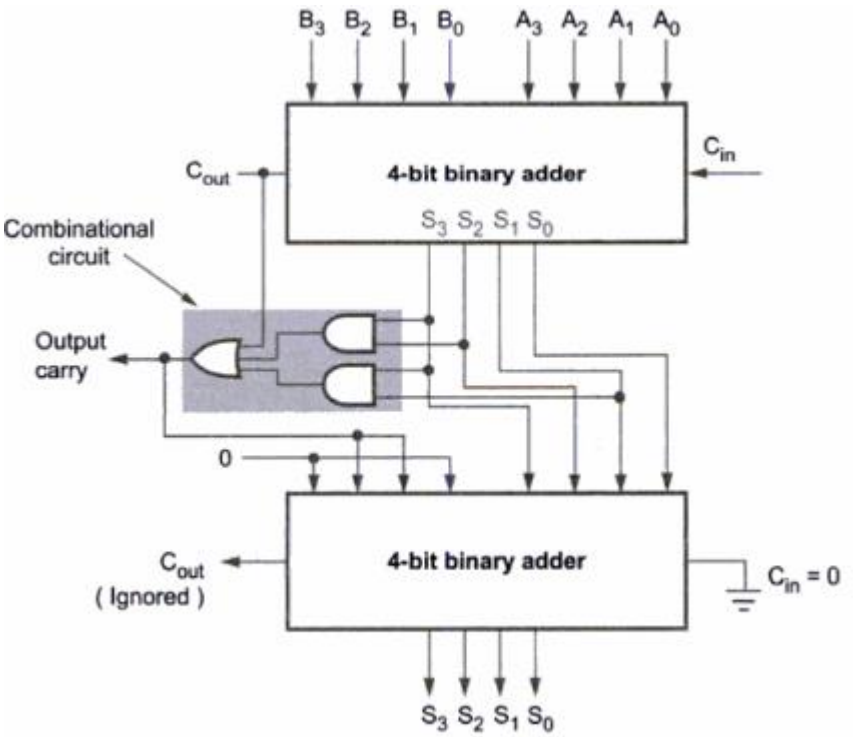
Performs full adder operation for two 4-bit numbers with a carry in, outputs the 4bit addition result and the carry out bit. Uses 4 Full adders to add the two 4-bit numbers.



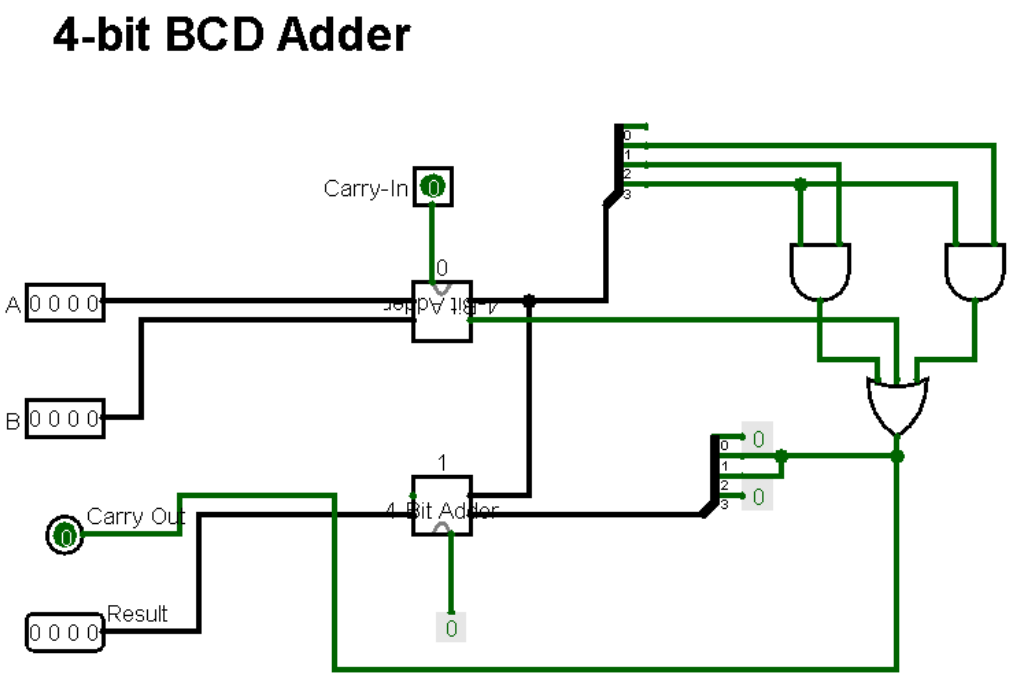
3. 4-Bit BCD Adder

Performs addition for two 4-Bit BCD Numbers, along with a Carry-In, outputs the 4-Bit BCD Result for A+B, and the Carry-Out.

$$OUTPUT\ CARRY = Cout + Z_3Z_2 + Z_3Z_1$$



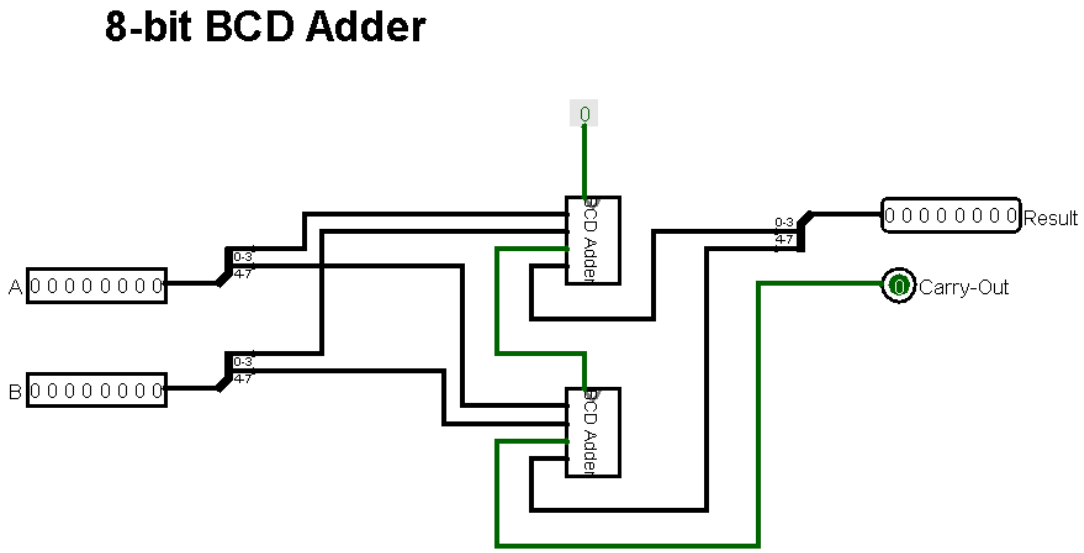
Implementation in Logisim:



4. 8-Bit BCD Adder

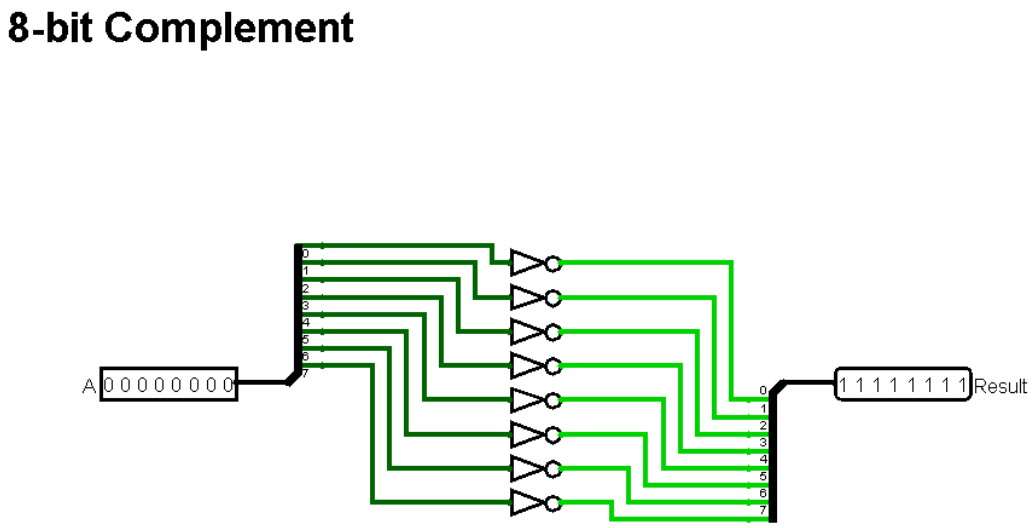
Two 4-bit BCD Adders are Cascaded together, performs addition for two 8-bit BCD Numbers, or can be thought of the addition of two 2-Digit BCD Numbers, Outputs the Result of A+B and the Carry-Out.

Uses 4-Bit BCD Adder.



5. 8-Bit Complement

Performs bit-wise NOT for a given number in Binary. Each bit of the input is passed through individual NOT gates that does the complement, bitwise.



6. 8-bit Comparator

Compares two Binary Numbers, Outputs 1 if they are equal, else outputs X.

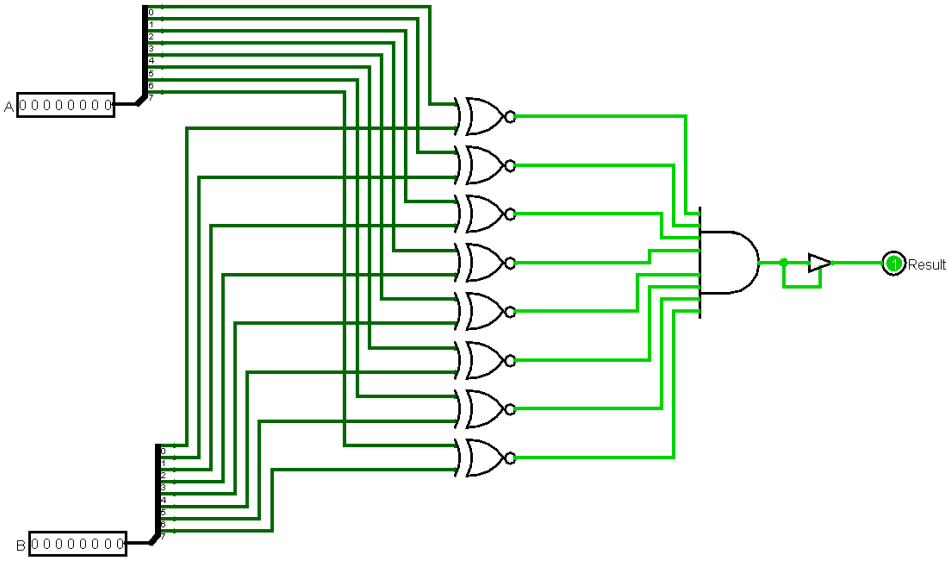
The circuit of the equality comparator consists of an exclusive NOR gate (XNOR) per pair of input bits. If the two inputs are identical (both 1s or both 0s) an output of logic 1 is obtained.

The outputs of the XNOR gates are then combined in an AND gate, the output of which will be 1, only when all the XNOR gates indicate matched inputs.

Truth Table for a 1-bit XNOR Gate.

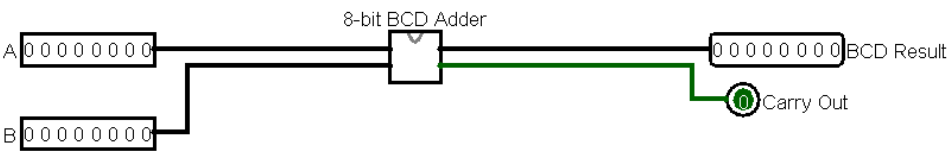
Inputs		Outputs
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

8-bit Comparator

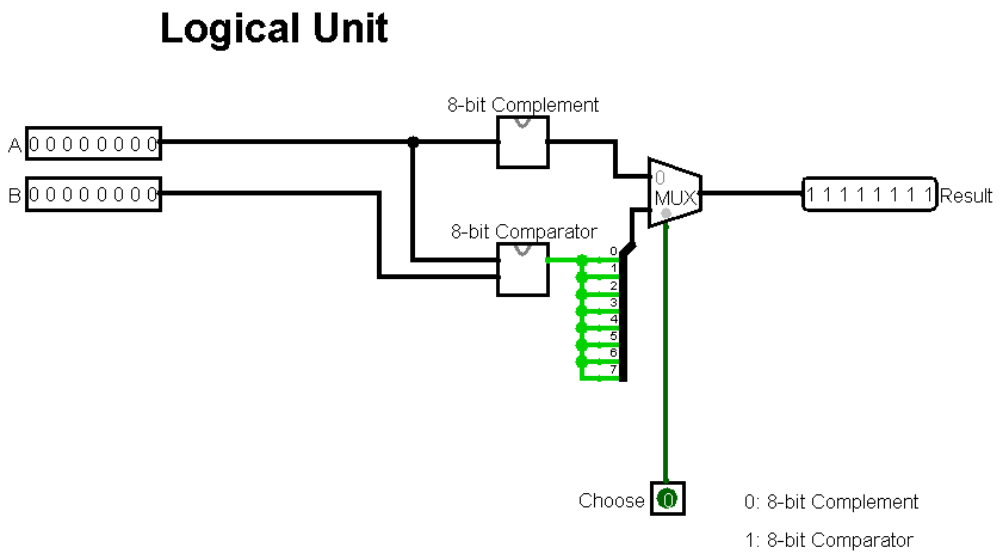


7. AU

Arithmetic Unit



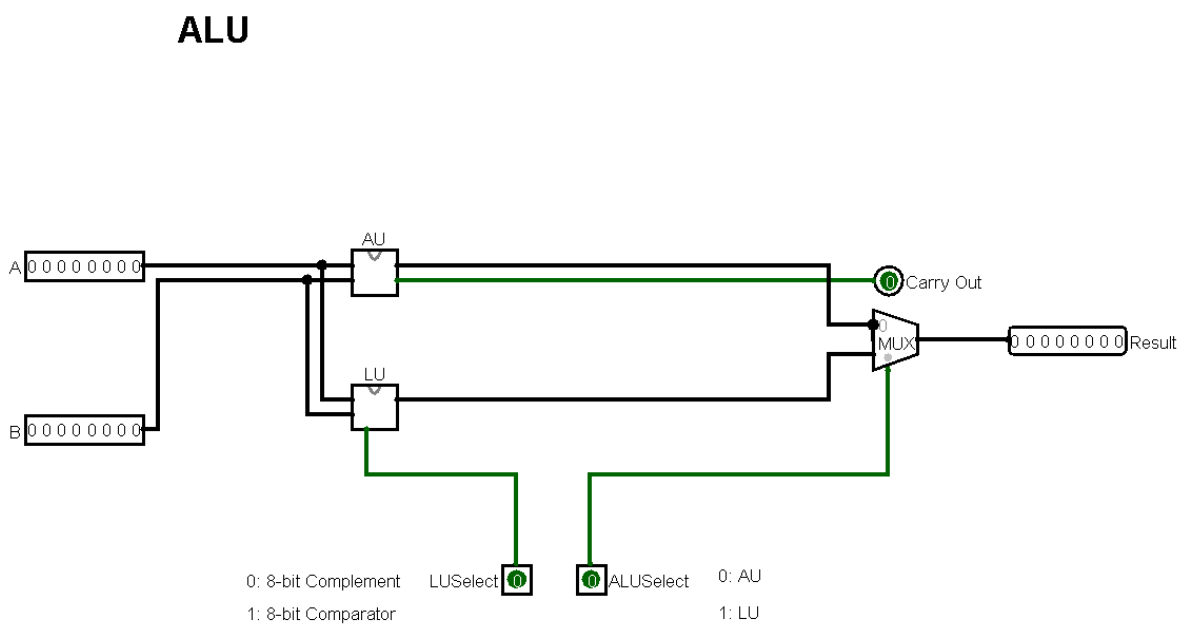
8. LU



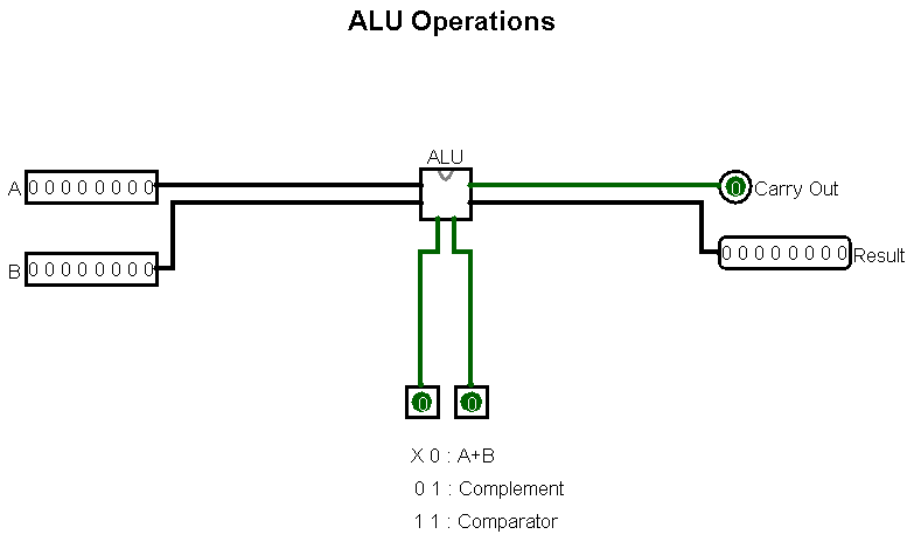
Uses a 2:1 MUX, to choose from either Complement or Comparator, 0 is for Complement and 1 for Comparator.

9. ALU

Combination of the AU and the LU, contains a Select Operation of ALU to switch between AU and LU, where 0 is for AU and 1 for LU, also the select line from LU is to select between Comparator and Complement. The output of ALU is always an 8-bit data and the input are two 8-bit binary numbers.



Simulations and Testing:



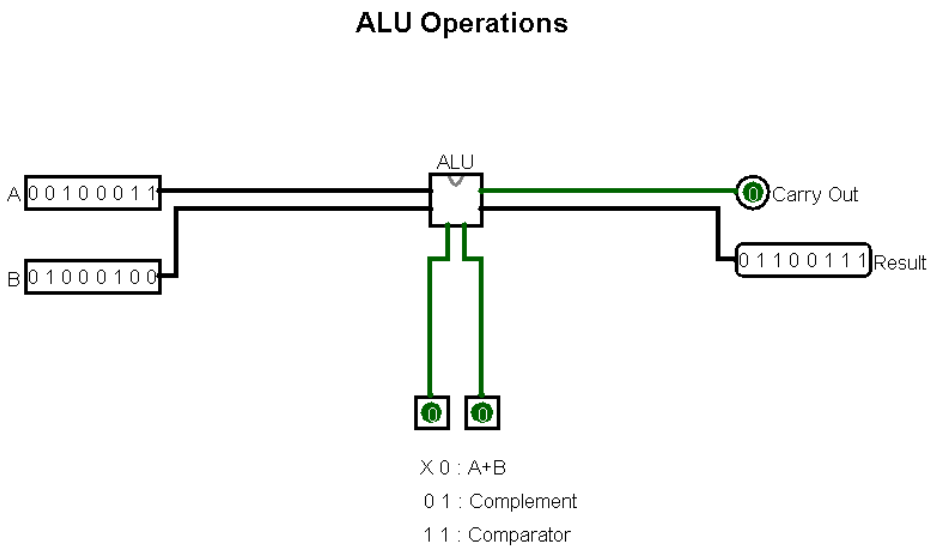
Simulation:

Perform various ALU operations to test the working of the ALU

1. Arithmetic Unit
   a. BCD Adder, A + B

      A = 23 = 00100011 in BCD

      B = 42 = 01000100 in BCD

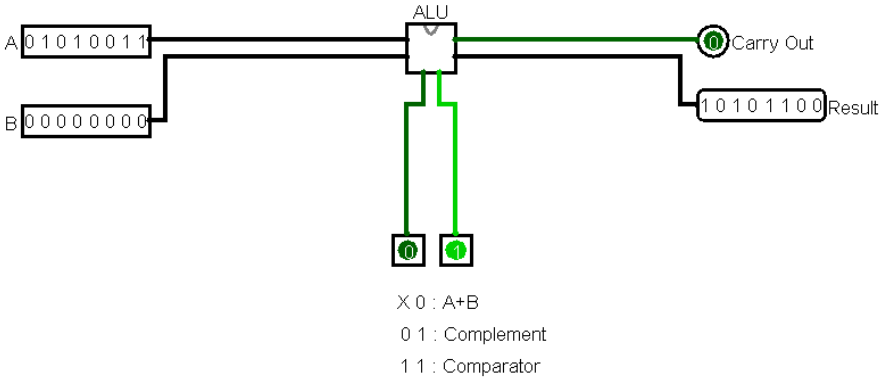


The Result obtained is 01100111 in BCD which is 65, which is the expected result for 23+42.

2. Logical Unit
   a. Complement

      Set A = 01010011

ALU Operations



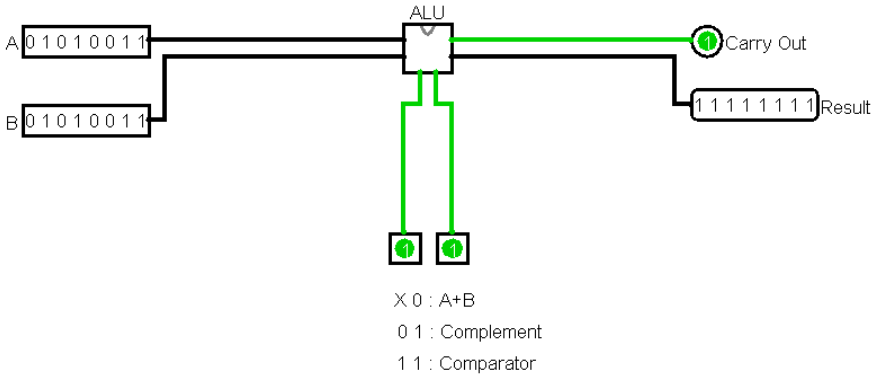
The Result obtained is 10101100 which is the complement of A

b. Comparator

Set A = 01010011

Set B = 01010011

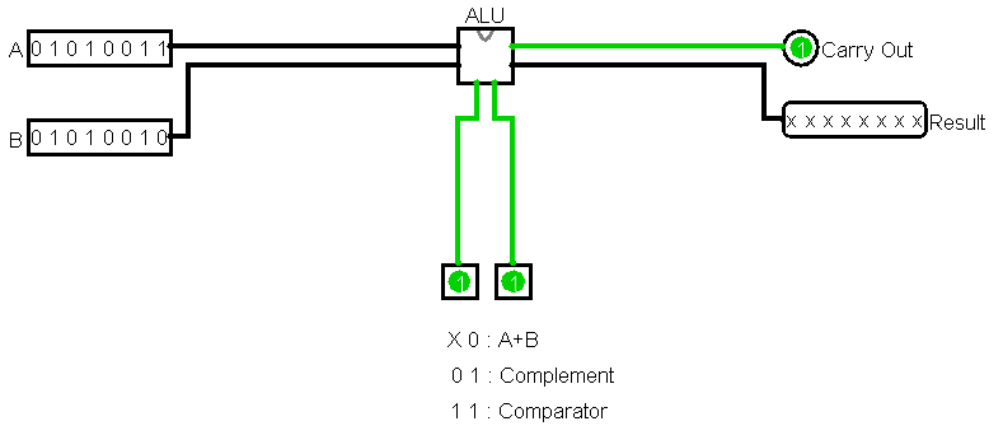
ALU Operations



Since A = B, the Output is 11111111, i.e. all the output bits are 1.

Change B to 01010010

ALU Operations



The Output is now xxxxxxxx, i.e. A ≠ B.

B 2.3 Issues involving connecting four such ALUs to the CPU bus:

Since multiple devices are connected to the bus, it is quite possible that more than one device wishes to send information on the bus. However, for correct operation of the bus, only one device may send data at one time. Several devices can simultaneously listen, however, normally, the data is intended for one device and only that one receives.

Consider a scenario in which 'n' ALU's connected on a link are waiting to output data through the bus. In this case al the 'n' ALU's would want to access the link/channel to transfer their own data. Problem arises when more than one ALU transmits the data at the moment. In this case, there will be collisions in the data from different ALU's.

Another problem would be to connect the four ALU's in the first place, i.e. if the CPU does not have a bus of width that equals to the four ALU's at once, then they cannot be connected.

Memory Access Collision, since there are four such ALU's they would either require a memory unit each to fetch the operands in parallel, this will lead to memory access collisions. And its inefficient for the processor and the memory to retain the bus

B 2.4 Resolution of the issues identified above:

CSMACD:

To detect whether a collision has occurred, the station simultaneously checks whether the transmitted signal is identical to that on the transmission medium. If this is not the case, another station carries out a transmission simultaneously and falsifies the signal on the bus. In general, collisions cannot be completely avoided with half-duplex systems. They are scheduled errors and CSMA/CD ensures that

collisions do not lead to problems during transmission. However, this only applies to data loss. CSMA protocol decides which station will transmit when so that data reaches the destination without corruption. The Algorithm for CSMACD is as follows:

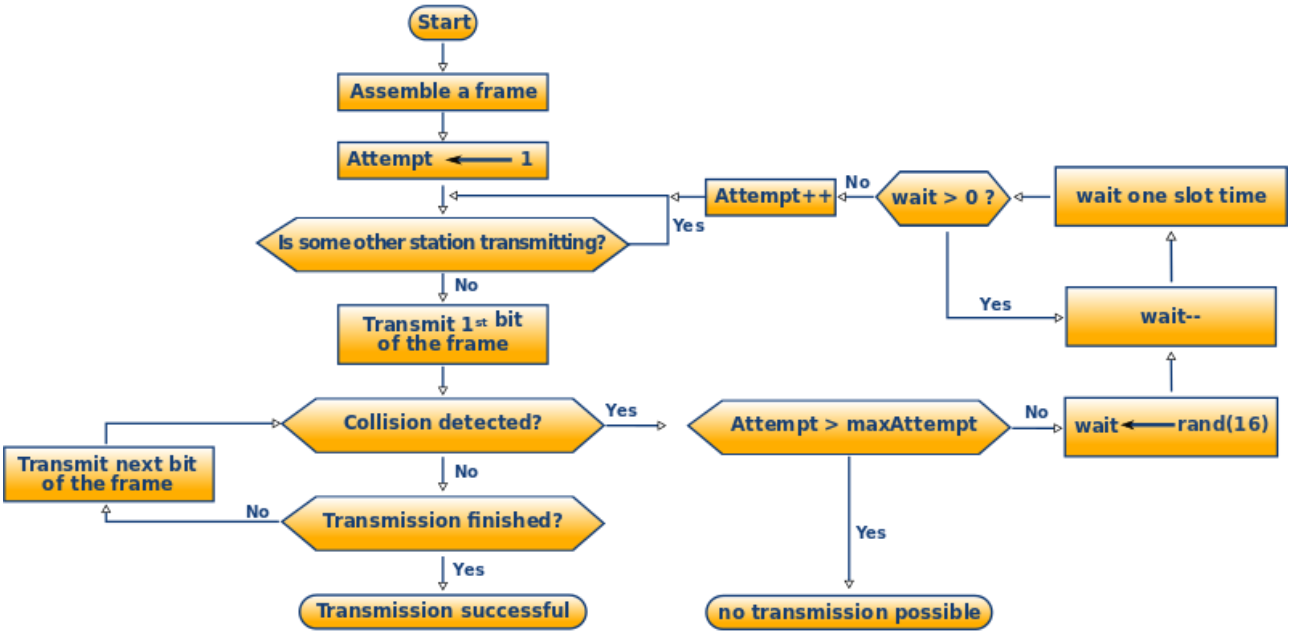


Figure B2.1: CSMACD Algorithm

Bus Arbitration:

The device that initiates the transaction and controls the bus is the bus master. Other devices respond to the master. A slave is a device that responds to a transaction initiated by the bus master. The bus master can be a sender or a receiver. Bus Arbitration schemes allow to decide who gets control of the bus when there is more than one device requesting control od the bus. Arbitration must be fast and fair. The fairness criterion assures that every device that wants to use the bus will eventually get it, or in other words, no device will be starved for the bus.

1. https://www.geeksforgeeks.org/collision-detection-csmacd/
2. Computer Science and Engineering, Zainalabedin Navabi, David R Kaeli
3. https://www.geeksforgeeks.org/digital-logic-carry-look-ahead-adder/
4. Relative Analysis of 32 Bit Ripple Carry Adder and Carry Lookahead Adder, B.Tharanum sulthana, B.Ramya ,G.sai Divya