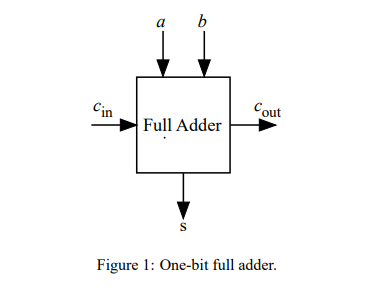
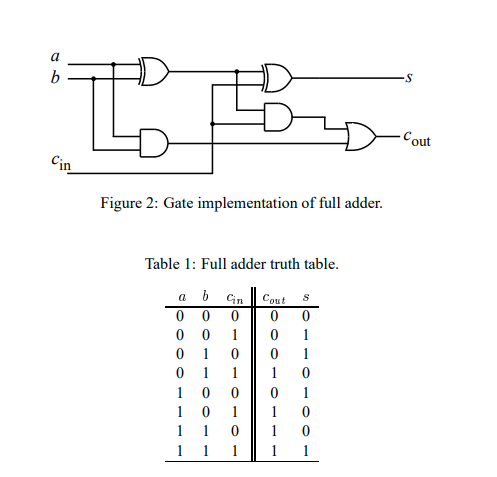
**Ripple Carry and Carry Look Ahead Adder**

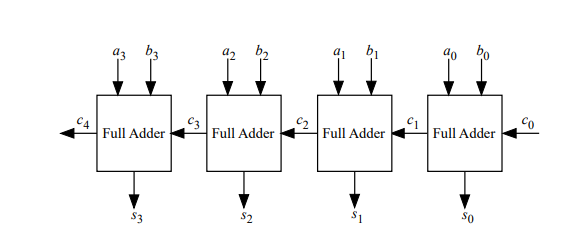
We will start by explaining the operation of one-bit full adder which will be the basis for constructing ripple carry and carry lookahead add.

and   
The truth table of the full adder is listed in Table 1. The gate implementation of 1-bit full adder is shown in Figure.



****

Ripple carry adder A ripple carry adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascaded (see section 2.1), with the carry output from each full adder connected to the carry input of the next full adder in the chain. Figure 3 shows the interconnection of four full adder (FA) circuitsto provide a 4-bit ripple carry adder. Notice from Figure 3 that the input is from the right side because the first cell traditionally represents the least significant bit (LSB). Bits and in the figure represent the least significant bits of the numbers to be added. The sum output is represented by the bits.



The drawback of ‘[Ripple carry adder](https://vlsiverify.com/verilog/verilog-codes/ripple-carry-adder)‘ is that it has a carry propagation delay that introduces slow computation. Since adders are used in designs like multipliers and divisions, it causes slowness in their computation. To tackle this issue, a carry look-ahead adder (CLA) can be used that reduces propagation delay with additional hardware complexity.

CLA has introduced some functions like ‘carry generate (G)’ and ‘carry propagate (P)’ to boost the speed.

**Carry Generate (G):** This function denotes how the carry is generated for single-bit two inputs regardless of any input carry.

As we have seen in the full adder, carry is generated using the equation as AB.

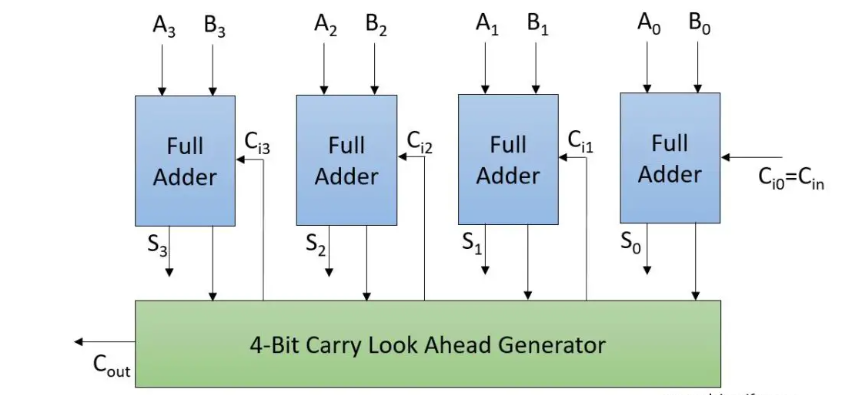
Hence, G = A·B (similar to how carry is generated by full adder)

**Carry Propagate (P):** This function denotes when the carry is propagated to the next stage with an addition whenever there is an input carry.

Let’s consider single bit two inputs A and B.

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **Carry In** | **Description** |
| 0 | 0 | 1 | Carry is not propagated (0) |
| 0 | 1 | 1 | Carry is propagated (1) |
| 1 | 0 | 1 | Carry is propagated (1) |
| 1 | 1 | 1 | Carry is propagated (1) |

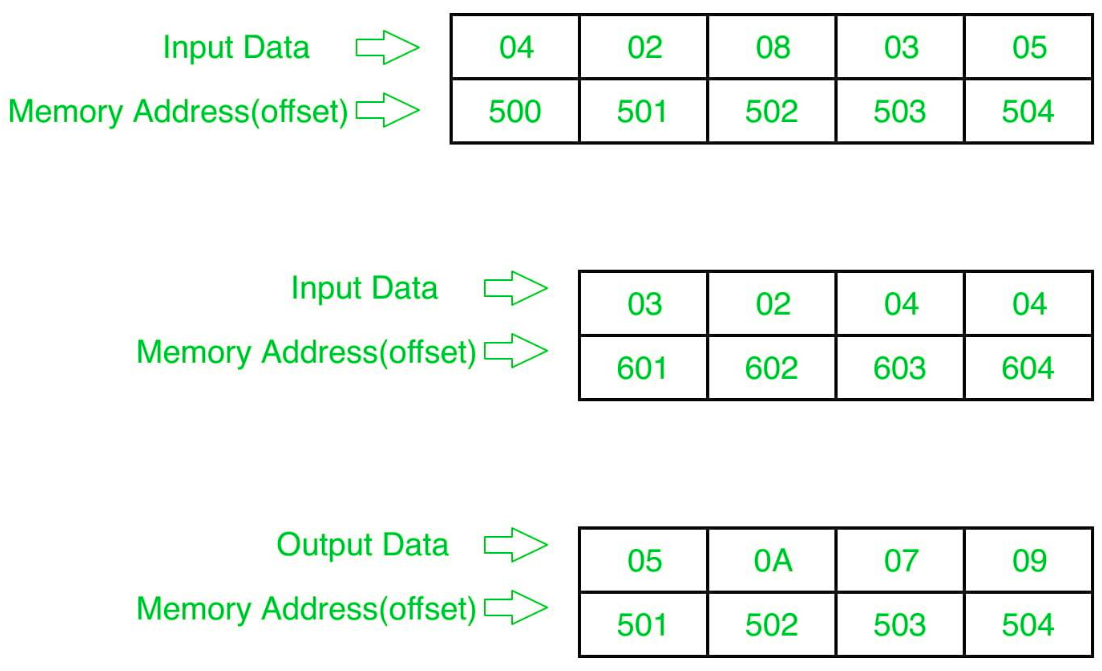
## Block Diagram:-



b)

Write a program in 8086 microprocessor to find out the sum of two arrays of 8-bit n numbers, where size “n” is stored at offset 500 and the numbers of first array are stored from offset 501 and the numbers of second array are stored from offset 601 and store the result numbers into first array i.e offset 501.

**Example –**



Algorithm:-

1. Store 500 to SI and 601 to DI and Load data from offset 500 to register CL and set register CH to 00 (for count).
2. Increase the value of SI by 1.
3. Load first number(value) from next offset (i.e 501) to register AL.
4. Add the value in register AL by value at offset DI.
5. Store the result (value of register AL ) to memory offset SI.
6. Increase the value of SI by 1.
7. Increase the value of DI by 1.
8. Loop above 5 till register CX gets 0.

**Program –**

| **MEMORY ADDRESS** | **MNEMONICS** | **COMMENT** |
| --- | --- | --- |
| 400 | MOV SI, 500 | SI<-500 |
| 403 | MOV CL, [SI] | CL<-[SI] |
| 405 | MOV CH, 00 | CH<-00 |
| 407 | INC SI | SI<-SI+1 |
| 408 | MOV DI, 601 | DI<-601 |
| 40B | MOV AL, [SI] | AL<-[SI] |
| 40D | ADD AL, [DI] | AL=AL+[DI] |
| 40F | MOV [SI], AL | AL->[SI] |
| 411 | INC SI | SI<-SI+1 |
| 412 | INC DI | DI<-DI+1 |
| 413 | LOOP 40B | JUMP TO 40B IF CX!=0 and CX=CX-1 |
| 415 | HLT | end |

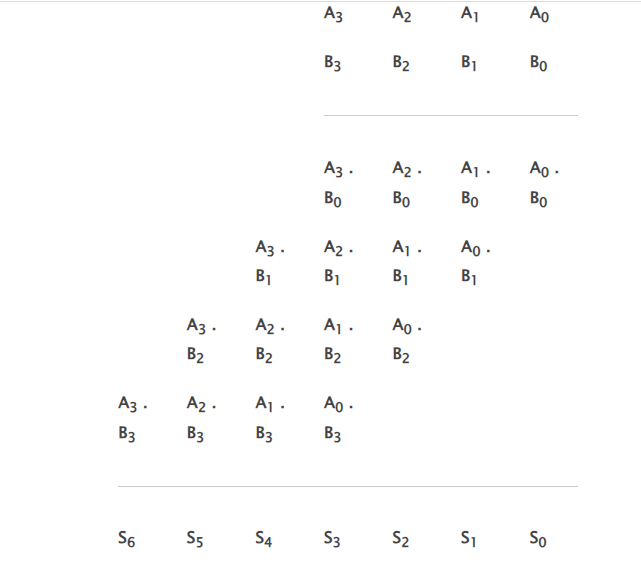
**Explanation –**

1. **MOV SI, 500:** set the value of SI to 500
2. **MOV CL, [SI]:** load data from offset SI to register CL
3. **MOV CH, 00:** set value of register CH to 00
4. **INC SI:** increase value of SI by 1.
5. **MOV DI, 600:** set the value of DI to 600.
6. **MOV AL, [SI]:** load value from offset SI to register AL
7. **ADD AL, [DI]:** Add value of register AL by content at offset DI.
8. **MOV [SI], AL:** store value of register AL at offset SI.
9. **INC SI:** increase value of SI by 1.
10. **INC DI:** increase value of DI by 1.
11. **LOOP 408:** jump to address 408 if CX not 0 and CX=CX-1.
12. **HLT:** stop

**4 a)Design of Combinational Multiplier :**

Combinational Multipliers do multiplication of two unsigned binary numbers.Each bit of the multiplier is multiplied against the multiplicand, the product is aligned according to the position of the bit within the multiplier, and the resulting products are then summed to form the final result. Main advantage of binary multiplication is that the generation of intermediate products are simple: if the multiplier bit is a 1, the product is an appropriately shifted copy of the multiplicand; if the multiplier bit is a 0, the product is simply 0.

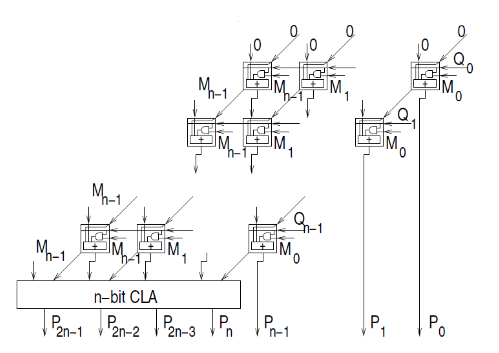
The design of a combinational multiplier to multiply two 4-bit binary number is illustrated below:



If two n-bit numbers are multiplied then the output will be less than or equals to 2n bits.

Some features of the multiplication scheme: it can be designed by unrolling the multiplier loop instead of handling the carry out of partial product summation bit,the carry out can be sent to the next bit of the next step this scheme of handling the carry is called carry save addition this scheme is more regular and modular

Logic diagram:



b)

Write an assembly language program in 8085 microprocessor to separate odd and even numbers from the given array?

Algorithm:

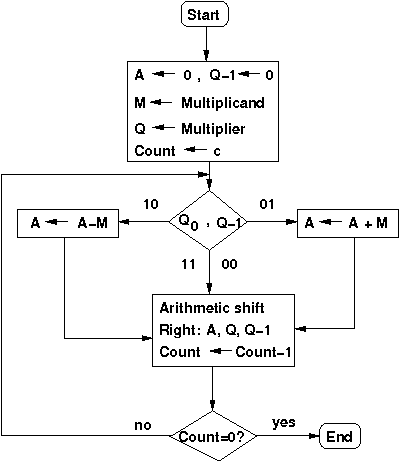
1. Load the memory location 2000 in HL register pair.
2. Load the memory location 2100 in DE register pair for storing odd numbers.
3. Store the number of elements in register C.
4. Move the next number in the list to accumulator.
5. Perform AND operation with 01H to check whether the number is even or odd.
6. If even, jump to step 9.
7. Get the number in accumulator and store in the memory location pointed by DE.
8. Increment DE.
9. Increment HL. Decrement C.
10. If C is not zero, jump to step 4.
11. Program:

| 1. Memory Location | 1. Mneumonics | 1. Comments |
| --- | --- | --- |
| 1. 2000H | 1. LXI H, 2000H | 1. Initialize memory pointer 1 |
| 1. 2003H | 1. LXI D, 2100H | 1. Initialize memory pointer 2 |
| 1. 2006H | 1. MVI C, 32H | 1. Initialize counter |
| 1. 2008H | 1. MOV A, M | 1. Get the number |
| 1. 2009H | 1. ANI 0lH | 1. Check for odd number |
| 1. 200BH | 1. JZ 2011H | 1. If EVEN, don’t store |
| 1. 200EH | 1. MOV A, M | 1. Get the number |
| 1. 200FH | 1. STAX D | 1. Store the number in result list |
| 1. 2010H | 1. INX D | 1. Increment pointer 2 |
| 1. 2011H | 1. INX H | 1. Increment pointer l |
| 1. 2012H | 1. DCR C | 1. Decrement counter |
| 1. 2013H | 1. JNZ 2008H | 1. If not zero, repeat |
| 1. 2016H | 1. LXI H, 2000H | 1. Initialize memory pointer l |
| 1. 2019H | 1. LXI D, 2200H | 1. Initialize memory pointer2 |
| 1. 201CH | 1. MVI C, 32H | 1. Initialize counter |
| 1. 201EH | 1. MOV A, M | 1. Get the number |
| 1. 201FH | 1. ANI 0lH | 1. Check for even number |
| 1. 2021H | 1. JNZ 2027H | 1. If ODD, don’t store |
| 1. 2024H | 1. MOV A, M | 1. Get the number |
| 1. 2025H | 1. STAX D | 1. Store the number in result list |
| 1. 2026H | 1. INX D | 1. Increment pointer 2 |
| 1. 2027H | 1. INX H | 1. Increment pointer l |
| 1. 2028H | 1. DCR C | 1. Decrement counter |
| 1. 2029H | 1. JNZ 201EH | 1. If not zero, repeat |
| 1. 202CH | 1. HLT | 1. Stop |

**Lab No 5:- Booth's Multipliers :**

Booth's multiplication algorithm is an algorithm which multiplies 2 signed integers in 2's complement. The algorithm is depicted in the following figure with a brief description. This approach uses fewer additions and subtractions than more straightforward algorithms.

The multiplicand and multiplier are placed in the m and Q registers respectively. A 1 bit register is placed logically to the right of the LSB (least significant bit) Q0 of Q register. This is denoted by Q-1. A and Q-1 are initially set to 0. Control logic checks the two bits Q0 and Q-1. If the twi bits are same (00 or 11) then all of the bits of A, Q, Q-1 are shifted 1 bit to the right. If they are not the same and if the combination is 10 then the multiplicand is subtracted from A and if the combination is 01 then the multiplicand is added with A. In both the cases results are stored in A, and after the addition or subtraction operation, A, Q, Q-1 are right shifted. The shifting is the arithmetic right shift operation where the left most bit namely, An-1 is not only shifted into An-2 but also remains in An-1. This is to preserve the sign of the number in A and Q. The result of the multiplication will appear in the A and Q.

****

**Design Issues :**

Booth's algorithm can be implemented in many ways. This experiment is designed using a controller and a datapath. The operations on the data in the datapath is controlled by the control signal received from the controller. The datapath contains registers to hold multiplier, multiplicand, intermediate results, data processing units like ALU, adder/subtractor etc., counter and other combinational units. Following is the schemetic diagram of the Booth's multiplier which multiplies two 4-bit numbers in 2's complement of this experiment. Here the adder/subtractor unit is used as data processing unit.M, Q, A are 4-bit and Q-1 is a 1-bit rigister. M holds the multiplicand, Q holds the multiplier, A holds the results of adder/subtractor unit. The counter is a down counter which counts the number of operations needed for the multiplication. The data flow in the data path is controlled by the five control signals generated from the controller. these signals are load (to load data in registers), add (to initiate addition operation), sub (to initiate subtraction operation), shift (to initiate arithmetis right shift operation), dc (this is to decrement counter). The controller generates the control signals according to the input received from the datapath. Here the inputs are the least significant Q0 bit of Q register, Q-1 bit and count bit from the down counter.

b) Assembly language rogram to find prime numbers between a given range?