PARALLEL ADDER

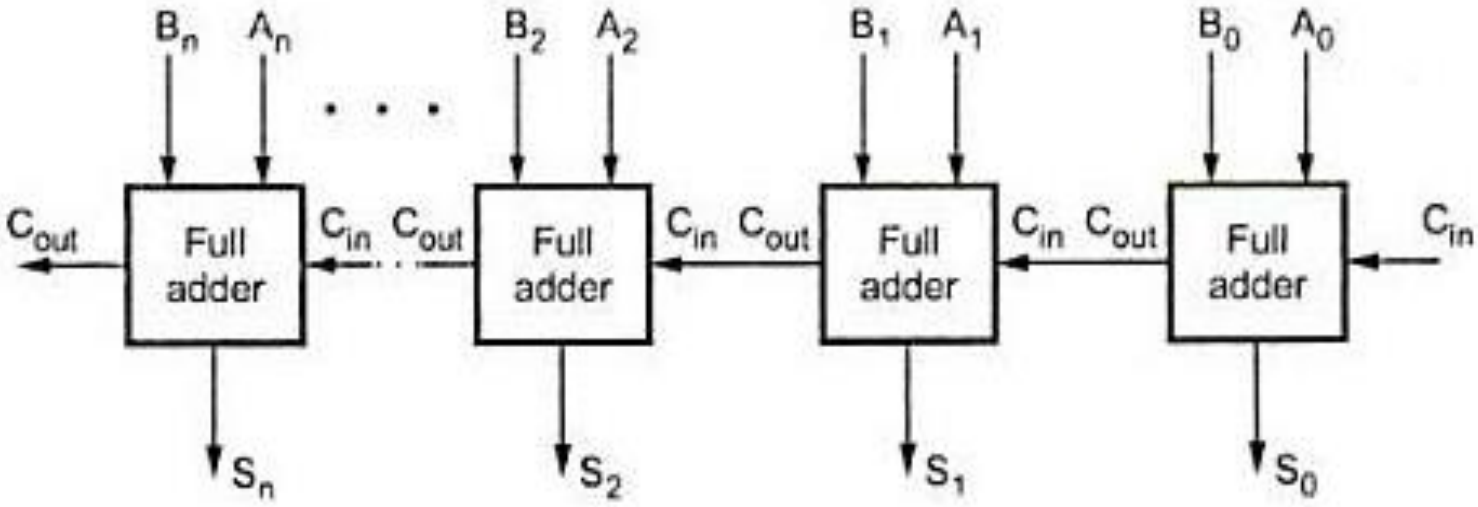
PARALLEL ADDER

- A single full adder is capable of adding two one bit numbers and an input carry. In order to add a binary number with more than one bit an additional full adders must be employed.

- The n-bit parallel adder can be constructed using “n” number of full adder circuits in parallel.

- The block diagram of n-bit parallel adder using number of full adder circuits connected in cascade

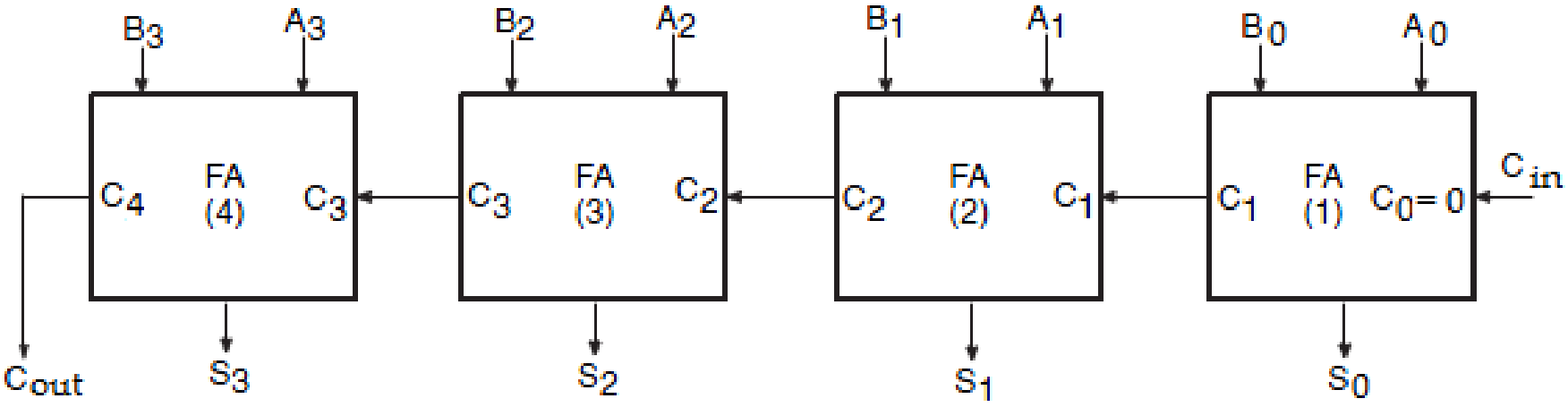
PARALLEL ADDER



n-bit parallel Adder

PARALLEL ADDER

- The 4-bit binary adder using full adder circuits is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in figure below.



4-bit binary parallel Adder

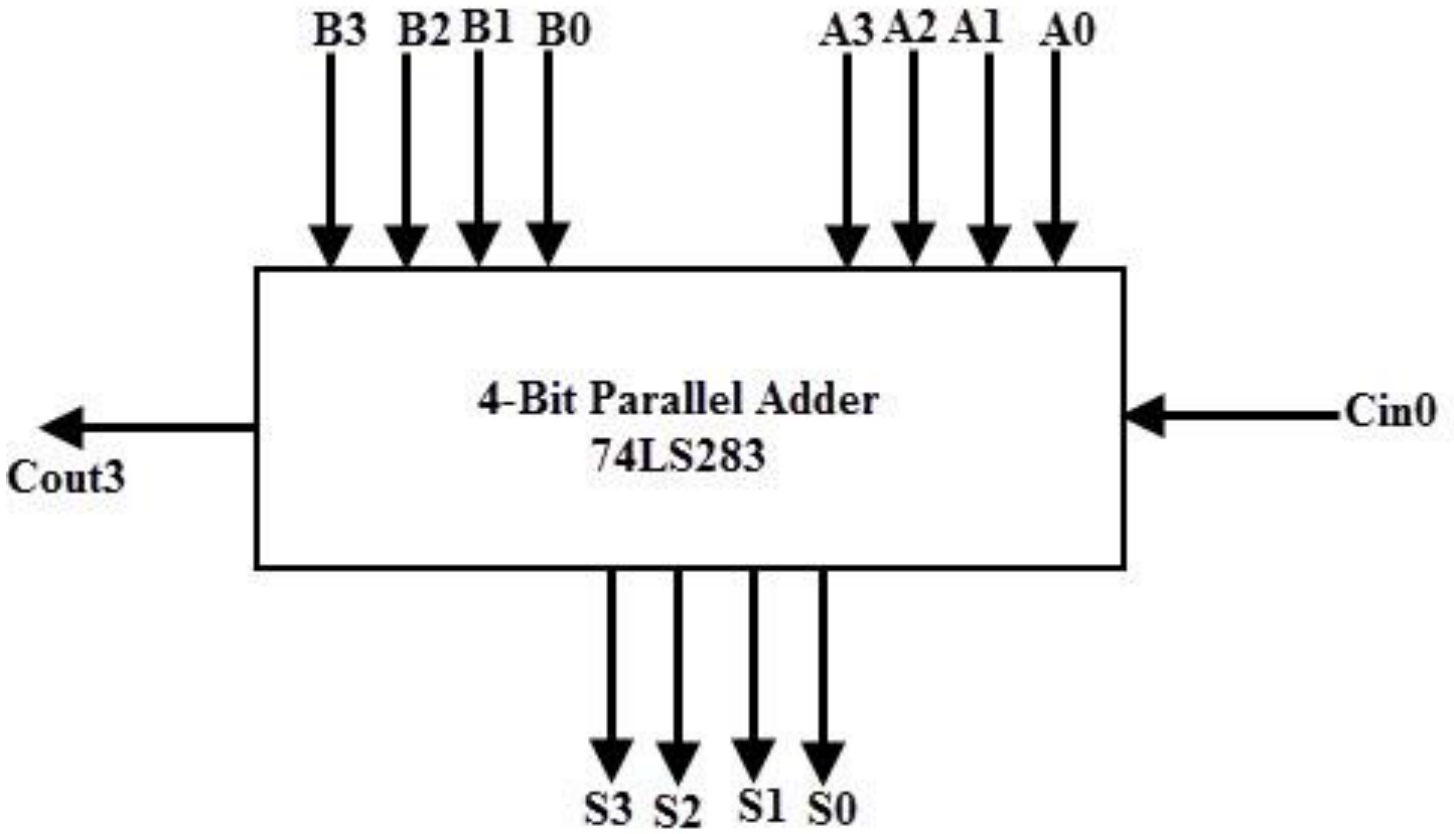
PARALLEL ADDER

- Since all the bits of augend and addend are fed into the adder circuits simultaneously and the additions in each position are taking place at the same time, this circuit is known as parallel adder.
- Let the 4-bit words to be added be represented by, $A_3 A_2 A_1 A_0$= 1 1 1 1 and $B_3 B_2 B_1 B_0$= 0 0 1 1.

Significant place		4	3	2	1
Input carry		1	1	1	0
Augend word A :		1	1	1	1
Addend word B :		0	0	1	1
	1	0	0	1	0 ← Sum

↑
Output Carry

PARALLEL ADDER



Logic diagram of 4-bit parallel adder

Full Adder Module

```
1  module fulladder(a,b,ic,o,oc);
2
3  input a,b,ic;
4
5  output o,oc;
6
7  assign o = (~ic & ((a & ~b) | (~a & b)) ) | (ic & ~((a & ~b
8
9  assign oc = (a & b) | (b & ic) | (ic & a);
10
11 endmodule
```

Parallel Adder Module

```
1  module main(in1,in2,ic,out,oc);
2
3  input [3:0]in1;
4
5  input [3:0]in2;
6
7  input ic;
8
9  output [3:0]out;
10
11 output [3:0]oc;
12
13 fulladder fa1(in1[0],in2[0],ic,out[0],oc[0]);
14
15 fulladder fa2(in1[1],in2[1],oc[0],out[1],oc[1]);
16
17 fulladder fa3(in1[2],in2[2],oc[1],out[2],oc[2]);
18
19 fulladder fa4(in1[3],in2[3],oc[2],out[3],oc[3]);
20
21 endmodule
```

```
module fa_su(A,B,CIN,OUT,OUTC);
input A,B,CIN;
output OUT,OUTC;
assign OUT=A^B^CIN;
Assign OUTC=(A&B)|(B&CIN)|(CIN&A);
endmodule
```

```
module pa_add(in1,in2,ic,out,oc);
input [3:0]in1;
input [3:0]in2;
input ic;
output [3:0]out;
output [3:0]oc;
fa_su fa1(in1[0],in2[0],ic,out[0],oc[0]);
fa_su  fa2(in1[1],in2[1],oc[0],out[1],oc[1]);
fa_su  fa3(in1[2],in2[2],oc[1],out[2],oc[2]);
fa_su  fa4(in1[3],in2[3],oc[2],out[3],oc[3]);

endmodule
```

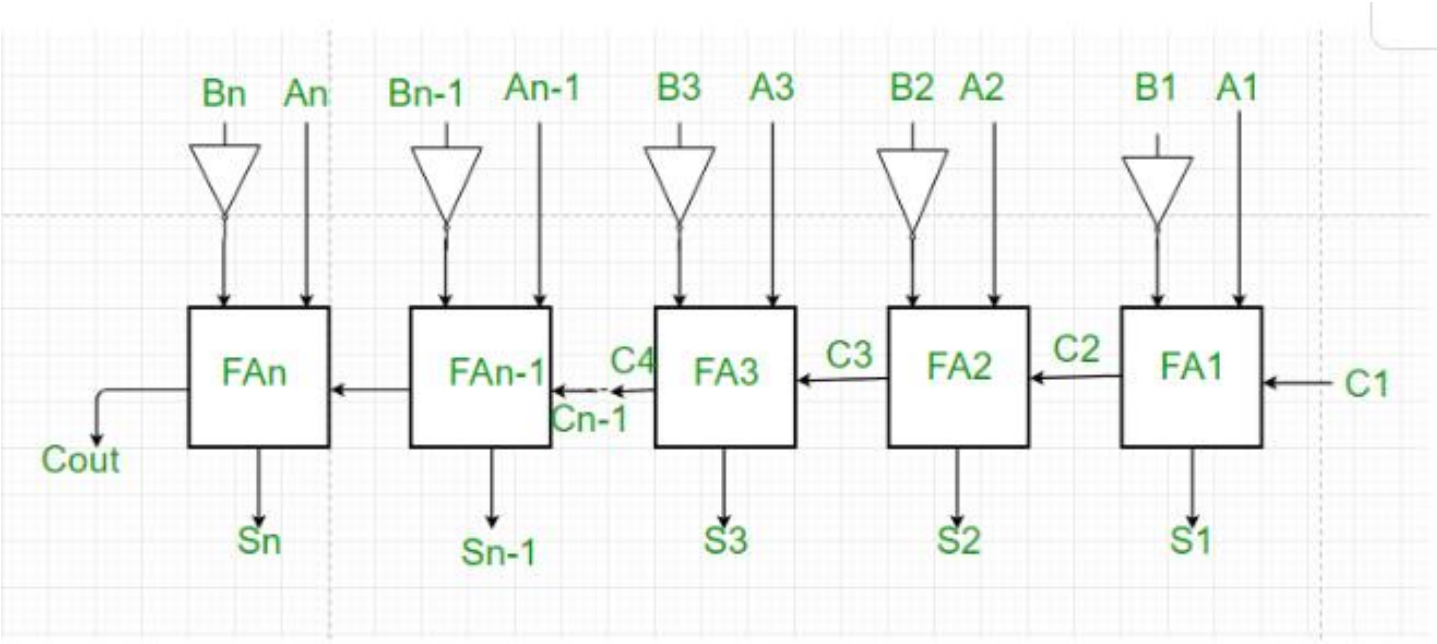
```
module fa_su(A,B,CIN,OUT,OUTC);
input A,B,CIN;
output OUT,OUTC;
assign OUT=A^B^CIN;
assign OUTC=(A&B)|(B&CIN)|(CIN&B);
endmodule
```

```
module pa_add(in1,in2,ic,out,oc);
input [3:0]in1;
input [3:0]in2;
input ic;
output [3:0]out;
output [3:0]oc;
fa_su fa1(in1[0],in2[0],ic,out[0],oc[0]);
fa_su  fa2(in1[1],in2[1],oc[0],out[1],oc[1]);
fa_su  fa3(in1[2],in2[2],oc[1],out[2],oc[2]);
fa_su  fa4(in1[3],in2[3],oc[2],out[3],oc[3]);

endmodule
```

```
module pa_tb;
reg [3:0]in1;
reg [3:0]in2;
reg ic;
wire [3:0]oc;
wire [3:0]out;
pa_add
dut(.in1(in1),.in2(in2),.ic(ic),.out(out),.oc(oc));
initial
begin
$monitor(in1,in2,ic,out,oc[3]);
in1=4'b1010; in2=4'b1001;ic=0;
#100;
end
endmodule
```

Parallel Subtractor



Logic diagram of 4-bit parallel subtractor

Parallel Subtractor

Working of Parallel Subtractor

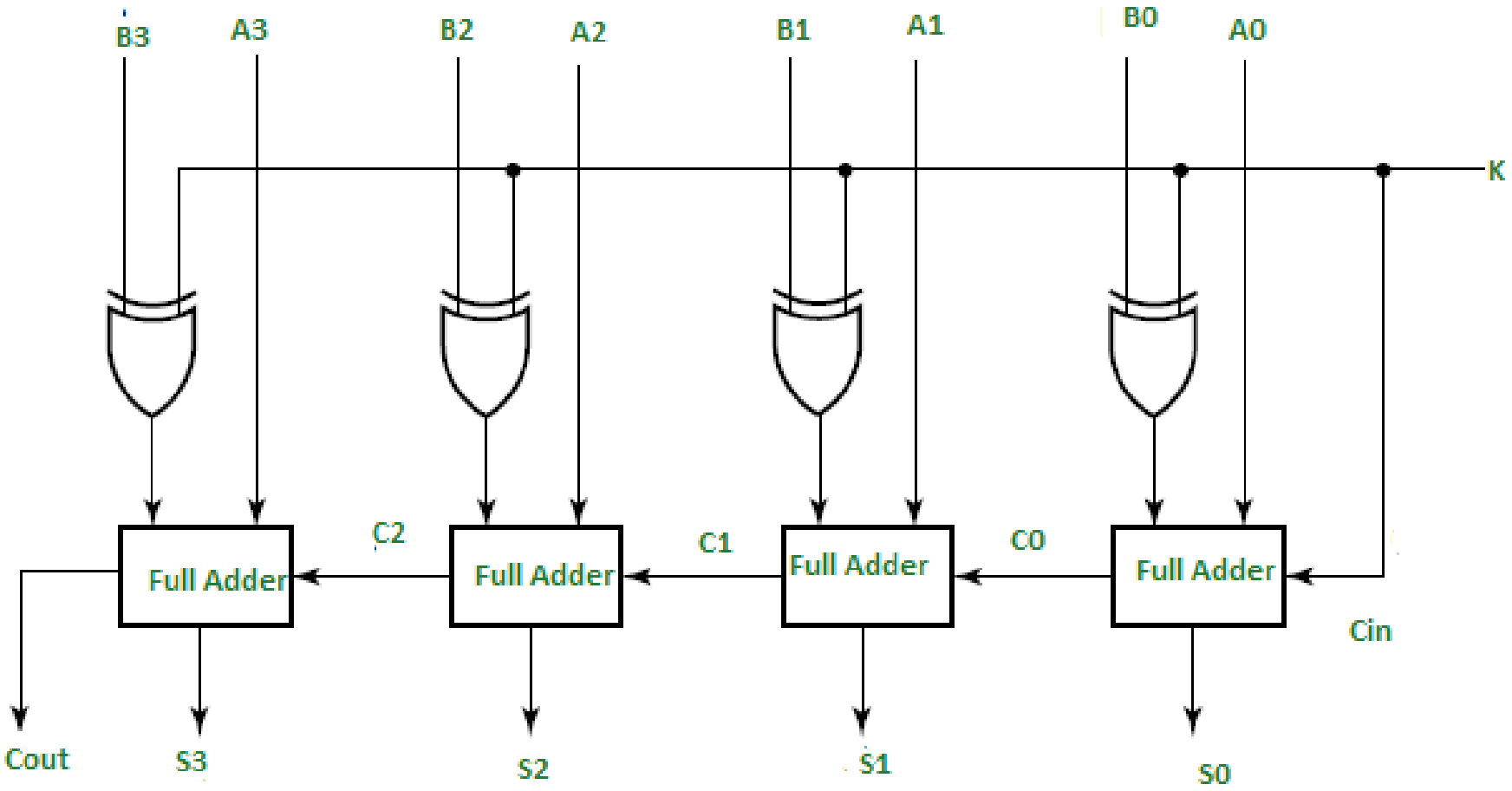
- The parallel binary subtractor is formed by combination of all full adders with subtrahend complement input

- This operation considers that the addition of minuend along with the 2's complement of the subtrahend is equal to their subtraction

- Firstly the 1's complement of B is obtained by the NOT gate and 1 can be added through the carry to find out the 2's complement of B. This is further added to A to carry out the arithmetic subtraction

- The process continues till the last full adder FAn uses the carry bit Cn to add with its input An and 2's complement of Bn to generate the last bit of the output along last carry bit Cout

Four Bit Adder Subtractor



Carry Look-Ahead Adder

- The adder produce carry propagation delay while performing other arithmetic operations like multiplication and divisions as it uses several additions or subtraction steps.
- This is a major problem for the adder and hence improving the speed of addition will improve the speed of all other arithmetic operations.
- Hence reducing the carry propagation delay of adders is of great importance. There are different logic design approaches that have been employed to overcome the carry propagation problem
- One widely used approach is to employ a carry look-ahead which solves this problem by calculating the carry signals in advance, based on the input signals
- This type of adder circuit is called a carry look-ahead adder
- Here a carry signal will be generated in two cases:
- Input bits A and B are 1
- When one of the two bits is 1 and the carry-in is 1

Carry Look-Ahead Adder

- A carry look-ahead adder reduces the propagation delay by introducing more complex hardware
- In this design, the ripple carry design is suitably transformed such that the carry logic over fixed groups of bits of the adder is reduced to two-level logic
- We define two variables as **'carry generate'** and **'carry propagate'**

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

where G_i produces the carry when both A_i, B_i are 1 regardless of the input carry. P_i is associated with the propagation of carry from C_i to C_{i+1}.

A	B	C	C +1	Condition
0	0	0	0	
0	0	1	0	No Carry
0	1	0	0	Generate
0	1	1	1	
1	0	0	0	No Carry
1	0	1	1	Propogate
1	1	0	1	Carry
1	1	1	1	Generate

Carry Look-Ahead Adder

The carry output Boolean function of each stage in a 4 stage carry look-ahead adder can be expressed as

$$C_1 = G_0 + P_0 C_{in}$$
$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

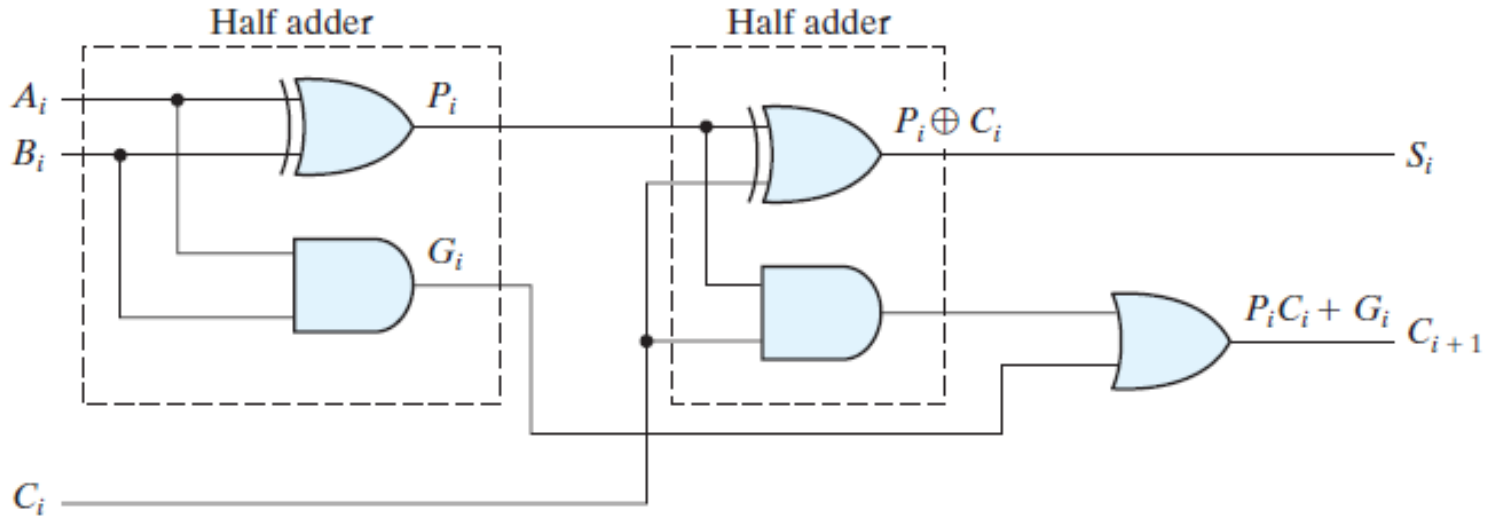


FIGURE 4.10
Full adder with *P* and *G* shown

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

the output sum and carry can respectively be expressed as

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

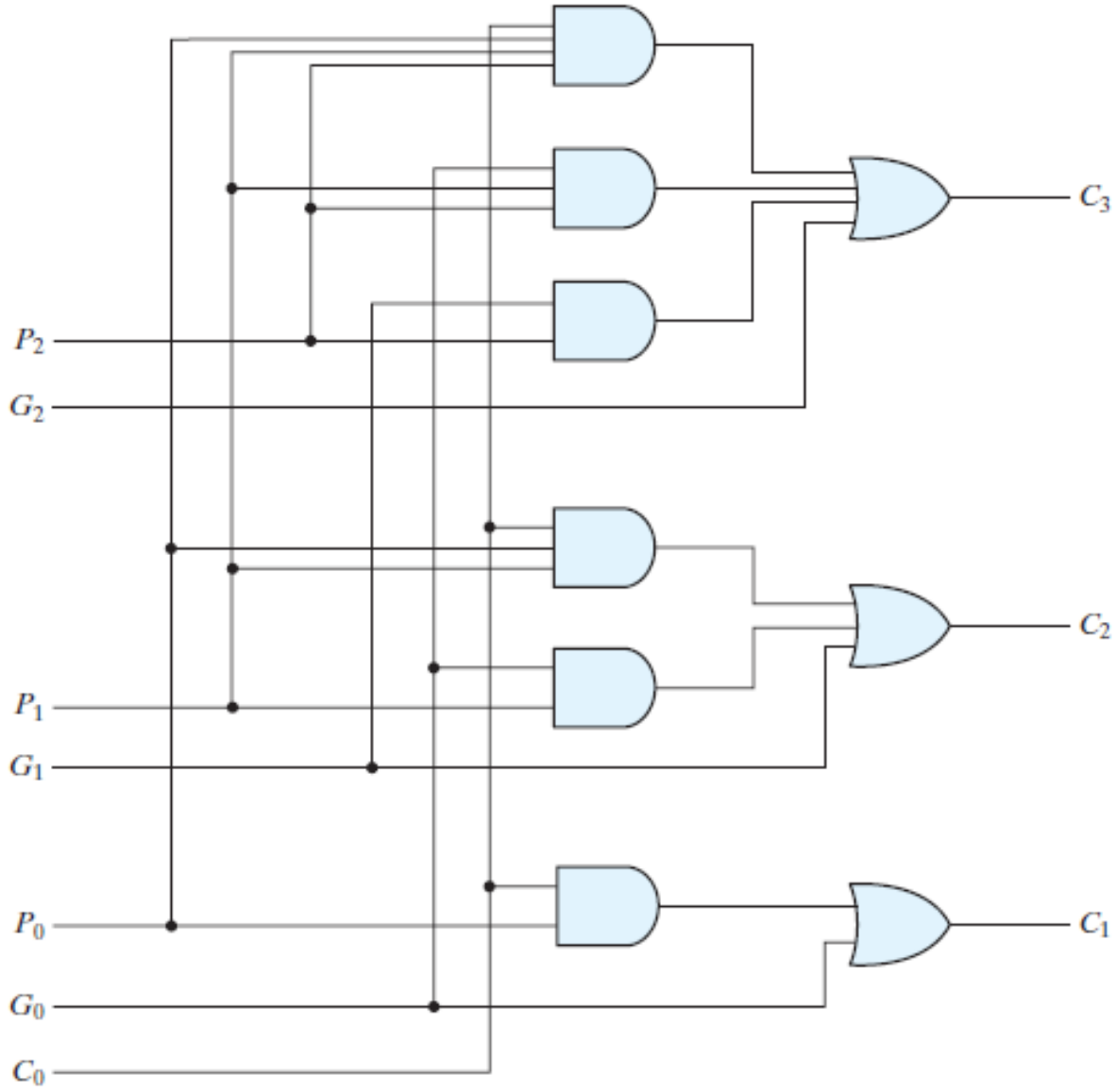
$C_0 =$ input carry

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 = P_2 P_1 P_0 C_0$$

- From the above Boolean equations we can observe that does not have to wait for and
- To propagate but actually is propagated at the same time
- Since the Boolean expression for each carry output is the sum of products so these can be implemented with one level of AND gates followed by an OR gate



URE 4.11

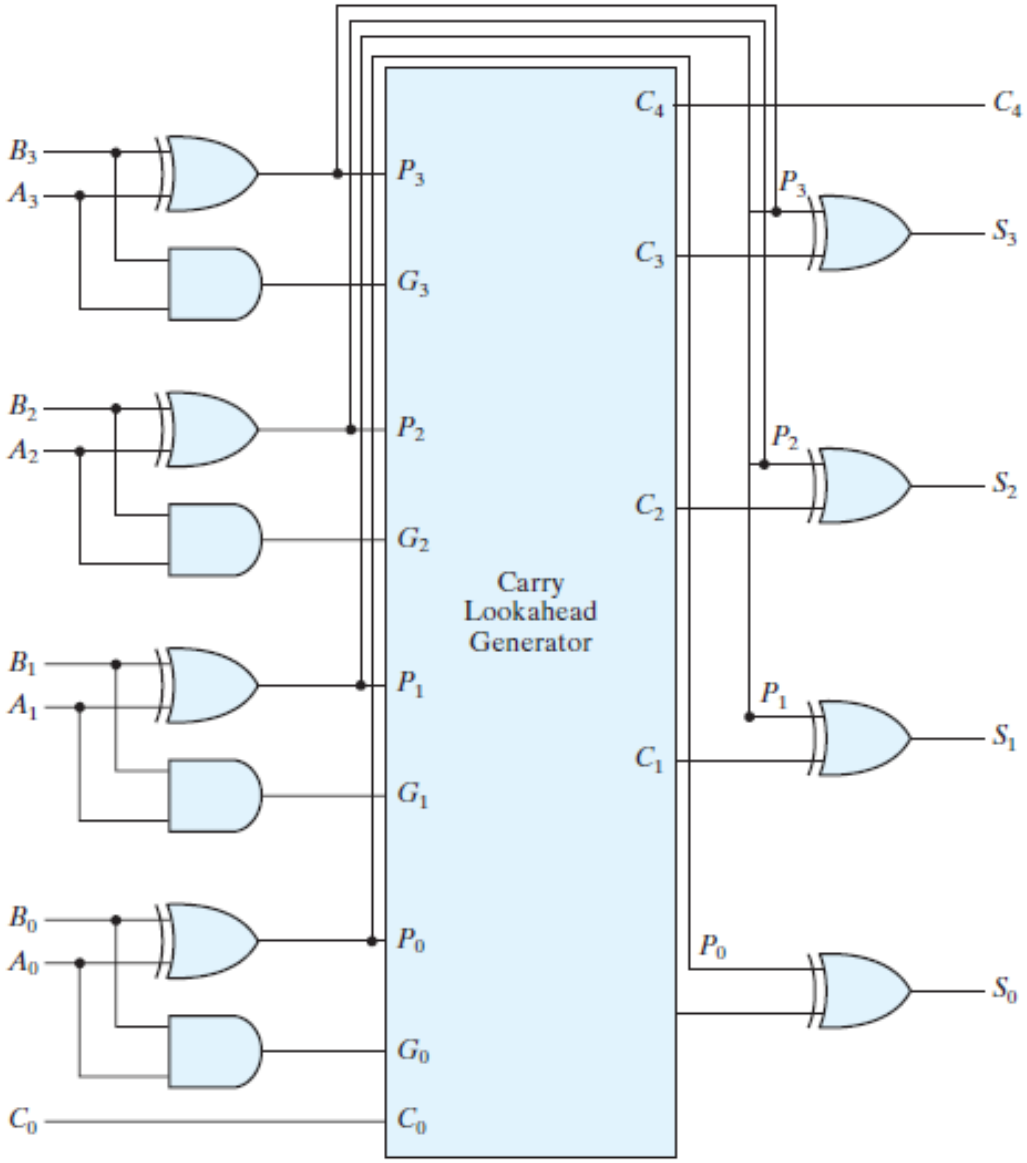


FIGURE 4.12
Four-bit adder with carry lookahead

Carry Look-Ahead Adder

Advantages –

The propagation delay is reduced.
It provides the fastest addition logic.

Disadvantages –

The Carry Look-ahead adder circuit gets complicated as the number of variables increase.
The circuit is costlier as it involves more number of hardware

Array Multiplier

- An **array multiplier** is a digital combinational circuit used for multiplying two binary numbers by employing an array of full adders and half adders
- This array is used for the nearly simultaneous addition of the various product terms involved
- To form the various product terms, an array of AND gates is used before the Adder array
- Checking the bits of the multiplier one at a time and forming partial products is a sequential operation that requires a sequence of add and shift micro-operations
- consider the multiplication of two 2-bit numbers
- The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3c2c1c0

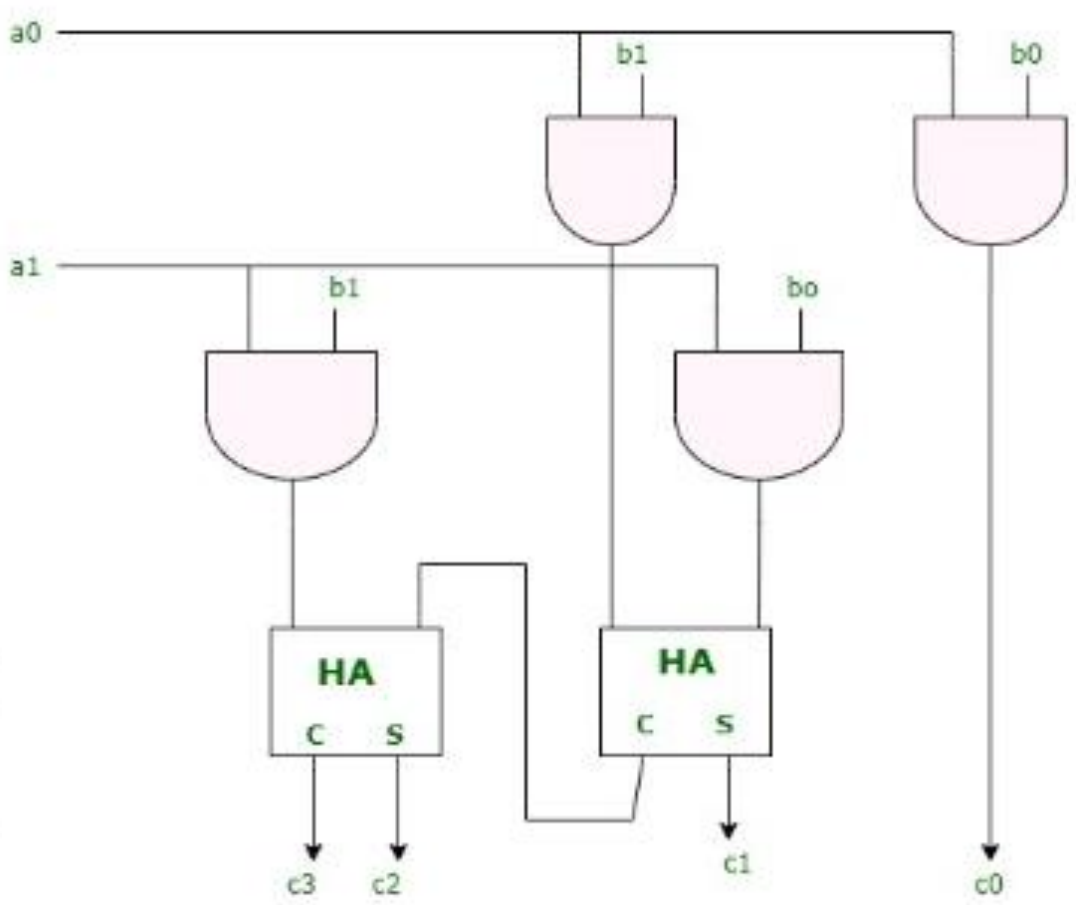
		b1	b0
		a1	a0
		a0b1	a0b0
	a1b1	a1b0	
c3	c2	c1	c0

Array Multiplier

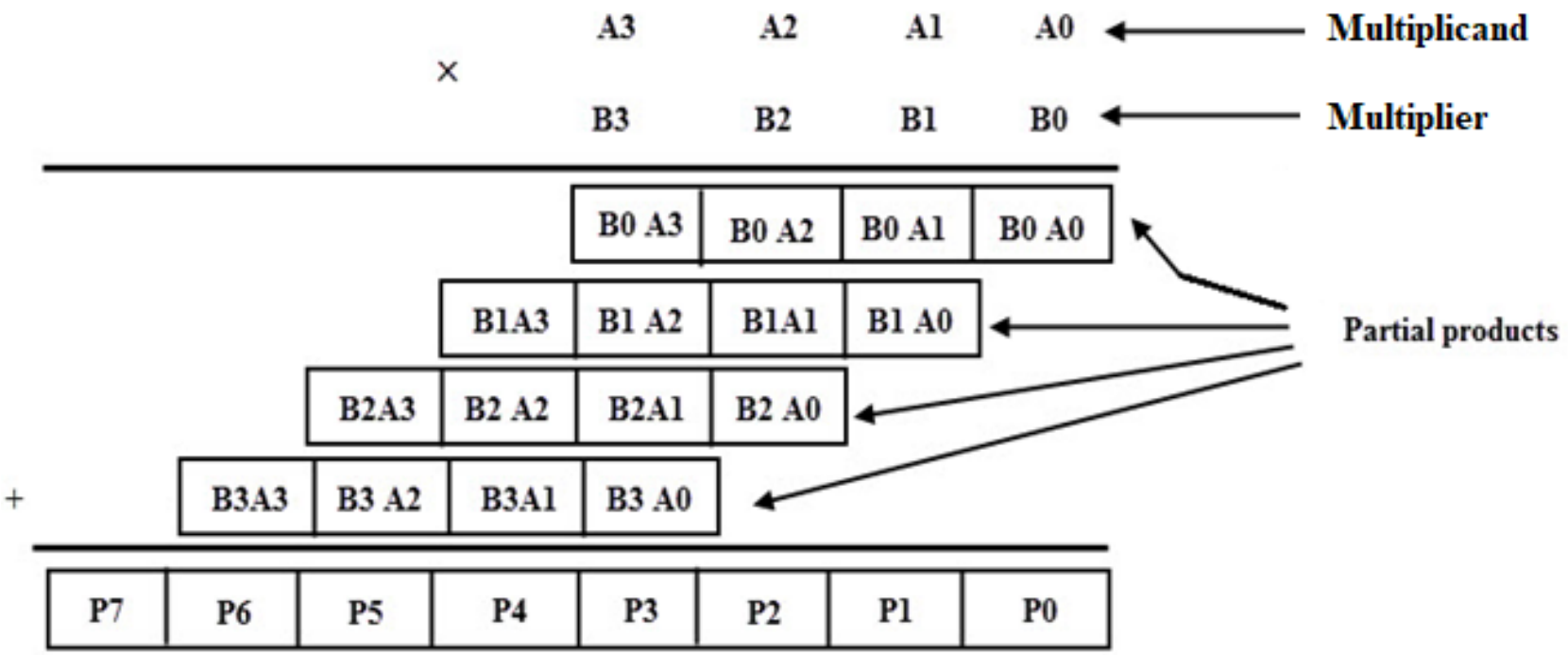
- The first partial product is formed by multiplying a0 by b1, b0. The multiplication of two bits such as a0 and b0 produces a 1 if both bits are 1; otherwise, it produces 0. This is identical to an AND operation and can be implemented with an AND gate
- The first partial product is formed by means of two AND gates
- The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left
- The above two partial products are added with two half-adder(HA) circuits. Usually there are more bits in the partial products and it will be necessary to use full-adders to produce the sum
- Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate

Array Multiplier

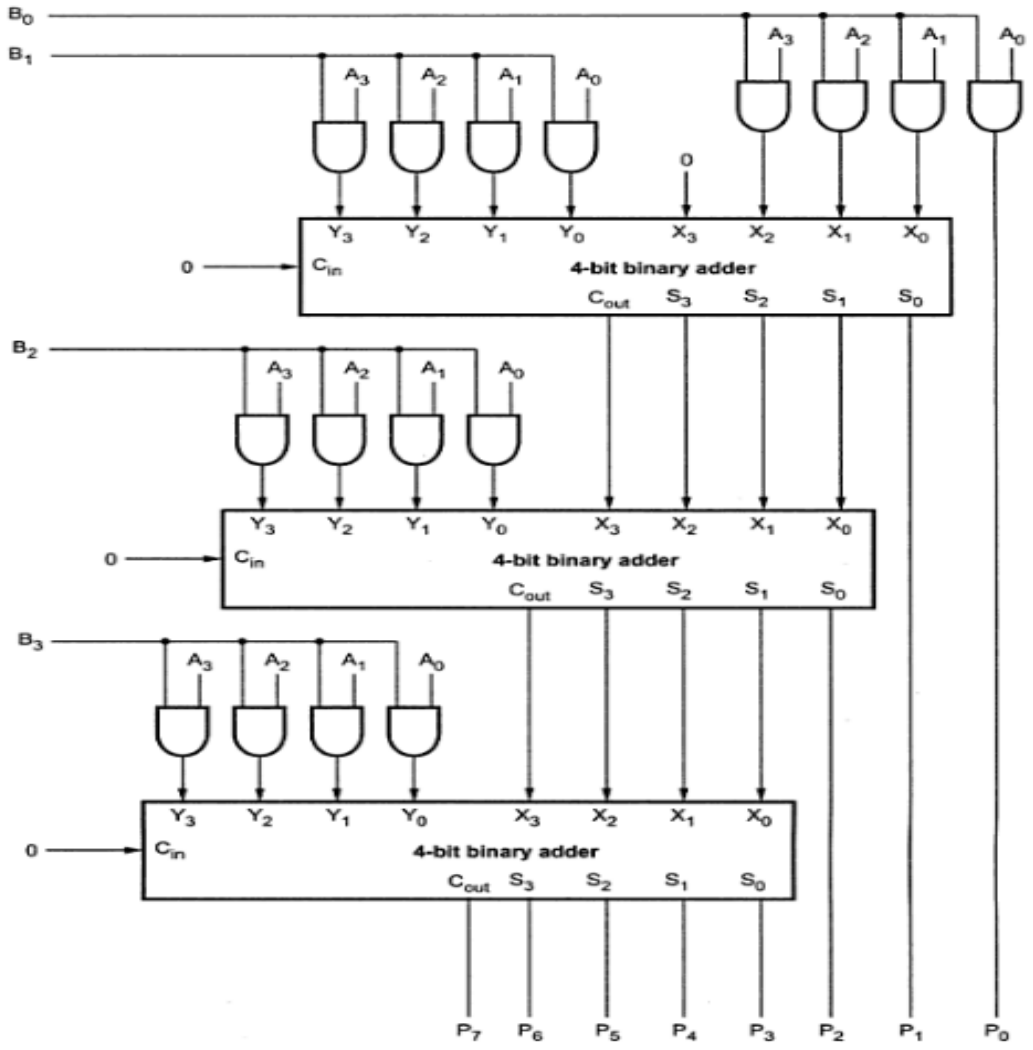
		b1	b0
		a1	a0
		a0b1	a0b0
	a1b1	a1b0	
c3	c2	c1	c0



Array Multiplier



Array Multiplier



Array Multiplier

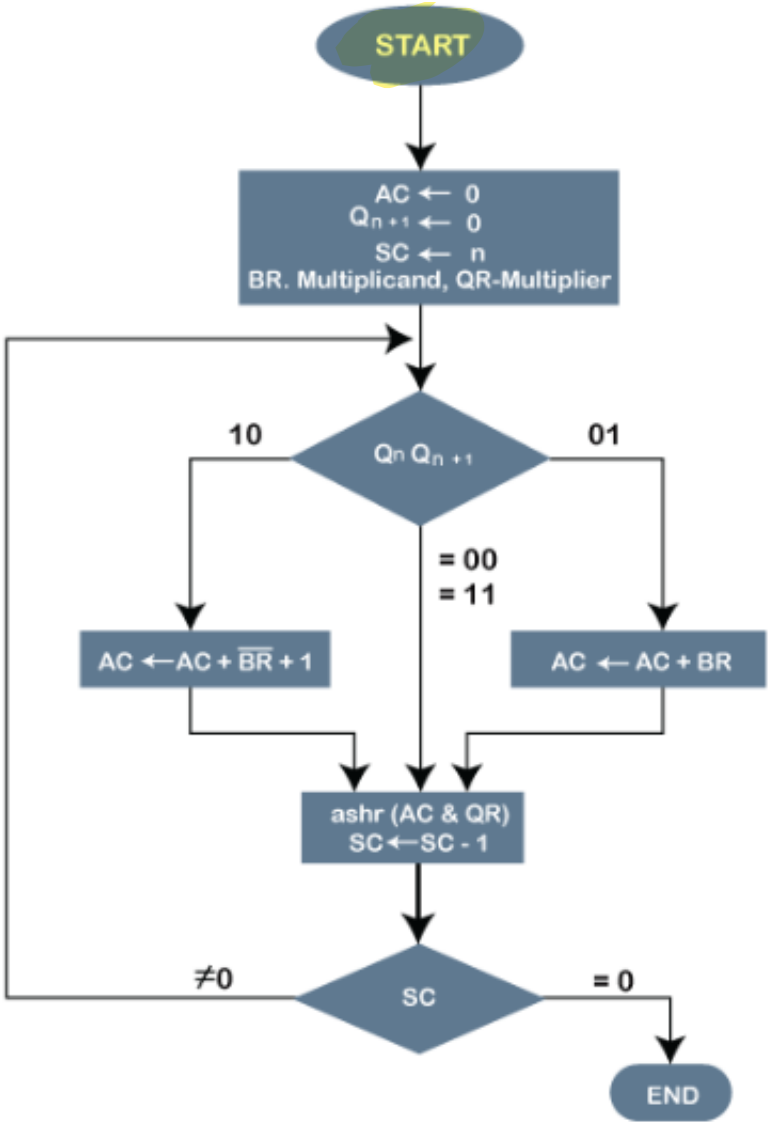
```
module Multiply_4x4(
   input [3:0] a,
   input [3:0] b,
   output [7:0] p );
         wire [3:0]pp0,pp1,pp2,pp3;
         wire c1,c2,c3,c4,c5,c6,c7,c8;
         wire s1,s2,s3,s4,s5,s6;


assign pp0=a[0]*b[3:0];
assign pp1=a[1]*b[3:0];
assign pp2=a[2]*b[3:0];
assign pp3=a[3]*b[3:0];
assign p[0]=pp0[0];
```

```
halfadder HA1(pp0[1],pp1[0],p[1],c1);
fulladder FA1(c1,pp0[2],pp1[1],s1,c2);
halfadder HA2(s1,pp2[0],p[2],c3);
fulladder FA2(c2,c3,pp0[3],s2,c4);
fulladder FA3(s2,pp1[2],pp2[1],s3,c5);
halfadder HA3(s3,pp3[0],p[3],c6);
fulladder FA4(c4,c5,c6,s4,c7);
fulladder FA5(s4,pp1[3],pp2[2],s5,c8);
halfadder HA4(s5,pp3[1],p[4],c9);
fulladder FA6(c7,c8,c9,s6,c10);
fulladder FA7(s6,pp2[3],pp3[2],p[5],c11);
fulladder FA8(c10,c11,pp3[3],p[6],p[7]);
endmodule
```

Booth's Multiplication Algorithm

- Booth algorithm gives a procedure for **multiplying binary integers** in signed 2's complement representation **in efficient way**,
- ie less number of additions/subtractions required
- initially, **AC** and $\mathbf{Q_{n+1}}$ bits are set to 0
- The **SC** is a sequence counter that represents the total bits set **n,** which is equal to the number of bits in the multiplier
- There are **BR** that represent the **multiplicand bits,** and QR represents the **multiplier bits**



Booth's Multiplication Algorithm

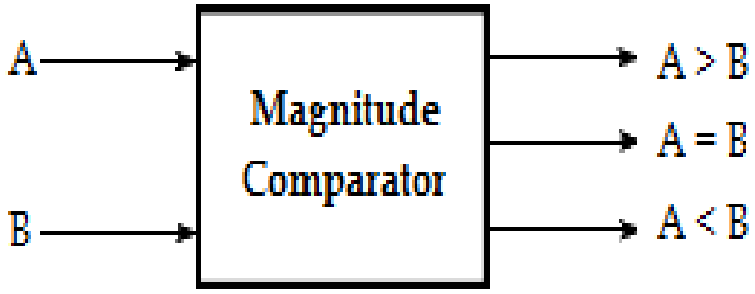
Q_n	Q_{n+1}	M = (0111) M' + 1 = (1001) & Operation	AC	Q	Q_{n+1}	SC
1	0	Initial	0000	0011	0	4
		Subtract (M' + 1)	1001			
			1001			
		Perform Arithmetic Right Shift operations (ashr)	1100	1001	1	3
1	1	Perform Arithmetic Right Shift operations (ashr)	1110	0100	1	2
0	**1**	Addition (A + M)	0111			
			0101	0100		
		Perform Arithmetic right shift operation	0010	1010	0	1
0	0	Perform Arithmetic right shift operation	**0001**	**0101**	0	0

The numerical example of the Booth's Multiplication Algorithm is 7 x 3 = 21 and the binary representation of 21 is 10101. Here, we get the resultant in binary 00010101. Now we convert it into decimal, as $(00010101)_{10}$ = 2*4 + 2*3 + 2*2 + 2*1 + 2*0 => 21.

MAGNITUDE COMPARATOR

MAGNITUDE COMPARATOR

- A magnitude comparator is a combinational circuit that compares two given numbers (A and B) and determines whether one is equal to, less than or greater than the other.
- The output is in the form of three binary variables representing the conditions A = B, A>B and A<B, if A and B are the two numbers being compared.



Block diagram of magnitude comparator

MAGNITUDE COMPARATOR

2-Bit Magnitude Comparator

Inputs				Outputs		
A_1	A_0	B_1	B_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Truth Table

MAGNITUDE COMPARATOR

2-Bit Magnitude Comparator

For A>B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$$

For A=B

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$$A=B = A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A_0B_1B_0 + A_1A_0'B_1B_0'$$
$$= A_1'B_1' (A_0'B_0' + A_0B_0) + A_1B_1 (A_0B_0 + A_0'B_0')$$
$$= (A_0 \odot B_0)(A_1 \odot B_1)$$

For A<B

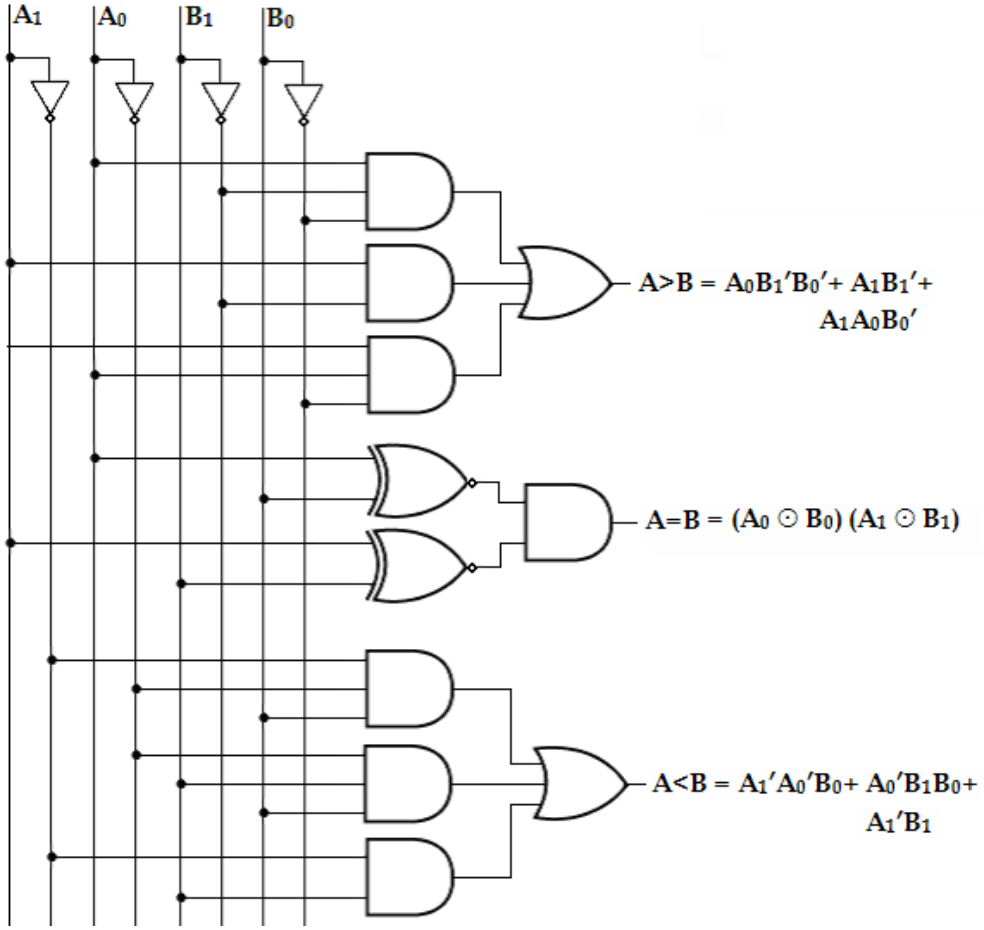
A_1A_0 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$$

K-Map Simplification

MAGNITUDE COMPARATOR

2-Bit Magnitude Comparator



Logic Diagram

MAGNITUDE COMPARATOR

4-Bit Magnitude Comparator

- In a 4-bit comparator the condition of A>B can be possible in the following four cases:
  1. If A3 = 1 and B3 = 0
  2. If A3 = B3 and A2 = 1 and B2 = 0
  3. If A3 = B3, A2 = B2 and A1 = 1 and B1 = 0
  4. If A3 = B3, A2 = B2, A1 = B1 and A0 = 1 and B0 = 0

- Similarly the condition for A<B can be possible in the following four cases:
  1. If A3 = 0 and B3 = 1
  2. If A3 = B3 and A2 = 0 and B2 = 1
  3. If A3 = B3, A2 = B2 and A1 = 0 and B1 = 1
  4. If A3 = B3, A2 = B2, A1 = B1 and A0 = 0 and B0 = 1

MAGNITUDE COMPARATOR

4-Bit Magnitude Comparator

COMPARING INPUTS				OUTPUT		
A3, B3	**A2, B2**	**A1, B1**	**A0, B0**	**A > B**	**A < B**	**A = B**
A3 > B3	X	X	X	H	L	L
A3 < B3	X	X	X	L	H	L
A3 = B3	A2 >B2	X	X	H	L	L
A3 = B3	A2 < B2	X	X	L	H	L
A3 = B3	A2 = B2	A1 > B1	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 > B0	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 < B0	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	H

H = High Voltage Level. L = Low Voltage. Level. X = Don't Care

MAGNITUDE COMPARATOR

4-Bit Magnitude Comparator

- Let us consider the two binary numbers A and B with four digits each. Write the coefficient of the numbers in descending order as,

$$\mathbf{A = A_3 A_2 A_1 A_0}$$
$$\mathbf{B = B_3\, B_2\, B_1\, B_0}$$

- Each subscripted letter represents one of the digits in the number. It is observed from the bit contents of two numbers that A = B when $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$.

MAGNITUDE COMPARATOR

4-Bit Magnitude Comparator

- Therefore, we can derive the logical expression of such sequential comparison by,

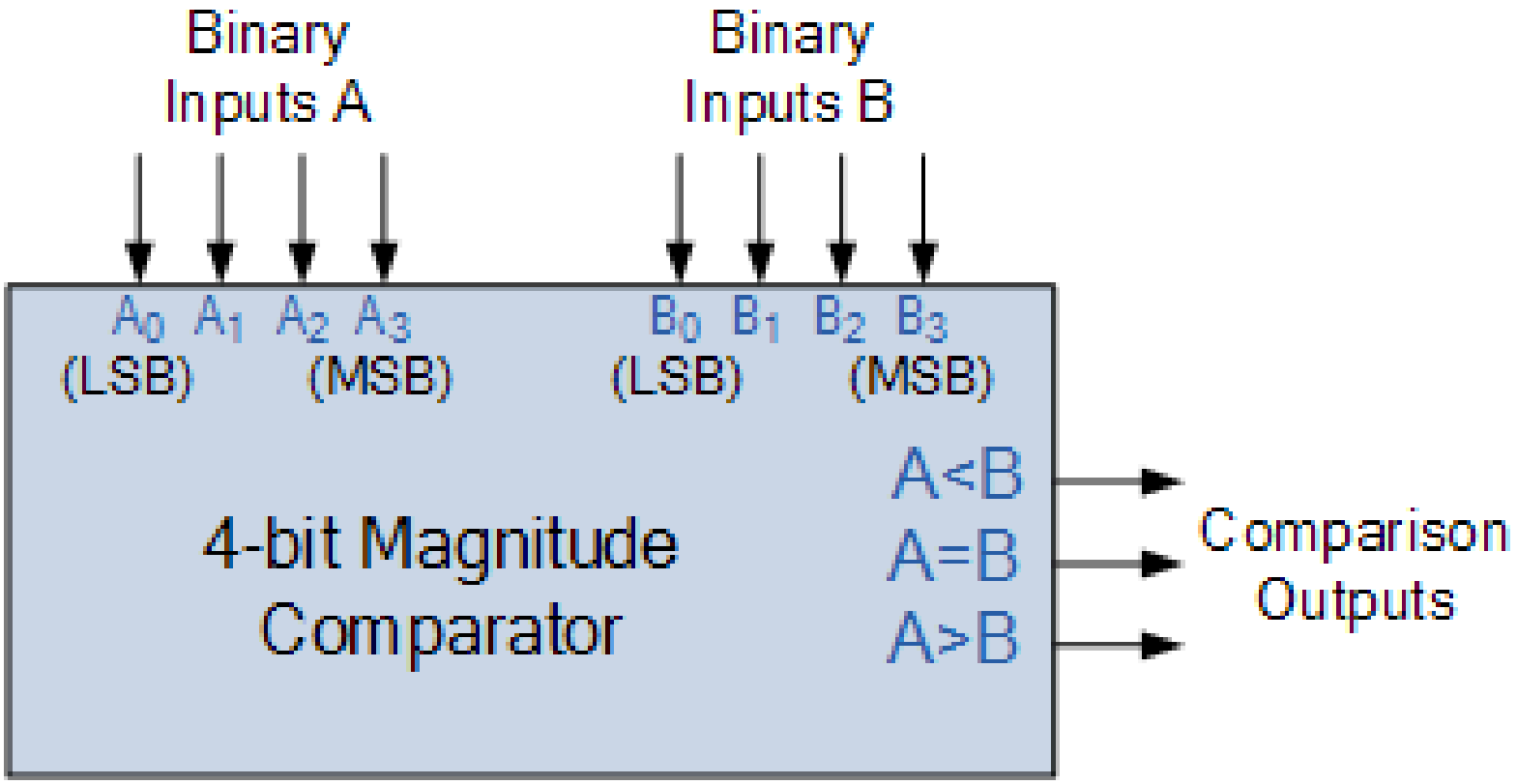
$$(A>B) = A_3B_3' + X_3A_2B_2' + X_3X_2A_1B_1' + X_3X_2X_1A_0B_0'$$

$$(A<B) = A_3'B_3 + X_3A_2'B_2 + X_3X_2A_1'B_1 + X_3X_2X_1A_0'B_0$$

- The symbols (A>B) and (A<B) are binary output variables that are equal to 1 when A>B or A<B, respectively.

- The unequal outputs can use the same gates that are needed to generate the equal output.

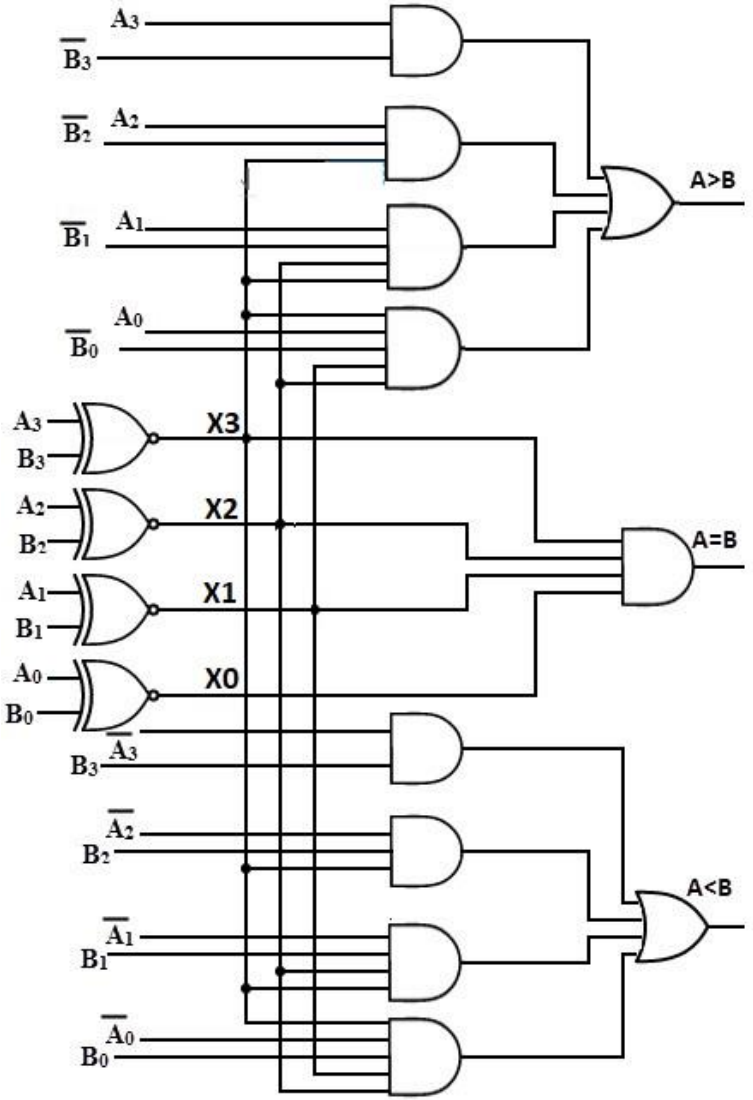
MAGNITUDE COMPARATOR



Block diagram of 4-Bit magnitude comparator
(IC7485)

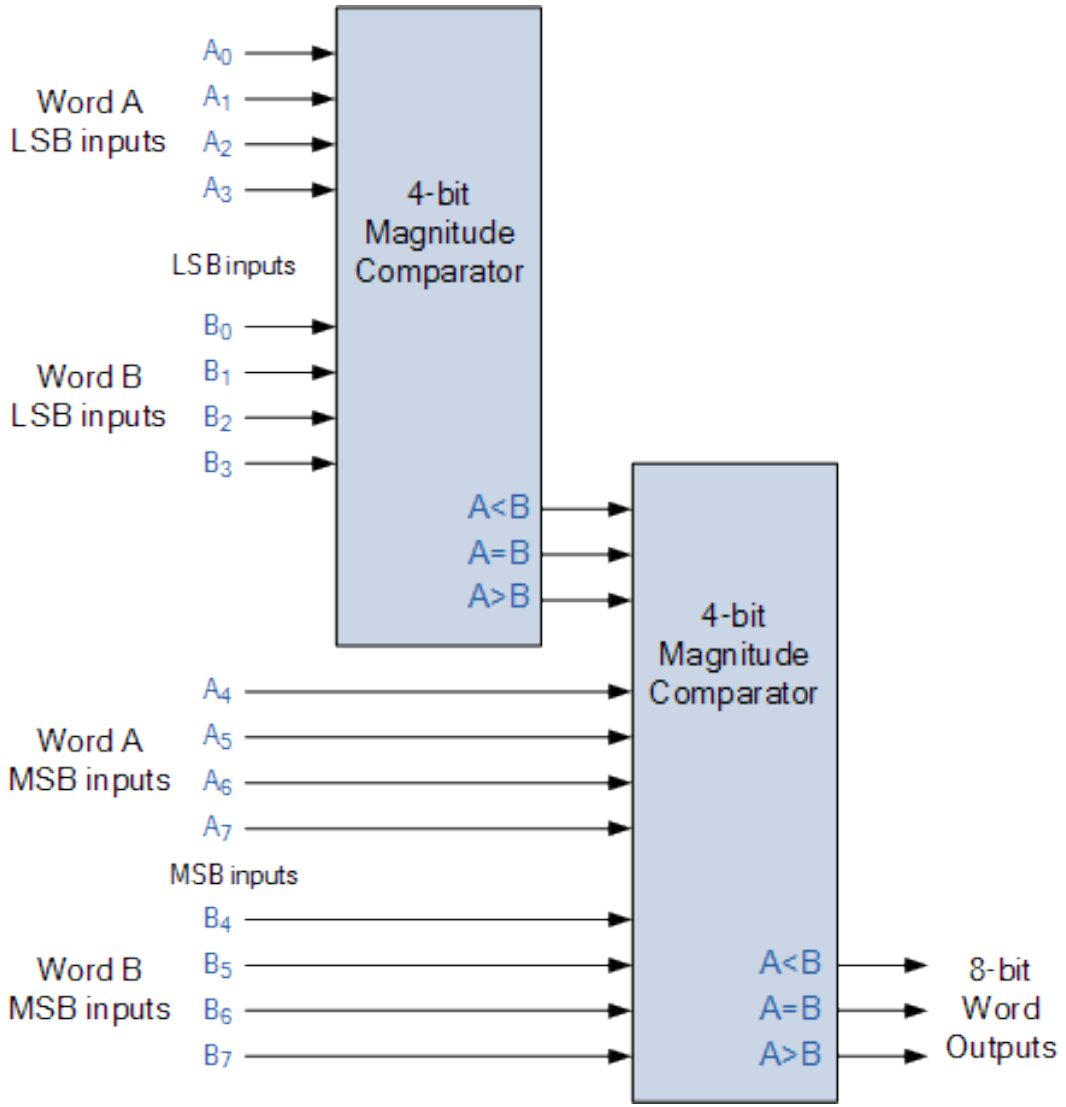
MAGNITUDE COMPARATOR

Logic diagram of 4-Bit magnitude comparator



MAGNITUDE COMPARATOR

Block diagram of 8-Bit magnitude comparator



MAGNITUDE COMPARATOR

4 bit Comparator:

```
module comparator(
   Data_in_A,  //input A
   Data_in_B,  //input B
   less,      //high when A is less than B
   equal,      //high when A is equal to B
    greater        //high when A is greater than B );

      input [3:0] Data_in_A;
   input [3:0] Data_in_B;
      output less;
    output equal;
    output greater;
    //Internal variables
    reg less;
    reg equal;
    reg greater;

always @(Data_in_A or Data_in_B)
    begin
      if(Data_in_A > Data_in_B)   begin
         less = 0;
         equal = 0;
         greater = 1;    end
      else if(Data_in_A == Data_in_B) begin
         less = 0;
         equal = 1;
         greater = 0;    end
      else    begin.
         less = 1;
         equal = 0;
         greater =0;
      end
   end
endmodule
```