16-Bit Booth Multiplier

By
SUMAN MAHATO

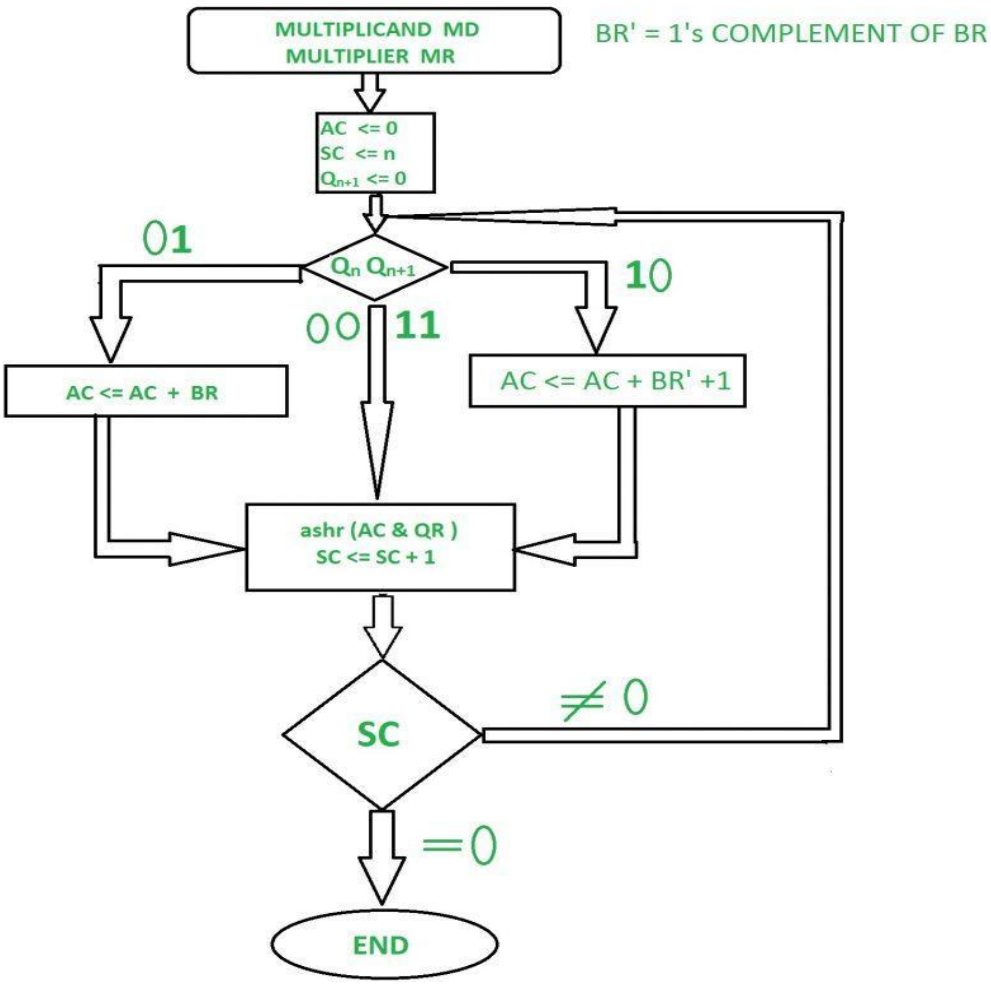
Problem Description

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation

Implementation

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values S(sum of multiplicand and accumulator) and D(difference of multiplicand and accumulator) to a product P, then performing a rightward arithmetic shift on P.

Flowchart Diagram :



Example

Let **A: 3** and **B: 17**

Multiplicand -	
Decimal:	3
Binary:	00000011

Multiplier -	
Decimal:	17
Binary:	00010001
Two's Complement:	11101111
Steps -	
Starting Out:	0000000000000011
Subtract:	1110111100000011
Shift:	1111011110000001
Shift:	1111101111000000
Add:	0000110011000000
Shift:	0000011001100000
Shift:	0000001100110000
Shift:	0000000110011000
Shift:	0000000011001100
Shift:	0000000001100110
Shift:	0000000000110011

Final Product (Binary):	0000000000110011
Final Product (Decimal):	51

Implementation of a Booth's algorithm uses various sub-modules, as described below :

Worst and Ideal Case

The **worst case of an implementation** using Booth's algorithm is when pairs of 01s or 10s occur very frequently in the multiplier.

Modules and Sub Modules

Implementation of Booth's algorithm uses various sub-modules, as described below.

1. **boothmul():** This module is the main module which uses the help of other sub-modules or counterparts to solve our problem.

   This module takes in two **8-bit signed** inputs, which are our multiplicand and multiplier. It has one **16-bit signed** output. Inside the module, we have **eight** 8-bit signed wires hold the value of the changed bits after shifting so that we can manipulate them later.

2. **booth_substep()**: This sub-module does the main operation of either adding/subtracting or just shifting the bits according to the last two positions.

   This module takes in an 8-bit signed **accumulator**, 8-bit signed **multiplier**, the last bit of the accumulator, 8-bit signed **multiplicand**, and the output consists of two 8-bit signed registers containing first 8 and last 8 bits of the product, and cq0 is the changed q0 after the shift operation.
3. **Adder()**: This sub-module adds two 8-bit register values, and gives out their sum. This uses a library module of **fa** which is nothing but a simple full adder.

4. **Subtractor()**: This sub-module subtracts two 8-bit register values, and gives out their difference. This uses a library module of **invert** to invert each bit separately, and then uses **fa** which is

nothing but a simple full adder as described above.

5. **Lib.v:**
    a. **invert(output ib,input b);**
    b. and2 (input wire i0, i1, output wire o);
    c. or2 (input wire i0, i1, output wire o);
    d. xor2 (input wire i0, i1, output wire o);
    e. **nand2 (input wire i0, i1, output wire o);**
    f. **nor2 (input wire i0, i1, output wire o);**
    g. xnor2 (input wire i0, i1, output wire o);
    h. and3 (input wire i0, i1, i2, output wire o);
    i. or3 (input wire i0, i1, i2, output wire o);
    j. nor3 (input wire i0, i1, i2, output wire o);
    k. nand3 (input wire i0, i1, i2, output wire o);
    l. xor3 (input wire i0, i1, i2, output wire o);
    m. xnor3 (input wire i0, i1, i2, output wire o);
    n. **fa (input wire i0, i1, cin, output wire sum, cout);**

Apart from using the above modules, we use a testbench to supply the initial values.

```
a = 8'b11110000;

b = 8'b11110000;

#10

a = 8'b10010101;

b = 8'b100000;
```

And so on and so forth.

Final Result on Screen

The results on the screen are printed like this:

VCD info: dumpfile tb_boothsalgo.vcd opened for output.

0	-16	X	-16	=	256
10	-107	X	32	=	-3424
20	7	X	0	=	0
30	1	X	1	=	1
40	60	X	5	=	300
50	-86	X	35	=	-3010
60	17	X	28	=	476
70	8	X	-65	=	-520
