PES UNIVERSITY, BENGALURU

FINAL SEMESTER ASSESSMENT (FSA) – B.TECH. (CSE) – III SEM

SESSION: AUGUST – DECEMBER, 2018

UE17CS206 – DIGITAL DESIGN & COMPUTER ORGANIZATION LABORATORY

Project Report

on

"16-Bit Booth's Multiplier"

Submitted by

Aekansh Dixit **PES1201701808**

Name of the Examiners	**Signature with Date**
1.________________________	________________________
2.________________________	________________________
3.________________________	________________________
4.________________________	________________________

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

PES UNIVERSITY

(ESTABLISHED UNDER KARNATAKA ACT NO. 16 OF 2013)

100 FEET RING ROAD, BENGALURU – 560 085, KARNATAKA, INDIA

16-Bit Booth's Multiplier

Problem Description

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

Booth's algorithm examines adjacent pairs of bits of the 'N'-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, $y_{-1} = 0$. For each bit y_i for i running from 0 to N − 1, the bits y_i and y_{i-1} are considered.

Where these two bits are equal, the product accumulator P is left unchanged. Where $y_i = 0$ and $y_{i-1} = 1$, the multiplicand times 2^i is added to P; and where $y_i = 1$ and $y_{i-1} = 0$, the multiplicand times 2^i is subtracted from P. The final value of P is the signed product.

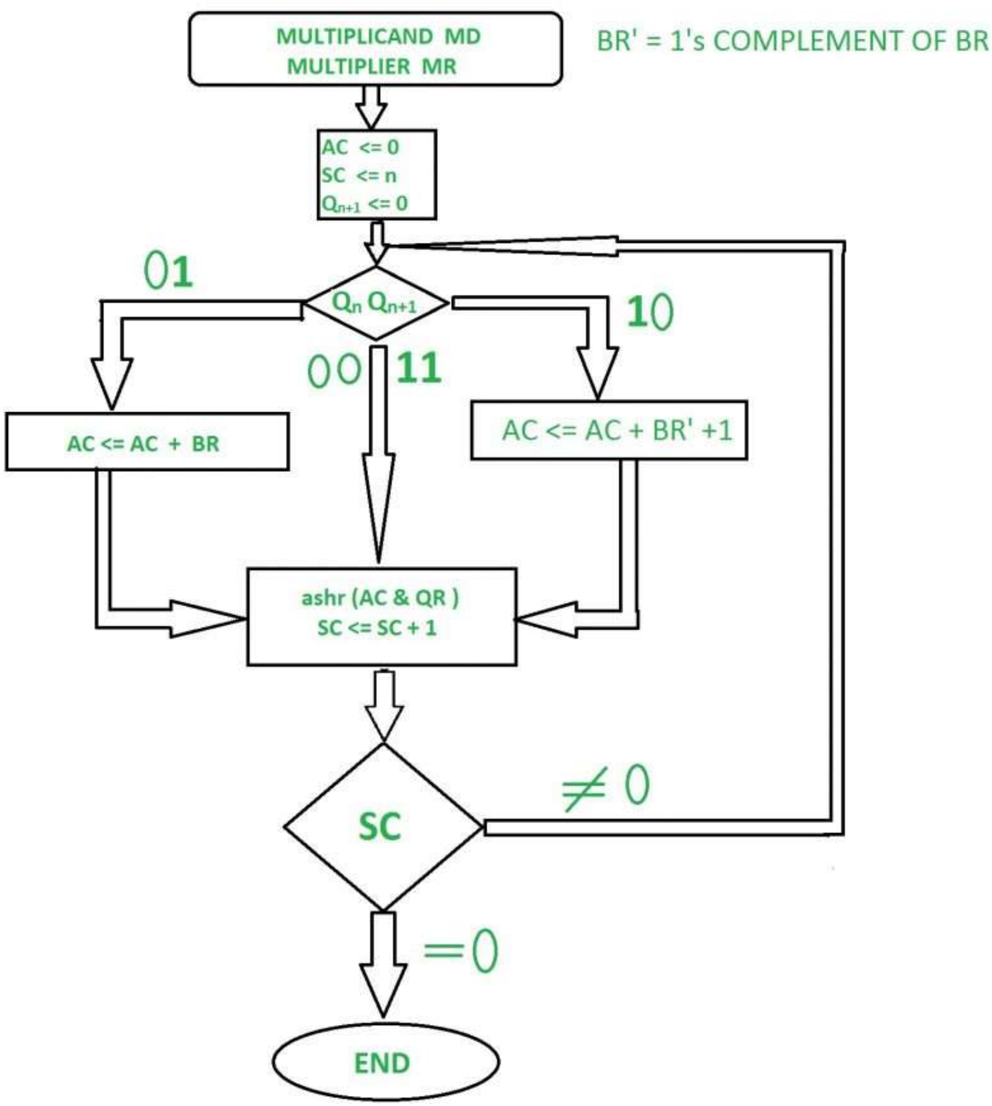
Implementation

Booth's algorithm can be implemented by repeatedly adding (with ordinary unsigned binary addition) one of two predetermined values A and S to a product P, then performing a rightward arithmetic shift on P. Let m and r be the multiplicand and multiplier, respectively; and let x and y represent the number of bits in m and r.

1. Determine the values of A and S, and the initial value of P. All of these numbers should have a length equal to (x + y + 1).
    a. A: Fill the most significant (leftmost) bits with the value of m. Fill the remaining (y + 1) bits with zeros.
    b. S: Fill the most significant bits with the value of (−m) in two's complement notation. Fill the remaining (y + 1) bits with zeros.
    c. P: Fill the most significant x bits with zeros. To the right of this, append the value of r. Fill the least significant (rightmost) bit with a zero.

2. Determine the two least significant (rightmost) bits of P.
    a. If they are 01, find the value of P + A. Ignore any overflow.
    b. If they are 10, find the value of P + S. Ignore any overflow.
    c. If they are 00, do nothing. Use P directly in the next step.
    d. If they are 11, do nothing. Use P directly in the next step.
3. Arithmetically shift the value obtained in the 2nd step by a single place to the right. Let P now equal this new value.
4. Repeat steps 2 and 3 until they have been done y times.
5. Drop the least significant (rightmost) bit from P.
   This is the product of m and r.

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 a 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
