Assignment No :A5

Roll No.4431

1 Title:

Booth's Multiplication on a cluster.

2 Problem Definition

Build a small compute cluster using Raspberry Pi/BBB modules to implement Booths Multiplication algorithm.

3 Learning Objectives

1. To understand the concept of clusters.
2. To understand the latency issue in cluster computing.
3. To be able to perform booth's multiplication on a cluster.

4 Learning Outcomes

1. Ability to form clusters and perform required computations on them.
2. Understanding of Booth's algorithm for multiplication.

5 Related Mathematics

Let S be the solution perspective of the given problem.

The set S is defined as:

$S = \{ s, e, X, Y, F, DD, NDD, S_c, F_c | \varnothing_s \}$

Where,

s= Start state, Such that $Y = \{\varnothing\}$

e= End state

X= Input Set.

$X = \{ \{x_1, x_2\} \mid x_i \in \text{Natural numbers} \}$

Y=Output set.

$Y = \{x_1 * x_2 \text{ (in binary) } \}$

F= Set of functions used.

F=$\{master(), slave(), boothMultiplication(), timeAnalyse()\}$

master()= function for master purposes.

slave()= function for slave purposes.

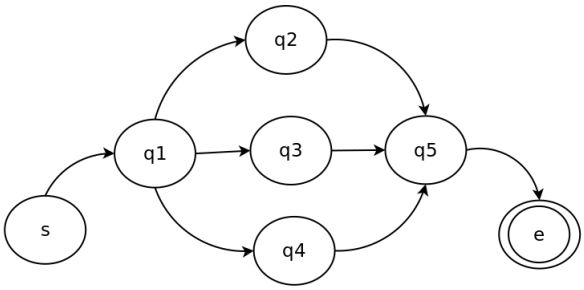
boothMultiplication()= function for computing the multiplication of the two numbers in binary.

DD=Deterministic data.
DD=

1. both numbers in the pair are proper.
2. multiplication is defined and exists.
3. computation terminates.

NDD= Non-deterministic data.
NDD= $\cup$ - DD

6 State Transition diagram



s = start state
q1 = master thread
q2 = slave thread
q3 = slave thread
q4 = slave thread
q5 = Collect the multiplication of the two numbers using booth's multiplicatior
e = end state

7 Concepts related theory

7.1 Booth's Multiplication

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture.

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 yi, for i running from 0 to N-1, the bits yi and yi-1 are considered. Where these two bits are equal, the product accumulator P is left unchanged. Where yi = 0 and yi-1 = 1, the multiplicand times 2i is added to P; and where yi = 1 and yi-1 = 0, the multiplicand times 2i is subtracted from P. The final value of P is the signed product.

The representations of the multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P.[1] There are many variations and optimizations on these details.

The algorithm is often described as converting strings of 1's in the multiplier to a high-order +1 and a low-order -1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

7.2 Cluster

A computer cluster consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system.

The components of a cluster are usually connected to each other through fast local area networks ("LAN"), with each node (computer used as a server) running its own instance of an operating system.

Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low-cost microprocessors, high speed networks, and software for high-performance distributed computing.

8 Algorithm

- Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation.
- The algorithm was invented by Andrew Donald Booth in 1950 while doing research on crystallography at Birkbeck College in Bloomsbury, London. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed.
- Booth's algorithm is of interest in the study of computer architecture. The algorithm 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. The multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined.
- Typically, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2 i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P.[1] There are many variations and optimizations on these details.
- The algorithm is often described as converting strings of 1's in the multiplier to a highorder +1 and a low-order -1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

9 Example

Find 3 x (−4), with m = 3
and r = −4, and x = 4 and y = 4:

m = 0011,
−m = 1101,
r = 1100

A = 0011 0000 0
S = 1101 0000 0
P = 0000 1100 0

Perform the loop four times:

1. P = 0000 1100 0. The last two bits are 00.
P = 0000 0110 0. Arithmetic right shift.
2. P = 0000 0110 0. The last two bits are 00.
P = 0000 0011 0. Arithmetic right shift.
3. P = 0000 0011 0. The last two bits are 10.
P = 1101 0011 0. P = P + S.
P = 1110 1001 1. Arithmetic right shift.

4. P = 1110 1001 1. The last two bits are 11.
P = 1111 0100 1. Arithmetic right shift.

The product is 1111 0100, which is −12.

The above mentioned technique is inadequate when the multiplicand is the most negative number that can be represented (e.g. if the multiplicand has 4 bits then this value is −8). One possible correction to this problem is to add one more bit to the left of A, S and P. This then follows the implementation described above, with modifications in determining the bits of A and S; e.g., the value of m, originally assigned to the first x bits of A, will be assigned to the first x+1 bits of A.

Below, we demonstrate the improved technique by multiplying −8 by 2 using 4 bits for the multiplicand and the multiplier:

* A = 1 1000 0000 0
* S = 0 1000 0000 0
* P = 0 0000 0010 0

Perform the loop four times:
1. P = 0 0000 0010 0. The last two bits are 00.
* P = 0 0000 0001 0. Right shift.
2. P = 0 0000 0001 0. The last two bits are 10.
* P = 0 1000 0001 0. P = P + S.
* P = 0 0100 0000 1. Right shift.
3. P = 0 0100 0000 1. The last two bits are 01.
* P = 1 1100 0000 1. P = P + A.
* P = 1 1110 0000 0. Right shift.
4. P = 1 1110 0000 0. The last two bits are 00.
* P = 1 1111 0000 0. Right shift.
* The product is 11110000 (after discarding the first and the last bit) which is −16.

10 Program Listing

```
// Assignment No. 5(Booth's algo)

#include <stdio.h>
#include <math.h>
int a = 0,b = 0, c = 0, a1 = 0, b1 = 0, com[5] = { 1, 0, 0, 0, 0};
int anum[5] = {0}, anumcp[5] = {0}, bnum[5] = {0};
int acomp[5] = {0}, bcomp[5] = {0}, pro[5] = {0}, res[5] = {0};
void binary(){
    a1 = fabs(a);
    b1 = fabs(b);
    int r, r2, i, temp;
    for (i = 0; i < 5; i++){
        r = a1 % 2;
        a1 = a1 / 2;
        r2 = b1 % 2;
        b1 = b1 / 2;
        anum[i] = r;
        anumcp[i] = r;
        bnum[i] = r2;
        if(r2 == 0){
            bcomp[i] = 1;
            }
            if(r == 0){
                acomp[i] =1;
                }
        }
       //part for two's complementing
    c = 0;
    for ( i = 0; i < 5; i++){
        res[i] = com[i]+ bcomp[i] + c;
        if(res[i] >= 2){
         c = 1;
        }
        else
         c = 0;
        res[i] = res[i] % 2;
        }
    for (i = 4; i >= 0; i--){
         bcomp[i] = res[i];
       }
      //in case of negative inputs
    if (a < 0){
       c = 0;
       for (i = 4; i >= 0; i--){
        res[i] = 0;
        }
        for ( i = 0; i < 5; i++){
```

```
            res[i] = com[i] + acomp[i] + c;
            if (res[i] >= 2){
                c = 1;
               }
               else
                c = 0;
            res[i] = res[i]%2;
        }
        for (i = 4; i >= 0; i--){
            anum[i] = res[i];
            anumcp[i] = res[i];
        }
    }
    if(b < 0){
        for (i = 0; i < 5; i++){
            temp = bnum[i];
            bnum[i] = bcomp[i];
            bcomp[i] = temp;
        }
    }
}
void add(int num[]){
    int i;
    c = 0;
    for ( i = 0; i < 5; i++){
        res[i] = pro[i] + num[i] + c;
        if (res[i] >= 2){
            c = 1;
        }
        else{
            c = 0;
        }
        res[i] = res[i]%2;
    }
    for (i = 4; i >= 0; i--){
        pro[i] = res[i];
        printf("%d",pro[i]);
    }
    printf(":");
    for (i = 4; i >= 0; i--){
        printf("%d", anumcp[i]);
    }
}
void arshift(){//for arithmetic shift right
    int temp = pro[4], temp2 = pro[0], i;
    for (i = 1; i < 5  ; i++){//shift the MSB of product
        pro[i-1] = pro[i];
    }
    pro[4] = temp;
    for (i = 1; i < 5  ; i++){//shift the LSB of product
```

```
        anumcp[i-1] = anumcp[i];
    }
    anumcp[4] = temp2;
    printf("\nAR-SHIFT: ");//display together
    for (i = 4; i >= 0; i--){
        printf("%d",pro[i]);
    }
    printf(":");
    for(i = 4; i >= 0; i--){
        printf("%d", anumcp[i]);
    }
}
void main(){
    int i, q = 0;
    printf("\t\tBOOTH'S MULTIPLICATION ALGORITHM");
    printf("\nEnter two numbers to multiply: ");
    printf("\nBoth must be less than 16");
    //simulating for two numbers each below 16
    do{
        printf("\nEnter A: ");
        scanf("%d",&a);
        printf("Enter B: ");
        scanf("%d", &b);
    }while(a >=16 || b >=16);
    printf("\nExpected product = %d", a * b);
    binary();
    printf("\n\nBinary Equivalents are: ");
    printf("\nA = ");
    for (i = 4; i >= 0; i--){
        printf("%d", anum[i]);
    }
    printf("\nB = ");
    for (i = 4; i >= 0; i--){
        printf("%d", bnum[i]);
    }
    printf("\nB'+ 1 = ");
    for (i = 4; i >= 0; i--){
        printf("%d", bcomp[i]);
    }
    printf("\n\n");
    for (i = 0;i < 5; i++){
        if (anum[i] == q){//just shift for 00 or 11
            printf("\n--------------------------------------------");
            arshift();
            q = anum[i];
        }
        else if(anum[i] == 1 && q == 0){//subtract and shift for 10
            printf("\n--------------------------------------------");
            printf("\nSUB B: ");
            add(bcomp);//add two's complement to implement subtraction
```

```
            arshift();
            q = anum[i];
        }
        else{//add ans shift for 01
            printf("\n-----------------------------------------");
            printf("\nADD B: ");
            add(bnum);
            arshift();
            q = anum[i];
        }
    }
    printf("\n\nProduct is = \n");
    for (i = 4; i >= 0; i--){
        printf("%d", pro[i]);
    }
    for (i = 4; i >= 0; i--){
        printf("%d", anumcp[i]);
    }
}
```

11 Output

```
botman@botmatrix:~$ scp booth.c root@192.168.7.2:
Debian GNU/Linux 7

BeagleBoard.org BeagleBone Debian Image 2014-05-14

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian
booth.c                                       100% 5148
5.0KB/s   00:00
botman@botmatrix:~$ ssh root@192.168.7.2
Debian GNU/Linux 7

BeagleBoard.org BeagleBone Debian Image 2014-05-14

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian
Last login: Thu May 15 03:36:15 2014 from botmatrix-2.local
root@beaglebone:~# ls
booth.c  direction  export  sys  unexport  value
root@beaglebone:~# gcc booth.c
root@beaglebone:~# ./a.out
                BOOTH'S MULTIPLICATION ALGORITHM
Enter two numbers to multiply:
Both must be less than 16
Enter A: 12
Enter B: 3

Expected product = 36

Binary Equivalents are:
```

```
A = 01100
B = 00011
B'+ 1 = 11101

---------------------------------------------------------------------------

AR-SHIFT: 00000:00110

---------------------------------------------------------------------------

AR-SHIFT: 00000:00011

---------------------------------------------------------------------------

SUB B: 11101:00011
AR-SHIFT: 11110:10001

---------------------------------------------------------------------------

AR-SHIFT: 11111:01000

---------------------------------------------------------------------------

ADD B: 00010:01000
AR-SHIFT: 00001:00100

Product is = 0000100100

root@beaglebone:~#
```

12 TESTING :

12.1 Positive Testing

Sr. No.	Test Condition	Steps to be executed	Expected Result	Actual Result
1.	Enter the A & B as int	Press Enter	Multiplication of A&B	Same as Expected

12.2 Negative Testing

Sr. No.	Test Condition	Steps to be executed	Expected Result	Actual Result
1.	Entered A&B as a character	Press Enter	error message	Multiplication of A&B
2.	Entered A&B greater than 16 bit	Press enter	Error message	Multiplication of A&B

13 CONCLUSION :

We have successfully implemented Booths Multiplication algorithm using BBB modules by building a compute cluster which include availability of low-cost, communitysupported development platform.