Assignment No. B5

Aim

Booths Multiplication Algorithm.

Problem Definition

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

Learning Objectives

- To use mathematical modeling for problem solving and to revel efficient use of data structures.
- To understand effective use of data flow programming architecture
- To understand use of job scheduling primaries
- To demonstrate use advanced capabilities of hardware beneath by designing algorithms using data structure.
- To design program to nullify the redundant conditions and loops for effective use of pipe-lined architecture
- Understanding distributed environment architecture.

Learning Outcome

- To generate desired multiplication outcome by identifying and nullifying the redundant if-then-else while taking 2s complement of an answer irrespective of the signs of multiplier and multiplicand.
- Implemented Booths Multiplication Algorithm.

Software And Hardware Requirements

- Latest 64-BIT Version of Linux Operating System
- Eclipse with Python
- Modelio Software

Mathematical Model

Let S be the system of solution set for given problem statement such that,
S = { s, e, X, Y, F, DD, NDD, Su, Fu }
where,

s = start state
such that, y = { }
e = end state
such that, y = { ans }
ans = multiplication of X1 and X2 $\in I^+$
X = set of input
such that X = { X1, X2 }
X1 $\in I^+, X2 \in I^+$
Y = set of output
such that Y = { ans}
F = set of function
such that F = { f1, f2, f3, f4 }
where,
f1 = function to accept numbers from server.
f2 = function to perform 2's compliment on client1.
f3 = function to perform rotate and addition on client2.
f4 = function to revert answerer back to server.
DD = Deterministic data
|X| ie. 2
NDD = Nondeterministic data
ans ie. product of two numbers.
number of client posting requests to server.
Su = Success
If numbers entered are valid.
Booth Multiplication works correctly on converted binary equivalent of integers.
Fu = Failure
If numbers entered are valid.

State Diagram

Theory

Booths Multiplication 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

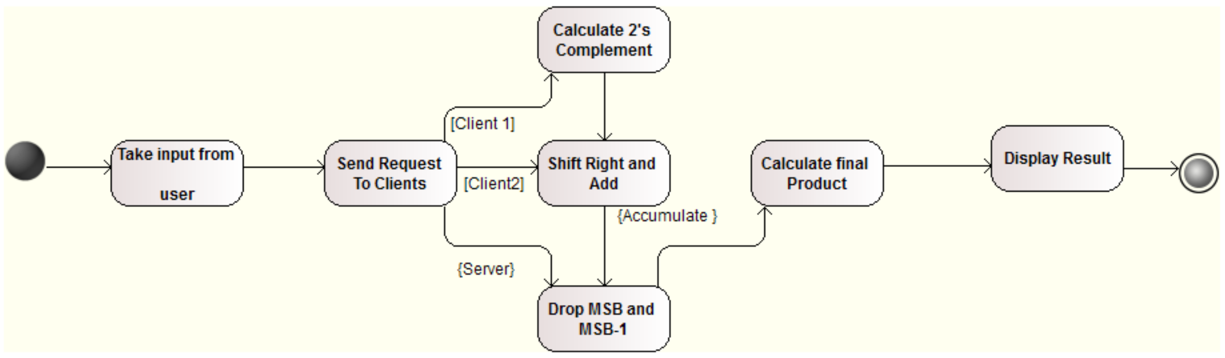

Figure 1: State Diagram

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 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.

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.

Program Code

```
Server.py
# take values A,S from clients and run the main loop for calculating P
#TODO- change the code for accomodating different sequence of exec(client2 r

import socket

def addition(op1, op2):
        # length of P and A and S is same.
        result=""
        carry="0"
        for i in range(len(op1)-1, -1, -1):          #run reverse loop
                if op1[i]=="1" and op2[i]=="1":
                        if carry=="1":
                                result="1"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="0"+result
                                carry="1"

                elif op1[i]=="0" and op2[i]=="0":
                        if carry=="1":
                                result="1"+result
                                carry="0"
                        else:                                   #carry = 0
                                result="0"+result
                                carry="0"

                elif op1[i]=="0" and op2[i]=="1":
                        if carry=="1":
                                result="0"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="1"+result
                                carry="0"

                else:                                           # 1, 0
                        if carry=="1":
                                result="0"+result
                                carry="1"
                        else:                                   #carry = 0
                                result="1"+result
                                carry="0"
        return result

s = socket.socket()         # Create a socket object
s.bind(("192.168.0.103", 9001))        # Bind to the port


M=int(input("Enter a multiplicant:"))
R=int(input("Enter a  multiplier:"))
M, R=bin(M), bin(R)
print "Binary representation: ", M, R

s.listen(2)                     # Now wait for client connection.
client, addr = s.accept()      # Establish connection with client.
print 'Got connection from', addr

client2, addr2 = s.accept()
print 'Got connection from', addr2
```

```
'''
Send the value of both. Client will return A, S. It will also return length_I
Send the value of length of R and value of R. Client will return P.     P<P>
'''
client.send(M+" "+R)

AS=client.recv(1024)     # recv A, S
index_A=AS.index("A")
index_S=AS.index("S")
A=AS[index_A+1:index_S]
S=AS[index_S+1:]

length_R=int(AS[:index_A])
client2.send(str(length_R)+" "+R)
P=client2.recv(1024)     # recv P
index_P=P.index("P")
P=P[index_P+1:]
P_length=len(P)

#we've got A,S,P in strings
for i in range(length_R):

        last_two_digits=P[P_length-2:P_length]

        if last_two_digits == "01":
                #add A in P and store that in P and ignore overflows
                P=addition(A, P)
        elif last_two_digits == "10":
                #add S in P aND store the result in P and IGNORE OVerflows
                P=addition(S, P)

        #print "After addn", P
        #arithmetic right shift (copy the sign bit as well). Start looping f
        P=P[0]+P[0:P_length-1]

P=P[:P_length-1]
print P



multiplicand.py
# calculate values A, S and send it back to server.

import socket
```

```
def twos_comp(binM):
        S = [int(x) for x in binM]
        flag = 0
        for i in range(len(S)-1, -1, -1):
                if flag==1:
                        #invert
                        if S[i]==1:
                                S[i]=0
                        else:
                                S[i]=1
                        continue

                if S[i]==1:
                        flag=1
        return S

s = socket.socket()         # Create a socket object
s.connect(("192.168.0.103", 9001))
temp=s.recv(1024)
temp=temp.split()
M, R=temp[0], temp[1]
origM, origR="", ""
Max_length=0

flag,flag_R=0,0                  # flag=1: -M, flag=2: M. flag_R=1: -R, flag_
if M[0]=="-":
        M=M[3:]
        origM=M
        M=twos_comp(M)
        M=[str(x) for x in M]
        M=''.join(M)
        flag=1
else:
        M=M[2:]
        flag=2

if R[0]=="-":
        R=R[3:]
        origR=R
        R=twos_comp(R)
        R=[str(x) for x in R]
        R=''.join(R)
        flag_R=1
else:
        R=R[2:]
        flag_R=2
```

```
if len(M)>= len(R):
        padding=len(M)-len(R)+1 #+1 for sign bit
        if flag==1:
                M="1"+M
        else:
                M="0"+M
        for i in range (padding):
                if flag_R==1:
                        R="1"+R
                else:
                        R="0"+R
        Max_length=len(M)
else:
        padding=len(R)-len(M)+1
        if flag_R==1:
                R="1"+R
        else:
                R="0"+R
        for i in range (padding):
                if flag==1:
                        M="1"+M
                else:
                        M="0"+M
        Max_length=len(R)

print M, R

#now calc A, S using the length of M and R and 1 (lenM+lenR+1)
A = []
for i in range(len(M)+len(R)+1):
        A.append(0)

for i in range(len(M)):
        A[i]=int(M[i])
A=[str(x) for x in A]
print "A: ", A
#A is ready at this point

if flag==1:             # orignal M was -ve. So we need origM with the minus
        for i in range(Max_length-len(origM)):
                origM="0"+origM
        S=[str(x) for x in origM]

else:
        S=twos_comp(M)
```

```
for i in range(len(M)+len(R)+1-len(S)):
        S.append(0)

S=[str(x) for x in S]

#S is ready at this point
print "S: ", S

#pack the A ans S in a buffer string
Send_AS= str(len(R))+"A"+''.join(A)      #secret- length of both operands is
Send_AS += "S"+''.join(S)
print Send_AS

#send the A and S to server and the job here is done
s.send(Send_AS)

multiplier.py
# calculate value P and send it back to server.

import socket

def twos_comp(binM):
        S = [int(x) for x in binM]
        flag = 0
        for i in range(len(S)-1, -1, -1):
                if flag==1:
                        #invert
                        if S[i]==1:
                                S[i]=0
                        else:
                                S[i]=1
                        continue

                if S[i]==1:
                        flag=1
        return S

s = socket.socket()          # Create a socket object
s.connect(("192.168.0.103", 9001))
temp=s.recv(1024)
temp=temp.split()
length_R, R= int(temp[0]), temp[1]

if R[0]=="-":
        R=R[3:]
        #origR=R
        R=twos_comp(R)
        R=[str(x) for x in R]
        R=''.join(R)
        for i in range (length_R-len(R)):
                R="1"+R
else:
        R=R[2:]
        for i in range (length_R-len(R)):
                R="0"+R
        #flag_R=2

P = []
for i in range(2*length_R + 1):
        P.append(0)

print "check length of P: ", P

for i in range(len(R)):
        P[length_R+i]=int(R[i])

P=[str (x) for x in P]
P="".join(P)
print P
s.send("P"+P)
```

Output

```
Server side :
root@Student5 Booth's Algorithm# python server.py
Enter a multiplicant:5
Enter a  multiplier:2
Binary representation:  0b101 0b10
Got connection from ('192.168.6.67', 36176)
Got connection from ('192.168.6.79', 36224)
00001010
```

```
Client1 :
root@student-OptiPlex-3010:/home/student/Documents# python client_multiplier
[0, 0, 0, 0, 0, 0, 0, 0, 0]
000000100
root@student-OptiPlex-3010:/home/student/Documents#
```

```
Client2 :
root@student-OptiPlex-3010:/home/student/Documents# python client_multiplica
0101 0010
A:  ['0', '1', '0', '1', '0', '0', '0', '0', '0']
S:  ['1', '0', '1', '1', '0', '0', '0', '0', '0']
4A010100000S101100000
root@student-OptiPlex-3010:/home/student/Documents#
```

Conclusion

Thus we have successfully implemented distributed booths algorithm.