# Summery of 7/5/98 lesson:

By: Oren Malerevich.
ID : 024064495.

# Array multipliers:

Another way to implement an array multiplier of signed numbers in two's complement notation is to add +/- sign to every partial multiplication.

In order to implement [p]=[a]*[b] multiplication
which it's inputs are [a],[b] signed numbers of length n in two's complement notation.
And it's output is [p] signed numbers in two's complement notation.

We will use the following strings in order to represent the inputs:

a= a(0).a(-1) ... a-(n-1).

b= b(0).b(-1) ... b-(n-1).

a(0),b(0) are the most significant bits of the numbers [a],[b].

In order to calculate the multiplication we will use an array which is depicted in figure 1.

The multiplication (Π) of two numbers in two's complement is :

(1) $\Pi = [a]*[b] = (-a(0) + <a'>)*(-b(0)+<b'>)$

a'= a(-1) ..... a(-n+1).

b'= b(-1) ..... b(-n+1).

So the sign for every partial multiplication should be :

+a(i)*b(j) => if ((i,j <0) or (i=j=0)) .

- a(i)*b(j) => if ((i=0 & j<0 ) or (I<0 & j=0 )).

In order to implement an array with sign for each partial multiplication, lets declare 3 blocks that we will use in the array multiplier :

PPP : Block that has :
Three input bits :
three positive : x,y,z.
Two outputs bits:
positive s (sum) and
positive c (carry).
It's equation is x+y+z = 2c+s( FA).

PPM : Block that has:
Three input bits :
two positive : x,y.
one negative : z.
Two outputs bits:
negative s (sum) and
positive c (carry).
It's equation is x+y-z = 2c-s.

MMP : Block that has:
Three input bits :
two negative: y,z .
one positive: x.
Two ouput bits :
negative s (sum) and
positive c (carry).
It's equation is x-y-z = -2c+s.

We will build a net with the above blocks in order to implement an array multiplier according to the following rules :

1. Output with + sign should enter to input with + sign.
   Output with - sign should enter to input with - sign.

2. Output from s(sum) should enter to a block in the same column .

3. Output from c(carry) should enter to a block the is in one column to the left.

4. The inputs and the outputs are in two's complement notation .

5. The outputs are : p(1)p(0).p(-1).... p-(2n-2).
   p(1) is the sign bit.

A 5x5 bit multiplication is depicted in Figure 1.

The net in Figure 1 is implementing equation 1 above.
Every negative bit result is propagate through the net . It can be canceled if it sums with positive bit .If the result of p(1) equal ‘1’ then the whole result number is negative.

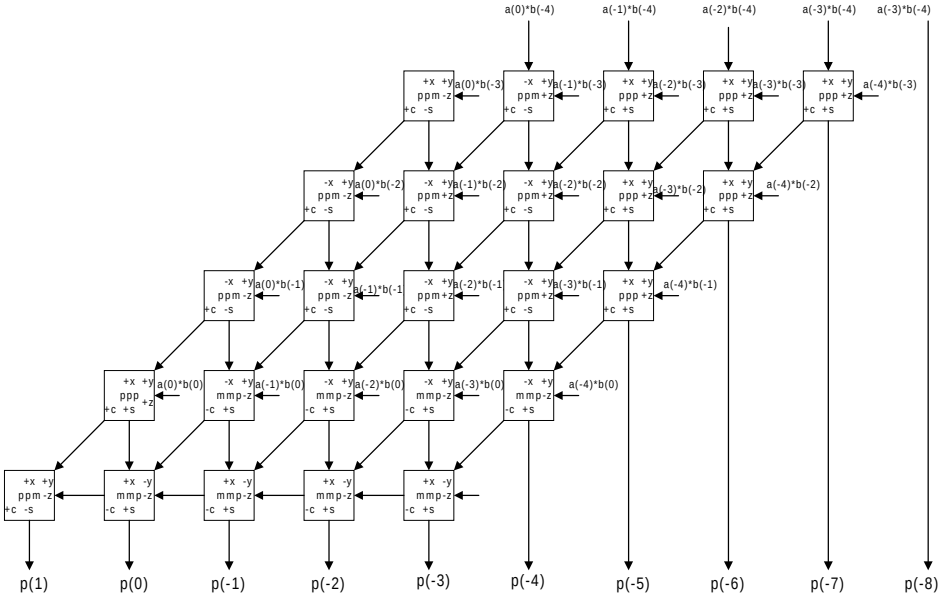


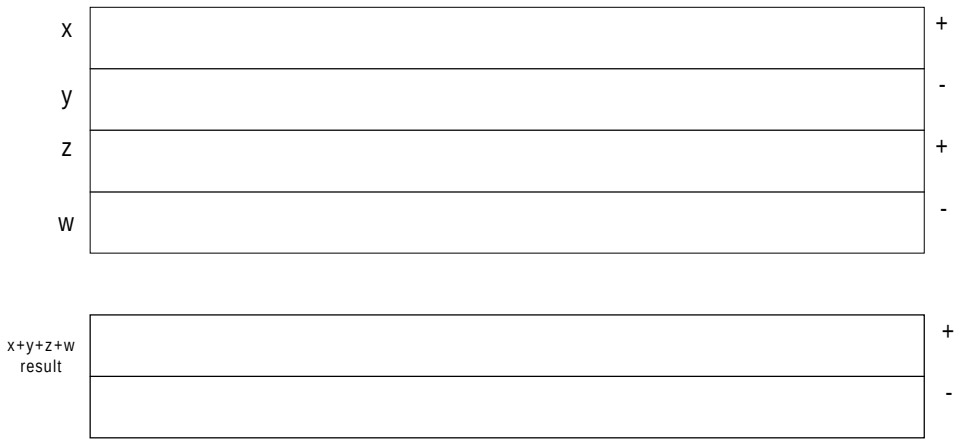

**Figure 1** array multiplier for two's complement

The delay of  array multiplier in figure 1 is linear with n ( O(n)).

Another way to calculate an array multiplier in logarithm delay is to separate the partial multiplication into positive rows and negative rows.
The sum of the rows will be done by 4:2 adders as depicted in figure 2 until we will get 2 rows, one is positive sum and the other is negative sum.

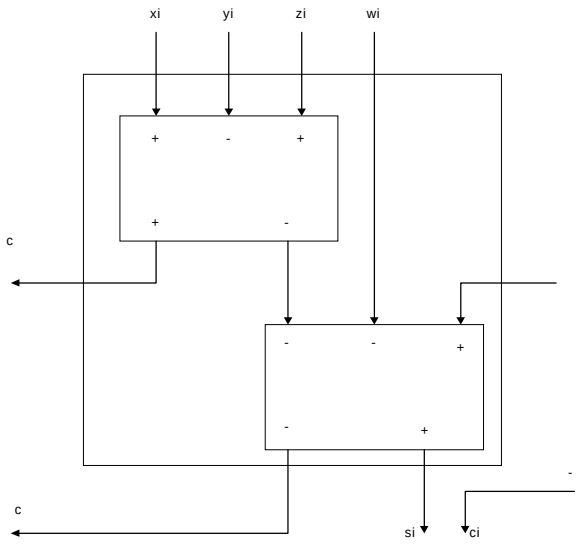
The sum of the two result rows into two's complement notation number  can be done with the blocks  of  PPM and MMP.

Figure 2: 4:2 adder array for summing two's complement array multipliers




Depicted in figure 3 is a Bit -Slice of 4:2 adder that accept 2 negative vectors and 2 positive vectors and one positive carry and one negative carry ,and it's outputs are: positive sum ,positive carry and negative carry.

Figure 3: bit slice of 4:2 adder.



## Reducing the number of partial products.

In order to reduce the number of partial products we may examine two or more bits of the multiplier at a time. However , this scheme requires the generation of the multiples A,2A,3A .. where A is the multiplicand.

We may represent binary number with less digits by doing the following transform:

$\{0,1\}^n \rightarrow \{-2^{k-1},....,2^{k-1}\}^{n'}$

n'= $\lceil (n+1)/k \rceil$. $\lceil \rceil$ = round up.

For example if we have a binary number <b> with n digits we will transform it to x with n' digits but every digit in x can represent the range $\{-2^{k-1},....,2^{k-1}\}$ :
So we can write the transformation of <b> to x as:

<b> = b(n-1) .... b(0) $\rightarrow$ x = x(n'-1) ... x(0).

And in order to get <b> from x we will do:

$$< b >= \sum_{i=0}^{n'-1} x(i) * (2^k)^i$$

So there is a reduction in the number of digits in x but every digit in x contains more bits then a digit in <b>.

# Booth recoding.

## Booth 1 :

Lets a = a(n-1) ..... a(0) $\in \{0,1\}^n$ .

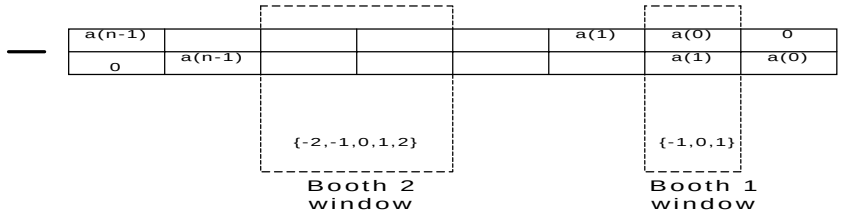
So we can write the equation (2): <a> = 2<a>- <a>.

let x be the transformation of <a> with k=1.

So every digit in x represent $\{-2^{k-1},....,2^{k-1}\}$ = {-1,0,1}.

And we can find x by subtracting the upper row (in figure 4 ) from the lower one using equation (2).

Figure 4: Boot recording windowing




So we can represent x in booth 1 by doing:

x(j) = a(j-1)-a(j).

For j=0 a(j-1) is extended to be '0'.

The range of digit in x is {-1,0,1}

In order to get <b> from x we will use the following equation:

$$< b >= \sum_{i=0}^{n'-1} x(i) * 2^i$$ because k =1.

Booth 2 :

In booth 2 we are looking on wider window (see figure 4 ) and the representation of x is:

x(j) = -2*a(j+1)+a(j)+a(j-1).

For j=0 a(j-1) is extended to be ‘0’.

The range of digit in x is {-2,-1,0,1,2}

Booth K:

In the general case we will use a window of size k and the range of a digit in x will be:

$\{-2^{k-1},....,2^{k-1}\}$ .

Multiplication using booth recording :

K=1:

By extending the range of the digits to {-1,0,1} we can implement multiplication by shift, add and subtract operations.
The advantage of using this method is that we can replace a sequence of 111111111 with sequence of digits with only two digits that are not zero : $100000001\bar{}$.
(So we need to do only two subtraction/addition) .
So we will represent the multiplier with string of length

n'= ⌈(n+1)/1⌉. ⌈⌉= round up.

K=2:

We will represent the multiplier with string {-2,-1,0,1,2} of length

n'= ⌈(n+1)/2⌉.

K=3:

We will represent the multiplier with string {-4,-3,-2,-1,0,1,2,3,4} of length

n'= ⌈(n+1)/3⌉.

And in the general case we will represent the multiplier b with string b' of length :

n'= ⌈(n+1)/k⌉.

When b'(i) = '0' we will just do a shift otherwise line (i) will be:
b'(i)*<a> and we will then have to sum all the rows, in order to get the multiplication result.

Arrangement of the adder tree in multiplication of unsign numbers when the multiplicand is coded in Booth-2.

The problem:
We have two inputs that we want to multiply :

$A , B \in \{0,1\}^n$

The output should be: <A>*<B>.
We will re coded B =>

B'= b'(n')b'(n'-1)....b'(0).

While :

1. b(i') $\in$ {-2,-1,0,1,2}.
2. n'=$\lceil (n+1)/2 \rceil$.
3. $< B >= \sum_{i=0}^{n'-1} b(i) * 4^i$

Declaration of the j line:

Trial 1 : $\alpha$(j')= <A>*b(j')

But $\alpha$(j') can be negative so we will add constant of $3*2^{n+1}$ to every line ,in order to make line j positive. It is easier to sum just positive lines then summing positive and negative ones.

Trial 2 : $\alpha(j'') = \alpha(j') + 3*2^{n+1}$

This assured us that line j will be positive and the addition is much simplified .
But we need to eliminate the addition of $3*2^{n+1}$ from the result. We will do it by module the result by $(2^{2n})$.

So the multiplication solution will be :

$$< A > * < B > \equiv 2^{n+1} + \sum_{j=0}^{n'-1} \alpha(j'') * 4^j \ (\mathrm{mod}(2^{2n})) .$$