Adder/Subtractor Unit :

This unit consist of :

- Carry Lookahead adder or *CLA*
- Combinational logic to take care of subtraction
- Logic for storing carry and considering this carry into the next operation if required.(memory operation)

The important part of the unit is Carry Lookahead Adder (CLA).
First we will explain why we have decided to use CLA and then we will discuss about why and how we have modified original CLA design.

Why CLA is used ?

Adders are usually implemented by combining multiple copies of simple components. The natural components of such a building block would be (3,2) full adder which is define by following equations :

$$s = ab'c' + a'bc' + a'b'c + abc$$

$$c_{out} = ab + ac + bc$$

where a and b being inputs and c is carry input to the circuit, s is sum bit and c_{out} is the carry output of the adder.
The principal problem in constructing an adder for large numbers out of this smaller components is propogating the carries from one piece to the next. In the most obvious way to solve is to connect carry output of previous block to carry input of next block. In this case, carry is ripple through stages and so the time it takes to produce an output is propotional to the maximum number of logiv levels through which a signal travels. This can be a very slow process particularly for larger numbers.

After comparing different adders like *Carry Skip Adder, Carry Select Adder, Ripple Carry Adder and Carry Lookahead* Adder in perspective of time and the space required, we have decided to use Carry Lookahead Adder (CLA) .

Carry Lookahead Adder (CLA) :

For a n-bit number the sum bit of ith stage is given by following formula :

$$s_i = a_i b_i' c_i' + a_i' b_i c_i' + a_i' b_i' c_i + a_i b_i c_i$$

where c_i is both the carry-in to the ith adder and carry-out from the (i-1)st adder.

We know values of ai and bi, but we don't know the c_i, if we can find out c_i for each stage the we can calculate sum bits for all the adders simultaneously.

If we define g_i as carry generate and p_i as carry propogate, the we can write c_i in terms of g_i and p_i.

$$c_i = g_{i-1} + p_{i-1} * c_{i-1}$$

where g_{i-1} is carry generated and p_{i-1} is carry prpogated which is given by :

$$g_{i-1} = a_{i-1} * b_{i-1}$$

$$p_{i-1} = a_{i-1} + b_{i-1}$$

So, carry into the ith stage c_i is true if and only if carry is generated in the i-1th stage i.e. g_{i-1} is true or carry into previous stage c_{i-1} is true and carry is propogated from i-1 the stage i.e. p_{i-1} is true.

From above equation of carry, we can say that the formula for carry is recursive and we can put c_i in terms of p_0 and g_0.

$$c_i = g_{i-1}+p_{i-1}*g_{i-2}+.........+p_{i-1}*p_{i-2}..*p_1*g_0+p_{i-1}*g_{i-2}*p_1*p_0*c_0$$

We can see that we can calculate carries for each stage in parallel and so the sum bits. This will be a vast improvement on speed. This is the reason why we decided to use Carry Lookahead Adder.

Modifications to original CLA :

We decided to modify the basic CLA due to following problems.

- A CLA of n bits requires a fan-in of n+1 at the OR gate as well as at the rightmost AND gate.
- n-1 signal must drive n AND gates.
- Irregular structure and long wired can create the problem in building the circuit.

Here the main idea is to build up p's and g's in steps. We know that

$$c_1 = g_0 + c_0 * p_0$$

This says there is a carry-out of the 0th position (c_1) if there is either a carry generated in the 0th position, or if there is a carry into the 0th postion and carry propogates. Similiarly,

$$c_2 = G_{01} + P_{01} * c_0$$

G_{01} means there is a carry generated out of the block consisting of the first two bits. P_{01} indicates that a carry propogates through this block. P and G have following logic equations.

$$G_{01} = g_1 + p_1 * g_0$$

$$P_{01} = p_1 * p_0$$

More generally, for any j with i < j , j+1 < k, we have the recursive relations,

$$c_{k+1} = G_{ik} + P_{ik} * c_i$$

$$G_{ik} = G_{j+1,k} + P_{j+1,k} * G_{ij}$$

$$P_{ik} = P_{ij} * P_{j+1,k}$$

With these preliminaries, design of a practical CLA can be shown as below. The adder consist of two parts. The first part computes various values of P and G from p's and g's. The second part uses P and G values to compute all carries as the above equation shows.

The first part of the design is shown in fig. 1. At the top of the diagram, input numbers $a_7....a_0$ and $b_7....b_0$ are converted to p's and g's using cells of the same type. Then various P's and G's are generated by combining cells in a binary-tree structure.

The second part of the design is shown in fig. 2. By feeding c_0 at the bottom of this tree, all the carry bits come out at top. Each cell must know a pair of P,G values in order to do conversion, and the value it needs is written inside the cell. Comapring two figures, we can see the one-to-one corrospondence between cells, and the value of (P, G) needed by the carry-generating cells is eactly the value known by the corrosponding (P, G) generating cells.

The combined cell is shown in fig.3 The numbers to be added flow into the top and downward through the tree, combining with c_0 at the bottom and flowing back up the tree to form the carries.

So, this shows that the small improvement in size pays off in a drametic improvement in speed.

Here the height of the tree is $\log_2 n$ levels (n being number of input bits) as we are allowing fan-in and fan-out of two.

For our application fan-in and fan-out of 4 bits can be allowed which will reduce the height of tree to $\log_4 n$ i.e. it will saves lots of space. So, we have further modified our circuit to take advantage of this feature. For that, we need to develope new equations for each stage.

By using the same recursive property we derived new equations as under.

$c_{k+j} = G_{j+1,k} + P_{j+1,k} * G_{ij} + P_{j+1,k} * P_{ij} * c_i$

$G_{ik} = G_{j+1,k} + P_{j+1,k} * G_{l+1,j} + P_{j+1,k} * P_{l+1,j} * G_{il}$

$P_{ik} = P_{il} * P_{l+1,j} * P_{j+1,k}$

This reduces the number of blocks to be used, which is considerable saving in space.

Subtraction Unit :

By making small changes as explained under, we can incorporate subtraction function in the same circuit.

Input B is fed to two input ex-or gate before going to the A block above. The other input to ex-or gate is mode signal, during subtraction operation is one and for addition it will be zero.

So, when the mode signal is one, B input is inverted and fed to the adder unit, but when mode is zero, the B input stays same.

We also need to provide c_0 input as one for subtraction operation. So, the idea is when subtracting, we are adding two's complement of B which will give us the correct result.

We also modified the carry input to the unit to take into account of carry of borrow generated by previous add or sub operation and next operation is add or sub from the memory contents.

Whenever there is addition from memory and there is carry out of the previous result, we should add that carry in this operation. Also, if we have borrow in the previous operation, for the next subtraction, we should subtract one from the addition.

We have derived the formula for carry input to the adder unit by considering the above mentioned facts.

$$\text{carry-in} = c_{out} * \text{opcode}(2) + c_{out} * \text{opcode}(1) + \text{opcode}(1) * \text{opcdoe}(2)$$

where opcode(1) and opcode(2) being bits from our opcode. The gate-level diagram is shown in the fig.