High-Speed VLSI Arithmetic Units: Adders and Multipliers

by

Prof. Vojin G. Oklobdzija

Fall 1999

Introduction

Digital computer arithmetic is an aspect of logic design with the objective of developing appropriate algorithms in order to achieve an efficient utilization of the available hardware [1-4]. Given that the hardware can only perform a relatively simple and primitive set of Boolean operations, arithmetic operations are based on a hierarchy of operations that are built upon the simple ones. Since ultimately, speed, power and chip area are the most often used measures of the efficiency of an algorithm, there is a strong link between the algorithms and technology used for its implementation.

High-speed Addition: Algorithms and VLSI Implementation:

First we will examine a realization of a one-bit adder which represents a basic building block for all the more elaborate addition schemes.

Full Adder:

Operation of a Full Adder is defined by the Boolean equations for the sum and carry signals:

$$s_i = a_i\bar{b}_i\bar{c}_i + \bar{a}_i b_i\bar{c}_i + \bar{a}_i\bar{b}_i c_i + a_i b_i c_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i + a_i b_i\bar{c}_i + a_i b_i c_i$$

Where: a_i, b_i, and c_i are the inputs to the *i-th* full adder stage, and s_i and c_{i+1} are the sum and carry outputs from the *i-th* stage, respectively.

From the above equation we realize that the realization of the Sum function requires two XOR logic gates.

The Carry function is further rewritten defining the Carry-Propagate p_i and Carry-Generate g_i terms:

$$p_i = a_i \oplus b_i \quad , \quad g_i = a_i \bullet b_i$$

At a given stage *i*, a carry is generated if g_i is true (i.e., both a_i and b_i are ONEs), and if p_i is true, a stage propagates an input carry to its output (i.e., either a_i or b_i is a ONE). The logical implementation of the full adder is shown in Fig. 1.a.

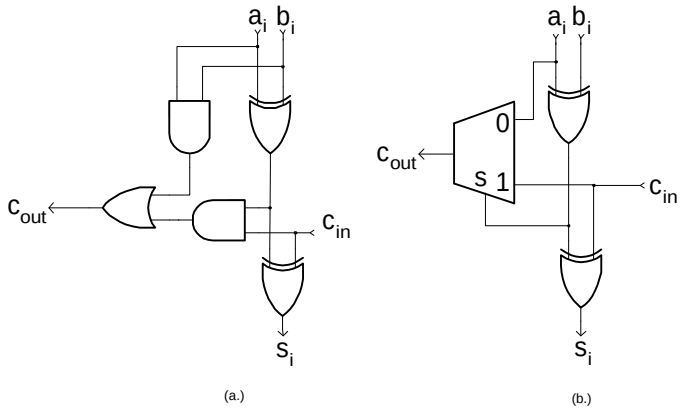

Fig. 1.a.b. Full-Adder implementation (a) regular (b) using multiplexer in the critical path

For this implementation, the delay from either a or b_i to s_i is two XOR delays and the delay from c_i to c_{i+1} is 2 gate delays. Some technologies, such as CMOS, implement the functions more efficiently by using pass-transistor circuits. For example, the critical path of the carry-in to *carry-out* uses a fast pass-transistor multiplexer [8] in an alternative implementation of the Full Adder shown in Fig.1.b.

The ability of pass-transistor logic to provide an efficient multiplexer implementation has been exploited in CPL and DPL logic families [10,11]. Even an XOR gate is more efficiently implemented using multiplexer topology. A Full-Adder cell which is entirely multiplexer based as published by Hitachi [11] is shown in Fig.2. Such a Full-Adder realization contains only two transistors in the *Input-to-Sum* path and only one transistor in the *Cin-to-Cout* path (not counting the buffer). The short critical path is a factor that contributes to a remarkable speed of this implementation.

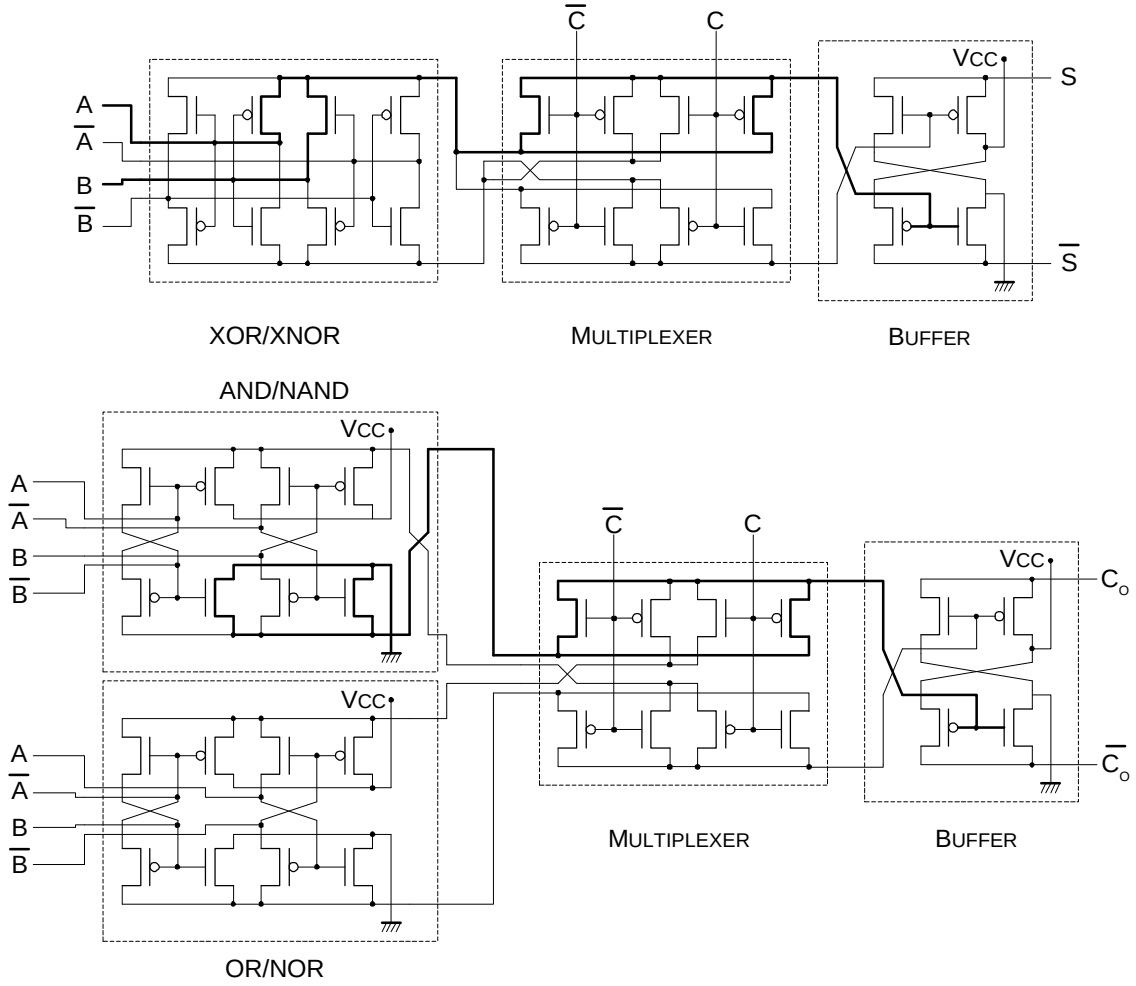

Fig.2. Pass-Transistor realization of a Full-Adder in DPL [11]

Ripple Carry Adder:

A ripple carry adder for N-bit numbers is implemented by concatenating N full adders as shown on Figure 3. At the *i-th* bit position, the *i-th* bits of operands A and B and a carry signal from the preceding adder stage are used to generate the *i-th* bit of the sum, s_i, and a carry, c_{i+1}, to the next adder stage. This is called a Ripple Carry Adder (RCA), since the carry signal "ripple" from the least significant bit position to the most significant [3-4]. If the ripple carry adder is implemented by concatenating N full adders, the delay of such an adder is 2N gate delays from C_{in}-*to*-C_{out}.

The path from the input to the output signal that is likely to take the longest time is designated as a *"critical path"*. In the case of a RCA, this is the path from the least significant input a_0 or b_0 to the last sum bit s_n. Assuming a multiplexer based XOR gate implementation, this critical path will consist of N+1 pass transistor delays. However, such a long chain of transistors will significantly degrade the signal, thus some amplification points are necessary. In practice, we can use a multiplexer cell to build this critical path using standard cell library as shown in Fig.3 [8].

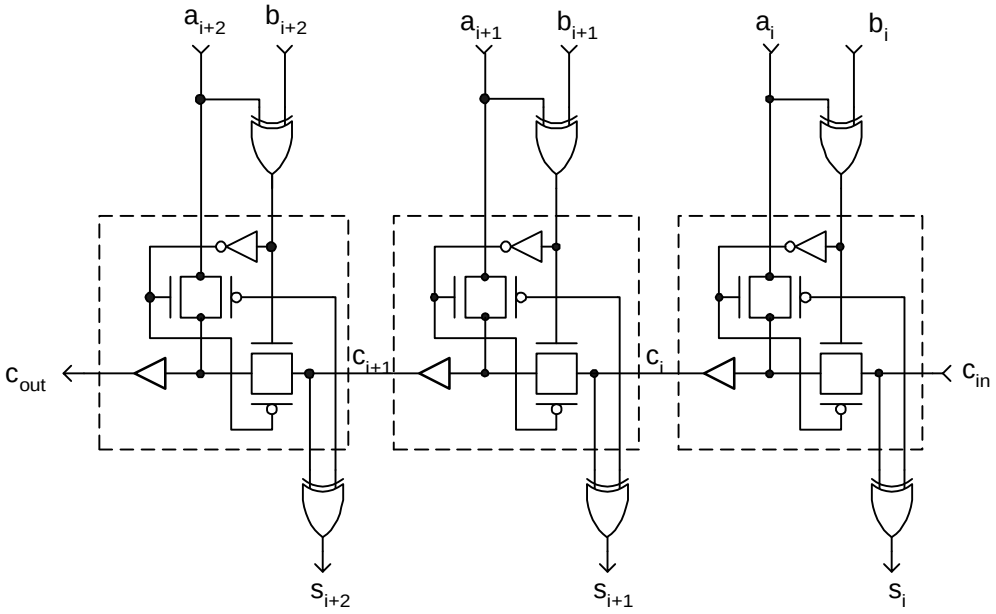

Fig. 3. Carry-Chain of an RCA implemented using multiplexer from the standard cell library [8]

Carry Skip Adder:

Since the C_{in}-*to*-C_{out} represents the longest path in the ripple-carry-adder an obvious attempt is to accelerate carry propagation through the adder. This is accomplished by using Carry-Propagate *pi* signals within a group of bits. If all the p_i signals within the group are $p_i = 1$, the condition exist for the carry to bypass the entire group:

$$P = p_i \bullet p_{i+1} \bullet p_{i+2} \bullet \ldots\ldots \bullet p_{i+k}$$

The Carry Skip Adder (CSKA) divides the words to be added into groups of equal size of k-bits. The basic structure of an N-bit Carry Skip Adder is shown on Fig. 4. Within the group, carry propagates in a ripple-carry fashion. In addition, an AND gate is used to form the group

propagate signal *P*. If *P = 1* the condition exists for carry to bypass (skip) over the group as shown in Fig.4.

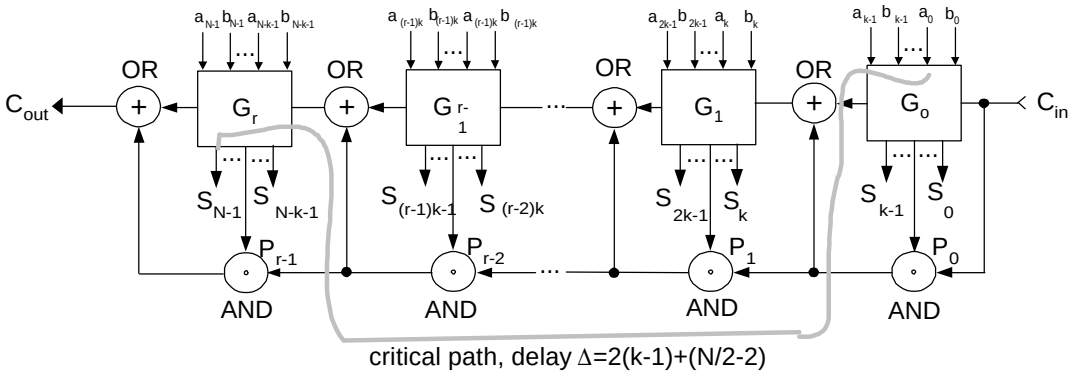

Fig.4. Basic Structure of a CSA: N-bits, k-bits/group, r=N/k groups

The maximal delay of a Carry Skip Adder is encountered when carry signal is generated in the least-significant bit position, rippling through *k-1* bit positions, skipping over *N/k-2* groups in the middle, rippling to the *k-1* bits of most significant group and being assimilated in the Nth bit position to produce the sum S_N:

$$\Delta_{CSA} = (k-1)\Delta_{rca} + (\frac{N}{2} - 2)\Delta_{SKIP} + (k-1)\Delta_{rca} = 2(k-1)\Delta_{rca} + (\frac{N}{2} - 2)\Delta_{SKIP}$$

Thus, CSKA is faster than RCA at the expense of a few relatively simple modifications. The delay is still linearly dependent on the size of the adder N, however this linear dependence is reduced by a factor of *1/k* [3].

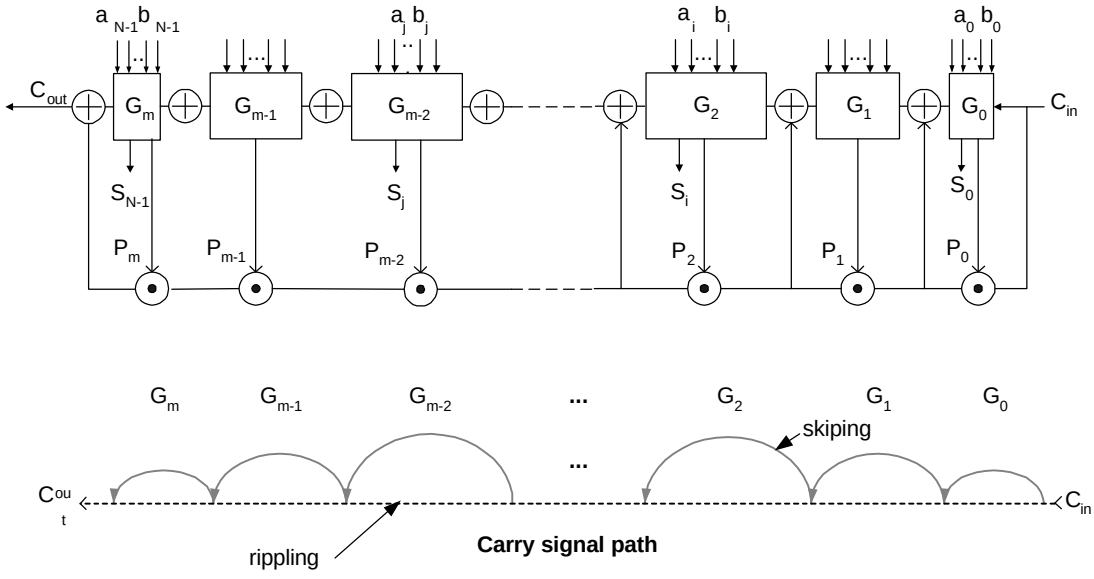

Fig.5. Carry-chain of a 32-bit Variable Block Adder

Variable Block Adder:

The idea behind Variable Block Adder (VBA) is to minimize the longest critical path in the carry chain of a CSKA, while allowing the groups to take different sizes [2,7]. Such an optimization in general does not result in an enhanced complexity as compared to the CSKA. A carry-chain of a 32-bit VBA is shown in Fig.5.

The first and last blocks are smaller, and the intermediate blocks are larger. That compensates for the critical paths originating from the ends by shortening the length of the path used for the carry signal to ripple in the end groups, allowing carry to skip over larger groups in the middle.

There are two important consequences of this optimization:

(a.) First, the total delay is reduced as compared to CSKA

(b.) Second, the delay dependency is not a linear function of the adder size *N* as in CSKA. This dependency follows a square root function of *N* instead.

For an optimized VBA, it is possible to obtain a close form solution expressing this delay dependency which is given as: $\Delta_{VBA} = c_1 + \sqrt{c_2 N + c_3}$ where: c_1, c_2, c_3 are constants.

It is also possible to extend this approach to multiple levels of carry skip as done in [7]. A determination of the optimal sizes of the blocks on the first and higher levels of skip blocks is a linear programming problem which does not yield a close form solution. Such types of problems are solved with the use of dynamic programming techniques. The speed of such a multiple-level VBA adder surpasses single-level VBA and that of fixed group Carry-Lookahead Adder (CLA). [15]. There are two reasons why this is possible:

(1.) First, the speed of the logic gates used for CMOS implementation depends on the output load: *fan-out*, as well as the number of inputs: *fan-in*. CLA implementation is characterized with a large *fan-in* which limits the available size of the groups. On the other hand VBA implementation is simple. Thus, it seems that CLA has passed the point of diminishing returns as far as an efficient implementation is concerned. This example also points to the importance of modeling and incorporating appropriate technology parameters into the algorithm. Most of the computer arithmetic algorithms developed in the past use a simple *constant gate delay* model.

(2.) Second, a fixed-group CLA is not the best way to build an adder. It is a sub-optimal structure which after being optimized for speed, consists of groups that are different in size yielding a largely irregular structure [15].

There are other advantages of VBA adder that are direct result of its simplicity and efficient optimization of the critical path. Those advantages are exhibited in the lower area and power consumption while retaining its speed. Thus, VBA has the lowest energy-delay product as compared to the other adders in its class. [9].

Carry Lookahead Adder:

A significant speed improvement in the implementation of a parallel adder was introduced by a Carry-Lookahead-Adder (CLA) developed by Weinberger and Smith in 1958 [13]. The CLA adder is theoretically one of the fastest schemes used for the addition of two numbers, since the delay to add two numbers depends on the logarithm of the size of the operands: $\Delta \approx \log\lceil N \rceil$

The Carry Loookahead Adder uses modified full adders (modified in the sense that a carry output is not formed) for each bit position and Lookahead modules which are used to generate carry signals independently for a group of *k*-bits. In most common case *k*=4. In addition to carry signal for the group, Lookahead modules produce group carry generate (G) and group carry propagate (P) outputs that indicate that a carry is generated within the group, or that an incoming carry would propagate across the group.
Extending the carry equation to a second stage in a Ripple-Carry-Adder we obtain:

$$\begin{aligned} c_{i+2} &= g_{i+1} + p_{i+1}c_{i+1} \\ &= g_{i+1} + p_{i+1}(g_i + p_i c_1) \\ &= g_{i+1} + p_{i+1}g_i + p_{i+1}p_i c_1 \end{aligned}$$

This carry equation results from evaluating the carry equation for the *i+1*-st stage and substituting c_{i+1}. Carry c_{i+2} exits from stage *i+1* if:

(a.) a carry is generated in the stage *i+1* or

(b.) a carry is generated in stage *i* and propagates across stage *i+1* or

(c.) a carry enters stage *i* and propagates across both stages *i* and *i+1*, etc.

Extending the carry equation to a third stage yields:

$$\begin{aligned} c_{i+3} &= g_{i+2} + p_{i+2}c_{i+2} \\ &= g_{i+2} + p_{i+2}(g_{i+1} + p_{i+1}g_i + p_{i+1}p_i c_i) \\ &= g_{i+2} + p_{i+2}g_{i+1} + p_{i+2}p_{i+1}g_i + p_{i+2}p_{i+1}p_i c_i \end{aligned}$$

Although it would be possible to continue this process indefinitely, each additional stage increases the size (i.e., the number of inputs) of the logic gates. Four inputs (as required to implement c_{i+3} equation) is frequently the maximum number of inputs per gate for current technologies. To continue the process, Carry-Lookahead utilizes group generate and group propagate signals over four bit groups (stages *i* to *i+3*), G_i and P_i, respectively:

$$G_i = g_{i+3} + p_{i+3}g_{i+2} + p_{i+3}p_{i+2}g_{i+1} + p_{i+3}p_{i+2}p_{i+1}g_i$$

and:

$$P_i = p_{i+3}p_{i+2}p_{i+1}p_i$$

The carry equation can be expressed in terms of the four bit group generate and propagate signals:

$$c_{i+1} = G_i + P_i c_i$$

Thus, the carry out from a 4-bit wide group c_{i+4} can be computed in four gate delays: one gate delay to compute p_i and g_i for $i = i$ through $i+3$, a second gate delay to evaluate P_i, the second and the third to evaluate G_i, and the third and fourth to calculate carry signals c_{i+1}, c_{i+2} , c_{i+3} and c_{i+4}. Actually, if not limited by fan-in constraints, c_{i+4} could be calculated concurrently with G_i and will be available after three gate delays.

In general, an k bit lookahead group requires $0.5(3k+k^2)$ logic gates, where k is the size of the group. In a recursive fashion, we can create a *"group of groups"* or a *"super-group"*. The inputs to the *"super-group"* are G and P signals from the previous level. The *"super-group"* produces P^* and G^* signals indicating that the carry signal will be propagated across all of the groups within the *"super-group"* domain, or that the carry will be generated in one of the groups encompassed by the *"super-group"*. Similarly to the group, a *"super-group"* produces a carry signal out of the *"super-group"* as well as an input carry signal for each of the groups in the level above:

$$G^*_j = G_{i+3} + P_{i+3}G_{i+2} + P_{i+3}P_{i+2}G_{i+1} + P_{i+3}P_{i+2}P_{i+1}G_i$$
$$P^*_j = P_{i+3}P_{i+2}P_{i+1}P_i$$
$$c_j = G_j + P_j c_i$$

A construction of a 32-bit Carry Lookahead Adder is illustrated in Fig. 6.

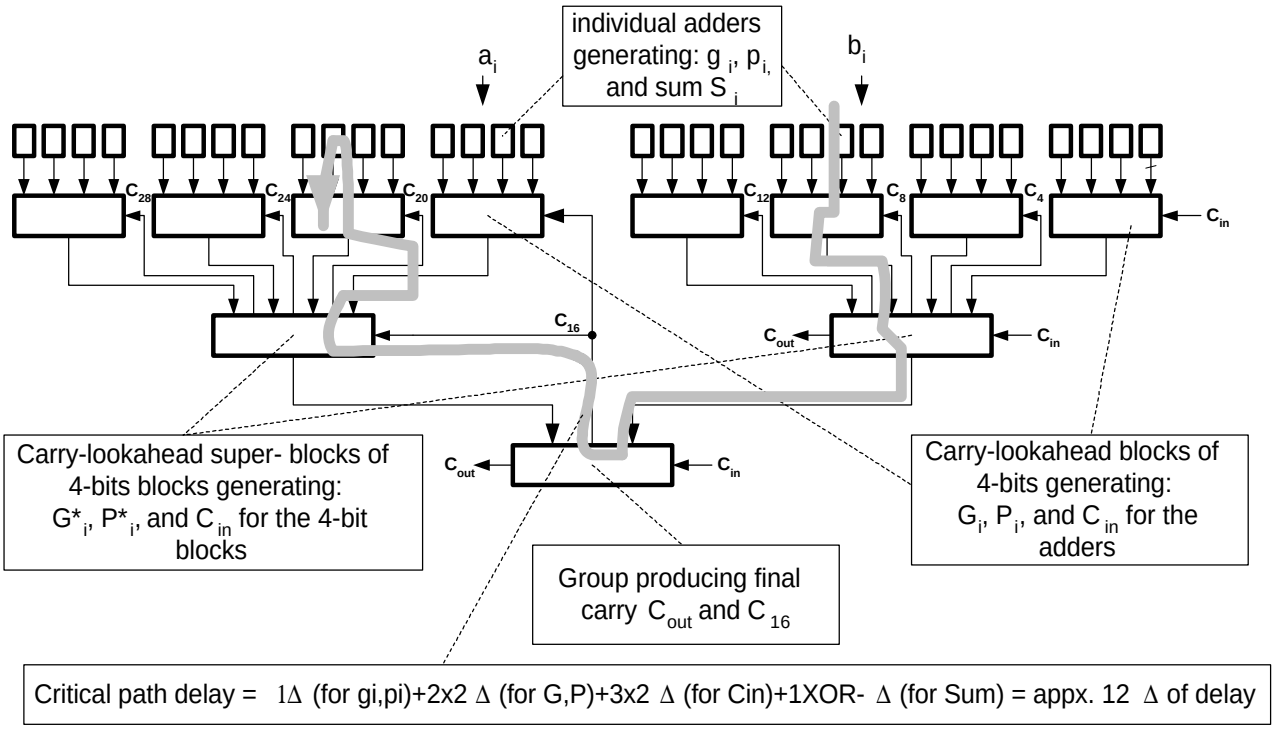

Fig. 6. 32-bit Carry Lookahead Adder

As opposed to RCA or CSA the critical path in the CLA travels in vertical direction rather than a horizontal one as shown in Fig.6. Therefore the delay of CLA is not directly proportional to the size of the adder N, but to the number of levels used. Given that the groups and super-groups in the CLA resemble a tree structure the delay of a CLA is thus proportional to the *log* function of the size N.

CLA delay is evaluated by recognizing that an adder with a single level of carry lookahead (four bit words) contains three gate delays in the carry path. Each additional level of lookahead increases the maximum word size by a factor of *k* and adds two additional gate delays. Generally the number of lookahead levels for an N-bit adder is $\lceil log_k N \rceil$ where *k+1* is the maximum number of inputs per gate. Since an *k*-bit group carry-lookahead adder introduces three gate delays per CLA level, and there are two additional gate delays: one for g_i and p_i, and other for the final sum s_i, CLA delay Δ is:

$$\Delta_{CLA} = 1 + 2(\log\lceil N \rceil - 1) + 1 = 4\log\lceil N \rceil$$

This *log* dependency makes CLA one of the theoretically fastest structures for addition [2-4]. However, it can be argued that the speed efficiency of the CLA has passed the point of diminishing returns given the *fan-in* and *fan-out* dependencies of the logic gates and inadequacy of the delay model based on counting number of gates in the critical path. In reality, CLA is indeed achieving lesser speed than expected, especially when compared to some techniques that consume less hardware for the implementation as shown in [7,8].

One of the simple schemes for addition that was very popular at the time when transition into MOS technology was made, is *Manchester Carry Chain* (MCC) [6,38]. MCC is an alternative switch based technique implemented using pass-transistor logic. The speed realized using MCC is impressive which is due to its simplicity and the properties of the pass-transistor logic. MCC does not require a large area for its implementation, consuming substantially less power as compared to CLA or other more elaborate schemes. A realization of the MCC is shown in Fig. 7. Due to the RC delay properties of the MCC the signal needs to be regenerated by inserting inverters at appropriately chosen locations in the carry chain.

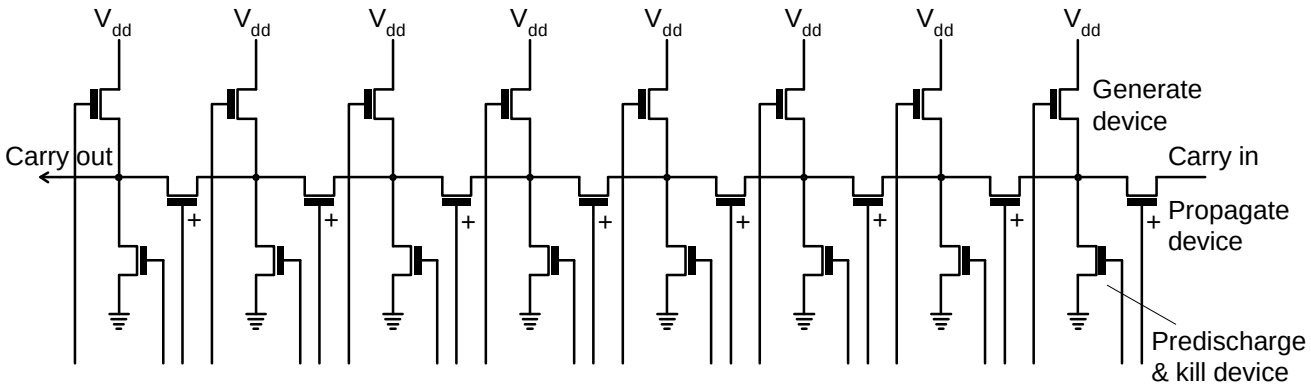

Fig. 7. Manchester Carry-Chain realization of the carry path.

In the same way a CLA can be built using MCC for the implementation of the lookahead group. Further, pass-transistor MCC structure can be incorporated in the logic group of the circuit technology used for CLA realization. One such an example is shown in Fig. 8.a. representing a four bit group of an 64-bit CLA built using CMOS Domino logic [14]. Each CLA group is implemented as a separate CMOS Domino function. This adder built by Motorola using 1.0u CMOS technology achieved a remarkable speed of 4.5nS at V_{DD}=5V and 25^{o}C. The critical path of this design is shown in Fig. 8.b. Using selection technique and careful analysis of the critical path the same adder was extended to 96-bits at the same speed of 4.5nS.
As with RCA, the carry lookahead adder complexity grows linearly with the word size (for k = 4, this occurs at a 40% faster rate than the RCA).

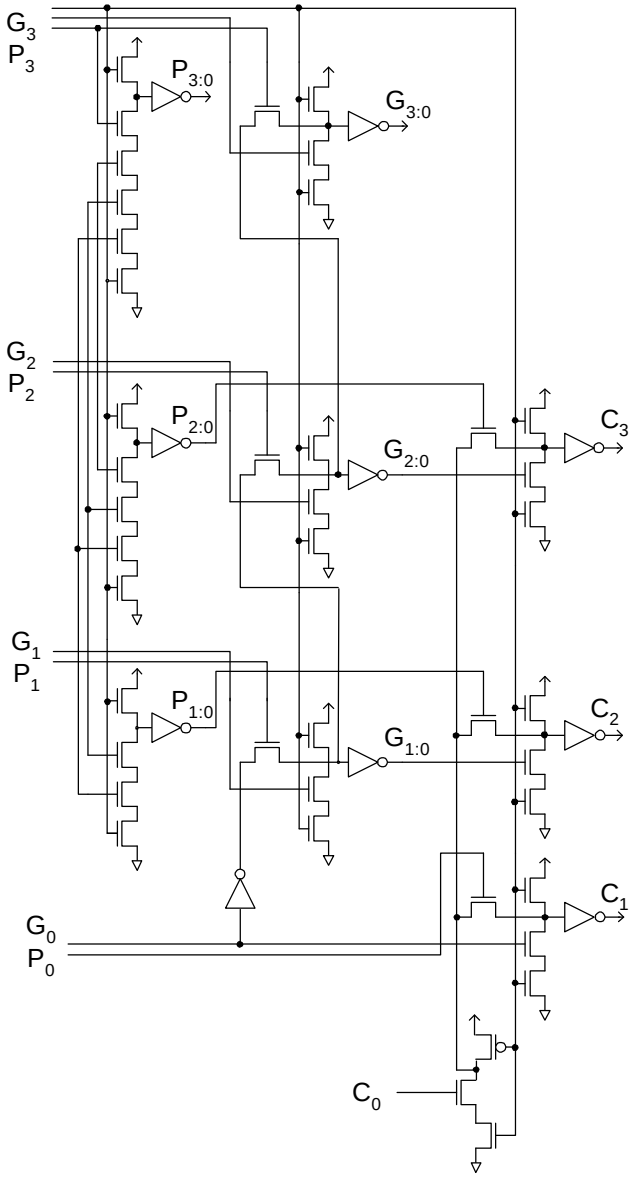

Fig. 8.a. CMOS Domino realization of a 64-bit CLA

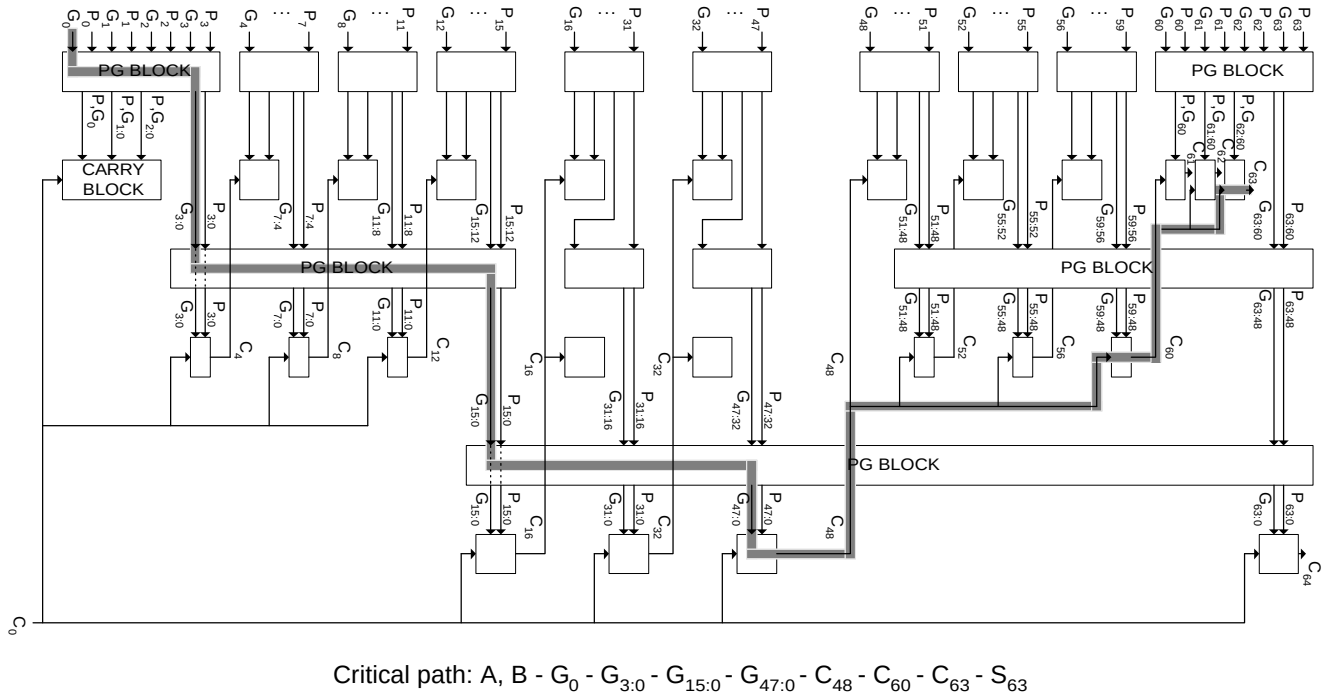

Fig. 8.b. Critical path in Motorola's 64-bit CLA [14]

Recurrence Solver Based Adders:

The class of adders known as based on solving recurrence equations was first introduced by Biolgory and Gajski and Brent and Kung [17],[18] based on the previous work by Koggie and Stone [16].

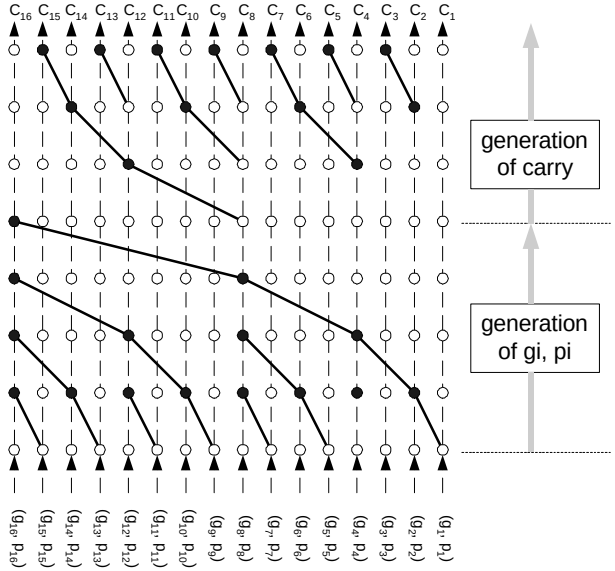

Fig. 9. Recurrence Solver adder topology

They realized that if $C_{in}=0$ can be assumed, the carry-lookahead equations can be written in a simple form of a recurrence:

$$(g, p) \bullet (g, p) = (g + pg', pp')$$

where an operator $\bullet$ termed "black" operator was introduced. By application of this recurrence equation various topologies of an adder can be obtained with several desirable properties:

a. a good layout
b. the fan-out can be controlled and limited to no more than 2
c. trade-offs between *fan-in*, *fan-out* and hierarchical layout topologies can be achieved.

The above reasons were the cause for a relative popularity of the *"recurrence-solver"* schemes. In essence, "*recurrence solver*" based adders are nothing else but a variation of many possible different CLA topologies [2]. An example of a "*recurrence solver*" adder is shown in Fig. 9.

Ling Adder:

Ling adder is a scheme developed at IBM to take advantage of the ability of the ECL technology to perform wired-OR operation with a minimal additional delay [19]. Ling redefined the equations for Sum and Carry by encoding pairs of digit positions: (a_i,b_i, a_{i-1}, b_{i-1}). To understand the advantage of Ling Adder, we will consider the generation of C_3 carry-out bit using conventional CLA and using modified Ling equations. Without the wired-OR function, C_3 can be implemented in three gate delays. The expansion of those equation will yield 15 terms and a maximum fan-in of 5. Ling equations on the other hand will perform the same evaluation (of Ling's modified carry H_3) using 8 terms with the maximal fan-in of 4. Thus, in a particular IBM's ECL technology (for which this adder was developed) with the limitation of fan-in of 4 for the wired-OR term, Ling's adder yields substantial advantage.

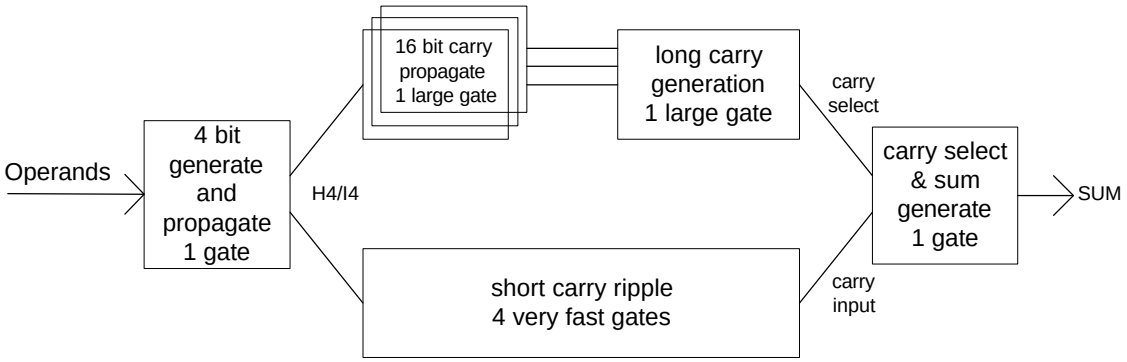

Fig. 10. a. Organization of a 64-bit Ling adder realized in CMOS technology [20]

Ling adder can realize a sum delay in:

$$\Delta = \left\lceil \log_r \frac{N}{2} \right\rceil + 1$$

The Ling adder was also found to be adequate for realizations using CMOS technology. The advantage of high-gain and fan-in capabilities of dynamic CMOS combined with the dual rail DCVS logic were used in Hewlett-Packard's sub-nanosecond adder which was design in 0.5u

CMOS technology [20]. The organization of this adder is shown in Fig.10.a. while the circuits used for generation of H4 and I4 terms are shown in Fig.10.b.

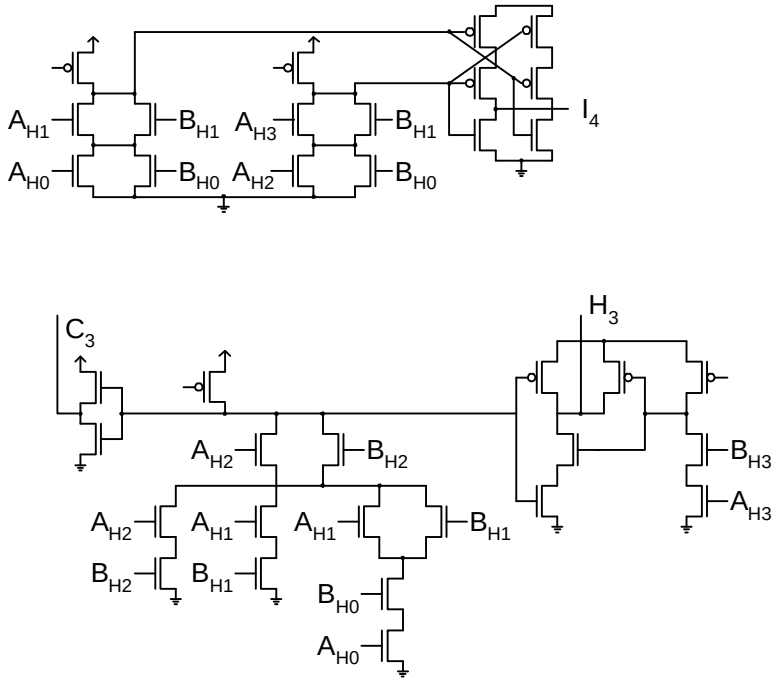

Fig. 10. b. Circuit used for generation of H4 and I4 terms [20]

Conditional-Sum Addition:

The theoretically fastest scheme for addition of two numbers is *"Conditional-Sum Addition"* (CNSA) proposed by Sklansky in 1960 [3,5,21]. The essence of the CNSA scheme is in the realization that we can add two numbers without waiting for the carry signal to be available. Simply, the numbers are added in two instances: one assuming $C_{in} = 0$ and the other assuming $C_{in} = 1$. The conditionally produced results: Sum_0, Sum_1 and $Carry_0$, $Carry_1$ are selected by a multiplexer using an incoming carry signal C_{in} as a multiplexer control. Similarly to the CLA the input bits are divided into groups which are in case of CSNA added "conditionally".

It is apparent that while building CSNA the hardware complexity starts to grow rapidly starting from the Least Significant Bit (LSB) position. Therefore, in practice, the full-blown implementation of the CNSA is not found.

However, the idea of adding the Most Significant (MS) portion of the operands conditionally and selecting the results once the carry-in signal is computed in the Least Significant (LS) portion, is attractive. Such a scheme (which is a subset of CNSA) is known as *"Carry-Select Adder"* (CSLA) [22].

Carry Select Adder:

The Carry Select Adder (CSLA) divides the words to be added into blocks and forms two sums for each block in parallel (one with a carry in of ZERO and the other with a carry in of ONE)

[2,3,5,22]. As shown in an example of a 16 bit carry select adder in Fig. 11, the carry-out from the previous LS 4-bit block controls a multiplexer that selects the appropriate sum from the MS portion. The carry out is computed using the equation for the carry out of the group, since the group propagate signal P_i is the carry out of an adder with a carry input of ONE and the group generate G_i signal is the carry out of an adder with a carry input of ZERO. This speeds-up the computation of the carry signal necessary for selection in the next block. The upper 8-bits are computed conditionally using two CSLAs similar to the one used in the LS 8-bit portion. The delay of this adder is determined by the speed of the LS k-bit block (4-bit RCA in the example, Fig. 11) and delay of multiplexers in the MS path. Generally this delay is:

$$\Delta = \mathrm{d}_{MUX}\lceil \log_k N \rceil + \mathrm{d}_{k-bit-adder}$$

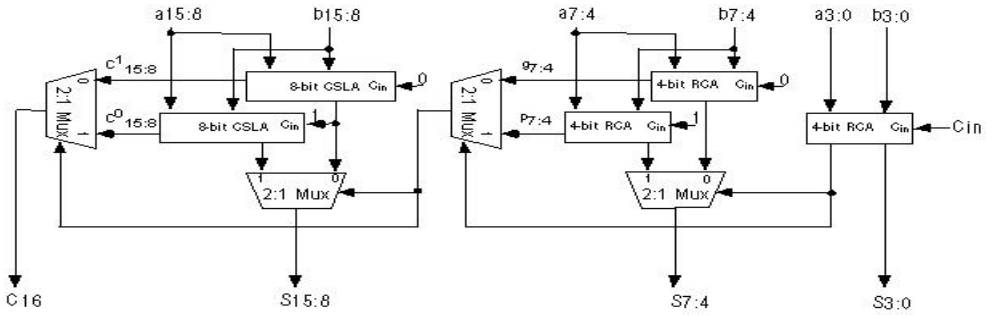

Fig. 11: 16-bit Carry-Select Adder

DEC "Alpha" 21064 Adder:

The 64-bit adder used in the first 200MHz Digital's WD21064 RISC microprocessor employed a combination of techniques in order to reach 5nS cycle required from the 0.75u CMOS technology of implementation [23]. There were four different techniques used on the various levels of this 64-bit adder:

a. On the 8-bit level Manchester Carry Chain technique was used. MCC seems to be the most effective for the short adders, especially when the word length is below 16-bits. The carry chain was further optimized by tapering down each chain stage in order to reduce the load caused by the remainder of the chain. The chain was pre-discharged at the beginning of the operation and three signals were used: Propagate P, Generate G and Carry-Kill (assimilate) K. The local carry signals were amplified using ratioed inveters. There were two MCC employed: one that assumes *Cin = 0* and other that assumes *Cin = 1*.
b. Carry-Lookahead Addition (CLA) was used on the least significant 32-bits of the adder. The CLA section was implemented as a distributed differential circuit producing the carry signal that controls the most-significant 32-bit portion of the adder.

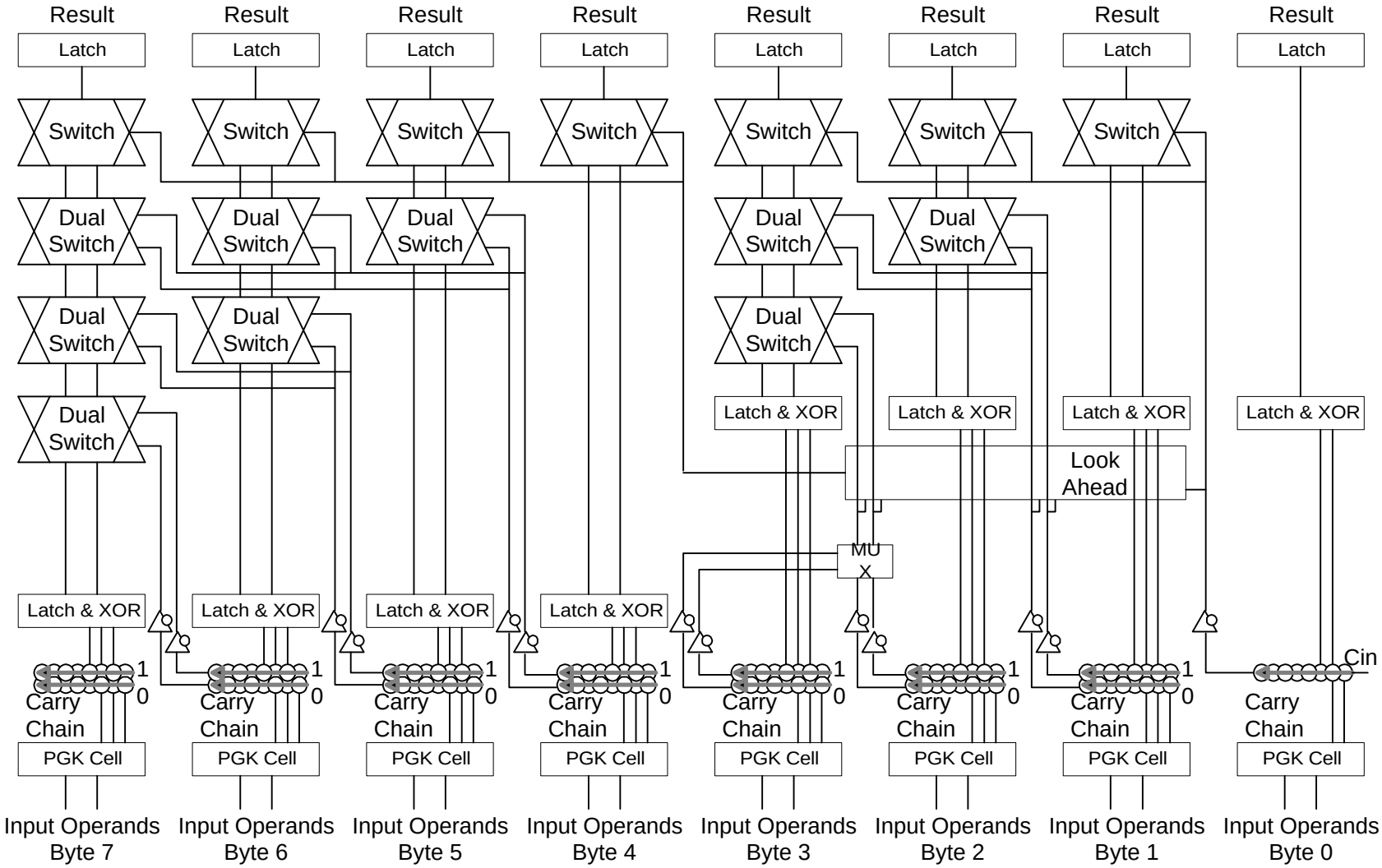

Fig. 12. Block diagram of DEC *"Alpha"* 64-bit adder [23]

c. Conditional Sum Addition (CNSA) was used for the most-significant 32-bits of the adder. There were six 8-bit select switches used to implement conditional summation on the 8-bit level.
d. Finally, Carry Select (CSLA) method was used in order to produce the most-significant 32-bits of the 64-bit word. The selection of the final result was done using nMOS byte carry-select switches.

The block diagram of DEC "Alpha's" adder is shown in Fig. 12 [23].

Multiplication

Algorithm:

In microprocessors multiplication operation is performed in a variety of forms in hardware and software depending on the cost and transistor budget allocated for this particular operation. In the beginning stages of computer development any complex operation was usually programmed in software or coded in the micro-code of the machine. Some limited hardware assistance was provided. Today it is more likely to find full hardware implementation of the multiplication in order to satisfy growing demand for speed and due to the decreasing cost of hardware [2-5]. For simplicity, we will describe a basic multiplication algorithm which operates on positive n-bit long integers X and Y resulting in the product P which is *2n* bit long:

$$P = XY = X \times \sum_{i=0}^{n-1} y^i r^i = \sum_{i=0}^{n-1} X \times y^i r^i$$

This expression indicates that the multiplication process is performed by summing *n* terms of a *partial product* P_i. This product indicates that the *i-th* term P_i is obtained by simple arithmetic left shift of X for the *i* positions and multiplication by the single digit y_i. For the binary radix (*r=2*), y_i is 0 or 1 and multiplication by the digit y_i is very simple to perform. The addition of *n* terms can be performed at once, by passing the *partial products* through a network of adders or sequentially, by adding *partial products* using an adder *n* times. The algorithm to perform the multiplication of X and Y can be described as [5]:

$$p^{(0)} = 0$$

$$p^{j+1} = \frac{1}{r}(p^j + r^n X y_j) \qquad \text{for } j=0,.....n-1$$

It can be easily proved that this recurrence results in $p^{(n)}=XY$.

High-Performance Multipliers

The speed of multiply operation is of great importance in digital signal processing as well as in the general purpose processors today, especially since the media processing took off. In the past multiplication was generally implemented via a sequence of addition, subtraction, and shift operations.

Parallel Multipliers:

An alternative approach to sequential multiplication involves the combinational generation of all bit products and their summation with an array of full adders. This approach uses an *n* by *n* array

of AND gates to form the bit products, an array of *n x n* adders (and half adders) to sum the n^2 bit products in a carry-save fashion. Finally a *2n* Carry-Propagate Adder (CPA) is used in the final step to finish the summation and produce the result [2-5].

Wallace/Dadda Multiplier:
In his historic paper C. S. Wallace introduced a way of summing the partial product bits in parallel using a tree of Carry Save Adders which became generally known as the *"Wallace Tree"* [2,25]. This method was further refined by Dadda [26].
With Wallace method, a three step process is used to multiply two numbers:

(1) the bit products are formed
(2) the bit product matrix is *"reduced"* to a two row matrix by using a carry-save adders (known as *Wallace Tree*)
(3) the remaining two rows are summed using a fast carry-propagate adder to produce the product.

Although this may seem to be a complex process, it yields multipliers with delay proportional to the logarithm of the operand size *n*.

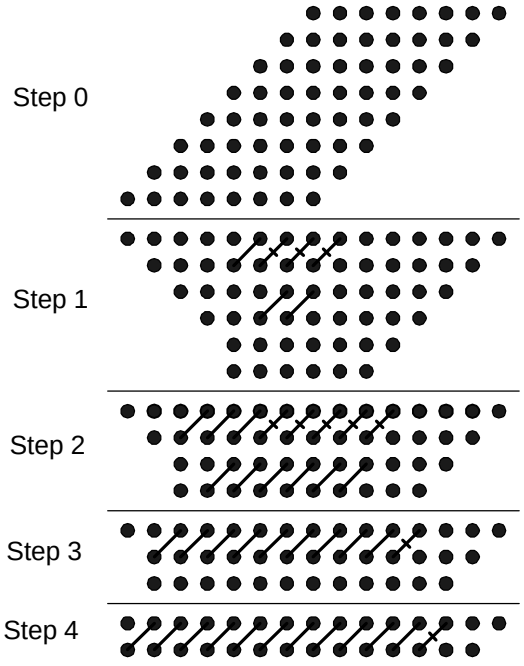

Fig. 13. 8 by 8 Dadda multiplier example [26]

A suggestion for improved efficiency of addition of the partial was published by Dadda [26]. In his historic 1965 paper, Dadda introduces a notion of a *counter* structure which will take a number of bits *p* in the same bit position (of the same *"weight"*) and output a number *q* which represent the count of ones at the input. Dadda has introduced a number of ways to compress the partial product bits using such a counter, which later became known as *"Dadda's counter"*.
This process is shown for an 8 by 8 *Dadda multiplier* in Fig. 13 [2]. An input 8 by 8 matrix of dots (each dot represents a bit product) is shown as Matrix 0. Columns having more than six dots (or that will grow to more than six dots due to carries) are reduced by the use of half adders

(each half adder takes in two dots and outputs one in the same column and one in the next more significant column) and full adders (each full adder takes in three dots and outputs one in the same column and one in the next more significant column) so that no column in Matrix 1 will have more than six dots. Half adders are shown by a "crossed" line in the succeeding matrix and full adders are shown by a line in the succeeding matrix. In each case the right most dot of the pair that are connected by a line is in the column from which the inputs were taken for the adder. In the succeeding steps reduction to Matrix 2 with no more than four dots per column, Matrix 3 with no more than three dots per column, and finally Matrix 4 with no more than two dots per column is performed. The height of the matrices is determined by working back from the final (two row) matrix and limiting the height of each matrix to the largest integer that is no more than 1.5 times the height of its successor. Each matrix is produced from its predecessor in one adder delay. Since the number of matrices is logarithmically related to the number of bits in the words to be multiplied, the delay of the matrix reduction process is proportional to *log(n)*. Since the adder that reduces the final two row matrix can be implemented as a carry lookahead adder (which also has logarithmic delay), the total delay for this multiplier is proportional to the logarithm of the word size [2,4].

An extensive study of the use of *"Dadda's counters"* was undertaken by Stenzel and Kubitz in 1977. In their paper [27] they have also demonstrated a parallel multiplier built using ROM to implement [5,5,4] counters used for partial product summation.

The quest for making the parallel multiplier even faster continued for almost 30 years. However, the pursuit for inventing a fastest *"counter"* did not result in a structure yielding faster partial product summation than the one which uses Full-Adder (FA) cell, or *"3:2 counter"*. Therefore *"Wallace Tree"* was widely used in the implementation of the parallel multipliers.

4:2 Compressor:

In 1981 Weinberger disclosed a structure which he called *"4-2 carry-save module"* [28]. This structure contained a combination of FA cells in an intricate interconnection structure which was yielding a faster partial product compression than the use of 3:2 counters.

The structure actually compresses five partial product bits into three, however it is connected in such a way that four of the inputs are coming from the same bit position of the weight j while one bit is fed from the neighboring position *j-1* (known as carry-in). The output of such a 4:2 module consists of one bit in the position *j* and two bits in the position *j+1*.

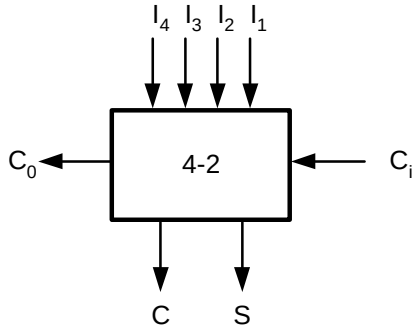

Fig. 14. 4:2 Compressor arrangement [28]

This structure does not represent a counter (though it became erroneously known as *"4:2 counter"*) but a *"compressor"* that would compress four partial product bits into two (while using one bit laterally connected between adjacent 4:2 compressors). The structure of 4:2 compressor is shown in Fig. 14. The efficiency of such a structure is higher (it reduces the number of partial product bits by one half at each stage). The speed of such a 4:2 compressor has been determined by the speed of 3 XOR gates in series, in the redesigned version of 4:2 compressor [36], making such a scheme more efficient that the one using 3:2 counters in a regular *"Wallace Tree"*. The other equally important feature of the use of 4:2 compressor is that the interconnections between 4:2 cells follow more regular pattern than in case of the *"Wallace Tree"*.

TDM:
The further work in improving the speed of a multiplier by optimizing Partial Product Reduction Tree (PPRT) was extended by Oklobdzija, Villeger and Liu [30]. Their approach was to optimize the entire PPRT in one pass, thus the name *Three Dimensional optimizaiton Method* (TDM). The important aspect of this method is in sorting of fast inputs and fast outputs. It was realized that the most important step is to properly interconnect the elements used in the PPRT. Thus, appropriate counters (3:2 adders in a particular case) were characterized in a way which identifies delay of each input to each output. Interconnecting of the PPRT was done in a way in which signals with large delays are connected to *"fast inputs"* and signals with small delay to *"slow inputs"* in a way that minimizes the critical paths in the PPRT.

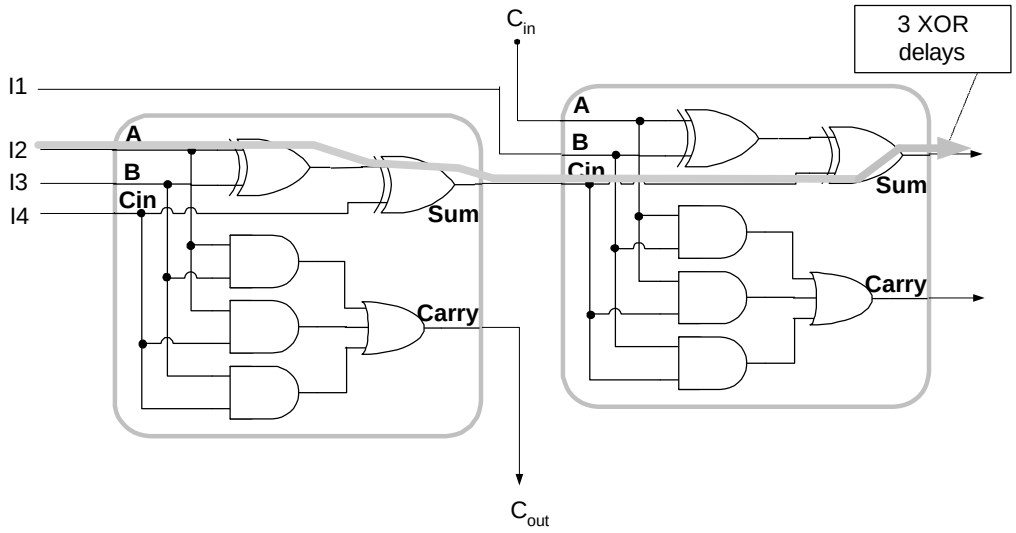

Fig. 15. An example of TDM method producing a balanced 4:2 compressor [30]

An example of this method is illustrated in Fig. 15. producing a 3 XOR gate delay 4:2 compressor, without resorting to a redesign as done in [36]. It was further proven that TDM indeed produces an optimal PPRT and that further optimization is not possible [37, 30]. An example of TDM generation of PPRT is shown in Fig. 16.

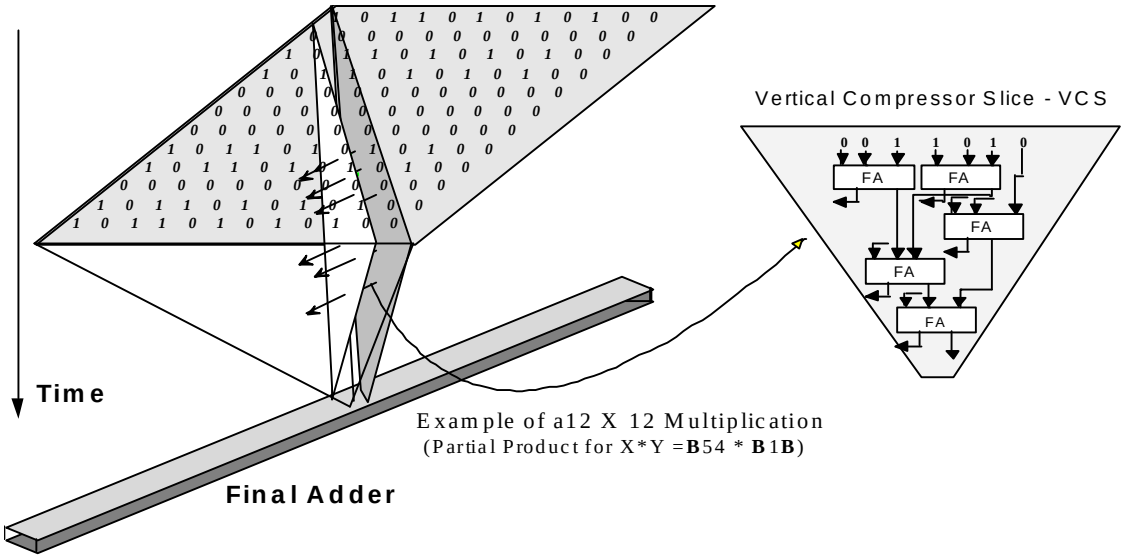

Fig. 16. Generation of the Partial Product Reduction Tree in TDM multiplier [30]

Booth Recoding Algorithm:

One of the best known variations of the multiplication algorithm is *"Booth Recoding Algorithm"* described by Booth in 1951[31]. This algorithm allows for the reduction of the number of partial products, thus speeding up the multiplication process. Generally speaking, Booth algorithm is a case of using the redundant number system with the radix *r* higher than *r=2* [1]. Earlier two's complement multipliers required data dependent correction cycles if either operand is negative. Both algorithm can be used for both sign-magnitude numbers as well as 2's complement numbers with no need for a correction term or a correction step.

A modification of the Booth algorithm was proposed by Mac Sorley in which a triplet of bits is scanned instead of two bits [32]. This technique has the advantage of reducing the number of partial products by roughly one half.

This method is actually an application of a sign-digit representation in radix 4 [1]. The *Booth-MacSorley Algorithm*, usually called the *Modified Booth Algorithm* or simply the *Booth Algorithm*, can be generalized to any radix. However, a 3-bit recoding (case of radix 8) would require the following set of digits to be multiplied by the multiplicand : 0, ±1, ±2, ±3, ±4. The difficulty lies in the fact that ±3Y is computed by summing (or subtracting) Y to ±2Y, which means that a carry propagation occurs. The delay caused by the carry propagation renders this scheme to be slower than a conventional one. Consequently, only the 2 bit (radix 4) Booth recoding is used.

Booth recoding necessitates the internal use of 2's complement representation in order to efficiently perform subtraction of the partial products as well as additions. However, floating point standard specifies sign magnitude representation which is also followed by most of the non-standard floating point numbers in use today. The Booth algorithm [31] is widely used for two's complement multiplication, since it is easy to implement.

Booth recoding is performed within two steps: *encoding* and *selection*. The purpose of the encoding is to scan the triplet of bits of the multiplier and define the operation to be performed on the multiplicand, as shown in Table 1.

Table 1: *Booth Recoding*

$x_{i+2}x_{i+1}x_i$	Add to partial product
000	+0Y
001	+1Y
010	+1Y
011	+2Y
100	-2Y
101	-1Y
110	-1Y
111	-0Y

The advantage of Booth recoding is that it generates roughly one half of the partial products as compared to the multiplier implementation, which does not use Booth recoding. However, the benefit achieved comes at the expense of increased hardware complexity. Indeed, this implementation requires hardware for the *encoding* and for the *selection* of the partial products (0, ±Y, ±2Y).

Hitachi's DPL Multiplier:

Hitachi's DPL multiplier was the first one to achieve under 5nS speed for a double-precision floating-point mantissa imposed by the increasing demands on the operating frequency of modern micro-processors [12,33]. This multiplier is of a regular structure including: (a.) A Booth Recoder, (b.) A Partial Product Reduction Tree (Wallace Tree) and (c.) A final Carry Propagate Adder (CPA) as shown in Fig. 17.

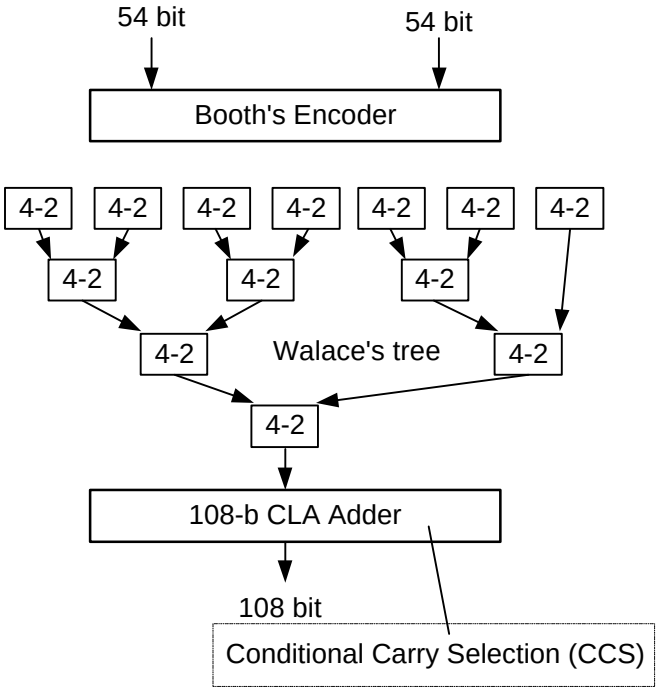

Fig. 17. Organization of Hitachi's DPL multiplier [33]

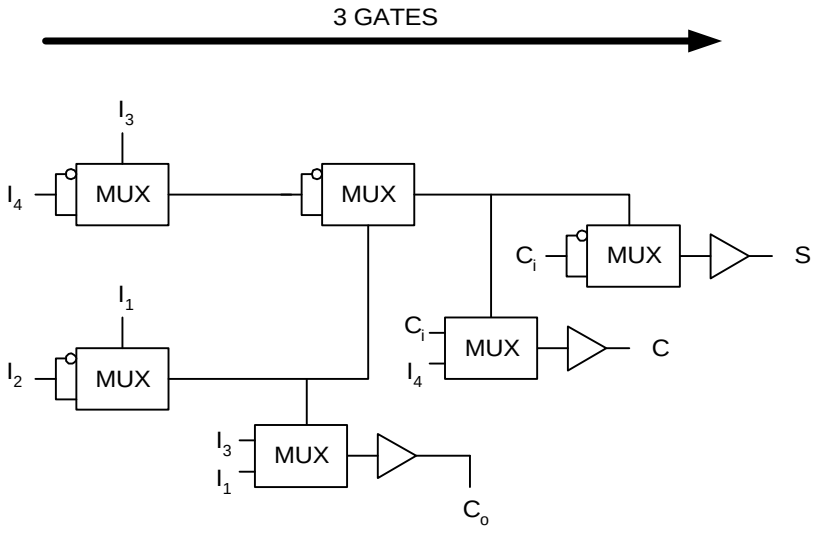

Fig. 18.a. b. (a.) Hitachi's 4:2 compressor structure (b.) DPL multiplexer circuit [12]

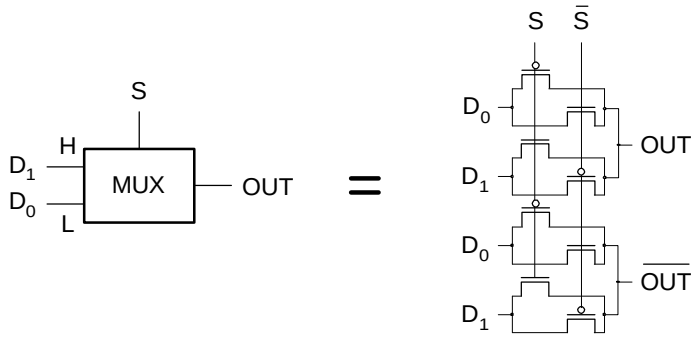

Fig. 19. (a.) Regular *Booth Selector* (b.) Modified Booth Selector [35]

The key to performance of Hitachi's multiplier lays in the use of DPL circuits and the efficiency with which DPL can realize 4:2 compressor. The structure of Hitachi's 4:2 compressor is shown in Fig. 18.a. The realization of the 4:2 function consists entirely of DPL multiplexers which introduce only one pass-transistor delay in the critical path as shown in Fig.18.b. Indeed later studies [35] recognized this structure as one of the fastest Partial Product Reduction Tree (PPRT) realizations. For larger size multipliers this PPRT may start showing degraded performance because of the long pass-transistor chain which is equal to the number of 4:2 compressors used in the PPRT.

Inoue's Multiplier:
High speed multiplier published by Inoue, et al. [35] employs two novel techniques in achieving very fast (4.1nS delay) 54X54-bit parallel multiplier implemented in 0.25u CMOS technology.

The first novelty introduced is in a new design of the *Booth Encoder* and *Booth selector* for generation of partial products. The encoding used in Inoue's multiplier is shown in Table 2.

Table 2: Truth Table for Second-Order Modified Booth Encoding [35]

Inputs				Usual			Sign select			
b_{j+1}	b_j	b_{j-1}	Func.	X_j	$2X_j$	M_j	X_j	$2X_j$	PL_j	M_j
0	0	0	0	0	0	0	0	1	0	0
0	0	1	+A	1	0	0	1	0	1	0
0	1	0	+A	1	0	0	1	0	1	0
0	1	1	+2A	0	1	0	0	1	1	0
1	0	0	-2A	0	1	1	0	1	0	1
1	0	1	-A	1	0	1	1	0	0	1
1	1	0	-A	1	0	1	1	0	0	1
1	1	1	0	0	0	1	0	1	0	0

X_j - partial product , PL_j - positive partial product, M_j - negative partial product
B - Multiplier (encoded), A - Multiplicand,
P = AxB

There are two bits used for generation of sign of the partial product: Mj (for negative) and PLj (for positive). Though, this may seem to be redundant at the first sight, it allows for a simpler implementation of the *Booth Encoder* which does not require an XOR gate in the critical path. The equations for *Booth Selector* using regular and modified *Booth Encoding* are listed:

$$P_{i,j} = (a_i \cdot X_j + a_{i-1} \cdot 2X_j) \oplus M_j \qquad (i = 0,1,2,...., n-1 \;\; j = 0,2,4,..., n-4, n-2)$$

.....regular *Booth Encoder*

(a.)

$$P_{i,j} = (a_i \cdot PL_j + \overline{a}_i \cdot M_j) X_j + (a_{i-1} \cdot PL_j + \overline{a}_{i-1} \cdot M_j) \cdot 2 \cdot X_j \qquad (i = 0,1,2,...., n-1 \;\; j = 0,2,4,..., n-4, n-2)$$

..... Modified *Booth Encoder*

(b.)

A modified equations (b.) obtained from the Table 2. yield simpler *Booth Selector* implementation than the regular case. Modified *Booth Selector* is shown in Fig. 19. (b.) versus regular *Booth Selector* shown in Fig. 19. (a.).

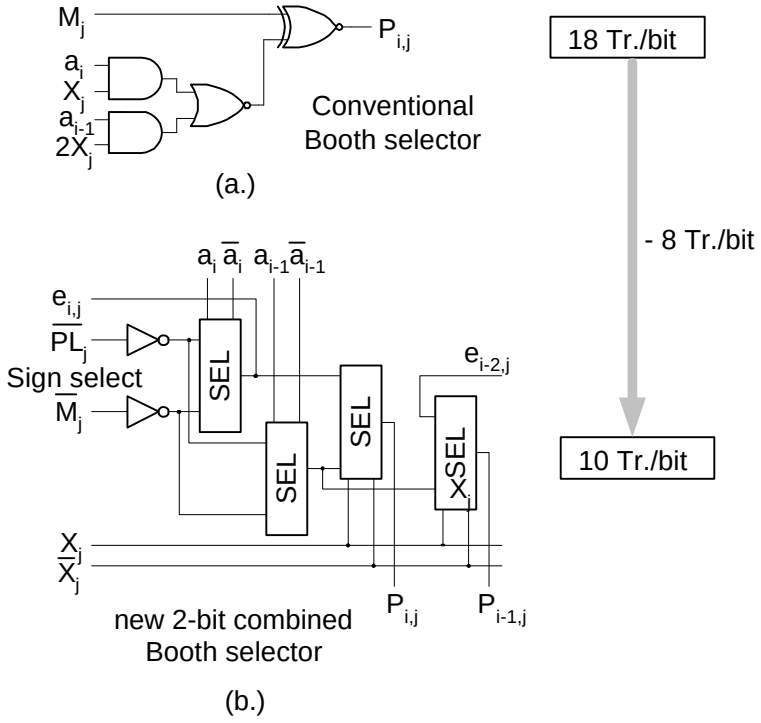

Fig. 19. (a.) Regular *Booth Selector* (b.) Modified Booth Selector [35]

The modified *Booth Selector* requires 10 transistors per bit as compared to the regular *Booth Selector* which requires 18 transistors per bit for its implementation. The modification shown in Table 2. yields 44% reduction in the transistor count for the *Booth Selector* of the 54X54-bit multiplier. Given that the total number of transistor used for Booth Encoder in a 54X54-bit multiplier is only 1.2% of the total, modification of the *Booth Encoder* resulting from the Table 2. does not result in significant transistor savings. However, the use of the new *Booth Encoder* resulted in a slight improvement in speed.

The second novelty in Inoue's multiplier is the pass-transistor implementation of the 4:2 compressor, which is shown in Fig. 20.

Inoue realized that there are 2^6 possible implementations of the 4:2 compressor. Out of the total number of 2^6 they have chosen the one that yields the minimal transistor count yet maintaining the speed within the 5% of the fastest possible realization. This resulted in 24% savings in transistor count in the partial product reduction tree as compared to the earlier designs [34]. The transistor savings more than offset the 5% speed degradation by yielding more area and power efficient design. It could be argued that the area improvement resulted in a better speed in the final implementation, which the simulation tools were not able to show.

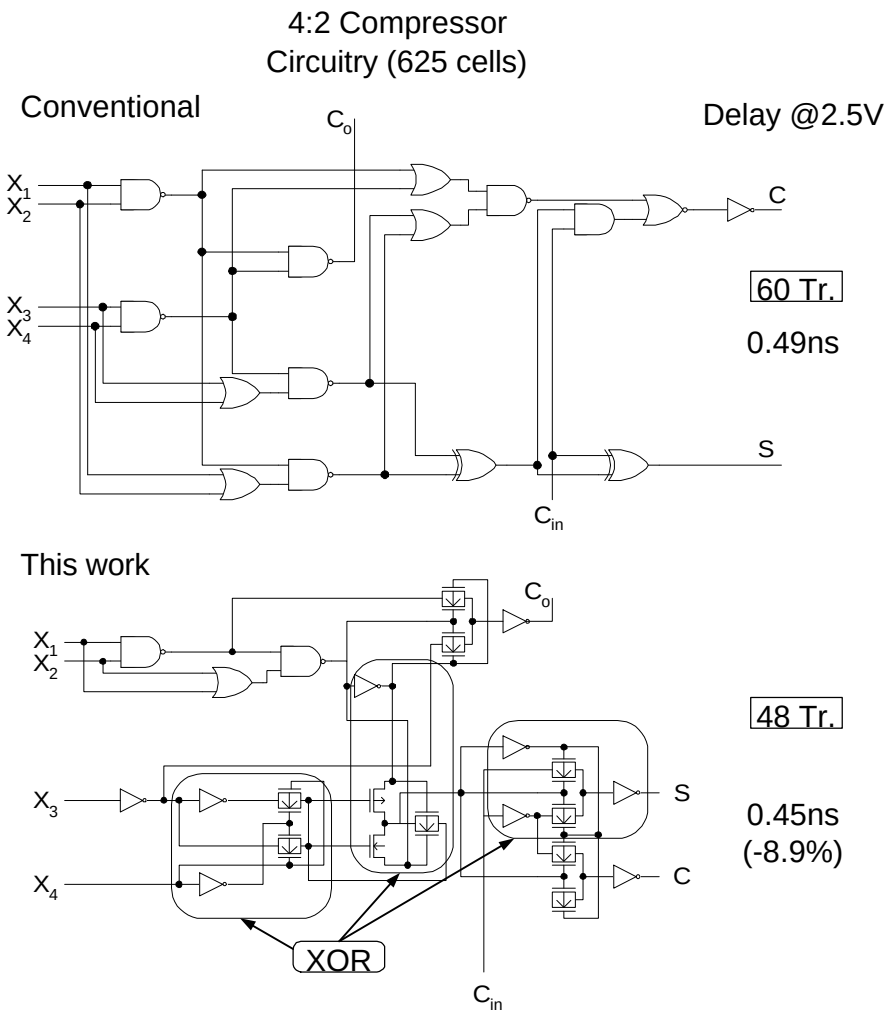

Fig. 20. Pass-Transistor Implementation of the 4:2 Compressor [35]

Conclusion

In the past, a thorough examination of the algorithms with the respect to particular technology has only been partially done. The merit of the new technology is to be evaluated by its ability to efficiently implement the computational algorithms. In the other words, the technology is developed with the aim to efficiently serve the computation. The reverse path; evaluating the merit of the algorithms should also be taken. Therefore, it is important to develop computational structures that fit well into the execution model of the processor and are optimized for the current technology. In such a case, optimization of the algorithms is performed globally across the critical path of its implementation.

Ability to integrate 100 millions of transistors onto the silicon has changed our focus and the way we think. Measuring the quality of the algorithm by the minimum number of devices used has simply vanished from the picture. However, new concerns such as power, have entered it.

References

[1] A. Avizienis, *"Digital Computer Arithmetic: A Unified Algorithmic Specification"*, Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971.

[2] Earl E. Swartzlander, *"Computer Arithmetic"* Vol. 1 & 2, IEEE Computer Society Press, 1990.

[3] K. Hwang, "*Computer Arithmetic : Principles, Architecture and Design*", John Wiley and Sons, 1979.

[4] S.Waser, M.Flynn, *"Introduction to Arithmetic for Digital Systems Designers"*, Holt, Rinehart and Winston 1982.

[5] M. Ercegovac, *"Digital Systems and Hardware/Firmware Algorithms*", Chapter 12: Arithmetic Algorithms and Processors, John Wiley & Sons, 1985.

[6] T. Kilburn, D.B.G. Edwards and D. Aspinall, Parallel Addition in Digital Computers: A New Fast "Carry" Circuit, Proc. IEE, Vol.106, Pt.B. p.464, September 1959.

[7] V.G. Oklobdzija, E.R. Barnes, *"Some Optimal Schemes for ALU Implementation in VLSI Technology"*, Proceedings of 7th Symposium on Computer Arithmetic, June 4-6, 1985, University of Illinois, Urbana, Illinois.

[8] V. G. Oklobdzija, "Simple And Efficient CMOS Circuit For Fast VLSI Adder Realization," *Proceedings of the International Symposium on Circuits and Systems*, pp. 1-4, 1988.

[9] C. Nagendra, et al, *"A Comparison of the Power-Delay Characteristics of CMOS Adders"*, Proceedings of the International Workshop on Low-Power Design, 1994.

[10] Yano, K, et al, "A 3.8 ns CMOS 16x16-b Multiplier Using Complementary Pass-Transistor Logic," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 388-395, April 1990.

[11] M. Suzuki et al, *"A 1.5nS 32b CMOS ALU in Double Pass-Transistor Logic"*, Digest of Technical Papers, 1993 IEEE Solid-State Circuits Conference, San Francisco, February 24-26, 1993.

[12] N. Ohkubo, et al, *"A 4.4-ns CMOS 54x54-b Multiplier Using Pass-transistor Multiplexer"*, Proceedings of the Custom Integrated Circuits Conference, San Diego, California, May 1-4, 1994.

[13] Weinberger, J.L. Smith, *"A Logic for High-Speed Addition"*, National Bureau of Standards, Circulation 591, p. 3-12, 1958.

[14] Naini, D. Bearden, W. Anderson, *"A 4.5nS 96-b CMOS Adder Design"*, IEEE 1992 Custom Integrated Circuits Conference, 1992.

[15] B.D. Lee, V.G. Oklobdzija, *"Improved CLA Scheme with Optimized Delay"*, Journal of VLSI Signal Processing, Vol. 3, p. 265-274, 1991.

[16] Kogge, P.M., Stone, H.S., *"A Parallel Algorithms for the Efficient Solution of a General Class of Recurrence Equations"*, IEEE Transactions on Computers, Vol. C-22, No 8, Aug. 1973. p. 786-93.

[17] A. Bilgory and D.D. Gajski, Automatic Generation of Cells for Recurence Structures, Proc. of 18th Design Automation Conference, Nashville, Tennessee, 1981.

[18] Brent, R.P.; Kung, H.T. A regular layout for parallel adders. IEEE Transactions on Computers, vol.C-31, (no.3), March 1982. p.260-4.

[19] H. Ling, *"High Speed Binary Adder"*, IBM Journal of Research and Development, Vol. 25, No 3, May 1981. p. 156.

[20] Naffziger, S., *"A Sub-Nanosecond 0.5 um 64 b Adder Design"*, 1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, San Francisco, February 8-10, 1996. p. 362-3.

[21] Sklanski, *"Conditional-Sum Addition Logic"*, IRE Transaction on Electronic Computers, EC-9, pp. 226-231, 1960.

[22] O. J. Bedrij, *"Carry-Select Adder"*, IRE Transaction on Electronic Computers, June 1962.

[23] D. Dobberpuhl et al, *"A 200MHz 64-b Dual-Issue CMOS Microprocessor"*, IEEE Journal of Solid-State Circuits, Vol 27, No 11. November 1992.

[24] S. D. Pezaris *"A 40 ns 17-bit array multiplier"*, IEEE Trans. on Computers., Vol. C-20, pp. 442-447, Apr. 1971.

[25] C.S. Wallace, "A Suggestion for a Fast Multiplier", IEE Transactions on Electronic Computers, EC-13, p.14-17, 1964.

[26] L. Dadda, "Some Schemes for Parallel Multipliers", Alta Frequenza, Vol.34, p.349-356, March 1965.

[27] W. J. Stenzel, *"A Compact High Speed Parallel Multiplication Scheme"*, IEEE Transaction on Computers, Vol. C-26, pp. 948-957, February 1977.

[28] A. Weinberger, *"4:2 Carry-Save Adder Module*", IBM Technical Disclosure Bulletin.,Vol.23, January 1981.

[29] J. Fadavi-Ardekani, *"M x N Booth Encoded Multiplier Generator Using optimized Wallace Trees"* IEEE trans. on VLSI Systems, Vol. 1 ,No.2, June 1993.

[30] V.G. Oklobdzija, D. Villeger, S. S. Liu, *"A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using and Alghoritmic Approach"*, IEEE Transaction on Computers, Vol. 45, No 3, March 1996.

[31] A. D. Booth, "A Signed Binary Multiplication Technique", Qarterly J. Mechan. Appl. Math., Vol. IV, 1951.

[32] O. L. Mac Sorley, "High Speed Arithmetic in Binary Computers", Proceedings of IRE, Vol.49, No. 1, January, 1961.

[33] Ohkubo, N., Suzuki, M., Shinbo, T., Yamanaka, T., Shimizu, A.; Sasaki, K., Nakagome, Y., *"A 4.4 ns CMOS 54*54-b Multiplier using Pass-Transistor Multiplexer"*, IEEE Journal of Solid-State Circuits, Vol. 30, No 3, March 1995. p. 251-7.

[34] G. Goto, et al., *"A 4.1nS Compact 54X54-b Multiplier Utilizing Sign-Select Booth Encoders"*, IEEE Journal of Solid-State Circuits, Vol. 32, No 11, November 1997. p. 1676-82.

[35] A. Inoue, R. Ohe, S. Kashiwakura, S. Mitari, T. Tsuru, T. Izawa, G. Goto, *"A 4.1nS Compact 54X54b Multiplier Utilizing Sign Select Booth Encoders"*, 1997 IEEE International Solid State Circuits Conference, Digest of Papers, San Francisco, 1997. p.416.

[36] M. Nagamatsu et al, *"A 15nS 32X32-bit CMOS Multiplier with an Improved Parallel Structure"*, Digest of Technical papers, IEEE Custom Integrated Circuits Conference 1989.

[37] P. Stelling, C. Martel, V. G. Oklobdzija, R. Ravi, "Optimal Circuits for Parallel Multipliers," IEEE Transaction on Computers, Vol. 47, No.3, pp. 273-285, March, 1998.

[38] C. Mead, L. Conway, *"Introduction to VLSI systems"*, Addison-Wesley, 1980.