UNIT 6: Subsystem Design Processes Illustration

Objectives: At the end of this unit we will be able to understand

- Design consideration, problem and solution
- Design processes
  - Basic digital processor structure
  - Datapath
  - Bus Architecture
- Design 4 – bit shifter
- Design of ALU subsystem
- 4 – bit Adder

General Considerations

- Lower unit cost
- Higher reliability
- Lower power dissipation, lower weight and lower volume
- Better performance
- Enhanced repeatability
- Possibility of reduced design/development periods

Some Problems

1. How to design complex systems in a reasonable time & with reasonable effort.
2. The nature of architectures best suited to take full advantage of VLSI and the technology
3. The testability of large/complex systems once implemented on silicon

Some Solution

Problem 1 & 3 are greatly reduced if two aspects of standard practices are accepted.

1. a) Top-down design approach with adequate CAD tools to do the job

   b) Partitioning the system sensibly

   c) Aiming for simple interconnections

   d) High regularity within subsystem

   e) Generate and then verify each section of the design
2. Devote significant portion of total chip area to test and diagnostic facility
3. Select architectures that allow design objectives and high regularity in realization

Illustration of design processes

1. Structured design begins with the concept of hierarchy

2. It is possible to divide any complex function into less complex subfunctions that is up to leaf cells
3. Process is known as top-down design
4. As a systems complexity increases, its organization changes as different factors become relevant to its creation
5. Coupling can be used as a measure of how much submodels interact
6. It is crucial that components interacting with high frequency be physically proximate, since one may pay severe penalties for long, high-bandwidth interconnects
7. Concurrency should be exploited – it is desirable that all gates on the chip do useful work most of the time
8. Because technology changes so fast, the adaptation to a new process must occur in a short time.

Hence representing a design several approaches are possible. They are:

- Conventional circuit symbols
- Logic symbols
- Stick diagram
- Any mixture of logic symbols and stick diagram that is convenient at a stage
- Mask layouts
- Architectural block diagrams and floor plans

General arrangements of a 4 – bit arithmetic processor

The basic architecture of digital processor structure is as shown below in figure 6.1. Here the design of datapath is only considered.

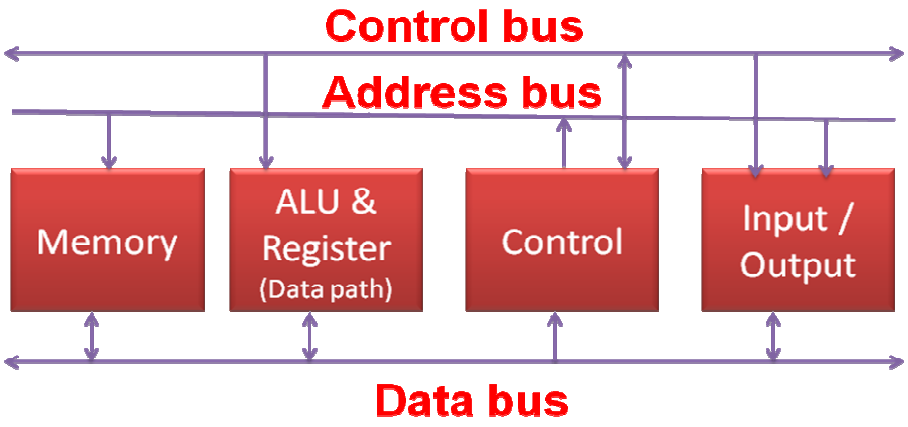


Figure 6.1: Basic digital processor structure

Datapath is as shown below in figure 6.2. It is seen that the structure comprises of a unit which processes data applied at one port and presents its output at a second port.

Alternatively, the two data ports may be combined as a single bidirectional port if storage facilities exist in the datapath. Control over the functions to be performed is effected by control signals as shown.

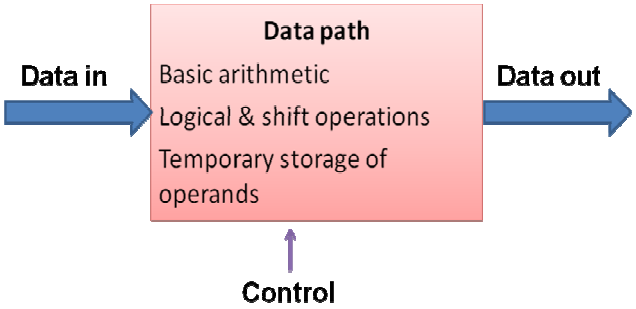


Figure 6.2: Communication strategy for the datapath

Datapath can be decomposed into blocks showing the main subunits as in figure 3. In doing so it is useful to anticipate a possible floor plan to show the planned relative decomposition of the subunits on the chip and hence on the mask layouts.

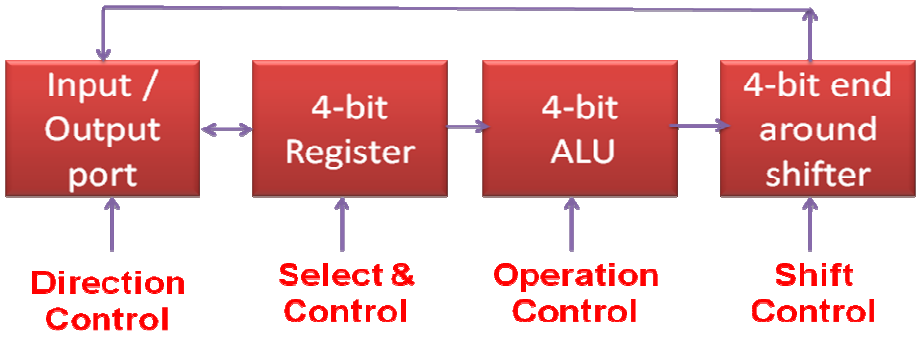


Figure 6.3: Subunits and basic interconnection for datapath

Nature of the bus architecture linking the subunits is discussed below. Some of the possibilities are:

One bus architecture:

Figure 6.4: One bus architecture

Sequence:

1. 1st operand from registers to ALU. Operand is stored there.
2. 2nd operand from register to ALU and added.
3. Result is passed through shifter and stored in the register

Two bus architecture:

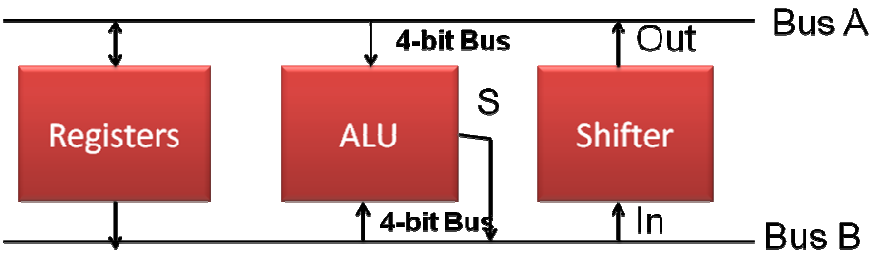


Figure 6.5: Two bus architecture

Sequence:

1. Two operands (A & B) are sent from register(s) to ALU & are operated upon, result S in ALU.
2. Result is passed through the shifter & stored in registers.

Three bus architecture:

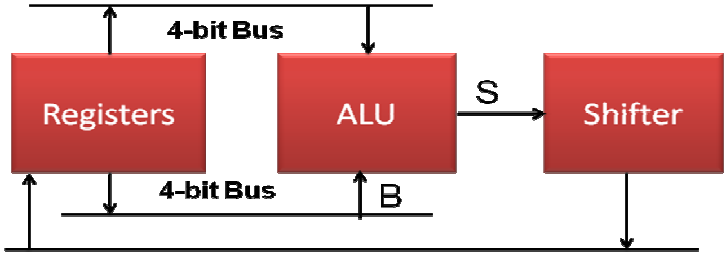


Figure 6.6: Three bus architecture

Sequence:

Two operands (A & B) are sent from registers, operated upon, and shifted result (S) returned to another register, all in same clock period.

In pursuing this design exercise, it was decided to implement the structure with a 2 – bus architecture. A tentative floor plan of the proposed design which includes some form of interface to the parent system data bus is shown in figure 6.7.

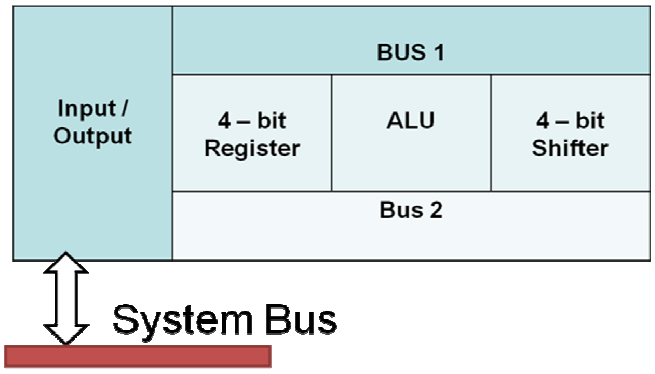


Figure 6.7: Tentative floor plan for 4 – bit datapath

The proposed processor will be seen to comprise a register array in which 4-bit numbers can be stored, either from an I/O port or from the output of the ALU via a shifter. Numbers from the register array can be fed in pairs to the ALU to be added (or subtracted) and the result can be shifted or not. The data connections between the I/O port, ALU, and shifter must be in the form of 4-bit buses. Also, each of the blocks must be suitably connected to control lines so that its function may be defined for any of a range of possible operations.

During the design process, and in particular when defining the interconnection strategy and designing the stick diagrams, care must be taken in allocating the layers to the various data or control paths. Points to be noted:

- ✓ Metal can cross poly or diffusion
- ✓ Poly crossing diffusion form a transistor
- ✓ Whenever lines touch on the same level an interconnection is formed
- ✓ Simple contacts can be used to join diffusion or poly to metal.
- ✓ Buried contacts or a butting contacts can be used to join diffusion and poly
- ✓ Some processes use 2^{nd} metal
- ✓ 1^{st} and 2^{nd} metal layers may be joined using a via
- ✓ Each layer has particular electrical properties which must be taken into account
- ✓ For CMOS layouts, p-and n-diffusion wires must not directly join each other
- ✓ Nor may they cross either a p-well or an n-well boundary

Design of a 4-bit shifter

Any general purpose n-bit shifter should be able to shift incoming data by up to n – 1 place in a right-shift or left-shift direction. Further specifying that all shifts should be on an end-around basis, so that any bit shifted out at one end of a data word will be shifted in at the other end of the word, then the problem of right shift or left shift is greatly eased. It can be analyzed that for a 4-bit word, that a 1-bit shift right is equivalent to a 3-bit shift left and a 2-bit shift right is equivalent to a 2-bit left etc. Hence, the design of either shift right or left can be done. Here the design is of shift right by 0, 1, 2, or 3 places. The shifter must have:

- input from a four line parallel data bus
- four output lines for the shifted data
- means of transferring input data to output lines with any shift from 0 to 3 bits

Consider a direct MOS switch implementation of a 4 X 4 crossbar switches shown in figure 6.8. The arrangement is general and may be expanded to accommodate n-bit inputs/outputs. In this arrangement any input can be connected to any or all the outputs. Furthermore, 16 control signals (sw_{00} – sw_{15}), one for each transistor switch, must be provided to drive the crossbar switch, and such complexity is highly undesirable.

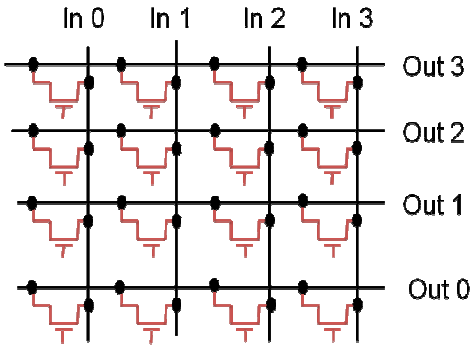


Figure 6.8: 4 X 4 crossbar switch

An adaptation of this arrangement recognizes the fact that we couple the switch gates together in groups of four and also form four separate groups corresponding to shifts of zero, one, two and three bits. The resulting arrangement is known as a barrel shifter and a 4 X 4 barrel shifter circuit diagram is as shown in the figure 6.9.

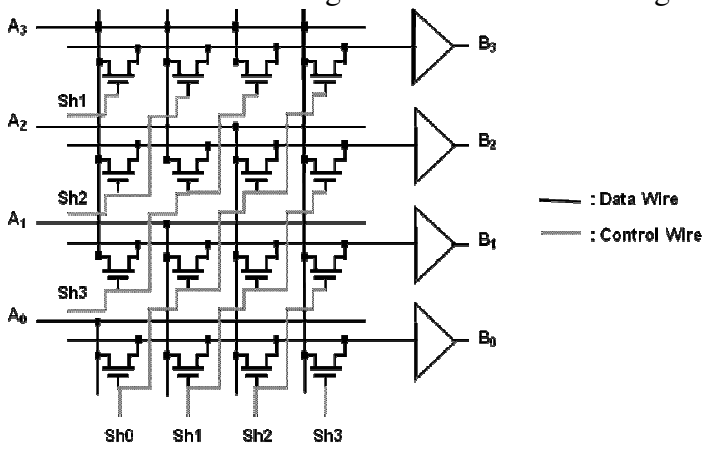


Figure 6.9: 4 X 4 barrel shifter

The interbus switches have their gate inputs connected in a staircase fashion in groups of four and there are now four shift control inputs which must be mutually exclusive in the active state. CMOS transmission gates may be used in place of the simple pass transistor switches if appropriate. Barrel shifter connects the input lines representing a word to a group of output lines with the required shift determined by its control inputs (sh0, sh1, sh2, sh3). Control inputs also determine the direction of the shift. If input word has n – bits and shifts from 0 to n-1 bit positions are to be implemented.

To summaries the design steps

- Set out the specifications
- Partition the architecture into subsystems
- Set a tentative floor plan
- Determine the interconnects
- Choose layers for the bus & control lines
- Conceive a regular architecture
- Develop stick diagram

- Produce mask layouts for standard cell
- Cascade & replicate standard cells as required to complete the design

Design of an ALU subsystem

Having designed the shifter, we shall design another subsystem of the 4-bit data path. An appropriate choice is ALU as shown in the figure 6.10 below.

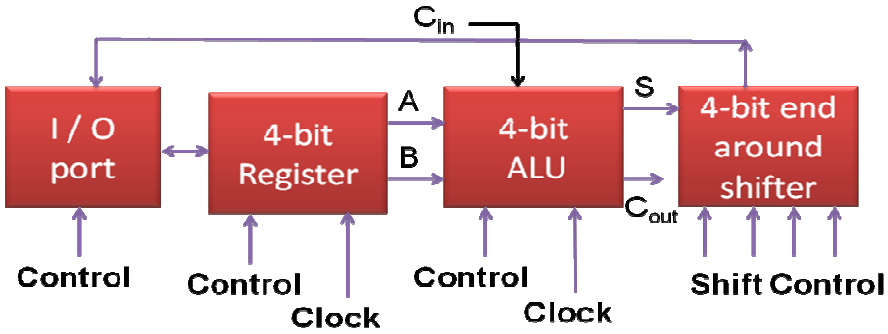


Figure 6.10: 4-bit data path for processor

The heart of the ALU is a 4-bit adder circuit. A 4-bit adder must take sum of two 4-bit numbers, and there is an assumption that all 4-bit quantities are presented in parallel form and that the shifter circuit is designed to accept and shift a 4-bit parallel sum from the ALU. The sum is to be stored in parallel at the output of the adder from where it is fed through the shifter and back to the register array. Therefore, a single 4-bit data bus is needed from the adder to the shifter and another 4-bit bus is required from the shifted output back to the register array. Hence, for an adder two 4-bit parallel numbers are fed on two 4-bit buses. The clock signal is also required to the adder, during which the inputs are given and sum is generated. The shifter is unclocked but must be connected to four shift control lines.

Design of a 4-bit adder:

The truth table of binary adder is as shown in table 6.1

Inputs			**Outputs**	
A_k	B_k	C_{k-1}	S_k	C_k
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

As seen from the table any column k there will be three inputs namely A_k , B_k as present input number and C_{k-1} as the previous carry. It can also be seen that there are two outputs sum S_k and carry C_k.

From the table one form of the equation is:

Sum $S_k = H_kC_{k-1}' + H_k'C_{k-1}$

New carry $C_k = A_kB_k + H_kc_{k-1}$

Where

Half sum $H_k = A_k'B_k + A_k B_k'$

Adder element requirements

Table 6.1 reveals that the adder requirement may be stated as:

If $A_k = B_k$ then $S_k = C_{k-1}$

Else $S_k = C_{k-1}'$

And for the carry C_k

If $A_k = B_k$ then $C_k = A_k = B_k$

Else $C_k = C_{k-1}$

Thus the standard adder element for 1-bit is as shown in the figure 6.11.

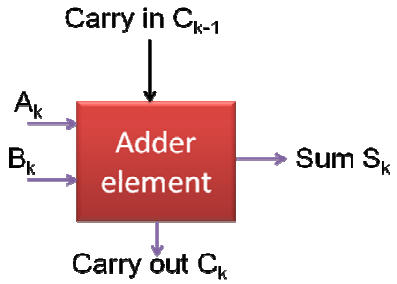


Figure 6.11: Adder element

Implementing ALU functions with an adder:

An ALU must be able to add and subtract two binary numbers, perform logical operations such as And, Or and Equality (Ex-or) functions. Subtraction can be performed by taking 2's complement of the negative number and perform the further addition. It is desirable to keep the architecture as simple as possible, and also see that the adder performs the logical operations also. Hence let us examine the possibility.

The adder equations are:

Sum $S_k = H_kC_{k-1}' + H_k'C_{k-1}$

New carry $C_k = A_kB_k + H_k C_{k-1}$

Where

Half sum $H_k = A_k'B_k + A_k B_k'$

Let us consider the sum output, if the previous carry is at logical 0, then

$S_k = H_k. 1 + H_k'. 0$

$S_k = H_k = A_k'B_k + A_k B_k'$ – An Ex-or operation

Now, if C_{k-1} is logically 1, then

$S_k = H_k. 0 + H_k'. 1$

$S_k = H_k'$ – An Ex-Nor operation

Next, consider the carry output of each element, first C_{k-1} is held at logical 0, then

$C_k = A_kB_k + H_k \cdot 0$

$C_k = A_kB_k$ - An And operation

Now if C_{k-1} is at logical 1, then

$C_k = A_kB_k + H_k \cdot 1$

On solving $C_k = A_k + B_k$ - An Or operation

The adder element implementing both the arithmetic and logical functions can be implemented as shown in the figure 6.12.

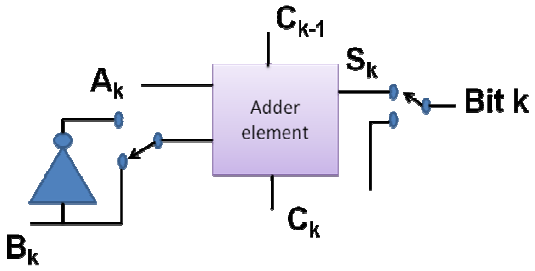


Figure 6.12: 1-bit adder element

The above can be cascaded to form 4-bit ALU.

A further consideration of adders

Generation:

This principle of generation allows the system to take advantage of the occurrences "a_k=b_k". In both cases (a_k=1 or a_k=0) the carry bit will be known.

Propagation:

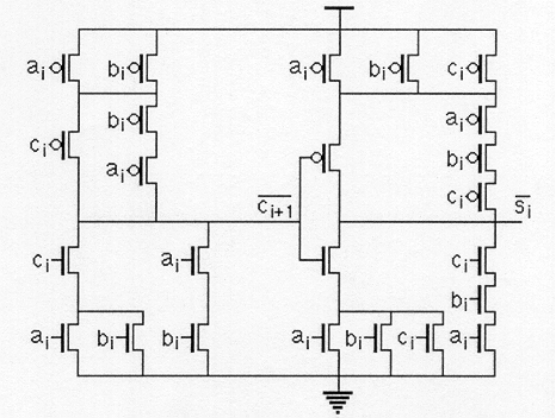
If we are able to localize a chain of bits $a_k\ a_{k+1}...a_{k+p}$ and $b_k\ b_{k+1}...b_{k+p}$ for which a_k not equal to b_k for k in [k,k+p], then the output carry bit of this chain will be equal to the input carry bit of the chain.

These remarks constitute the principle of generation and propagation used to speed the addition of two numbers.

All adders which use this principle calculate in a first stage.

$p_k = a_k$ XOR b_k

$g_k = a_k\ b_k$



Manchester carry – chain

This implementation can be very performant (20 transistors) depending on the way the XOR function is built. The carry propagation of the carry is controlled by the output of the XOR gate. The generation of the carry is directly made by the function at the bottom. When both input signals are 1, then the inverse output carry is 0.

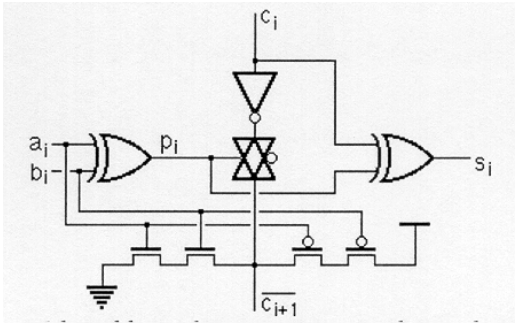


Figure-6.12: An adder with propagation signal controlling the pass-gate

In the schematic of Figure 6.12, the carry passes through a complete transmission gate. If the carry path is precharged to VDD, the transmission gate is then reduced to a simple NMOS transistor. In the same way the PMOS transistors of the carry generation is removed. One gets a Manchester cell.

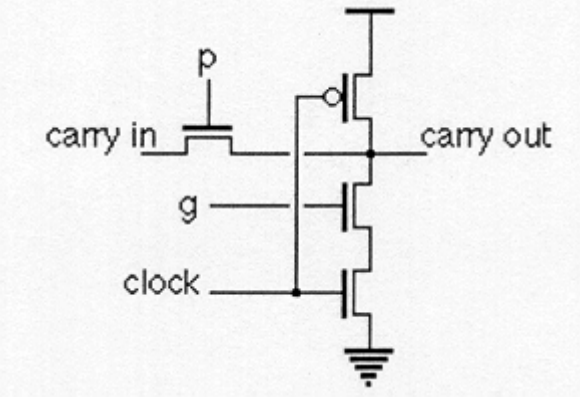


Figure-6.13: The Manchester cell

The Manchester cell is very fast, but a large set of such cascaded cells would be slow. This is due to the distributed RC effect and the body effect making the propagation time grow with the square of the number of cells. Practically, an inverter is added every four cells, like in Figure 6.14.

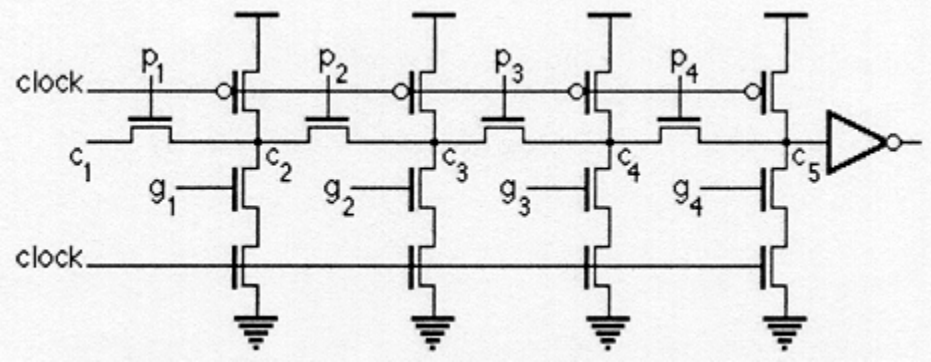


Figure-6.14**:** The Manchester carry cell

Adder Enhancement techniques

The operands of addition are the addend and the augend. The addend is added to the augend to form the sum. In most computers, the augmented operand (the augend) is replaced by the sum, whereas the addend is unchanged. High speed adders are not only for addition but also for subtraction, multiplication and division. The speed of a digital processor depends heavily on the speed of adders. The adders add vectors of bits and the principal problem is to speed- up the carry signal. A traditional and non optimized four bit adder can be made by the use of the generic one-bit adder cell connected one to the other. It is the ripple carry adder. In this case, the sum resulting at each stage need to wait for the incoming carry signal to perform the sum operation. The carry propagation can be speed-up in two ways. The first –and most obvious– way is to use a faster logic circuit technology. The second way is to generate carries by means of forecasting logic that does not rely on the carry signal being rippled from stage to stage of the adder.

The Carry-Skip Adder

Depending on the position at which a carry signal has been generated, the propagation time can be variable. In the best case, when there is no carry generation, the addition time will only take into account the time to propagate the carry signal. Figure 6.15 is an example illustrating a carry signal generated twice, with the input carry being equal to 0. In this case three simultaneous carry propagations occur. The longest is the second, which takes 7 cell delays (it starts at the 4th position and ends at the 11th position). So the addition time of these two numbers with this 16-bits Ripple Carry Adder is 7.k + k', where k is the delay cell and k' is the time needed to compute the 11th sum bit using the 11th carry-in.

With a Ripple Carry Adder, if the input bits Ai and Bi are different for all position i, then the carry signal is propagated at all positions (thus never generated), and the addition is completed when the carry signal has propagated through the whole adder. In this case, the Ripple Carry Adder is as slow as it is large. Actually, Ripple Carry Adders are fast only for some configurations of the input words, where carry signals are generated at some positions.

Carry Skip Adders take advantage both of the generation or the propagation of the carry signal. They are divided into blocks, where a special circuit detects quickly if all the bits to be added are different (Pi = 1 in all the block). The signal produced by this circuit will be called block propagation signal. If the carry is propagated at all positions in the block, then the carry signal entering into the block can directly bypass it and so be transmitted through a multiplexer to the next block. As soon as the carry signal is transmitted to a block, it starts to propagate through the block, as if it had been generated at the beginning of the block. Figure 6.16 shows the structure of a 24-bits Carry Skip Adder, divided into 4 blocks.

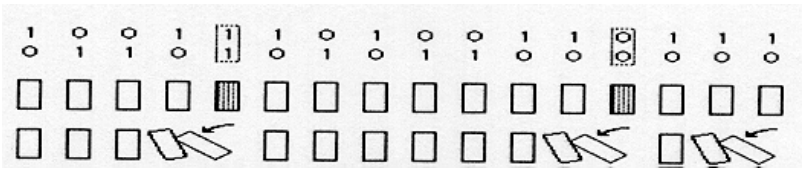


Figure 6.15: Example of Carry skip adder

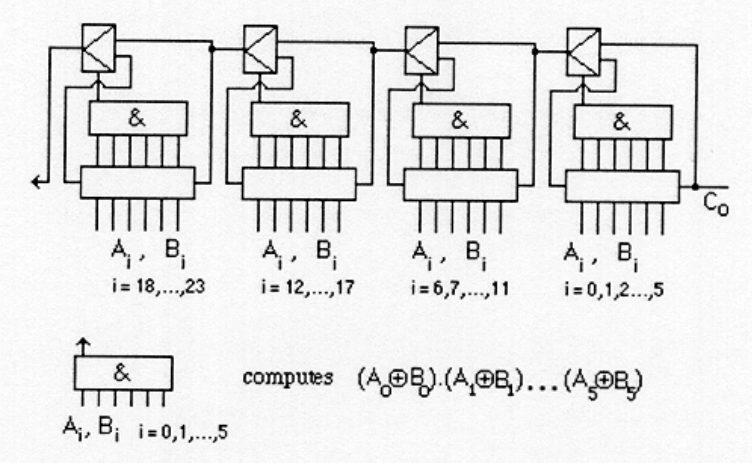


Figure-6.16: Block diagram of a carry skip adder

Optimization of the carry skip adder

It becomes now obvious that there exist a trade-off between the speed and the size of the blocks. In this part we analyze the division of the adder into blocks of equal size. Let us denote k1 the time needed by the carry signal to propagate through an adder cell, and k2 the time it needs to skip over one block. Suppose the N-bit Carry Skip Adder is divided into M blocks, and each block contains P adder cells. The actual addition time of a Ripple Carry Adder depends on the configuration of the input words. The completion time may be small but it also may reach the worst case, when all adder cells propagate the carry signal. In the same way, we must evaluate the worst carry propagation time for the Carry Skip Adder. The worst case of carry propagation is depicted in Figure 6.17.

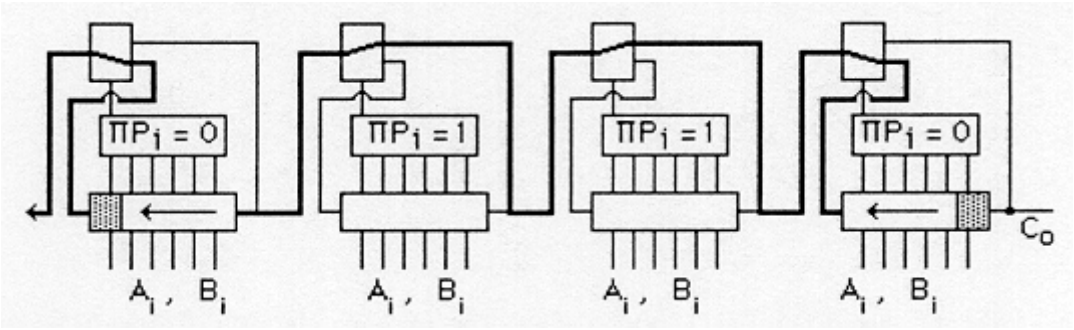


Figure-6.17: Worst case carry propagation for Carry Skip adder

The configuration of the input words is such that a carry signal is generated at the beginning of the first block. Then this carry signal is propagated by all the succeeding adder cells but the last which generates another carry signal. In the first and the last block the block propagation signal is equal to 0, so the entering carry signal is not transmitted to the next block. Consequently, in the first block, the last adder cells must wait for the carry signal, which comes from the first cell of the first block. When going out of the first

block, the carry signal is distributed to the 2nd, 3rd and last block, where it propagates. In these blocks, the carry signals propagate almost simultaneously (we must account for the multiplexer delays). Any other situation leads to a better case. Suppose for instance that the 2nd block does not propagate the carry signal (its block propagation signal is equal to zero), then it means that a carry signal is generated inside. This carry signal starts to propagate as soon as the input bits are settled. In other words, at the beginning of the addition, there exist two sources for the carry signals. The paths of these carry signals are shorter than the carry path of the worst case. Let us formalize that the total adder is made of N adder cells. It contains M blocks of P adder cells. The total of adder cells is then

$$N=M.P$$

The time T needed by the carry signal to propagate through P adder cells is

$$T=k_1.P$$

The time T' needed by the carry signal to skip through M adder blocks is

$$T'=k_2.M$$

The problem to solve is to minimize the worst case delay which is:

$$T_{worstcase} = 2 \cdot P \cdot k_1 + (M-2) \cdot k_2$$

$$T_{worstcase} = 2 \cdot \frac{N}{M} \cdot k_1 + (M-2) \cdot k_2$$

The Carry-Select Adder

This type of adder is not as fast as the Carry Look Ahead (CLA) presented in a next section. However, despite its bigger amount of hardware needed, it has an interesting design concept. The Carry Select principle requires two identical parallel adders that are partitioned into four-bit groups. Each group consists of the same design as that shown on Figure 6.18. The group generates a group carry. In the carry select adder, two sums are generated simultaneously. One sum assumes that the carry in is equal to one as the other assumes that the carry in is equal to zero. So that the predicted group carry is used to select one of the two sums.

It can be seen that the group carries logic increases rapidly when more high- order groups are added to the total adder length. This complexity can be decreased, with a subsequent increase in the delay, by partitioning a long adder into sections, with four groups per section, similar to the CLA adder.

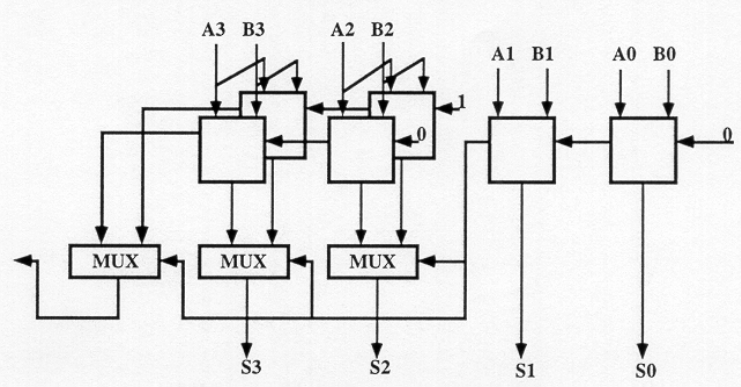


Figure-6.18: The Carry Select adder

Optimization of the carry select adder

- Computational time

$$T = K_1 n$$

- Dividing the adder into blocks with 2 parallel paths

$$T = K_1 n/2 + K_2$$

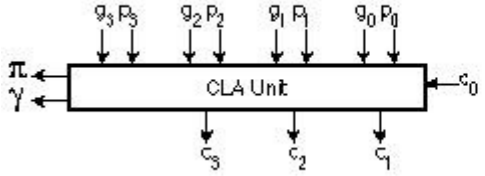
- For a n-bit adder of M-blocks and each block contains P adder cells in series $T = PK_1 + (M - 1)\,K_2$; $n = M.P$ minimum value for T is when $M=\sqrt{(K_1 n / K_1)}$

The Carry Look-Ahead Adder

The limitation in the sequential method of forming carries, especially in the Ripple Carry adder arises from specifying c_i as a specific function of c_{i-1}. It is possible to express a carry as a function of all the preceding low order carry by using the recursivity of the carry function. With the following expression a considerable increase in speed can be realized.

$$C_i = G_i + G_{i-2} P_{i-1} + G_{i-3} P_{i-2} P_{i-1} + \ldots \quad \ldots + G_0 P_1 P_2 \ldots P_{i-1} + C_0 P_0 P_1 P_2 \ldots P_{i-1}$$

Usually the size and complexity for a big adder using this equation is not affordable. That is why the equation is used in a modular way by making groups of carry (usually four bits). Such a unit generates then a group carry which give the right predicted information to the next block giving time to the sum units to perform their calculation.



$$\pi = p_0 p_1 p_2 p_3$$

$$\gamma = g_3 + p_3 g_2 + p_3 p_2 \cdot g_1 + p_3 p_2 p_1 \cdot g_0$$

$$c_4 = \gamma + \pi . c_0$$

Figure-6.19: The Carry Generation unit performing the Carry group computation

Such unit can be implemented in various ways, according to the allowed level of abstraction. In a CMOS process, 17 transistors are able to guarantee the static function (Figure 6.20). However this design requires a careful sizing of the transistors put in series.

The same design is available with less transistors in a dynamic logic design. The sizing is still an important issue, but the number of transistors is reduced (Figure 6.21).

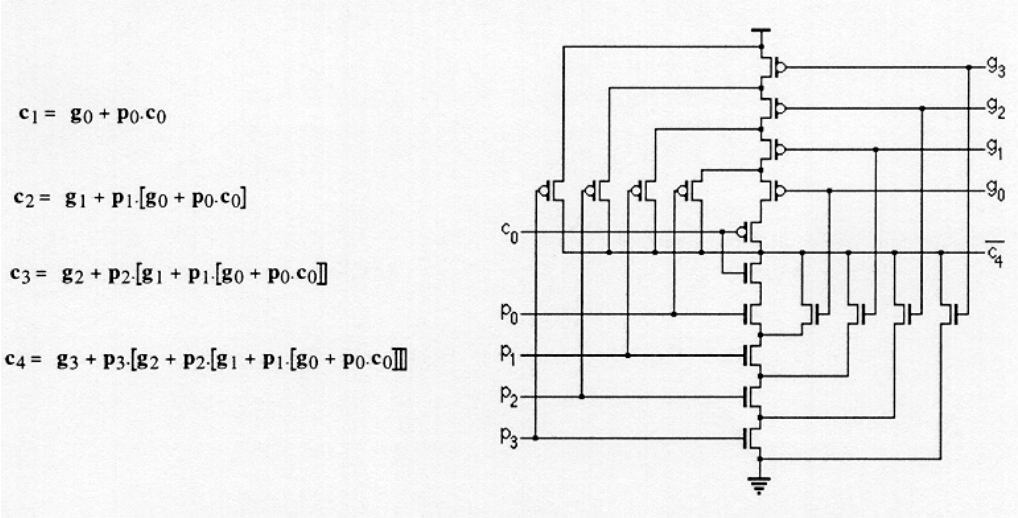


Figure-6.20: Static implementation of the 4-bit carry lookahead chain

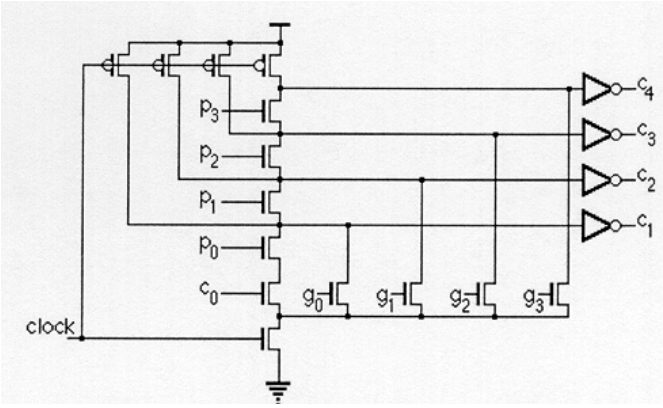


Figure-6.21: Dynamic implementation of the 4-bit carry lookahead chain

Figure 6.22 shows the implementation of 16-bit CLA adder.

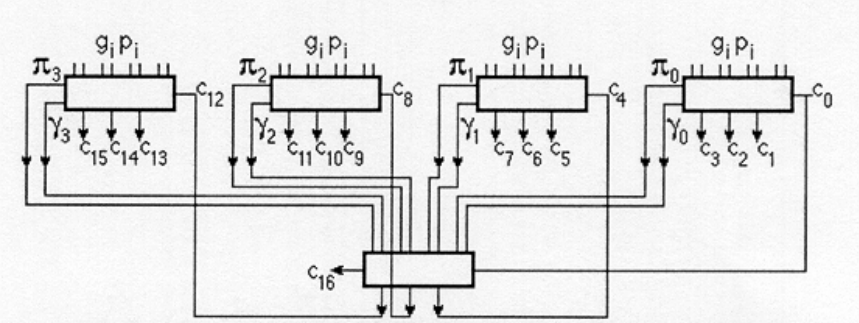


Figure-6.22: Implementation of a 16-bit CLA adder

Multipliers

Introduction

Multiplication can be considered as a series of repeated additions. The number to be added is the multiplicand, the number of times that it is added is the multiplier, and the result is the product. Each step of the addition generates a partial product. In most computers, the operands usually contain the same number of bits. When the operands are interpreted as integers, the product is generally twice the length of the operands in order to preserve the information content. This repeated addition method that is suggested by the arithmetic definition is slow that it is almost always replaced by an algorithm that makes use of positional number representation.

It is possible to decompose multipliers in two parts. The first part is dedicated to the generation of partial products, and the second one collects and adds them. As for adders, it is possible to enhance the intrinsic performances of multipliers. Acting in the generation part, the Booth (or modified Booth) algorithm is often used because it reduces the number of partial products. The collection of the partial products can then be made using a regular array, a Wallace tree or a binary tree

Serial-Parallel Multiplier

This multiplier is the simplest one, the multiplication is considered as a succession of additions.

$$\text{if } A = (a_n\ a_{n-1}\ldots\ldots a_0) \text{ and } B = (b_n\ b_{n-1}\ldots\ldots b_0)$$

The product A.B is expressed as :

$$A.B = A.2^n.b_n + A.2^{n-1}.b_{n-1} + \ldots + A.2_0.b^0$$

The structure of Figure 6.23 is suited only for positive operands. If the operands are negative and coded in 2's complement:

1. The most significant bit of B has a negative weight, so a subtraction has to be performed at the last step.

2. Operand A.2^k must be written on 2N bits, so the most significant bit of A must be duplicated. It may be easier to shift the content of the accumulator to the right instead of shifting A to the left.

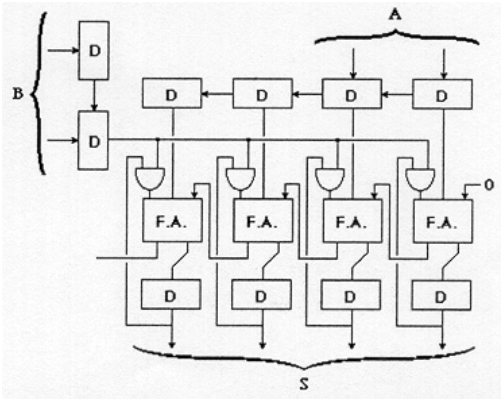


Figure-6.23: Serial-Parallel multiplier

Braun Parallel Multiplier

The simplest parallel multiplier is the Braun array. All the partial products A.bk are computed in parallel, and then collected through a cascade of Carry Save Adders. At the bottom of the array, the output of the array is noted in Carry Save, so an additional adder converts it (by the mean of carry propagation) into the classical notation (Figure 6.24). The completion time is limited by the depth of the carry save array, and by the carry propagation in the adder. Note that this multiplier is only suited for positive operands. Negative operands may be multiplied using a Baugh-Wooley multiplier.

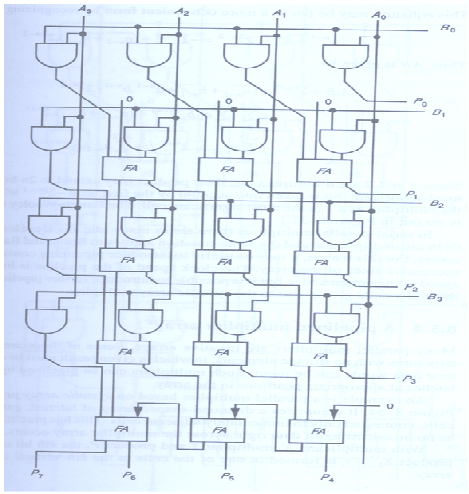


Figure 6.24: A 4-bit Braun Array

Baugh-Wooley Multiplier

This technique has been developed in order to design regular multipliers, suited for 2's-complement numbers.

Let us consider 2 numbers A and B:

$$A = (a_{n-1} \dots a_0) = -a_{n-1}.2^{n-1} + \sum_{0}^{n-2} a_i.2^i$$

$$B = (b_{n-1} \dots b_0) = -b_{n-1}.2^{n-1} + \sum_{0}^{n-2} b_i.2^i$$

The product A.B is given by the following equation:

$$A.B = a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_i.2^{i+j} - a_{n-1}\sum_{0}^{n-2} b_i.2^{n+i-1} - b_{n-1}\sum_{0}^{n-2} a_i.2^{n+i-1}$$

We see that subtraction cells must be used. In order to use only adder cells, the negative terms may be rewritten as:

$$-a_{n-1}\sum_{0}^{n-2} b_i.2^{i+n-1} = a_{n-1}.\left(-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{b_i}.2^{i+n-1}\right)$$

By this way, A.B becomes:

$$A.B = a_{n-1}.b_{n-1}.2^{2n-2} + \sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_j.2^{i+j}$$

$$+ b_{n-1}\left[-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{a_i}.2^{i+n-1}\right]$$

$$+ a_{n-1}\left[-2^{2n-2} + 2^{n-1} + \sum_{0}^{n-2} \overline{b_i}.2^{i+n-1}\right]$$

The final equation is:

$$A.B = -2^{2n-1} + \left(\overline{a_{n-1}} + \overline{b_{n-1}} + a_{n-1}.b_{n-1}\right).2^{2n-2}$$

$$+\sum_{0}^{n-2}\sum_{0}^{n-2} a_i.b_j.2^{i+j} + \left(a_{n-1} + b_{n-1}\right).2^{n-1}$$

$$+\sum_{0}^{n-2} b_{n-1}.\overline{a_i}.2^{i+n-1} + \sum_{0}^{n-2} a_{n-1}.\overline{b_i}.2^{i+n-1}$$

because:

$$-\left(b_{n-1} + a_{n-1}\right).2^{2n-2} = -2^{2n-1} + \left(\overline{a_{n-1}} + \overline{b_{n-1}}\right).2^{2n-2}$$

A and B are n-bits operands, so their product is a 2n-bits number. Consequently, the most significant weight is 2n-1, and the first term -2^{2n-1} is taken into account by adding a 1 in the most significant cell of the multiplier. The implementation is shown in figure 6.25.

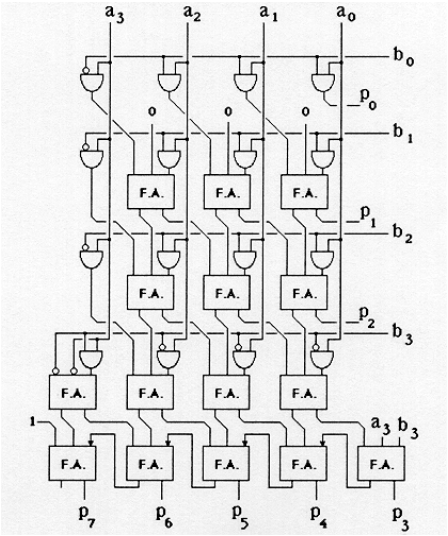


Figure-6.25: A 4-bit Baugh-Wooley Multiplier

Booth Algorithm

This algorithm is a powerful direct algorithm for signed-number multiplication. It generates a 2n-bit product and treats both positive and negative numbers uniformly. The idea is to reduce the number of additions to perform. Booth algorithm allows in the best case n/2 additions whereas modified Booth algorithm allows always n/2 additions.

Let us consider a string of k consecutive 1s in a multiplier:
..., i+k, i+k-1, i+k-2 , ..., i, i-1, ...
..., 0 , 1 , 1 , ..., 1, 0, ...

where there is k consecutive 1s.

By using the following property of binary strings:

$$2^{i+k}-2^{i}=2^{i+k-1}+2^{i+k-2}+...+2^{i+1}+2^{i}$$

the k consecutive 1s can be replaced by the following string

..., i+k+1, i+k, i+k-1, i+k-2, ..., i+1, i , i-1 , ...
..., 0 , 1 , 0 , 0 , ..., 0 , -1 , 0 , ...
k-1 consecutive 0s Addition Subtraction

In fact, the modified Booth algorithm converts a signed number from the standard 2's-complement radix into a number system where the digits are in the set {-1,0,1}. In this number system, any number may be written in several forms, so the system is called redundant.

The coding table for the modified Booth algorithm is given in Table 1. The algorithm scans strings composed of three digits. Depending on the value of the string, a certain operation will be performed.

A possible implementation of the Booth encoder is given on Figure 6.26.

Table-1: Modified Booth coding table

	BIT			**M is**
2^1	**2^0**	**2^{-1}**	**OPERATION**	**multiplied**
Y_{i+1}	**Y_i**	**Y_{i-1}**		**by**
0	0	0	add zero (no string)	+0
0	0	1	add multipleic (end of string)	+X
0	1	0	add multiplic. (a string)	+X
0	1	1	add twice the mul. (end of string)	+2X
1	0	0	sub. twice the m. (beg. of string)	-2X
1	0	1	sub. the m. (-2X and +X)	-X
1	1	0	sub . the m. (beg. of string)	-X
1	1	1	sub. zero (center of string)	-0

.

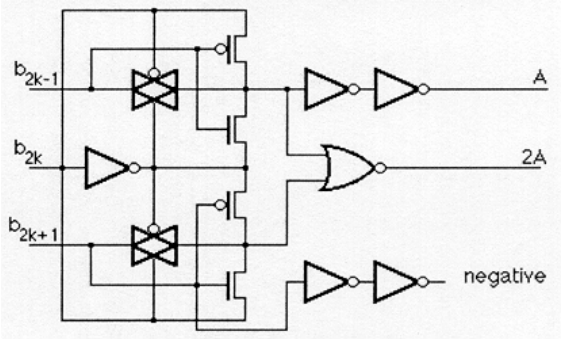


Figure-6.26: Booth encoder cell

To summarize the operation:

- Grouping multiplier bits into pairs
  - Orthogonal idea to the Booth recoding
  - Reduces the num of partial products to half
  - If Booth recoding not used ➔ have to be able to multiply by 3 (hard: shift+add)
- Applying the grouping idea to Booth ➔ Modified Booth Recoding (Encoding)
  - We already got rid of sequences of 1's ➔ no multiplication by 3
  - Just negate, shift once or twice

Wallace Trees

For this purpose, Wallace trees were introduced. The addition time grows like the logarithm of the bit number. The simplest Wallace tree is the adder cell. More generally, an n-inputs Wallace tree is an n-input operator and log2(n) outputs, such that the value of the output word is equal to the number of "1" in the input word. The input bits and the least significant bit of the output have the same weight (Figure 6.27). An important property of Wallace trees is that they may be constructed using adder cells. Furthermore, the number of adder cells needed grows like the logarithm log2(n) of the number n of input bits. Consequently, Wallace trees are useful whenever a large number of operands are to add, like in multipliers. In a Braun or Baugh-Wooley multiplier with a Ripple Carry Adder, the completion time of the multiplication is proportional to twice the number n of bits. If the collection of the partial products is made through Wallace trees, the time for getting the result in a carry save notation should be proportional to log2(n).

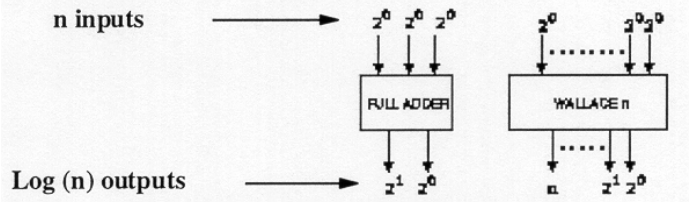


Figure-6.27: Wallace cells made of adders

Figure 6.28 represents a 7-inputs adder: for each weight, Wallace trees are used until there remain only two bits of each weight, as to add them using a classical 2-inputs adder. When taking into account the regularity of the interconnections, Wallace trees are the most irregular.

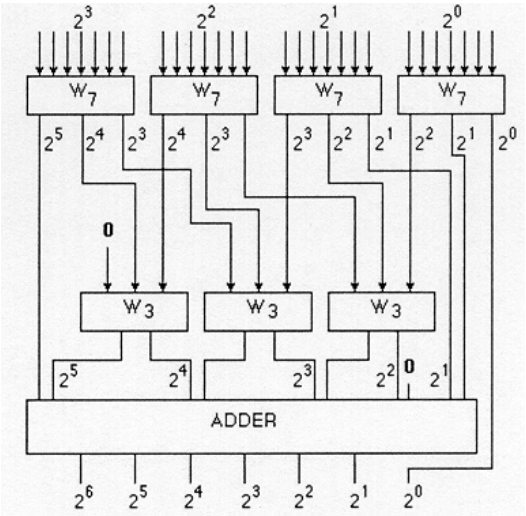


Figure-6.28: A 7-inputs Wallace tree

To summarize the operation:

The Wallace tree has three steps:

- Multiply (that is - AND) each bit of one of the arguments, by each bit of the other, yielding n^2 results.
- Reduce the number of partial products to two by layers of full and half adders.
- Group the wires in two numbers, and add them with a conventional adder.

The second phase works as follows.

- Take any three wires with the same weights and input them into a full adder.
- The result will be an output wire of the same weight and an output wire with a higher weight for each three input wires.
- If there are two wires of the same weight left, input them into a half adder.
- If there is just one wire left, connect it to the next layer.