TWO METHODS FOR FAST INTEGER BINARY-BCD CONVERSION

F.A. Schreiber, R. Stefanelli

Istituto di Elettrotecnica ed Elettronica
Politecnico di Milano
Piazza L. da Vinci 32
20133 Milano, Italy

Abstract

Two methods for performing binary-BCD conversion of positive integers are discussed. The principle which underlies both methods in the repeated division by five and then by two, obtained the first by means of substractions performed from left to right, the second by shifting bits before next subtraction.

It is shown that these methods work in a time which is linear with the length in bit of the number to be converted.

A ROM solution is proposed and its complexity is compared with that of other methods.

1 - Introduction

The problem of converting the binary numbers, used by the majority of arithmetical units in modern computers, into a decimal output form, which is still the most familiar to human users, has been considered by many authors, for integers as well as for fractionary numbers.

The core of the conversion process for integers is the repeated division of the binary number by ten; generally speaking, division is a complex and time consuming operation and the different methods which have been proposed are focused on fast algorithms which take advantage of the peculiarities of the divide-by-ten operation.

The integrated circuit technology made the hardware solution competitive with the traditional software routines from the point of view of the speed Vs. cost x bit ratio /1/, /2/.

In the following, two versions of a hardware con - verter are described for converting positive binary integers, presented in parallel form, into Binary Coded Decimals, which use a combinatorial array divider to divide by five and right shifts, obtained by appropriated wiring, to divide further by two.

The methods result in regular structures the conversion time of which grows linearly with the number of bits to be converted.

2 - The divide-by-five method

When converting a binary integer B to its BCD representation, the i-th decimal digit D_i is the remainder of the division of the binary integer B_i by ten that is /3/

$$D_i = B_i \bmod 10 = 2*(\lfloor B_i/2 \rfloor \bmod 5)+b_{io} \qquad (1)$$

where B_{io} is the least significant bit of B_i, i.e. $b_{io} = B_i \bmod 2$ and $\lfloor A \rfloor$ denotes the integer part of A. Moreover

$$B_{i+1} = \lfloor \lfloor (B_i - D_i)/2 \rfloor /5 \rfloor \qquad (2)$$

can be used in (1) for D_{i+1}, and the division can be iterated until the last (most significant) decimal digit is obtained, when the last binary integer has no digits left.

If B is a multiple of five, to divide B by five we can use the following relations

$$B = 5P \qquad P = B - 4P \qquad (3)$$

that is the quotient can be obtained by subtracting from B the quotient itself shifted left twice. As it can be seen from the following example the two least significant bits of the quotient correspond to the least significant bits of B

$$\begin{array}{cccccccc}
0 & 0 & b_n & b_{n-1} & b_{n-2}\cdots b_3 & b_2 & b_1 & b_0 \\
p_n & p_{n-1} & p_{n-2} & p_{n-3} & p_{n-4}\cdots p_1 & p_0 & 0 & 0 \\
\hline
0 & 0 & p_n & p_{n-1} & p_{n-2}\cdots p_3 & p_2 & p_1 & p_0
\end{array} \qquad (4)$$

In this way it is possible to calculate the result going from right to left, with a possible borrow which propagates in the same direction.

However this method can be applied only if the division has a null remainder. Otherwise we should add, as a borrow, at the beginning of the subtraction process, the very remainder we are looking for!

Things would go better if we could make the subtraction from left to right. This is really possible if we solve equation (3) as

$$4P = B - P$$

In this case the last generated borrow represents the remainder of the division by-five, while the difference, shifted right twice, consitutes the next dividend.

$$\begin{array}{cccccccccc} 0 & 0 & b_n & b_{n-1} & b_{n-2} \cdots b_3 & b_2 & b_1 & b_o & & \\ 0 & 0 & p_n & p_{n-1} & p_{n-2} \cdots p_3 & p_2 & p_1 & p_o & & \\ \hline p_n & p_{n-1} & p_{n-2} & p_{n-3} & p_{n-4} \cdots p_1 & p_o & 0 & 0 & & \end{array} \qquad (5)$$

Two methods based on the same technique will now be described.

In/1/Nicoud also suggests a method which works from left to right. In the appendix a comparison is made between the methods.

2.1 - The grouping-by-two method

For the first method, let us put, for the sake of semplicity, the following positions

$$A = B \text{ base } 4 \quad ; \quad Q = P \text{ base } 4$$

that is we consider the bits grouped by two. With such positions we get the bounds

$$0 \leqslant a_i \leqslant 3 \; ; \; 0 \leqslant q_i \leqslant 3 \; ; \; r_i \equiv \{0,1\} \qquad (6)$$

We must now establish the internal structure of a combinatorial cell wich can make the subtraction of any two bits: figure 1 shows the cell as a black-box. After a short reflection one finds the following equation

$$s_i = a_i - q_i + 4r_i = q_{i-1} + r_{i-1} \qquad (7)$$

With the limits given by (6), the left hand side of equation (5) tells us that

$$-3 \leqslant s_i \leqslant 7 \qquad (8)$$

while the right hand side of equation (7) allows for

$$0 \leqslant s_i \leqslant 4 \qquad (9)$$

Since both sides must be satisfied, we shall "a priori" exclude all the input configurations for which

$$-3 \leqslant s_i < 0 \quad \text{or} \quad 4 < s_i \leqslant 7$$

Moreover many ambiguous situations exist since, for given values of a_i, q_i and r_i, there are different possible combinations of q_{i-1} and r_{i-1} which satisfy equation (7).

In these cases the ambiguity can be solved looking one position ahead and imposing that $a_{i-1} - q_{i-1} + r_{i-1}$ stays in the limits of (9).

In this way we can build Table 1, which describes a 4-inputs-2-outputs combinatorial function. (Fig. 1).

				$a_{i-1} < s_i$		$a_{i-1} \geqslant s_i$	
s_i	a_i	q_i	r_i	q_{i-1}	r_{i-1}	q_{i-1}	r_{i-1}
0	0	0	0	X	X	0	0
0	1	1	0	X	X	0	0
0	2	2	0	X	X	0	0
0	3	3	0	X	X	0	0
1	1	0	0	0	1	1	0
1	2	1	0	0	1	1	0
1	3	2	0	0	1	1	0
1	0	3	1	0	1	1	0
2	2	0	0	1	1	2	0
2	3	1	0	1	1	2	0
2	0	2	1	1	1	2	0
2	1	3	1	1	1	2	0
3	3	0	0	2	1	3	0
3	0	1	1	2	1	3	0
3	1	2	1	2	1	3	0
3	2	3	1	2	1	3	0
4	0	0	1	3	1	X	X
4	1	1	1	3	1	X	X
4	2	2	1	3	1	X	X
4	3	3	1	3	1	X	X

Table 1

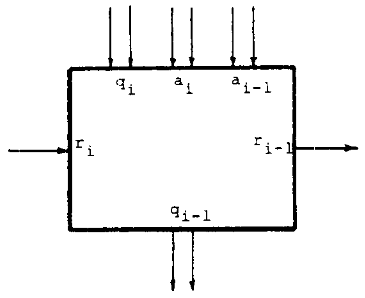


Figure 1

2.2 - The single bit method

The second method prescinds from bit grouping, since it considers each bit of the subtrahend by its own.

In this case, by direct inspection of (5), we find the expression for the left-to-right subtraction

$$b_i - (p_i + r_{i-1}) = p_{i-2} - 2r_i$$

which is easely rewritten as

$$b_i - p_i + 2r_i = p_{i-2} + r_{i-1} \qquad (10)$$

Expression (10) defines the cell of fig. 2.

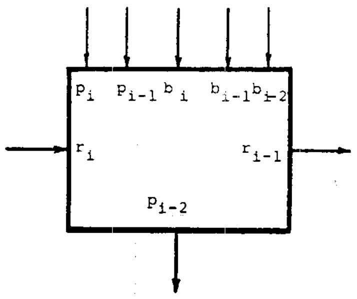


Figure 2

Since the right hand side of (10) is always positive we can admit as input configurations only those satisfying the following limits

$$0 \leqslant b_i - p_i + 2r_i \leqslant 3 \qquad (11)$$

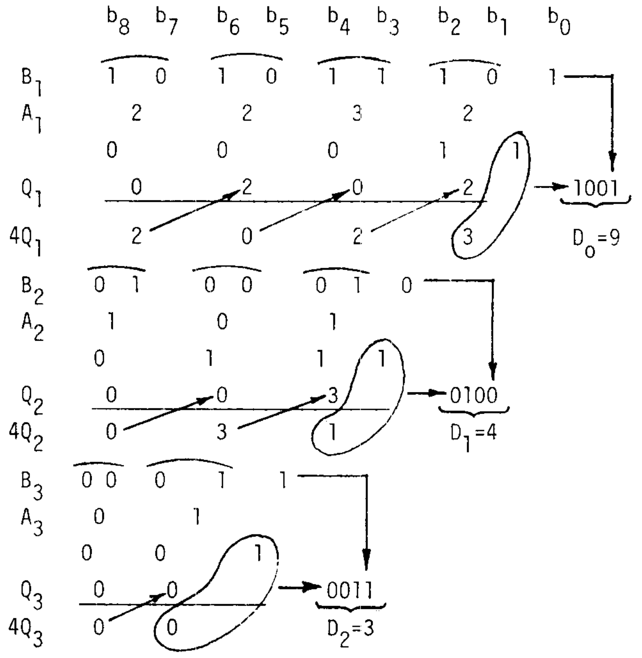
These limits are used in resolving ambiguous conditions which arise also with this algorithm. In this case however we must go ahead two position to determine the right values for p_{i-2} and r_{i-1}. The combinatorial function is given in Table 2.

3 - The structure of the array converter

We must now arrange the combinatorial cells in such a way that they can iteratively solve equation (7). This is made in Figure 3 for the grouping-by-two algorithm.

In the triangular array, cells belonging to a row accomplish the division by-five. The different rows work on the successive residues unti Ø is obtained as a quotient.

Since the least significant bit b_{io} must be appended to the remainder, one shift right of the quotient must be made before grouping bits for the next division. The division by-two is obtained by scaling all the row-cells one place right. The multiplication by-two is automatically obtained by appending b_{io} to the right of the remainder. We can follow the converter operation in the following example (borrows are appended as a left exponent to the minuend):



That is: $\underbrace{101011101}_{B} \longrightarrow \underbrace{0011\ 0100\ 1001}_{D} = 349$

b_i	p_i	r_i	b_{i-1}	p_{i-1}	b_{i-2}	p_{i-2}	r_{i-1}
0	0	0	X	X	X	0	0
0	0	1	X	X	X	1	1
0	1	0	X	X	X	-	-
0	1	1	0	0	0	0	1
			0	0	1	1	0
			0	1	X	0	1
			1	0	X	1	0
			1	1	0	0	1
			1	1	1	1	0
1	0	0	0	0	0	0	1
			0	0	1	1	0
			0	1	X	0	1
			1	0	X	1	0
			1	1	0	0	1
			1	1	1	1	0
1	0	1	X	X	X	-	-
1	1	0	X	X	X	0	0
1	1	1	X	X	X	1	1

Table 2

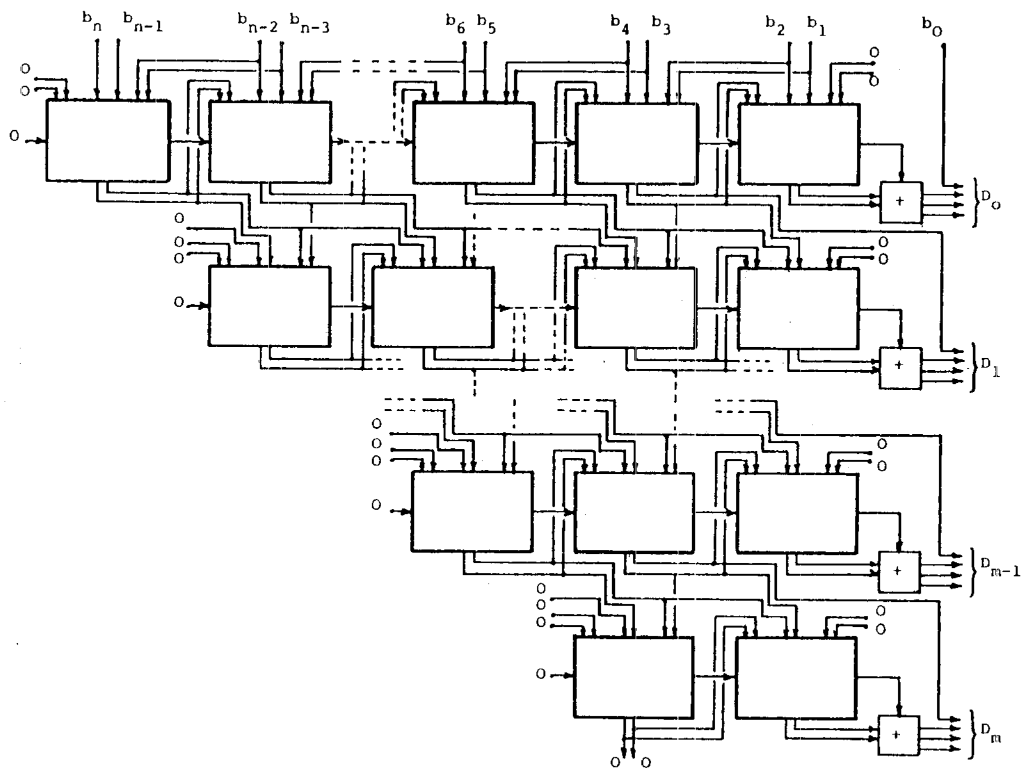


Figure 3

It must be noticed that if the lenght of the sub - trahend is even, a Ø must be considered for the most significant bit.

The time taken for the conversion process is propor-tional to the number n of bits to be converted *1*. More precisely, since the time taken from a single cell to produce the quotient q_{i-1} is the same as that to produce the borrow r_{i-1}, we can call this time the "cell delay" t_{cd}. Moreover, while the bor-row propagates only horizontally from one cell to the other, the q_i bits must propagate also vertical-ly, and the cells in the next row owing to the shift right, must wait until bits of the partial quotients q_i and q_{i-1} are ready. For this reason the vertical propagation time between any two rows is $2*t_{cd}$.
Since the number of rows is $\lfloor(n-1)/2\rfloor - 1$, the propagation time from the first to the second row is t_{cd} only, and to complete the operation one more horizontal step is required, we conclude that, <u>in the worst-case</u>, the conversion time is

$$T = \underbrace{\left(\left\lfloor \frac{n-1}{2} \right\rfloor - 1\right)*2t_{cd}}_{\text{propagation delay}} + \underbrace{2\tau}_{\text{sum delay}} \qquad (12)$$

where the second term takes into account the time to sum the remainder bits, being τ the delay of one logical gate.

The number N of cells required to convert n bits is <u>at most</u> *2*

---

1 Please notice that the least significant bit of the number to be converted is not processed, but it is only appended to the remainder. Therefore the actual lenght of the examples shown is n + 1 bits.

2 We noticed above that the general grouping-by-two structure works on an odd number of bits. If the number of bits is even, the values of T and N given here can be greatly improved. As an example, conversion of a 16 bits number requires only 23 cells and a time $T = 7\ t_{cd} + 2\tau$(see the example in the appendix).

$$N = \sum_{i=0}^{\mathrm{int}(\frac{n-1}{2})-1} (2+i)$$

Figure 4 shows the array when the non-grouping algorithm is used.

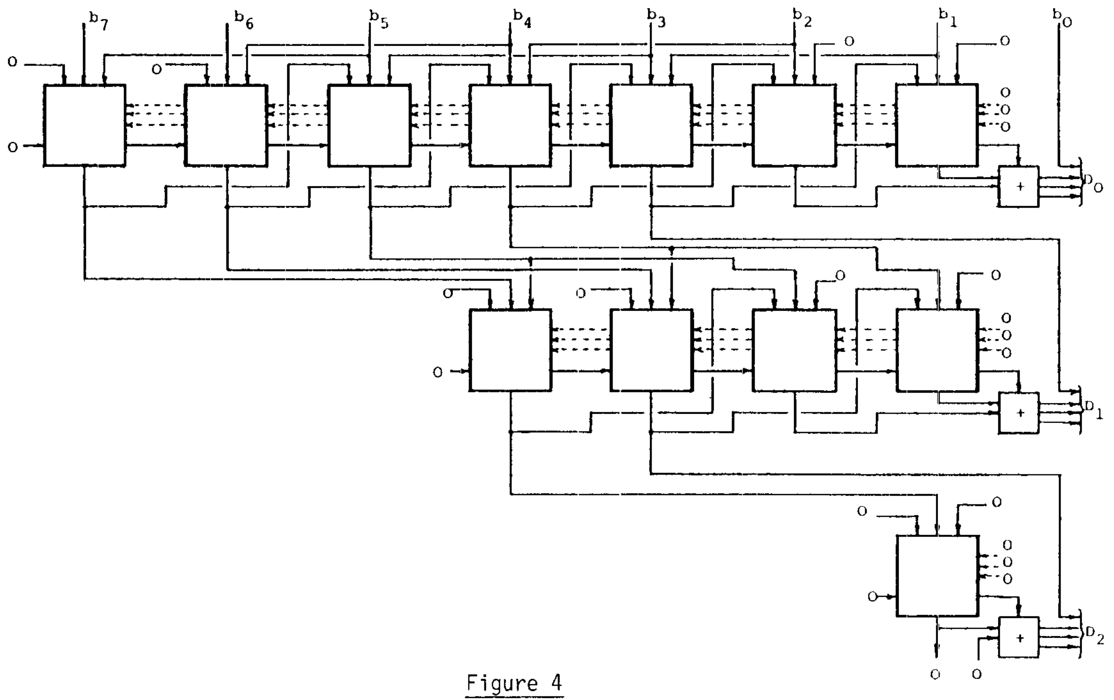
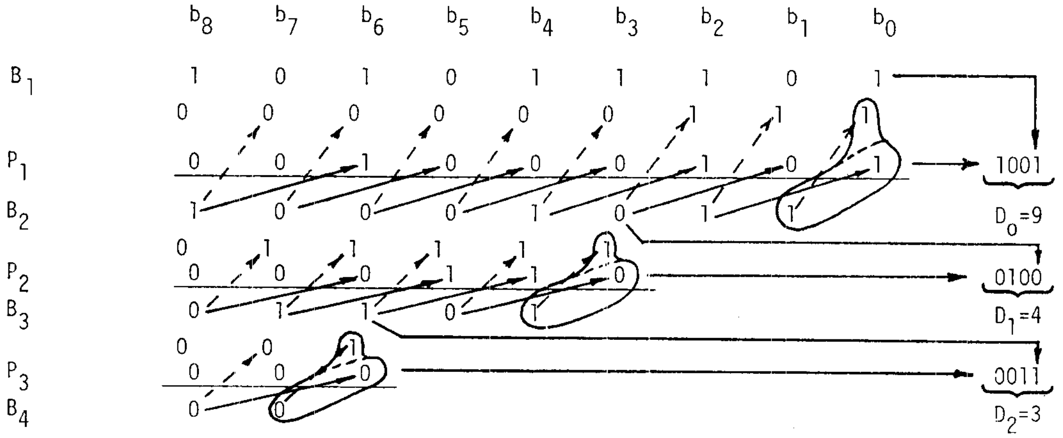


Figure 4

In this case there are three paths which propagate from right to left. This is true only appearently; in fact these paths represent partial results obtained from cell i-1, which computes them from data obtained from cell i+1 so that no time delay is introduced through these lines.

The following example shows the operation of the single bit method



If again t_{cd} represents the internal delay of a cell, the maximum horizontal propagation delay is $(n-1) * t_{cd}$. The vertical propagation delay is calculated by noticing that the i-th cell must wait until b_{i-2} is ready before producing p_{i-2} and r_{i-1}, so that a total delay of $3 * t_{cd}$ is introduced by each cell. However by inspection of the structure of figure 4 we find that the maximum length path takes

$$\left\lfloor \frac{n-1}{3} \right\rfloor * 3t_{cd} + (n-1) \bmod 3*t_{cd} = (n-1)*t_{cd}$$

so that the conversion time becomes

$$T = \underbrace{(n-1) * t_{cd}}_{\text{propagation delay}} + \underbrace{2\tau}_{\text{sum delay}} \qquad (13)$$

The number N of cells required to convert n bits is

$$N = \sum_{0}^{\left\lceil \frac{n-1}{3} \right\rceil}{}_{i} \left\lceil (n-1)-3i \right\rceil$$

By comparing expressions (12) and (13) it results that the grouping by two converter is slightly faster. However this result depends on the actual values of the cell delays t_{cd} which, in turn, heavily depend on their implementation.

It is important to notice, however, that the conversion time for both algorithms is proportional to the number of bits to be converted, a fact which compares favourably with times given in /2/ and /3/ .

As to the actual implementation issues, since they are heavily technology dependent; we are giving some comparison results in the appendix.

4 - References

/1/ J.D. Nicoud - "Iterative Arrays for Radix Conversion", IEEE Trans. Comp., Vol. C-20, Dec. 71, pp. 1479-1489.

/2/ J.F. Couleur - "BIDEC - A Binary-to-Decimal or Decimal-to-Binary Converter", IRE Trans. on Electronic Computers, Dec. 1958, pp. 313-316.

/3/ R.L. Sites - "Serial Division by ten", IEEE Trans. Comp., Vol. C-23, Dec. 1974, pp. 1299-1301.

Appendix

A rather straightforward way for implementing the combinatorial cell defined by Table 1, is to implement two functions for both cases $a_{i-1} < s$ and $a_{i-1} \geq s$. Such functions which have a five variables input $(a_{1,i}, a_{2,i}, q_{1,i}, q_{2,i}, r_i)$ and a three variable output $(q_{1,i-1}, q_{2,i-1}, r_{i-1})$ can be easily found as a two level minimal form using, for example, a simple method such as Karnaugh maps. A circuit is implemented apart with a two cell parallel adder for calculating $a_i + \overline{q}_i$ and an AND gate to add further $4r_i$, a two cell XOR circuit to make the comparison with a_{i-1}, and AND gates to make the selection between the two functions.

The boolean equations resulting for the functions are the following (we drop the index i):

a) $\dfrac{s \lesssim a_{i-1}}{r_{i-1} = 0}$

$$q_{1,i-1} = a_1\, r + a_2\, r + \overline{q}_1\, r + q_2\, r + a_1\overline{q}_1\overline{q}_2 + a_1a_2\overline{q}_1$$

$$q_{2,i-1} = a_2\, \overline{q}_2 + \overline{a}_2\, q_2$$

b) $\dfrac{s > a_{i-1}}{r_{i-1} = 1}$

$$q_{1,i-1} = a_1 r + \overline{q}_1\, r + a_1a_2\overline{q}_1\overline{q}_2 + \overline{a}_1\, a_2\, q_1\, \overline{q}_2$$

$$q_{2,i-1} = a_2\, q_2 + \overline{a}_2\, \overline{q}_2$$

We can notice that these functions can be implemented with a three level combinatorial network reducing the total gate count to 12.

The comparison circuit however requires a five level network and the selection circuit another two levels so that the total cell delay for this implementation amounts to $t_{cd} = 7\tau$, if τ is the single gate delay.

The cell defined by the combinatorial function of Table 2, which is a six inputs-two outputs function, is also described by the following equations:

$$p_{i-2} = r_i\, (\overline{b_i \oplus p_i}) + (b_i \oplus p_i)\, x_i$$

$$r_{i-1} = r_i\, (\overline{b_i \oplus p_i}) + (b_i \oplus p_i)\, \overline{x}_i$$

where

$$x_i = (\overline{b_{i-1} \oplus p_{i-1}})\, b_{i-2} + b_{i-1}\, \overline{p}_{i-1}$$

It is possible therefore to implement it with a four level AND/OR network. It is interesting to notice that, since OR gates can have at most one input at logic level "1" they can be realized as wired OR" connection, so reducing the number of levels to two only.

Signals involving b_{i-1} and p_{i-1} can be drawn from the first and second level gates of the i-1th cell, so reducing the overall circuit complexity (fig. 5).

The total delay for this cell is therefore at most $t_{cd} = 4\tau$.

However the fastest solutions for combinatorial functions implementation are to be found, with today and probably even more with future technologies, by using ROM's.

For the sake of comparison let us focus on a ROM implementation for the combinatorial cell defined by Tab. 1, which produces the fastest structure.

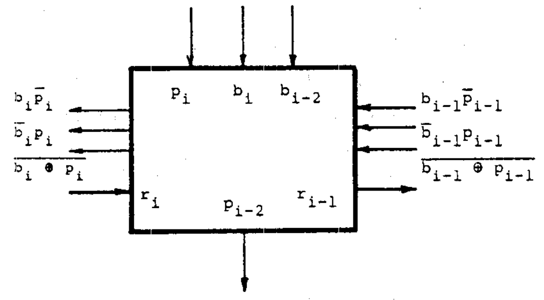


Figure 5

This cell, as shown in fig. 1, is a 7 bits input 3 bits outputs one and it could be implemented by a 128x4-bits words ROM.

However this would be a rather inefficient solution, since many of the 128 words would remain unused. Then let us decompose the cell as shown in figure 6.a; the upper subcell evaluates s_i, while the lower subcell produces the outputs. If we consider a row in figure 3 we can shift right 1 place the lower subcell to obtain the structure of fifure 6.b.

In this case the cell has a 5 bits input (2 for a_i and 3 for s_{i+1}) and a 5 bits output (2 for q_i and 3 for s_i) which make a 32x5-bits words ROM .

We can now compare our two algorithms and Nicoud's one in terms of number of bits required in a ROM implementation of the resulting structure. Let us consider an infinite row of cells (an infinite column for Nicourd's algorithm) and let us take n cells; since the cell of Tab. 1 is used to convert 2 bits, we must compare it with twice the number of cells for the algorithm of Tab. 2 and for Nicoud's algorithm, which both work on a single bit.

If now we should like to implement the n cells with a single ROM the number of bits in it would be given by

$$\underbrace{(n^0 \text{ of output bits})}_{\text{word length}} * 2^{\overbrace{(n^0 \text{ of input bits})}^{\text{address length}}}$$

Table 3 shows the ROM dimensions for the considered methods. It results from it that the grouping-by-two method requires the smallest ROM.

Moreover, if instead of considering an infinite row, we consider a finite one to be implemented with a single ROM, the boundary conditions for the grouping-by-two algorithm show that the most significant cell

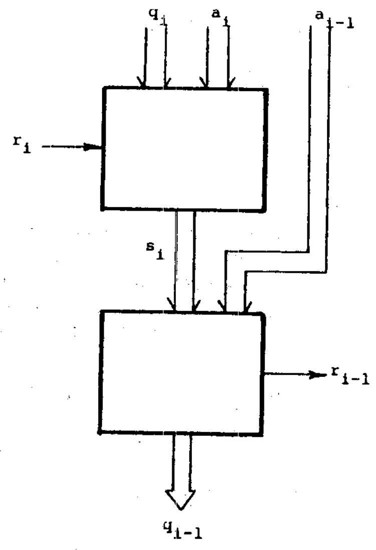


Figure 6.a

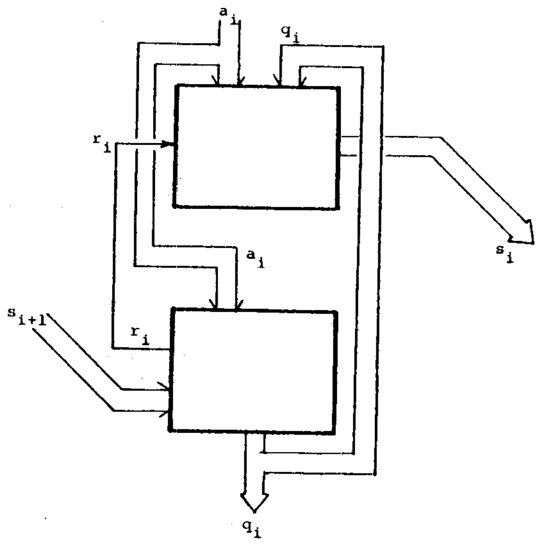


Figure 6.b

Method		n^o of bits in the ROM
grouping-by-two	(Tab. 1)	$(2n + 3) * 2^{(2n+3)}$
Nicoud's	/1/	$(2n + 3) * 2^{(2n+4)}$
single bit	(Tab. 2)	$(2n + 2) * 2^{(2n+5)}$

n = n^o of cells = n^o of bit couples

Table 3

(index m) reduces to the external input a_m, since $r_m = 0$, $q_m = 0$, and $s_{m+1} = 0$; the least significant cell (index 0) reduces to the lower subcell alone having the external input $a_o = 0$ which, being a constant, needs not to be implemented as an actual input, and the external outputs q_o and r_o.

The ROM implementation of an n-cells row in figure 3 requires then a number of bits given by

$$(2n + 1) * 2^{2n}$$

As a final example, a 16 bit converter is shown in Figure 7. All the intermediate results are also shown in the conversion process from $(1111111111111111)_2$ to $(65535)_{10}$; for clarity reasons, the values of the intermediate s_i are not shown in the figure.

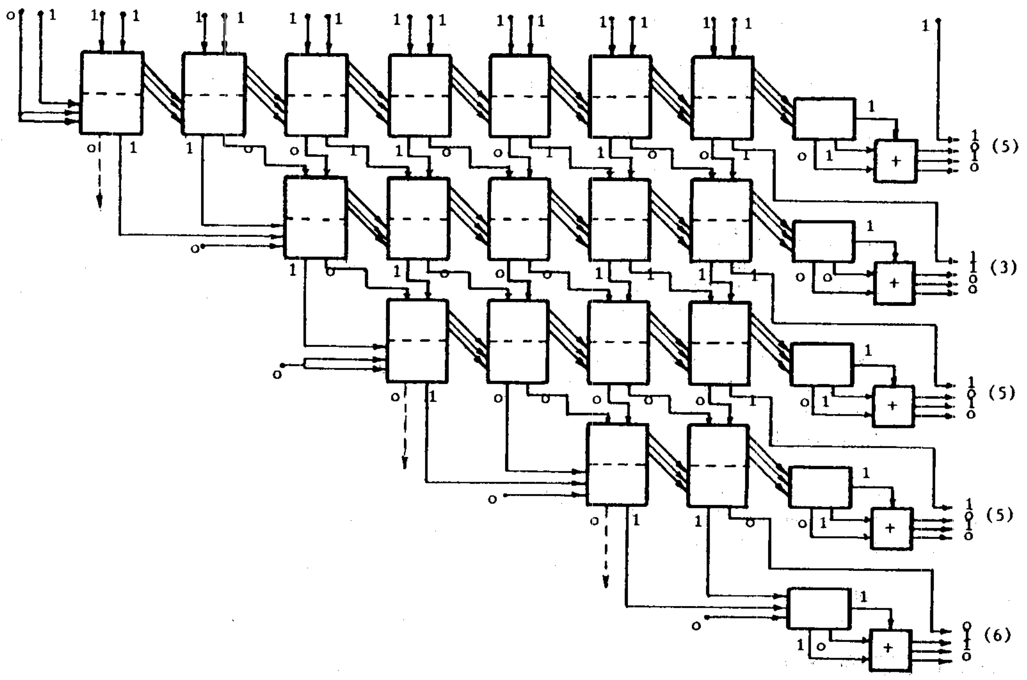


Figure 7